Opérateurs arithmétiques matériels
Multiplieurs

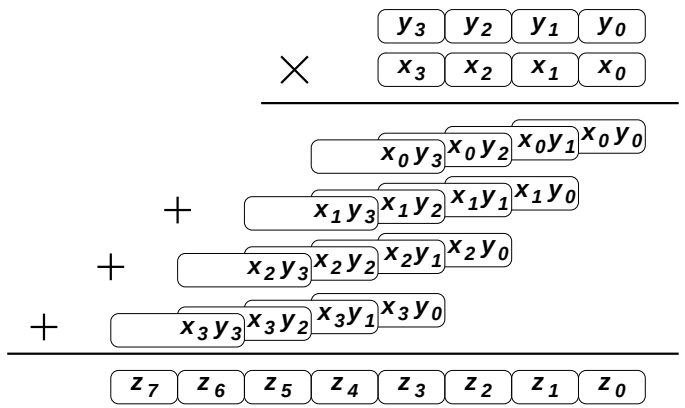
Florent de Dinechin

20 octobre 2005

Intro

Recyclage de transparent

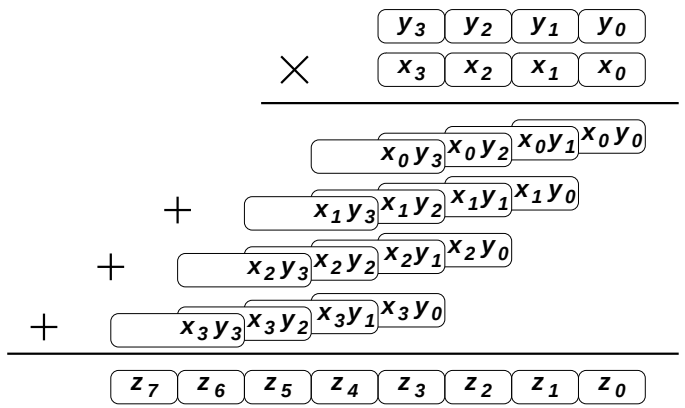
1: **for all** $i, j \in \{0..n-1\}$ **do**
2: $p_{ij} = x_i \times y_j$ *Calculer des produits partiels (sur deux chiffres)*
3: **end for**
4: Les sommer tous comme on peut



Le gros de la recherche concernant les multiplieurs, c'est trouver de bonnes manières de faire cette sommation...

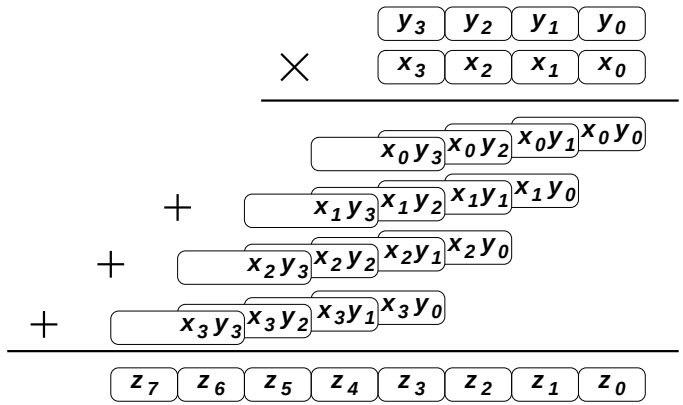
Recyclage de transparent

1: **for all** $i, j \in \{0..n-1\}$ **do**
2: $p_{ij} = x_i \times y_j$ *Calculer des produits partiels (sur deux chiffres)*
3: **end for**
4: Les sommer tous comme on peut



Le gros de la recherche concernant les multiplieurs, c'est trouver de bonnes manières de faire cette sommation...
D'ailleurs presque tout ce qui va se dire ici peut servir à l'addition de $n > 2$ nombres.

Un paquet de bits pondérés

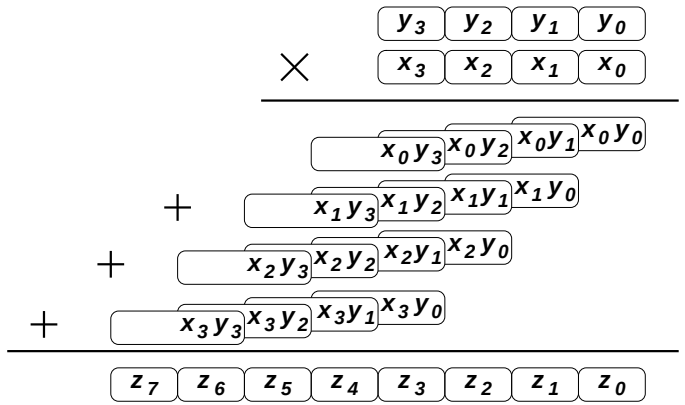


En base 2 j'ai un joli losange de bits pondérés

(par une puissance de 2).

Représentation dans le plan avec les poids en abscisse

Un paquet de bits pondérés



En base 2 j'ai un joli losange de bits pondérés

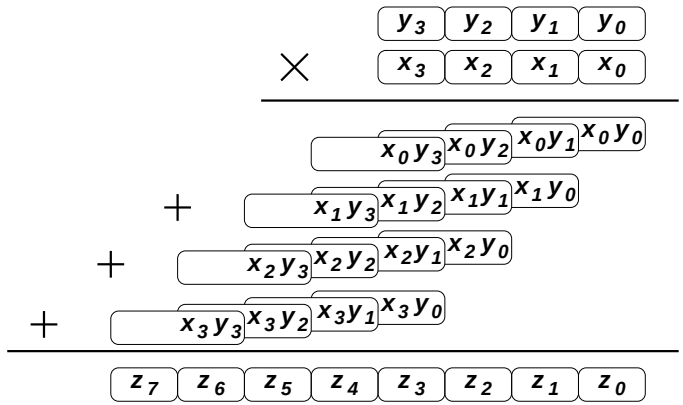
(par une puissance de 2).

Représentation dans le plan avec les poids en abscisse

Autres bases ? Le dessin il est un peu en base arbitraire, mais

- pour faire du matériel la base sera toujours une puissance de 2
- donc je pourrai toujours écrire chaque produit partiel comme un paquet de bits pondérés.

Un paquet de bits pondérés



En base 2 j'ai un joli losange de bits pondérés

(par une puissance de 2).

Représentation dans le plan avec les poids en abscisse

Autres bases ? Le dessin il est un peu en base arbitraire, mais

- pour faire du matériel la base sera toujours une puissance de 2
- donc je pourrai toujours écrire chaque produit partiel comme un paquet de bits pondérés.

Nouvelle formulation :

Le gros de la recherche concernant les multiplieurs, c'est trouver de bonnes manières de faire la somme d'un paquet de bits pondérés.

Grandes idées

- Obtenir le plus vite possible un paquet de bits pondérés le plus petit possible
- Faire la somme le plus vite possible, ou en le moins d'espace possible, ou...

Grandes idées

- Obtenir le plus vite possible un paquet de bits pondérés le plus petit possible
- Faire la somme le plus vite possible, ou en le moins d'espace possible, ou...

Et donc comme les additionneurs :

- plein de solutions algorithmiques (et de codages, et de recodages),
- certaines mieux adaptées que d'autres à la techno X ou Y ;
- travail sur les briques de base.

Grandes idées

- Obtenir le plus vite possible un paquet de bits pondérés le plus petit possible
- Faire la somme le plus vite possible, ou en le moins d'espace possible, ou...

Et donc comme les additionneurs :

- plein de solutions algorithmiques (et de codages, et de recodages),
- certaines mieux adaptées que d'autres à la techno X ou Y ;
- travail sur les briques de base.

Avertissement : un certain nombre des multiplieurs qui vont suivre sont obsolètes et inintéressants, et ne sont présenté que par souci de progression pédagogique.

Notations

- On multiplie X par Y
- X est le multiplicande
- Y est le multiplicateur
- La bestiole qu'on construit est un multiplieur
- C'est un vilain anglicisme mais au moins c'est clair
- Nos nombres X et Y seront sur n bits, ou $n+1$ bits quand cela me simplifiera les notations.
- Généralisations triviales pour multiplier des nombres de tailles différentes "multiplieurs rectangulaires"

Note sur la complexité

Montrons facilement que la multiplication est possible en temps logarithmique :

- Faisons tout en binaire

Note sur la complexité

Montrons facilement que la multiplication est possible en temps logarithmique :

- Faisons tout en binaire
- On a nos n lignes à additionner

Note sur la complexité

Montrons facilement que la multiplication est possible en temps logarithmique :

- Faisons tout en binaire
- On a nos n lignes à additionner
- On a un additionneur en temps constant (carry-save)

Note sur la complexité

Montrons facilement que la multiplication est possible en temps logarithmique :

- Faisons tout en binaire
- On a nos n lignes à additionner
- On a un additionneur en temps constant (carry-save)
- Yaka faire un arbre d'additions en temps $0(log_2 n)$

Note sur la complexité

Montrons facilement que la multiplication est possible en temps logarithmique :

- Faisons tout en binaire
- On a nos n lignes à additionner
- On a un additionneur en temps constant (carry-save)
- Yaka faire un arbre d'additions en temps $0(log_2 n)$
- Bien sûr on récupère un nombre carry-save, il faut encore $0(log_2 n)$ pour le convertir en binaire

Note sur la complexité

Montrons facilement que la multiplication est possible en temps logarithmique :

- Faisons tout en binaire
- On a nos n lignes à additionner
- On a un additionneur en temps constant (carry-save)
- Yaka faire un arbre d'additions en temps $0(log_2 n)$
- Bien sûr on récupère un nombre carry-save, il faut encore $0(log_2 n)$ pour le convertir en binaire
- Surface en $O(n \log_2 n)$

Note sur la complexité

Montrons facilement que la multiplication est possible en temps logarithmique :

- Faisons tout en binaire
- On a nos n lignes à additionner
- On a un additionneur en temps constant (carry-save)
- Yaka faire un arbre d'additions en temps $0(log_2 n)$
- Bien sûr on récupère un nombre carry-save, il faut encore $0(log_2 n)$ pour le convertir en binaire
- Surface en $O(n \log_2 n)$

C'est théorique tout cela, il y a de grosses constantes pour la surface. On va faire plus minimaliste.

Débarassons nous des virgules

- La virgule se gère comme vous avez appris à gérer la virgule décimale

Débarassons nous des virgules

- La virgule se gère comme vous avez appris à gérer la virgule décimale
- Dans le doute, écrire des puissances de 2

Débarassons nous des virgules

- La virgule se gère comme vous avez appris à gérer la virgule décimale
- Dans le doute, écrire des puissances de 2
- Deux cas importants :
  - Entiers
    - en général on garde tous les $2n$ chiffres
    - parfois on tronque les poids forts avec détection facile de dépassement de capacité
    - souvent on travaille en complément à 2 (entiers signés)

Débarassons nous des virgules

- La virgule se gère comme vous avez appris à gérer la virgule décimale
- Dans le doute, écrire des puissances de 2
- Deux cas importants :
  - Entiers
    - en général on garde tous les $2n$ chiffres
    - parfois on tronque les poids forts avec détection facile de dépassement de capacité
    - souvent on travaille en complément à 2 (entiers signés)
  - rationnels dans [01[:
    - on tronque les poids faibles
    - ou on calcule un arrondi : on ajoute 1/2 puis on tronque
    - si c'est pour un multiplieur flottant on ne gère pas le signe
    - sinon le complément à 2 marche aussi

Extension de signe

- La somme finale est sur $2n$ bits
- Donc faut étendre les signes de chaque ligne sur $2n$ bits.

"étendre le signe" kesaco ?

Extension de signe

- La somme finale est sur $2n$ bits
- Donc faut étendre les signes de chaque ligne sur $2n$ bits.

"étendre le signe" kesaco ?

- Soit X en complément à 2 sur $n+1$ bits :
  $X = -2^n x_n + \sum_{i=0}^{n-1} 2^i x_i$.

Extension de signe

- La somme finale est sur $2n$ bits
- Donc faut étendre les signes de chaque ligne sur $2n$ bits.

"étendre le signe" kesaco ?

- Soit X en complément à 2 sur $n+1$ bits : $X = -2^n x_n + \sum_{i=0}^{n-1} 2^i x_i$.
- Pour obtenir un nombre qui a la même valeur, mais codé en complément à 2 sur $n+p$ bits, il suffit de répliquer p fois x_n.

Extension de signe

- La somme finale est sur $2n$ bits
- Donc faut étendre les signes de chaque ligne sur $2n$ bits.

"étendre le signe" kesaco ?

- Soit X en complément à 2 sur $n+1$ bits :
  $X = -2^n x_n + \sum_{i=0}^{n-1} 2^i x_i$.
- Pour obtenir un nombre qui a la même valeur, mais codé en complément à 2 sur $n+p$ bits, il suffit de répliquer p fois x_n.
- Démonstration :
  - Si $x_n = 0$ cela marche.
  - Si $x_n = 1$ on utilise l'identité $2^p - \sum_{j=0}^{p-1} 2^j = 1$
    déjà vue sous la forme $7 = 8 - 1$, en binaire $111 = (1000 - 1)$.

Extension de signe

- La somme finale est sur $2n$ bits
- Donc faut étendre les signes de chaque ligne sur $2n$ bits.

"étendre le signe" kesaco ?

- Soit X en complément à 2 sur $n+1$ bits :
  $X = -2^n x_n + \sum_{i=0}^{n-1} 2^i x_i$.
- Pour obtenir un nombre qui a la même valeur, mais codé en complément à 2 sur $n+p$ bits, il suffit de répliquer p fois x_n.
- Démonstration :
  - Si $x_n = 0$ cela marche.
  - Si $x_n = 1$ on utilise l'identité $2^p - \sum_{j=0}^{p-1} 2^j = 1$
    déjà vue sous la forme $7 = 8 - 1$, en binaire $111 = (1000 - 1)$.

Inconvénient : à notre losange de bits pondérés à additionner, on a ajouté un triangle de bits supplémentaires.

Bricolage de Baugh et Wooley

- But :
  - faire une multiplication de nombres en complément à 2,
  - obtenir le résultat en complément à 2,
  - et n'avoir qu'un losange de bits à additionner.

Bricolage de Baugh et Wooley

- But :
  - faire une multiplication de nombres en complément à 2,
  - obtenir le résultat en complément à 2,
  - et n'avoir qu'un losange de bits à additionner.
- Idées :
  - écrire $X = -2^n x_n + \sum_{i=0}^{n-1} 2^i x_i$
  - écrire $Y = -2^n y_n + \sum_{i=0}^{n-1} 2^i y_i$
  - dessiner le losange avec des signes $-$ dedans

Bricolage de Baugh et Wooley

- But :
  - faire une multiplication de nombres en complément à 2,
  - obtenir le résultat en complément à 2,
  - et n'avoir qu'un losange de bits à additionner.
- Idées :
  - écrire $X = -2^n x_n + \sum_{i=0}^{n-1} 2^i x_i$
  - écrire $Y = -2^n y_n + \sum_{i=0}^{n-1} 2^i y_i$
  - dessiner le losange avec des signes $-$ dedans
  - utiliser intelligemment l'identité $-t = -1 + \overline{t}$

Bricolage de Baugh et Wooley

- But :
  - faire une multiplication de nombres en complément à 2,
  - obtenir le résultat en complément à 2,
  - et n'avoir qu'un losange de bits à additionner.
- Idées :
  - écrire $X = -2^n x_n + \sum_{i=0}^{n-1} 2^i x_i$
  - écrire $Y = -2^n y_n + \sum_{i=0}^{n-1} 2^i y_i$
  - dessiner le losange avec des signes $-$ dedans
  - utiliser intelligemment l'identité $-t = -1 + \overline{t}$
  - "compresser" les paquets de -1 obtenus en utilisant $-111 = -1000 + 1$

Bricolage de Baugh et Wooley, détail

- sur la diagonale de gauche, remplacer les $-y_i x_n$ par $-x_n + \overline{y_i} x_n$

Bricolage de Baugh et Wooley, détail

- sur la diagonale de gauche, remplacer les $-y_i x_n$ par $-x_n + \overline{y_i} x_n$
- sur la dernière ligne, remplacer les $-y_n x_i$ par $-yn + \overline{x_i} y_n$

Bricolage de Baugh et Wooley, détail

- sur la diagonale de gauche, remplacer les $-y_i x_n$ par $-x_n + \overline{y_i} x_n$
- sur la dernière ligne, remplacer les $-y_n x_i$ par $-yn + \overline{x_i} y_n$
- dans le coin on a $x_n y_n$, qu'on ne touche pas

Bricolage de Baugh et Wooley, détail

- sur la diagonale de gauche, remplacer les $-y_i x_n$ par $-x_n + \overline{y_i} x_n$
- sur la dernière ligne, remplacer les $-y_n x_i$ par $-yn + \overline{x_i} y_n$
- dans le coin on a $x_n y_n$, qu'on ne touche pas
- On se retrouve avec un losange de termes positifs, et une ligne de $-x_n$, et une ligne de $-y_n$

Bricolage de Baugh et Wooley, détail

- sur la diagonale de gauche, remplacer les $-y_i x_n$ par $-x_n + \overline{y_i} x_n$
- sur la dernière ligne, remplacer les $-y_n x_i$ par $-yn + \overline{x_i} y_n$
- dans le coin on a $x_n y_n$, qu'on ne touche pas
- On se retrouve avec un losange de termes positifs, et une ligne de $-x_n$, et une ligne de $-y_n$
- La valeur de la ligne de $-x_n$ est $-x_n \sum_{i=n}^{2n-1} 2^i \quad = \quad -x_n(2^{2n} - 2^n)$ (compression $-7 = -8 + 1$) et pareil pour la ligne de y_n.

Bricolage de Baugh et Wooley, détail

- sur la diagonale de gauche, remplacer les $-y_i x_n$ par $-x_n + \overline{y_i} x_n$
- sur la dernière ligne, remplacer les $-y_n x_i$ par $-yn + \overline{x_i} y_n$
- dans le coin on a $x_n y_n$, qu'on ne touche pas
- On se retrouve avec un losange de termes positifs, et une ligne de $-x_n$, et une ligne de $-y_n$
- La valeur de la ligne de $-x_n$ est $-x_n \sum_{i=n}^{2n-1} 2^i \quad = \quad -x_n(2^{2n} - 2^n)$ (compression $-7 = -8 + 1$) et pareil pour la ligne de y_n.
- On remet une couche de $-t = -1 + \overline{t}$ pour les deux termes négatifs qui restent :
  $-x_n 2^{2n} = (-1 + \overline{x_n})2^{2n}$ et pareil pour $-y_n 2^{2n}$

Bricolage de Baugh et Wooley, détail

- sur la diagonale de gauche, remplacer les $-y_i x_n$ par $-x_n + \overline{y_i} x_n$
- sur la dernière ligne, remplacer les $-y_n x_i$ par $-yn + \overline{x_i} y_n$
- dans le coin on a $x_n y_n$, qu'on ne touche pas
- On se retrouve avec un losange de termes positifs, et une ligne de $-x_n$, et une ligne de $-y_n$
- La valeur de la ligne de $-x_n$ est $-x_n \sum_{i=n}^{2n-1} 2^i = -x_n(2^{2n} - 2^n)$ (compression $-7 = -8 + 1$) et pareil pour la ligne de y_n.
- On remet une couche de $-t = -1 + \overline{t}$ pour les deux termes négatifs qui restent :
  $-x_n 2^{2n} = (-1 + \overline{x_n})2^{2n}$ et pareil pour $-y_n 2^{2n}$
- Les deux -1.2^{2n} superposés sont additionnés, ce qui donne -1.2^{2n+1}

Bricolage de Baugh et Wooley, détail

- sur la diagonale de gauche, remplacer les $-y_i x_n$ par $-x_n + \overline{y_i} x_n$
- sur la dernière ligne, remplacer les $-y_n x_i$ par $-yn + \overline{x_i} y_n$
- dans le coin on a $x_n y_n$, qu'on ne touche pas
- On se retrouve avec un losange de termes positifs, et une ligne de $-x_n$, et une ligne de $-y_n$
- La valeur de la ligne de $-x_n$ est $-x_n \sum_{i=n}^{2n-1} 2^i = -x_n(2^{2n} - 2^n)$ (compression $-7 = -8 + 1$) et pareil pour la ligne de y_n.
- On remet une couche de $-t = -1 + \bar{t}$ pour les deux termes négatifs qui restent :
  $-x_n 2^{2n} = (-1 + \overline{x_n})2^{2n}$ et pareil pour $-y_n 2^{2n}$
- Les deux -1.2^{2n} superposés sont additionnés, ce qui donne -1.2^{2n+1}
- Oh, $2n + 1$ c'est le poids du bit de signe dans le produit : en complément à deux, ce -1 s'écrit donc 1.

Bricolage de Baugh et Wooley, détail

- sur la diagonale de gauche, remplacer les $-y_i x_n$ par $-x_n + \overline{y_i} x_n$
- sur la dernière ligne, remplacer les $-y_n x_i$ par $-yn + \overline{x_i} y_n$
- dans le coin on a $x_n y_n$, qu'on ne touche pas
- On se retrouve avec un losange de termes positifs, et une ligne de $-x_n$, et une ligne de $-y_n$
- La valeur de la ligne de $-x_n$ est $-x_n \sum_{i=n}^{2n-1} 2^i = -x_n(2^{2n} - 2^n)$ (compression $-7 = -8 + 1$) et pareil pour la ligne de y_n.
- On remet une couche de $-t = -1 + \overline{t}$ pour les deux termes négatifs qui restent :
  $-x_n 2^{2n} = (-1 + \overline{x_n})2^{2n}$ et pareil pour $-y_n 2^{2n}$
- Les deux -1.2^{2n} superposés sont additionnés, ce qui donne -1.2^{2n+1}
- Oh, $2n + 1$ c'est le poids du bit de signe dans le produit : en complément à deux, ce -1 s'écrit donc 1.

Moralité : on a de nouveau un joli losange, avec quelques termes complémentés de ci de là, et cinq bits en plus.

Même bricolage pour la multiaddition

Supposons que vous ayez à ajouter p entiers en complément à deux (un rectangle, pas un losange). Ici p est supposé constant.

Même bricolage pour la multiaddition

Supposons que vous ayez à ajouter p entiers en complément à deux (un rectangle, pas un losange). Ici p est supposé constant. Eh ben on peut faire presque pareil :

- On remplace moralement tous les bits de signe x_n^i (qui valent $-x_n^i$) par $-1 + \overline{x_n^i}$
- On additionne tous ces -1, qui ont à présent le même poids 2^n
- Cela donne $-2^n p$
- On écrit $-2^n p$ en binaire et en complément à deux sur le nombre de bits du résultat (il faut log_2 bits)
- (Forcément faut bricoler un peu)
- Finalement on n'aura ajouté qu'une ligne à nos p lignes.

Moralité sur le complément à 2 dans les multiplieurs

- C'est pas très dur mais c'est très technique
- On s'en sortira toujours par ce genre de bricolage :
  - on se ramène à un paquet de bits pondérés,
  - le bricolage c'est quand on produit des bits dont le poids est celui du bit de signe

Moralité sur le complément à 2 dans les multiplieurs

- C'est pas très dur mais c'est très technique
- On s'en sortira toujours par ce genre de bricolage :
  - on se ramène à un paquet de bits pondérés,
  - le bricolage c'est quand on produit des bits dont le poids est celui du bit de signe
- En ce qui me concerne cela me suffit

Le reste de ce cours

- grandes idées pour construire un multiplieur :
  - multiplieurs cellulaires : mettre un FA derrière chaque bit du losange
  - multiplieurs séquentiels : sommer les lignes itérativement
  - multiplieurs récursifs
  - multiplieurs rapides : un arbre ad-hoc pour additionner un paquet de bits pondérés.
- Toutes ces idées bénéficient d'une réduction de la taille du paquet de bits pondérés :
  - pour les multiplieurs on va utiliser des recodages de Y

Bon, c'est pas du tout mon plan mais j'espère que vous avez une idée générale de ce qu'on va voir.

Multiplieurs cellulaires

Au moins c'est vite dit

- Seules idée à avoir

Au moins c'est vite dit

- Seules idée à avoir
  - ne pas faire une propagation de retenue à chaque ligne (passer la somme partielle en carry-save)

Au moins c'est vite dit

- Seules idée à avoir
  - ne pas faire une propagation de retenue à chaque ligne (passer la somme partielle en carry-save)
  - les n bits de poids faible sortent en n temps de traversée de FA, et on n'en parle plus
- On ne coupe pas à une propagation de retenue à la fin

Au moins c'est vite dit

- Seules idée à avoir
  - ne pas faire une propagation de retenue à chaque ligne (passer la somme partielle en carry-save)
  - les n bits de poids faible sortent en n temps de traversée de FA, et on n'en parle plus
- On ne coupe pas à une propagation de retenue à la fin
- temps en $2n$ au lieu de $3n$ si on faisait une retenue par ligne (dessin)

Au moins c'est vite dit

- Seules idée à avoir
  - ne pas faire une propagation de retenue à chaque ligne (passer la somme partielle en carry-save)
  - les n bits de poids faible sortent en n temps de traversée de FA, et on n'en parle plus
- On ne coupe pas à une propagation de retenue à la fin
- temps en $2n$ au lieu de $3n$ si on faisait une retenue par ligne (dessin)
- Placement régulier de cellules FA.

Au moins c'est vite dit

- Seules idée à avoir
  - ne pas faire une propagation de retenue à chaque ligne (passer la somme partielle en carry-save)
  - les n bits de poids faible sortent en n temps de traversée de FA, et on n'en parle plus
- On ne coupe pas à une propagation de retenue à la fin
- temps en $2n$ au lieu de $3n$ si on faisait une retenue par ligne (dessin)
- Placement régulier de cellules FA.

Petits noms : multiplieur de Braun, multiplieur de Baugh Wooley (signé)

Au moins c'est vite dit

- Seules idée à avoir
  - ne pas faire une propagation de retenue à chaque ligne (passer la somme partielle en carry-save)
  - les n bits de poids faible sortent en n temps de traversée de FA, et on n'en parle plus
- On ne coupe pas à une propagation de retenue à la fin
- temps en $2n$ au lieu de $3n$ si on faisait une retenue par ligne (dessin)
- Placement régulier de cellules FA.

Petits noms : multiplieur de Braun, multiplieur de Baugh Wooley (signé)
On va être capables de les réduire en recodant Y pour diminuer le nombre de ses chiffres.

Multiplieurs récursifs

Une approche algorithmique

- On veut calculer $X \times Y$, avec X et Y chacun sur n bits, avec n une puissance de 2.
- Coupons X et Y en deux : $X = 2^{n/2}X_1 + X_0$ et $Y = 2^{n/2}Y_1 + Y_0$

Une approche algorithmique

- On veut calculer $X \times Y$, avec X et Y chacun sur n bits, avec n une puissance de 2.
- Coupons X et Y en deux : $X = 2^{n/2}X_1 + X_0$ et $Y = 2^{n/2}Y_1 + Y_0$
- alors $XY = 2^n X_1 Y_1 + 2^{n/2}(X_1 Y_0 + X_0 Y_1) + X_0 Y_0$

Une approche algorithmique

- On veut calculer $X \times Y$, avec X et Y chacun sur n bits, avec n une puissance de 2.
- Coupons X et Y en deux : $X = 2^{n/2}X_1 + X_0$ et $Y = 2^{n/2}Y_1 + Y_0$
- alors $XY = 2^n X_1 Y_1 + 2^{n/2}(X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- On n'a plus que des produits de nombres de taille $n/2$ que l'on peut faire en parallèle.

Une approche algorithmique

- On veut calculer $X \times Y$, avec X et Y chacun sur n bits, avec n une puissance de 2.
- Coupons X et Y en deux : $X = 2^{n/2}X_1 + X_0$ et $Y = 2^{n/2}Y_1 + Y_0$
- alors $XY = 2^n X_1 Y_1 + 2^{n/2}(X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- On n'a plus que des produits de nombres de taille $n/2$ que l'on peut faire en parallèle.
- Additions de taille n et $n/2$

Une approche algorithmique

- On veut calculer $X \times Y$, avec X et Y chacun sur n bits, avec n une puissance de 2.
- Coupons X et Y en deux : $X = 2^{n/2}X_1 + X_0$ et $Y = 2^{n/2}Y_1 + Y_0$
- alors $XY = 2^n X_1 Y_1 + 2^{n/2}(X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- On n'a plus que des produits de nombres de taille $n/2$ que l'on peut faire en parallèle.
- Additions de taille n et $n/2$
- Si addition en temps constant (carry-save) il vient $T_\times(n) = T_\times(n/2) + 2T_+$

Une approche algorithmique

- On veut calculer $X \times Y$, avec X et Y chacun sur n bits, avec n une puissance de 2.
- Coupons X et Y en deux : $X = 2^{n/2}X_1 + X_0$ et $Y = 2^{n/2}Y_1 + Y_0$
- alors $XY = 2^n X_1 Y_1 + 2^{n/2}(X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- On n'a plus que des produits de nombres de taille $n/2$ que l'on peut faire en parallèle.
- Additions de taille n et $n/2$
- Si addition en temps constant (carry-save) il vient $T_\times(n) = T_\times(n/2) + 2T_+$
- En récursant : $T_\times(n) = 0(log_2 n)$

Une approche algorithmique

- On veut calculer $X \times Y$, avec X et Y chacun sur n bits, avec n une puissance de 2.
- Coupons X et Y en deux : $X = 2^{n/2}X_1 + X_0$ et $Y = 2^{n/2}Y_1 + Y_0$
- alors $XY = 2^n X_1 Y_1 + 2^{n/2}(X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- On n'a plus que des produits de nombres de taille $n/2$ que l'on peut faire en parallèle.
- Additions de taille n et $n/2$
- Si addition en temps constant (carry-save) il vient $T_\times(n) = T_\times(n/2) + 2T_+$
- En récursant : $T_\times(n) = 0(log_2 n)$
- Bien sûr on récupère un nombre carry-save, il faut encore $0(log_2 n)$ pour le convertir en binaire

Grandeur et misère des multiplieurs récursifs

- Surface importante au final.
- Seul intérêt : placement algorithmique fractal.

Grandeur et misère des multiplieurs récursifs

- Surface importante au final.
- Seul intérêt : placement algorithmique fractal.

(purement académique à mon avis)

Grandeur et misère des multiplieurs récursifs

- Surface importante au final.
- Seul intérêt : placement algorithmique fractal.
  (purement académique à mon avis)
- Compromis différent : décomposition 3M :
  - calculer $U = (X_1 + X_0)(Y_1 + Y_0)$, $P_1 = X_1 Y_1$, $P_0 = X_0 Y_0$
  - alors $XY = P_1 + 2^{n/2}(U - P_1 - P_0) + P_0$
  - Plus d'additions, mais seulement 3 multiplications de taille $n/2$
  - Utilisé pour faire de la multiprécision en soft (tout ceci marche avec n'importe quelle base)

Grandeur et misère des multiplieurs récursifs

- Surface importante au final.
- Seul intérêt : placement algorithmique fractal.
  (purement académique à mon avis)
- Compromis différent : décomposition 3M :
  - calculer $U = (X_1 + X_0)(Y_1 + Y_0)$, $P_1 = X_1 Y_1$, $P_0 = X_0 Y_0$
  - alors $XY = P_1 + 2^{n/2}(U - P_1 - P_0) + P_0$
  - Plus d'additions, mais seulement 3 multiplications de taille $n/2$
  - Utilisé pour faire de la multiprécision en soft (tout ceci marche avec n'importe quelle base)
- Une autre : décomposition 2M :
  - On peut ne décomposer que Y
  - ce qui donne $XY = XY_0 + XY_1$
  - les multiplications de X par un bit de Y sont triviales

Grandeur et misère des multiplieurs récursifs

- Surface importante au final.
- Seul intérêt : placement algorithmique fractal.
  (purement académique à mon avis)
- Compromis différent : décomposition 3M :
  - calculer $U = (X_1 + X_0)(Y_1 + Y_0)$, $P_1 = X_1 Y_1$, $P_0 = X_0 Y_0$
  - alors $XY = P_1 + 2^{n/2}(U - P_1 - P_0) + P_0$
  - Plus d'additions, mais seulement 3 multiplications de taille $n/2$
  - Utilisé pour faire de la multiprécision en soft (tout ceci marche avec n'importe quelle base)
- Une autre : décomposition 2M :
  - On peut ne décomposer que Y
  - ce qui donne $XY = XY_0 + XY_1$
  - les multiplications de X par un bit de Y sont triviales
  - Oh, c'est mon arbre d'addition des lignes !

Multiplieurs séquentiels

Version générale en base 2, entiers positifs

$$Y = \sum_{i=0}^{n-1} 2^i y_i$$

$$XY = \sum_{i=0}^{n-1} 2^i X y_i$$

Version générale en base 2, entiers positifs

$$Y = \sum_{i=0}^{n-1} 2^i y_i$$

$$XY = \sum_{i=0}^{n-1} 2^i X y_i$$

- Les multiplications par 2^i sont des décalages
- Les multiplications par x_i sont des ET logiques
- La somme partielle est conservée dans un registre
- Au lieu de décaler $y_i X$ à gauche, on peut décaler la somme partielle à droite (les poids faibles ne changeront plus)
  - Du coup, un additionneur n bits suffit

Version générale en base 2, entiers positifs

$$Y = \sum_{i=0}^{n-1} 2^i y_i$$

$$XY = \sum_{i=0}^{n-1} 2^i X y_i$$

- Les multiplications par 2^i sont des décalages
- Les multiplications par x_i sont des ET logiques
- La somme partielle est conservée dans un registre
- Au lieu de décaler $y_i X$ à gauche, on peut décaler la somme partielle à droite (les poids faibles ne changeront plus)
  - Du coup, un additionneur n bits suffit

Cette architecture a besoin de n cycles pour calculer la multiplication.
C'est aussi (surtout) comme cela qu'on fait des multiplications dans les microcontrôleurs sans multiplieurs.

Questions de signe

En complément à 2 on s'en tire en faisant des bricolages sur le bit de poids fort.
La difficulté est qu'on décale à droite au lieu de décaler à gauche, donc maintenir le signe de la somme partielle juste.
Ce qui aide c'est que le signe est le signe de la dernière ligne ajoutée, donc on peut faire l'extension de signe au vol.
(voir le Muller rose pour les détails)

Version carry-save

- Toujours n itérations, mais n fois plus rapides
- Une conversion de carry-save en binaire à la fin : addition (rapide)
- On peut convertir au vol les bits de poids faible qu'on sort
- Si on ne voulait que n bits de poids faibles c'est donc plus rapide

Version carry-save

- Toujours n itérations, mais n fois plus rapides
- Une conversion de carry-save en binaire à la fin : addition (rapide)
- On peut convertir au vol les bits de poids faible qu'on sort
- Si on ne voulait que n bits de poids faibles c'est donc plus rapide

(jamais vu dans la nature, celui-là)

Avec recodage

- But : moins d'itérations ($n/2$)
- Utilité plus générale que les multiplieurs séquentiels

Recodages

Changement de base ?

- Comme d'habitude, c'est pour augmenter la granularité
- On considère les entrées en base 2^p (bits regroupés par paquets de p)
- Le paquet de bits pondérés est produit par plein $((n/p)^2)$ de petits multiplieurs de chiffres.

Changement de base ?

- Comme d'habitude, c'est pour augmenter la granularité
- On considère les entrées en base 2^p (bits regroupés par paquets de p)
- Le paquet de bits pondérés est produit par plein $((n/p)^2)$ de petits multiplieurs de chiffres.
- Taille du paquet de bits obtenu : $2p.(n/p)^2$ au lieu de n^2

Changement de base ?

- Comme d'habitude, c'est pour augmenter la granularité
- On considère les entrées en base 2^p (bits regroupés par paquets de p)
- Le paquet de bits pondérés est produit par plein $((n/p)^2)$ de petits multiplieurs de chiffres.
- Taille du paquet de bits obtenu : $2p.(n/p)^2$ au lieu de n^2
- Pas très intéressant pour $p = 2$, coûteux pour $p > 2$

Recodage ?

Ce qu'on appelle "recodage" pour les multiplieurs, c'est en général un changement de base pour Y uniquement qui permet de calculer les $y_i X$ (presque) aussi facilement qu'en binaire.

- Diminution du nombre de lignes,
  - pour diminuer les itérations des multiplieurs séquentiels
  - pour diminuer le tas de bits pondérés à ajouter pour les multiplieurs à arbres

Calcul d'une ligne en parallèle

Ici "une ligne" signifie un produit $y_i X$.

- En base 2 une ligne $y_i X$ se calcule en // sur les chiffres de X.
  - Dans l'algo à la main en base 10 ce n'était pas vrai (propagation de retenue)

Calcul d'une ligne en parallèle

Ici "une ligne" signifie un produit $y_i X$.

- En base 2 une ligne $y_i X$ se calcule en // sur les chiffres de X.
  - Dans l'algo à la main en base 10 ce n'était pas vrai (propagation de retenue)
- En base plus grande :
  - tant que le multiplicateur Y est en base 2 on peut calculer les $y_i X$ en parallèle
    - mais cela ne nous fera pas moins de bits pondérés à ajouter

Calcul d'une ligne en parallèle

Ici "une ligne" signifie un produit $y_i X$.

- En base 2 une ligne $y_i X$ se calcule en // sur les chiffres de X.
  - Dans l'algo à la main en base 10 ce n'était pas vrai (propagation de retenue)
- En base plus grande :
  - tant que le multiplicateur Y est en base 2 on peut calculer les $y_i X$ en parallèle
    - mais cela ne nous fera pas moins de bits pondérés à ajouter
  - et même avec Y en base 4 et X en base 2, pour certains ensembles de chiffres (lesquels et pourquoi ?)

Calcul d'une ligne en parallèle

Ici "une ligne" signifie un produit $y_i X$.

- En base 2 une ligne $y_i X$ se calcule en // sur les chiffres de X.
  - Dans l'algo à la main en base 10 ce n'était pas vrai (propagation de retenue)
- En base plus grande :
  - tant que le multiplicateur Y est en base 2 on peut calculer les $y_i X$ en parallèle
    - mais cela ne nous fera pas moins de bits pondérés à ajouter
  - et même avec Y en base 4 et X en base 2, pour certains ensembles de chiffres (lesquels et pourquoi ?)
- Sinon on n'a qu'a précalculer (en parallèle) tous les cX pour tous les chiffres c.
  - Une fois ceci fait le calcul d'un $y_i X$ est en temps constant.

Base 4, bête

- Chiffres $\{0, 1, 2, 3\}$
- On calcule $3X = X + 2X$ une fois pour toutes
- On a deux fois moins de lignes

Coût ?

Recodage de Booth modifié

- Chiffres $\{-1, 0, 1, 2\}$
- On additionne des lignes ou des lignes décalées
- Recodage par récurrence de droite à gauche :
  - soit $v_i = 2y_{2i+1} + y_{2i}$
    (v_i est un chiffre de Y recodé en base 4 bête)
  - récurrence :
    $$\begin{cases} c_0 = 0 \\ w_i = v_i + c_i - 4c_{i+1} \end{cases}$$
  - et on choisit c_{i+1} en fonction de $v_i + c_i$ pour ne jamais avoir $w_i = 3$

Recodage de Booth modifié

- Chiffres $\{-1, 0, 1, 2\}$
- On additionne des lignes ou des lignes décalées
- Recodage par récurrence de droite à gauche :
  - soit $v_i = 2y_{2i+1} + y_{2i}$
    (v_i est un chiffre de Y recodé en base 4 bête)
  - récurrence :
    $$\begin{cases} c_0 = 0 \\ w_i = v_i + c_i - 4c_{i+1} \end{cases}$$
  - et on choisit c_{i+1} en fonction de $v_i + c_i$ pour ne jamais avoir $w_i = 3$

C'est un recodage séquentiel qui va bien

- pour la multiplication séquentielle
- pour les réseaux cellulaires (dans lesquels la propagation d'information est également séquentielle).

Un recodage parallèle en base 4

On n'y coupera pas, un recodage parallèle va nous donner un ensemble de chiffres redondant.

- En base 4, les chiffres $\{-2, -1, 0, 1, 2\}$ feraient bien l'affaire : On additionnera/soustraira des lignes ou des lignes décalées

Un recodage parallèle en base 4

On n'y coupera pas, un recodage parallèle va nous donner un ensemble de chiffres redondant.

- En base 4, les chiffres $\{-2, -1, 0, 1, 2\}$ feraient bien l'affaire : On additionnera/soustraira des lignes ou des lignes décalées
- Calcul du recodage : c'est la même technique que pour l'addition d'Avizienis
  - Soit $v_i = 2y_{2i+1} + y_{2i}$ (cela commence comme Booth)
  - On va reécrire v_i sous la forme $v_i = w_i + 4c_{i+1}$ donc $w_i = v_i - 4c_{i+1}$ (cela continue presque comme Booth)
  - Puis on va calculer $z_i = w_i + c_i$

Un recodage parallèle en base 4

On n'y coupera pas, un recodage parallèle va nous donner un ensemble de chiffres redondant.

- En base 4, les chiffres $\{-2, -1, 0, 1, 2\}$ feraient bien l'affaire : On additionnera/soustraira des lignes ou des lignes décalées
- Calcul du recodage : c'est la même technique que pour l'addition d'Avizienis
  - Soit $v_i = 2y_{2i+1} + y_{2i}$ (cela commence comme Booth)
  - On va reécrire v_i sous la forme $v_i = w_i + 4c_{i+1}$ donc $w_i = v_i - 4c_{i+1}$ (cela continue presque comme Booth)
  - Puis on va calculer $z_i = w_i + c_i$
  - Pour éviter la propagation de retenue il faut que c_{i+1} ne dépende pas de c_i :
    - on choisit c_{i+1} en fonction de v_i uniquement
    - on compense ensuite en fonction de c_i pour sortir le chiffre z_i

Un recodage parallèle en base 4

On n'y coupera pas, un recodage parallèle va nous donner un ensemble de chiffres redondant.

- En base 4, les chiffres $\{-2, -1, 0, 1, 2\}$ feraient bien l'affaire : On additionnera/soustraira des lignes ou des lignes décalées
- Calcul du recodage : c'est la même technique que pour l'addition d'Avizienis
  - Soit $v_i = 2y_{2i+1} + y_{2i}$ (cela commence comme Booth)
  - On va reécrire v_i sous la forme $v_i = w_i + 4c_{i+1}$ donc $w_i = v_i - 4c_{i+1}$ (cela continue presque comme Booth)
  - Puis on va calculer $z_i = w_i + c_i$
  - Pour éviter la propagation de retenue il faut que c_{i+1} ne dépende pas de c_i :
    - on choisit c_{i+1} en fonction de v_i uniquement
    - on compense ensuite en fonction de c_i pour sortir le chiffre z_i
  - L'algo à appliquer est donc :

$$\begin{cases} (c_{i+1}, w_j) = \begin{cases} (0, v_j) & si \quad v_j \leq 1 \\ (1, v_j - 4) & si \quad v_j > 1 \end{cases} \\ z_i = w_i + c_i \end{cases}$$

Exemples

$$\begin{cases} (c_{i+1}, w_j) = \begin{cases} (0, v_j) & si \quad v_j \leq 1 \\ (1, v_j - 4) & si \quad v_j > 1 \end{cases} \\ z_i = w_i + c_i \end{cases}$$

Exemples (deux volontaires en parallèle pour chaque) :

- 0 1 0 1 1 1 1 0

Exemples

$$\begin{cases} (c_{i+1}, w_j) = \begin{cases} (0, v_j) & si \quad v_j \leq 1 \\ (1, v_j - 4) & si \quad v_j > 1 \end{cases} \\ z_i = w_i + c_i \end{cases}$$

Exemples (deux volontaires en parallèle pour chaque) :

- 0 1 0 1 1 1 1 0
- 1 2 0 -2
- 1 0 0 0 1 1 0 1 (nombre en complément à 2)

Exemples

$$\begin{cases} (c_{i+1}, w_j) = \begin{cases} (0, v_j) & si \quad v_j \leq 1 \\ (1, v_j - 4) & si \quad v_j > 1 \end{cases} \\ z_i = w_i + c_i \end{cases}$$

Exemples (deux volontaires en parallèle pour chaque) :

- 0 1 0 1 1 1 1 0
- 1 2 0 -2
- 1 0 0 0 1 1 0 1 (nombre en complément à 2)
- -2 1 -1 1

Remarques

$$\begin{cases} (c_{i+1}, w_j) = \begin{cases} (0, v_j) & si \quad v_j \leq 1 \\ (1, v_j - 4) & si \quad v_j > 1 \end{cases} \\ z_i = w_i + c_i \end{cases}$$

- La condition, c'est la valeur du bit de poids fort de v_j, soit y_{2i+1}.

Remarques

$$\begin{cases} (c_{i+1}, w_j) = \begin{cases} (0, v_j) & si \quad v_j \leq 1 \\ (1, v_j - 4) & si \quad v_j > 1 \end{cases} \\ z_i = w_i + c_i \end{cases}$$

- La condition, c'est la valeur du bit de poids fort de v_j, soit y_{2i+1}.
- Pour faciliter le calcul des $z_i X$ le plus simple est de coder z_i sur 3 bits : (signe, un, deux)

Remarques

$$\begin{cases} (c_{i+1}, w_j) = \begin{cases} (0, v_j) & si \quad v_j \leq 1 \\ (1, v_j - 4) & si \quad v_j > 1 \end{cases} \\ z_i = w_i + c_i \end{cases}$$

- La condition, c'est la valeur du bit de poids fort de v_j, soit y_{2i+1}.
- Pour faciliter le calcul des $z_i X$ le plus simple est de coder z_i sur 3 bits : (signe, un, deux)
- Tout cela marche très bien pour le complément à 2 (bricolages déjà vus).
- et d'ailleurs, même pour un multiplieur de nombres positifs, les lignes recodées le sont en complément à 2 puisqu'on doit les ajouter ou les soustraire.

Base 8 et plus

On continue : recodages en base 8

- Le but c'est de diviser par 3 le nombre de lignes
- On ne coupe pas à précalculer les cX, ce qui peut se faire en parallèle au recodage
- Non redondant bête : précalculer $X, 2X, 3X = X + 2X$, $4X, 5X = X + 4X, 6X = 2X + 4X, 7X = 8X - X$
- Non redondant à la Booth : $\{-3, ...4\}$ pour n'avoir que $3X$ à précalculer (mais recodage séquentiel)
- Redondant : on utilisera $\{-4, ...4\}$ (recodage parallèle en exercice à la maison)
- Remarque : on ajoute quelques colonnes à notre tableau de bits pondérés.

Base 8 et plus

On continue : recodages en base 8

- Le but c'est de diviser par 3 le nombre de lignes
- On ne coupe pas à précalculer les cX, ce qui peut se faire en parallèle au recodage
- Non redondant bête : précalculer $X, 2X, 3X = X + 2X$, $4X, 5X = X + 4X, 6X = 2X + 4X, 7X = 8X - X$
- Non redondant à la Booth : $\{-3, ...4\}$ pour n'avoir que $3X$ à précalculer (mais recodage séquentiel)
- Redondant : on utilisera $\{-4, ...4\}$ (recodage parallèle en exercice à la maison)
- Remarque : on ajoute quelques colonnes à notre tableau de bits pondérés.

Le recodage en base 16 peut être utile pour accélérer le multiplieur séquentiel, mais alors moralement c'est deux étapes de recodage en base 4.

Arbres d'addition

Idée générale

On a obtenu d'une manière ou d'une autre un paquet de bits pondérés

- On a notre bon vieux *full adder* qui prend trois bits de même poids et renvoie leur somme sur deux poids consécutifs (compression trois-en-deux)

Idée générale

On a obtenu d'une manière ou d'une autre un paquet de bits pondérés

- On a notre bon vieux *full adder* qui prend trois bits de même poids et renvoie leur somme sur deux poids consécutifs (compression trois-en-deux)
- En assemblant plein de FA on peut réduire notre paquet de bits jusqu'à avoir un seul bit par poids.

Idée générale

On a obtenu d'une manière ou d'une autre un paquet de bits pondérés

- On a notre bon vieux *full adder* qui prend trois bits de même poids et renvoie leur somme sur deux poids consécutifs (compression trois-en-deux)
- En assemblant plein de FA on peut réduire notre paquet de bits jusqu'à avoir un seul bit par poids.
  Les additionneurs cellulaires et récursifs sont des cas particuliers

Idée générale

On a obtenu d'une manière ou d'une autre un paquet de bits pondérés

- On a notre bon vieux *full adder* qui prend trois bits de même poids et renvoie leur somme sur deux poids consécutifs (compression trois-en-deux)
- En assemblant plein de FA on peut réduire notre paquet de bits jusqu'à avoir un seul bit par poids.
  Les additionneurs cellulaires et récursifs sont des cas particuliers
- On saura faire en parallèle sur les bits une réduction jusqu'à deux bits par poids (somme en carry-save), après faudra propager une retenue.

Grosses questions :

- Comment agencer les FA ?

Idée générale

On a obtenu d'une manière ou d'une autre un paquet de bits pondérés

- On a notre bon vieux *full adder* qui prend trois bits de même poids et renvoie leur somme sur deux poids consécutifs (compression trois-en-deux)
- En assemblant plein de FA on peut réduire notre paquet de bits jusqu'à avoir un seul bit par poids.
  Les additionneurs cellulaires et récursifs sont des cas particuliers
- On saura faire en parallèle sur les bits une réduction jusqu'à deux bits par poids (somme en carry-save), après faudra propager une retenue.

Grosses questions :

- Comment agencer les FA ?
- Peut-on utiliser d'autres briques de base que le FA ? (compresseurs truc en machin)

Ca y est

On a un énorme espace de solutions dans lequel naviguer.

Ca y est

On a un énorme espace de solutions dans lequel naviguer.

Deux extrêmes :

Ca y est

On a un énorme espace de solutions dans lequel naviguer.

Deux extrêmes :

- compression par ligne, en utilisant des additionneurs
  - additionneurs carry-save, ou 3 :2
  - additionneur parallèle 4 :2 possible aussi
  - coût : sur combien de bits chaque addition ? de n à $2n$.

Ca y est

On a un énorme espace de solutions dans lequel naviguer.

Deux extrêmes :

- compression par ligne, en utilisant des additionneurs
  - additionneurs carry-save, ou 3 :2
  - additionneur parallèle 4 :2 possible aussi
  - coût : sur combien de bits chaque addition ? de n à $2n$.
- compression par colonne, en utilisant des “compresseurs $2^p - 1$ en p”.
  - ce n’est pas si simple parce qu’une colonne bave vers la gauche

Une brique de base récursive : l'arbre de Wallace

Un arbre de Wallace c'est une boîte qui prend p bits de meme poids et rend leur somme sur $\lceil \log_2 p \rceil$ bits.

Une brique de base récursive : l'arbre de Wallace

Un arbre de Wallace c'est une boîte qui prend p bits de meme poids et rend leur somme sur $\lceil \log_2 p \rceil$ bits.
Le minimal c'est notre FA, que l'on appelera alors W_3.

Une brique de base récursive : l'arbre de Wallace

Un arbre de Wallace c'est une boîte qui prend p bits de meme poids et rend leur somme sur $\lceil \log_2 p \rceil$ bits.
Le minimal c'est notre FA, que l'on appelera alors W_3.

- Exo1 : construire W_5 en assemblant des W_3

Une brique de base récursive : l'arbre de Wallace

Un arbre de Wallace c'est une boîte qui prend p bits de meme poids et rend leur somme sur $\lceil \log_2 p \rceil$ bits.
Le minimal c'est notre FA, que l'on appelera alors W_3.

- Exo1 : construire W_5 en assemblant des W_3
- Exo2 : construire $W_{2^{p+1}-1}$ en assemblant deux W_{2^p-1}, plus des miettes

Une brique de base récursive : l'arbre de Wallace

Un arbre de Wallace c'est une boîte qui prend p bits de meme poids et rend leur somme sur $\lceil \log_2 p \rceil$ bits.
Le minimal c'est notre FA, que l'on appelera alors W_3.

- Exo1 : construire W_5 en assemblant des W_3
- Exo2 : construire $W_{2^{p+1}-1}$ en assemblant deux W_{2^p-1}, plus des miettes
  Argh, une propagation de retenue ! mais sur $\lceil \log_2 p \rceil$ cellules seulement.

Exemple : décomposition en petits multiplieurs

Muller Rose p 121

Exemple : heuristique de Dadda

On veut réaliser un arbre de FA pour réduire un paquet de bits pondérés en juste deux lignes

- plusieurs niveaux de FA
- pas de propagation horizontale (au sein d'un niveau)
- minimiser avant tout le nombre de niveaux, ensuite le nombre de FA
- (parfois on n'utilisera que deux entrées du FA, qui est alors un HA)

Exemple : heuristique de Dadda

On veut réaliser un arbre de FA pour réduire un paquet de bits pondérés en juste deux lignes

- plusieurs niveaux de FA
- pas de propagation horizontale (au sein d'un niveau)
- minimiser avant tout le nombre de niveaux, ensuite le nombre de FA
- (parfois on n'utilisera que deux entrées du FA, qui est alors un HA)

Observation : si un niveau produit des colonnes de hauteur h, alors il a en entrée des colonnes de hauteur au plus $\lfloor 3h/2 \rfloor$
(la hauteur de la colonne j c'est le nombre de bits de poids 2^i du paquet)

Exemple : heuristique de Dadda (2)

- Soit la suite définie par $u_0 = 2$ et $u_j + 1 = \lfloor 3u_j/2 \rfloor$

$$(2, 3, 4, 6, 9, 13...)$$

Exemple : heuristique de Dadda (2)

- Soit la suite définie par $u_0 = 2$ et $u_j + 1 = \lfloor 3u_j/2 \rfloor$

$$(2, 3, 4, 6, 9, 13...)$$

- Soit h la hauteur max des colonnes au départ, et u_j le plus grand terme de la suite inférieur à h.

Exemple : heuristique de Dadda (2)

- Soit la suite définie par $u_0 = 2$ et $u_j + 1 = \lfloor 3u_j/2 \rfloor$

$$(2, 3, 4, 6, 9, 13...)$$

- Soit h la hauteur max des colonnes au départ, et u_j le plus grand terme de la suite inférieur à h.

Stratégie de Dadda :

- Utiliser le plus petit nombre possible de FA et HA pour se ramener à une hauteur max de u_j.
- Ensuite pour i allant de j à 2,
  - Utiliser le plus petit nombre possible de FA et HA pour se ramener à une hauteur max de u_{j-1}.

Exemple : heuristique de Dadda (2)

- Soit la suite définie par $u_0 = 2$ et $u_j + 1 = \lfloor 3u_j/2 \rfloor$

$$(2, 3, 4, 6, 9, 13...)$$

- Soit h la hauteur max des colonnes au départ, et u_j le plus grand terme de la suite inférieur à h.

Stratégie de Dadda :

- Utiliser le plus petit nombre possible de FA et HA pour se ramener à une hauteur max de u_j.
- Ensuite pour i allant de j à 2,
  - Utiliser le plus petit nombre possible de FA et HA pour se ramener à une hauteur max de u_{j-1}.

Stratégie gloutonne qui ne dit pas exactement quoi faire à chaque étape

Conclusions

- L'approche de Dadda a été généralisée et précisée

Conclusions

- L'approche de Dadda a été généralisée et précisée
- Elle suppose que $T_{c_{out}} = T_s$ dans le FA. Si (comme) ce n'est pas vrai, faut tout reprendre.
  - articles d'Oklobdzija

Conclusions

- L'approche de Dadda a été généralisée et précisée
- Elle suppose que $T_{c_{out}} = T_s$ dans le FA. Si (comme) ce n'est pas vrai, faut tout reprendre.
  - articles d'Oklobdzija
- Tout cela nous donne des circuits de forme irrégulière : leur placement sera tout un poème.

Conclusions

- L'approche de Dadda a été généralisée et précisée
- Elle suppose que $T_{c_{out}} = T_s$ dans le FA. Si (comme) ce n'est pas vrai, faut tout reprendre.
  - articles d'Oklobdzija
- Tout cela nous donne des circuits de forme irrégulière : leur placement sera tout un poème.
- On peut changer la granularité de notre compresseur de base (W_7 en transistors en plus de W_3 par exemple)

Conclusions

- L'approche de Dadda a été généralisée et précisée
- Elle suppose que $T_{c_{out}} = T_s$ dans le FA. Si (comme) ce n'est pas vrai, faut tout reprendre.
  - articles d'Oklobdzija
- Tout cela nous donne des circuits de forme irrégulière : leur placement sera tout un poème.
- On peut changer la granularité de notre compresseur de base (W_7 en transistors en plus de W_3 par exemple)
- On peut se préoccuper de l'ordre d'arrivée des chiffres

Conclusions

- L'approche de Dadda a été généralisée et précisée
- Elle suppose que $T_{c_{out}} = T_s$ dans le FA. Si (comme) ce n'est pas vrai, faut tout reprendre.
  - articles d'Oklobdzija
- Tout cela nous donne des circuits de forme irrégulière : leur placement sera tout un poème.
- On peut changer la granularité de notre compresseur de base (W_7 en transistors en plus de W_3 par exemple)
- On peut se préoccuper de l'ordre d'arrivée des chiffres
- L'additionneur rapide final commence en général sa propagation de retenue avant d'avoir tous les chiffres. Sur les poids faibles, c'est un CPA.

Pour finir

Un papier avec de l'algorithmique et du circuit

- Décomposition en blocs et étude des coûts
- Optimisation algorithmiques
- Optimisation des briques de bases

Gensuke Goto, Atsuki Inoue, Ryoichi Ohe, Shoichiro Kashiwakura, Shin Mitarai, Takayuki Tsuru, and Tetsuo Izawa : A 4.1-ns Compact 54 54-b Multiplier Utilizing Sign-Select Booth Encoders, *IEEE Journal of Solid-State Circuits*, vol. 32, no. 11, november 1997

Conclusions sur les multiplieurs

Dépendance à la technologie

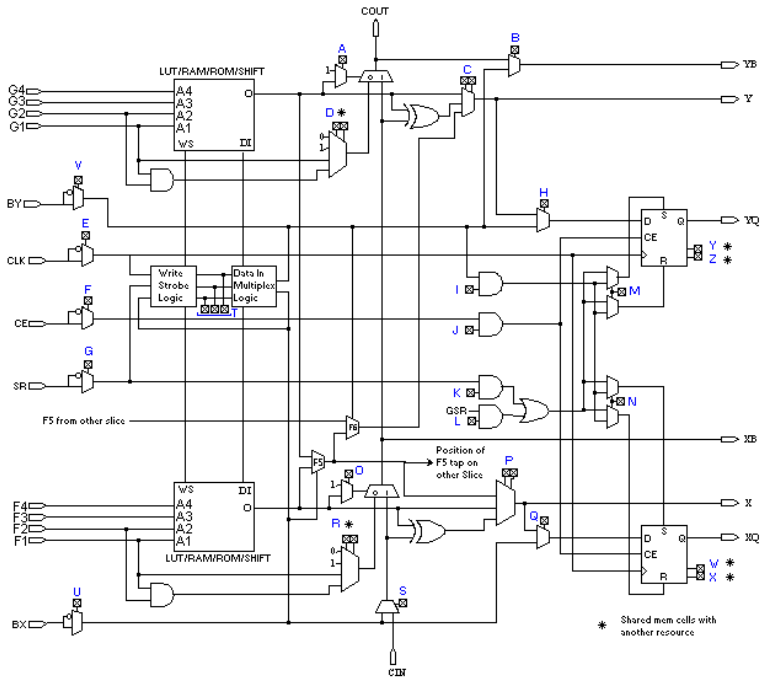
- Elle est encore plus terrible que pour les additionneurs

Dépendance à la technologie

- Elle est encore plus terrible que pour les additionneurs
- Nous on ne travaille plus sur des multiplieurs mais sur des générateurs de multiplieurs

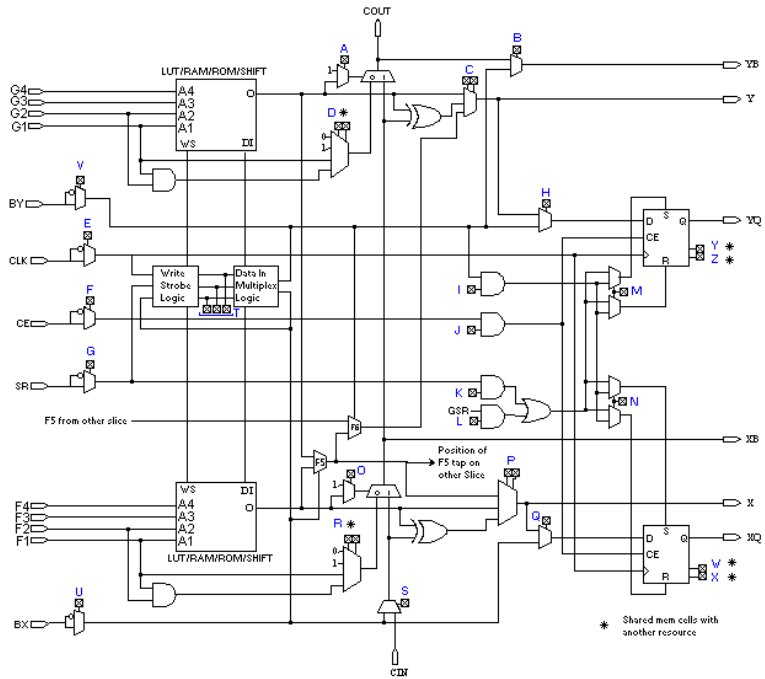
Dépendance à la technologie

- Elle est encore plus terrible que pour les additionneurs
- Nous on ne travaille plus sur des multiplieurs mais sur des générateurs de multiplieurs
- Exo : et dans les FPGA ?



Dépendance à la technologie

- Elle est encore plus terrible que pour les additionneurs
- Nous on ne travaille plus sur des multiplieurs mais sur des générateurs de multiplieurs
- Exo : et dans les FPGA ?



- Dans les FPGA récents, il y a des multiplieurs 18x18.
- Exo : comment faire un multiplieur 24x24 avec ?

Autres opérations liées

- Multiplication-accumulation ($XY + Z$) : coûte un epsilon de plus que la multiplication
  (suffit d'ajouter la ligne de bits de Z au paquet des produits partiels)
- Addition de plusieurs nombres
- Multiplication par une constante
- Somme de multiplications par des constantes
  ($12x + 42y + 23z$)
  utile pour les filtres paraît-t-il

Ce que j'ai escamoté

- Les multiplieurs bit-série, parallèle-série, en ligne