Une unique unité de calcul RISC pour les entiers et les flottants.

Bernard Goossens[1] Duc Thang Vu[2]
IBP-LITP, Université Paris 7 Denis Diderot,
2 place Jussieu, 75251 Paris cedex 05, France

Abstract

Nous proposons dans cet article de remplacer les unités de calcul entiers et flottants par une unique unité de calcul. Les opérations flottantes y sont émulées par des instructions entières. L'intérêt est double: D'une part cela permet d'intégrer des opérateurs entiers très performants dont le coût en place est amorti par le fait qu'ils sont aussi employés pour le calcul flottant. D'autre part, les calculs flottants sont plus rapides en moyenne que lorsqu'ils sont menés dans une unité spécialisée car d'abord, les algorithmes d'émulation que nous proposons s'appuient sur les données pour réduire le nombre d'étapes du calcul au strict nécessaire et ensuite, le cycle cadençant l'unité de calcul a été réduit à la traversée d'un additionneur 16 bits. Ainsi, la latence de l'addition flottante en double précision est-elle en moyenne de 6.7 cycles. Celle de la multiplication flottante double précision est de 8.3 cycles et celle de la division flottante double précision est de 18.8 cycles. La taille d'une telle unité mixte est estimée à 200000 transistors, à peine plus que les unités flottantes actuelles mais en faisant l'économie d'un banc de registres flottants séparés.

1 Introduction.

Les processeurs actuels, qu'ils soient CISC ou RISC, traitent le calcul flottant à partir d'une unité spécialisée. Dans le cas des processeurs RISC, le pipeline flottant se distingue du pipeline entier par son nombre d'étages. Cette latence accrue des opérations flottantes a conduit à séparer le processeur en deux parties tant pour les opérateurs que pour les registres; en fait, la totalité du pipeline une fois passé l'étage de décodage. Cette irrégularité de construction est en général présentée comme un avantage. En premier lieu, on avance que de cette façon les opérations flottantes ne présentent plus de conflit de ressources avec les opérations entières. La séparation règle aussi le problème épineux du traitement précis des interruptions puisque même si des opérations entières postérieures à une opération flottante déclenchant une trappe d'exception sont achevées avant l'interruption, comme les espaces flottants et entiers sont disjoints, les résultats entiers ne peuvent interférer sur le calcul flottant inachevé. Celui-ci, à défaut de devoir être abandonné, peut être repris par le traitement d'exception, l'exécution du programme continuant après la dernière instruction entière exécutée. Enfin, on suggère que deux unités séparées sont profitables aux architectures superscalaires, permettant d'exécuter simultanément une instruction entière et une instruction flottante.

[1]email: bg@litp.ibp.fr
[2]email: vu@litp.ibp.fr

Nous pensons qu'on peut obtenir les mêmes avantages à moindre frais avec une unité de calcul unique dont les opérateurs ont tous un pipeline de même longueur. Notons déjà que les espaces flottants et entiers sont naturellement disjoints par le logiciel, les opérations de conversions mises à part. De plus, les pipelines de longueurs égales assurent tout aussi naturellement un traitement précis des interruptions. Enfin, rien n'empêche qu'une instruction flottante émulée soit exécutée en même temps qu'une instruction entière.
Nous pensons encore que les opérations flottantes, qui sont des opérations complexes, ne devraient pas être traitées directement par le matériel, mais leur comportement simulé par l'exécution d'instructions plus élémentaires. En d'autres termes, dans un processeur RISC on devrait traiter le flottant comme on traite les modes d'adressage complexes.
La phase d'exécution des opérateurs flottants est composée de plusieurs cycles; d'autant plus que le cycle est court. Chacun de ces cycles d'exécution correspond à l'application d'une opération entière sur l'une ou l'autre des parties de la représentation flottante. Plutôt que de construire un matériel ad-hoc enchaînant les étapes nécessaires, il est plus dans l'esprit de RISC -donc potentiellement plus efficace- d'utiliser le matériel existant -les opérateurs entiers- en réduisant autant que possible le CPI moyen des instructions flottantes (nombre de Cycles Par Instruction). Chaque étape d'un calcul flottant est alors menée par une instruction commandant l'opérateur entier utilisé. On peut même pipeliner les opérateurs, donc chaque étape.
Or, si les additionneurs et décaleurs traditionnels sont suffisamment efficaces pour former la base des algorithmes des opérations flottantes, il n'en est pas de même des multiplieurs et diviseurs entiers qui travaillent en plusieurs étapes. De nos jours, l'intégration d'un réseau cellulaire complet pour effectuer une multiplication en une seule étape est possible. Mais on ne la considère pas comme indispensable car les mutiplications entières nécessitant un multiplieur complet sont très peu fréquentes dans les programmes [MAG88]. La plupart du temps, on peut s'en sortir avec quelques étapes, voire simplement par décalage et addition. C'est la raison pour laquelle on préfère économiser la place que nécessiteraient ces opérateurs.
Toutefois, leur emploi, tant dans les calculs entiers que flottants, permettrait d'en rendre le coût supportable. En effet, les applications scientifiques telles que Spice peuvent comporter jusqu'à 15% d'instructions flottantes [HEN90]. C'est sur la base de tels opérateurs entiers que nous proposons, dans la suite de l'article, d'émuler les opérations flottantes.
La section 2 présente les algorithmes d'émulation des opérations flottantes IEEE et la section 3, l'architecture des opérateurs entiers formant l'unité mixte ([GOO95-1] présente des opérateurs entiers -non adaptés aux calculs flottants- du même genre que ceux proposés ici; le lecteur intéressé y trouvera quelques détails d'architecture qui ne figurent pas dans le présent article).

2 Les algorithmes des opérations flottantes dans la norme IEEE-754.

Les trois opérations flottantes IEEE -l'addition, la multiplication et la division- sont toutes une combinaison d'opérations de décalage (alignement et normalisation des mantisses), d'addition (arrondi et traitement des signes) et d'opérations entières (addition, multiplication et division des mantisses). Seul le traitement des exposants, et de façon associée celui des représentations spéciales (zéro, infini, NaN et nombres dénormalisés) nécessitent un matériel supplémentaire mais réduit. L'ajustement des calculs de mantisses aux différentes précisions peut se faire simplement par un cadrage approprié des données lors de leur séparation à l'entrée du pipeline, et du résultat lors de son assemblage en sortie. Entre les deux, un additionneur, un multiplieur et un diviseur de largeur maximum calculent dans la plus grande précision.
La première étape de chaque algorithme traite les représentations spéciales. On les reconnait à partir des exposants, des signes, des mantisses et du type d'opération à réaliser. En voici l'algorithme dans lequel $s1$ et $s2$ désignent les signes des deux opérandes, $m1$ et $m2$, les mantisses et $e1$ et $e2$, les exposants. $Emax$ est le champ exposant tout à 1, r, un flottant normalisé ou non (mais ni un infini, ni un NaN), f, un flottant quelconque (y compris un infini ou un NaN) et d, un flottant dénormalisé (exposant nul et mantisse non nulle). Enfin, *op* est l'opération flottante considérée et $rés$, son résultat. Avant toute chose, si l'opération est une soustraction, on inverse $s2$.

1. Si e1=Emax=e2 et s1≠s2 et m1=m2=0 (+∞ op -∞ ou -∞ op +∞),

   si op=*, rés:= -∞; sinon, rés:= NaN;

2. Si e1=Emax=e2 et s1=s2 et m1=m2=0 (+∞ op +∞ ou -∞ op -∞),

   si op=*, rés:= +∞; sinon, si op=/, rés:= NaN; sinon, rés:= s1 ∞;

3. Si e1=Emax et m1=0 et e2=0 et m2=0 (∞ op 0),

   si op=*, rés:= NaN; sinon, rés:= s1 ∞;

4. Si e2=Emax et m2=0 et e1=0 et m1=0 (0 op ∞),

   si op=*, rés:= NaN; sinon, si op=/, rés:= 0; sinon, rés:= s2 ∞;

5. Si e1=0 et m1=0 et e2=0 et m2=0 (0 op 0),

   si op=/, rés:= NaN; sinon, rés:= 0;

6. Si e1=0 et m1=0 et e2≠Emax (0 op r),

   si op=* ou op=/, rés:= 0; sinon, rés:= r;

7. Si e1≠Emax et e2=0 et m2=0 (r op 0),

   si op=*, rés:= 0; sinon, si op=/, rés:= s1 ∞; sinon, rés := r;

8. Si e1=Emax et m1=0 et e2≠Emax (∞ op r), rés:= s1 ∞;
9. Si e1≠Emax et e2=Emax et m2=0 (r op ∞),

   si op=/, rés:= 0; sinon, rés:= s2 ∞;

10. Si (e1=Emax et m1≠0) ou (e2=Emax et m2≠0) (NaN op f ou f op NaN), rés:= f;
11. Si (e1=0 et m1≠0) ou (e2=0 et m2≠0) (d op f ou f op d),

    si op=* ou op=/, trappe;

12. Sinon (e1≠Emax et e2≠Emax et (e1≠0 ou m1≠0) et (e2≠0 ou m2≠0)),

    étape suivante.

Voici maintenant les algorithmes des opérations flottantes en faisant apparaître pour chacun les opérateurs entiers employés. Tout d'abord, celui de l'addition flottante:

Etape 1: Traitement des cas spéciaux (comparateurs spéciaux et soustracteur d'exposants).

Si un cas spécial est reconnu, le traiter selon l'algorithme ci-dessus et fin.
Sinon et si les exposants et les signes sont égaux, continuer par l'étape 3.
Sinon et si les exposants sont égaux, continuer par l'étape 4.

Etape 2: Alignement des mantisses (décaleur variable à droite).

Aligner les mantisses (le nombre de décalages sort de l'étape 1).
Si les signes ne sont pas égaux, continuer par l'étape 4.

Etape 3: Addition de nombres de mêmes signes avec arrondi (deux additionneurs).

Additionner les mantisses et l'arrondi (premier additionneur).
Additionner les mantisses (0 en tête, tronquées d'un bit à droite) et l'arrondi (deuxième additionneur en parallèle).
Si la première addition n'a pas de retenue, elle forme le résultat.
Sinon, c'est la seconde; incrémenter l'exposant; s'il vaut Emax, le résultat est s1 ∞;
Fin.

Etape 4: Addition de nombres de signes opposés (additionneur).

Additionner les mantisses en complémentant à deux celle de l'opérande négatif.

Etape 5: Signe et arrondi du résultat (signes opposés; additionneur).

Complémenter à deux le résultat et ajouter l'arrondi.
Si le chiffre haut est $\geq$8, fin.

Etape 6: Normalisation (décaleur variable à gauche).

Si la mantisse est nulle, le résultat est 0.
Sinon, décaler la mantisse de c=min(n,e) crans à gauche et e:= e-c. (e est l'exposant du résultat et n le nombre de 0 en tête de la mantisse)
Fin.

Les algorithmes des opérations flottantes, comme le montre celui de l'addition ci-dessus, s'expriment en terme d'opérations entières. Chaque étape est effectuée par une instruction commandant les opérateurs entiers qu'elle invoque (par exemple, l'étape 3 commande simultanément deux additionneurs). L'exécution de ces instructions, comme toutes celles du processeur, est pipelinée en 6 phases (le détail du découpage est donné en section 3). C'est la première phase qui, à chaque étape, établit l'étape suivante, selon l'algorithme (par exemple, à l'issue de la phase 1 de l'étape 1, on est en mesure de dire si la prochaine étape doit être la 2, la 3, la 4 ou si même le calcul est achevé; le choix ne repose que sur des comparaisons d'exposants et de signes).
Notons que dans l'algorithme précédent, toutes les étapes s'enchaînent sans délai par un mécanisme d'envoi par tranches (par exemple, tous les étages du pipeline détenant une tranche [0-15] de résultat la transmettent au premier étage, qui contient les opérateurs chargés de cette tranche). Seule la sixième doit être retardée jusqu'à ce que l'étape 5 ait produit la tranche haute de la mantisse pour déterminer le nombre de zéro de tête (la structure de l'additionneur -voir en section 3- impose 3 cycles de délai).
Les statistiques sur la distribution des flottants [KNU68] montrent que l'algorithme se réduit à l'étape 1 dans 10% des cas (cas spéciaux et opérande nul), se poursuit par l'étape 3, soit directement (15%), soit après l'étape 2 (30%), passe par les étapes 1, 4 et 5 dans 21% des cas, 1, 2, 4 et 5 dans 20% des cas et 1, 2, 4, 5 et 6 dans seulement 4% des cas. Ce qui fait qu'en moyenne, une addition flottante se déroule en 3 étapes (1*10% + 2*15% + 3*51% + 4*20% + 5*4%), permettant un débit d'un résultat tous les 3 cycles. L'architecture des opérateurs entiers (section 3) permet d'obtenir le résultat flottant définitif après le cycle 5 de la dernière étape (le cycle 2 quand seule l'étape 1 est nécessaire). Ainsi, la latence moyenne de l'addition flottante, hors cas spéciaux traités par trappe, est de 6.67 cycles (2*10% + 6*15% + 7*51% + 8*20% + 10*4%). Le cycle machine que nous considérons ici, comme le montrera la section suivante, est très court: il correspond à la traversée d'un additionneur 16 bits, soit la moitié du cycle d'un processeur comme le DEC 21164 [EDM95] (dont le chemin critique comprend un additionneur 64 bits) ou le cinquième du cycle du Power PC604 [SON94] (un multiplieur double précision dans son chemin critique). L'addition flottante du 21164 a une latence de 4 cycles (8 de nos cycles) et celle du PPC604, de 3 cycles (15 de nos cycles).
Les algorithmes de la multiplication et de la division sont plus simples que celui de l'addition. Voici celui de la multiplication:

Etape 1: Traitement des cas spéciaux (infini, NaN et 0).

> Calculer l'exposant du produit (somme des exposants).
>
> Trappe en cas de débordement.
>
> Si un cas spécial ou un débordement est signalé, fin.

Etape 2: Produit des mantisses (multiplieur).

Etape 3: Arrondi et normalisation (additionneur inclus dans le diviseur).

L'étape 3 nécessite un délai d'un cycle avant de commencer et s'effectue dans un additionneur placé en queue de l'opérateur de division (voir les raisons architecturales à la section suivante). Les nombres dénormalisés étant traités par trappe, le résultat de l'étape 2 est presque normalisé. Au pire, il nécessite un décalage à gauche d'une position si le bit fort de la mantisse n'est pas un 1, si bien que l'arrondi peut être effectué en même temps. En moyenne (10% de cas spéciaux et 90% de cas ordinaires) le multiplieur autorise un débit moyen d'un résultat simple ou double précision tous les 3.7 cycles (10%*1 + 90%*4 cycles) pour une latence moyenne de 8.3 cycles (10%*2 + 90%*9 cycles). Le 21164 calcule un produit flottant double précision en 4 cycles (8 de nos cycles; il est difficile de gagner sur la multiplication car la plupart des cas nécessitent le nombre maximum d'étapes; notons cependant qu'il est plus important de gagner sur l'addition qui est l'opération flottante la plus fréquemment utilisée) et le PPC604, en 3 cycles (15 de nos cycles). Pour la division flottante, l'algorithme est le suivant:

Etape 1: Traitement des cas spéciaux.

Calculer l'exposant du quotient (différence des exposants).

Trappe en cas de débordement.

Si un cas spécial ou un débordement est signalé, fin.

Etape 2: Calcul du quotient (diviseur; 7 cycles en simple précision, 14 cycles en double précision).

Si ce n'est pas le dernier cycle, continuer par l'étape 2.

Sinon, si q_haut≥8, continuer par l'étape 4.

Etape 3: Normalisation (décaleur variable à gauche).

Etape 4: Arrondi (additionneur).

Le quotient étant formé des poids forts vers les poids faibles, l'étape de normalisation peut s'effectuer sans délai. Le débit moyen de la division est d'un résultat simple précision tous les 8.8 cycles (60%*10 + 30%*9 + 10%*1 cycles) et d'un résultat double précision tous les 16 cycles (60%*17 + 30%*16 + 10%*1 cycles). La latence moyenne est de 12.5 cycles (60%*14 + 30%*13 + 10%*2 cycles) en simple précision et de 18.8 cycles (60%*21 + 30%*20 + 10%*2 cycles) en double précision. La division flottante sur le 21164 prend 31 cycles (62 de nos cycles) et autant sur le PPC604 (155 de nos cycles).
Le code exécutable contient, pour chaque opération flottante, la totalité des instructions de l'algorithme, qu'elles soient effectivement exécutées ou pas. Par exemple, une addition flottante se présente sous la forme de 6 instructions correspondant aux 6 étapes possibles. L'enchaînement correct des étapes est déterminé par un automate à états finis, dont les transitions sont assurées par le premier cycle d'exécution de chaque étape. Par exemple, l'étape 1 d'une addition flottante doit se poursuivre soit par l'étape 2, soit par l'étape 3, soit par l'étape 4, ce qui est établi à la fin du premier cycle à partir de l'égalité des exposants et des signes. Selon l'état nouveau de l'automate, les étapes incorrectes arrivant dans le flux des instructions sont ignorées (et les instructions correspondantes sont écartées au profit de l'étape attendue; cela ne concerne bien entendu que les étapes d'un calcul flottant; si des instructions ordinaires ont été insérées par le compilateur -pour

remplir profitablement des délais éventuels-, ces instructions sont exécutées normalement, mais elles ne modifient pas l'état de l'automate).

3 opérateurs entiers.

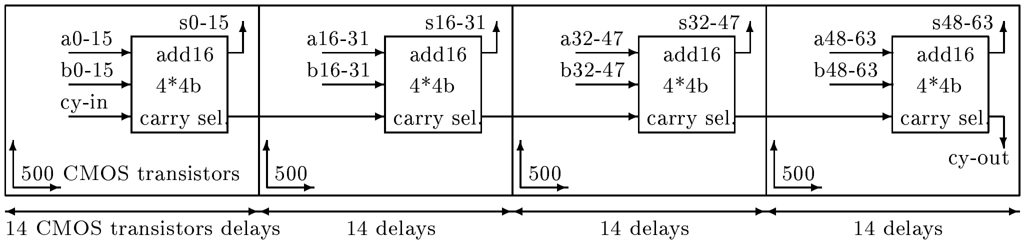

Figure 1: additionneur (2000 transistors CMOS)

Notre objectif est de fournir des opérateurs entiers de faible latence, découpés en pipelines de même longueur et permettant un fort débit. Pour cela, notre point de départ est de fixer le cycle du processeur au temps de traversée d'un additionneur rapide 16 bits. C'est ce découpage plus fin qui permet d'unifier la longueur des pipelines. Il faudrait bien sûr adapter les autres étages du pipeline -extraction, décodage, accès aux caches, traduction d'adresses-, mais ceci sort du cadre de cet article; le lecteur intéressé peut se reporter à [GOO95-2]). Basé sur ce cycle, l'additionneur présenté par la figure 1, effectuant $s[0-63] = a[0-63] + b[0-63] + cy\_in$, est composé de 4 tranches placées en série. Sa taille (2000 transistors pour la partie combinatoire et 300 bits mémorisés entre les étages) et son délai de propagation (14 transistors en série) sont estimés en nombre de transistors CMOS. Un mécanisme d'envoi des tranches de résultat aux étages de calcul précédents permet de limiter la latence de l'addition entière à un cycle (enchaînement des additions sans délai, même dépendantes).

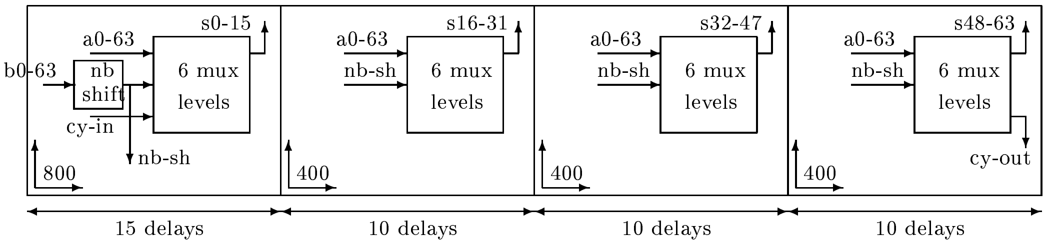

Figure 2: décaleur (2000 transistors CMOS)

Sur le même modèle, on construit deux décaleurs variables (0 à 63 crans, le premier décale à droite, $s[0-63] = a[0-63]/2^{nb\_sh}$, et le second à gauche, $s[0-63] = a[0-63] * 2^{nb\_sh}$) en 4 tranches. La figure 2 représente de façon générique les deux décaleurs. Le premier étage du pipeline produit la tranche basse du résultat et le dernier étage, la tranche haute.

Le décaleur à gauche permet d'effectuer des opérations de cadrage, nécessaires pour la normalisation des nombres flottants. Les différents niveaux du décaleur (décalage de 32, 16, 8, 4, 2 et 1 crans) sont commandés soit par une constante issue du code opération ($op[0-5]$) ou d'un registre ($b[0-5]$), soit par les sorties de comparateurs d'égalité, comparant à 0 les préfixes successifs du mot décalé ($a[0-63]$), de 32, 16, 8, 4, 2 et 1 bits. Le nombre de décalages (nb_sh) effectués est un résultat de la première tranche. Il faut aussi noter que si deux décalages à gauche peuvent s'enchaîner sans délai grâce au mécanisme d'envoi, il n'en est pas de même de deux décalages à droite. Le second devra attendre jusqu'à 3 cycles que le premier résultat soit formé avant de procéder au calcul de la première tranche. Cependant, comme ces opérations sont assez peu fréquentes (1% à 2% [HEN90]), on peut penser que le compilateur trouvera souvent de quoi remplir l'intervalle.
Les pipelines de l'additionneur et des décaleurs sont allongés de 2 étages supplémentaires en queue (simple propagation des résultats) pour qu'ils aient la même longueur que les pipelines du multiplieur et du diviseur. Cela ne nuit pas à la latence grâce au mécanisme d'envoi. Pour l'addition, ces deux étages peuvent être mis à profit pour effectuer la traduction d'adresse et le contrôle d'étiquette lors des opérations de rangement et de chargement. Six étages de pipeline pour l'addition et le décalage sont indispensables car le multiplieur, lui, ne peut pas se contenter de 4. Un réseau cellulaire pair-impair d'additionneurs à conservation de retenue (CSA), opérant en base 4, représente 16 étages de CSA pour délivrer un résultat de 128 bits en somme et retenue ([HEN90]). Dans le temps d'une addition 16 bits, il n'est pas possible de traverser plus de 4 niveaux de CSA. Donc, la traversée du réseau complet prend à elle seule 4 cycles. Les 64 bits bas du résultat sont formés en 5 cycles et les 64 bits hauts en 6 cycles ($p[0-127] = a[0-63] * b[0-63]$).

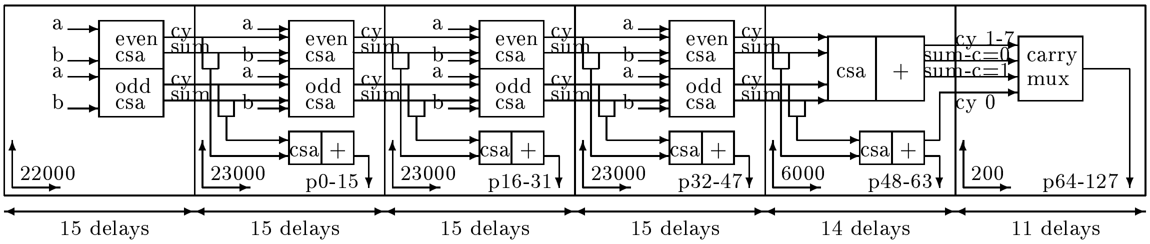

Figure 3: multiplieur (120000 transistors CMOS)

Les 4 premiers étages contiennent le réseau cellulaire pair-impair base 4. Les rejets de chaque ligne de CSA (4 bits faibles) sont assemblés dès que disponibles: les triplets de même colonne passent au travers de CSA et lorsque, par réduction 3-2 des CSA, il ne reste plus qu'un couple par colonne, il est sommé sans délai dans un additionneur à sélection de retenue. Ainsi, le second étage délivre les 16 bits faibles du résultat ($p[0-15]$), puis chaque étage suivant en fait de même, si bien que la moitié basse du produit ($p[0-63]$) est obtenue en sortie du cycle 5. Ce cinquième cycle effectue en outre l'addition en somme et retenue des 4 mots de 64 bits sortant de l'étage 4 (deux couples somme et retenue).

Cela nécessite deux lignes de CSA supplémentaires. Enfin, la somme et la retenue finales sont additionnées dans un additionneur 64 bits à sélection de retenue. Seules les sommes sont effectuées dans le cycle, avec et sans retenue d'entrée, par tranches de 8 bits. La sélection des bonnes tranches est reportée au cycle suivant. La partie haute ($p[64-127]$) est donc calculée au bout de 6 cycles.
Certes, ce multiplieur coûte cher: un réseau cellulaire complet pour un produit de 128 bits et divers additionneurs en sélection de retenue: à peu près 120000 transistors en tenant compte des mémorisations entre les étages du pipeline (un peu plus de 3000 bits). Mais il sert aussi pour le flottant, donc il sera plus utilisé qu'un multiplieur entier ordinaire. Et puis, on peut atteindre un débit maximum d'un produit 128 bits par cycle, avec un cycle réduit à une addition 16 bits. Toujours grâce au mécanisme d'envoi, la tranche faible du produit est disponible dès le deuxième cycle, ce qui donne à l'opérateur une latence d'à peine deux cycles (un seul cycle d'attente avant de se servir de la partie basse).
Le diviseur entier reste un opérateur à étapes. Même en travaillant en base 16, il lui faut tout de même 16 cycles pour fournir le quotient. Pas question donc d'incorporer ce calcul dans le pipeline. En se limitant aux 6 étages déjà nécessaires pour le multiplieur, une étape de division calcule 4 bits de quotient et le reste. Si bien qu'en 16 étapes, le quotient 64 bits et le reste 64 bits sont simultanément disponibles. La méthode employée est une division SRT en base 16 à redondance maximum (chiffres de quotient entre -15 et +15, 0 admis [ATK68]; le 0 doit être admis car le reste est calculé en notation somme et retenue, ce qui fait que son signe ne peut être déterminé sans une dernière somme de 64 bits; puisque pour des raisons de délai on se contente d'effectuer cette somme sur un préfixe de 7 bits, le signe du reste demeure indéterminé au voisinage de 0; on fixe donc arbitrairement le chiffre de quotient à 0, tant pour un reste très peu négatif que pour un reste très peu positif). Une table des quotients de 4K*4 bits est fournie, adressée par le préfixe du diviseur (6 bits, dont le poids fort est un 1) et le préfixe estimé de la valeur absolue du reste partiel (7 bits). On obtient en sortie le chiffre de quotient en valeur absolue.

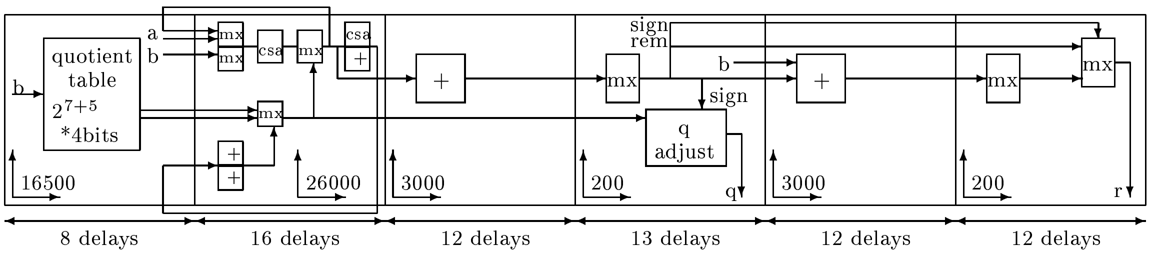

Figure 4: diviseur (50000 transistors CMOS)

Le premier cycle (voir la figure 4) sélectionne une ligne entière de la table à partir du diviseur $b[58-62]$), soit 128 quotients possibles correspondant aux 128 valeurs possibles du préfixe du reste non encore connu. Au second cycle, on procède au choix du bon quotient. Ce choix est basé sur le préfixe du reste calculé par le même étage de pipeline au cycle précédent.

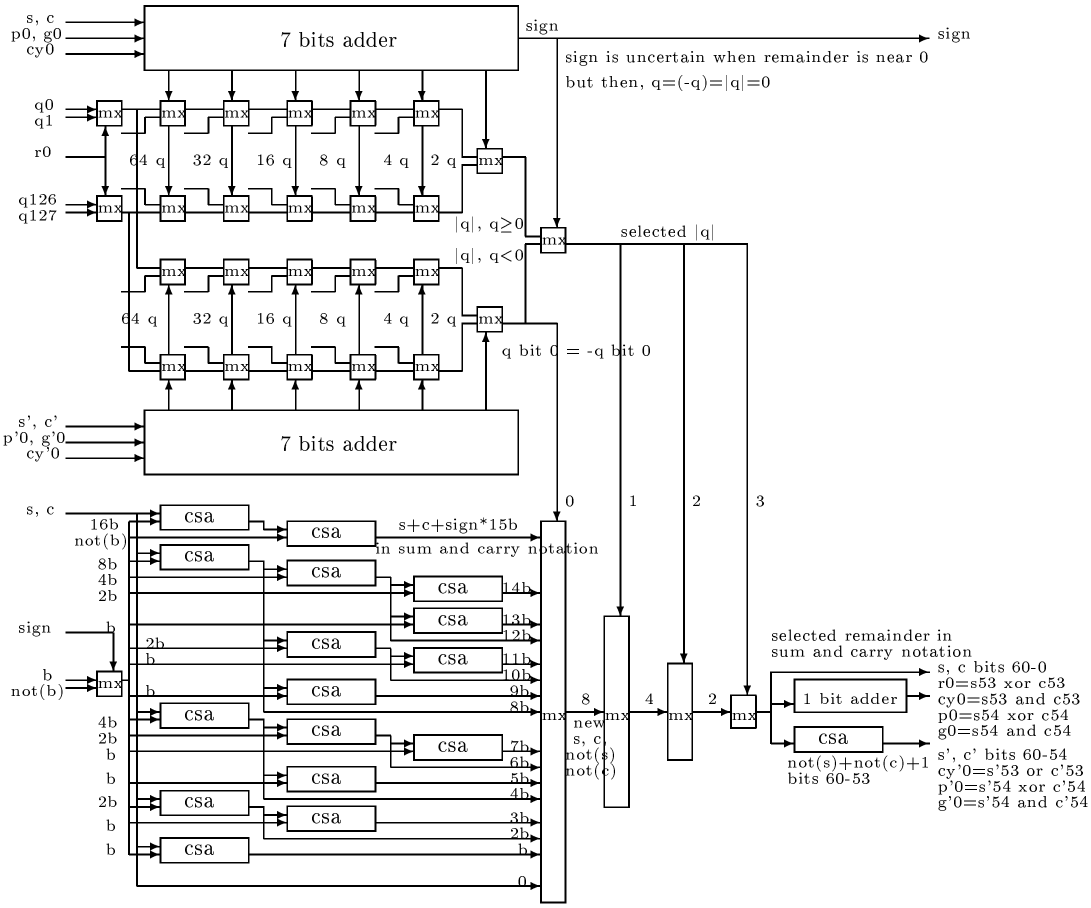

Figure 5: cycle 2 du diviseur

Plus précisément (voir la figure 5), pour équilibrer les étages et limiter chacun au temps de traversée d'un additionneur 16 bits, au cycle précédent on calcule le bit faible de ce préfixe (noté r_0 sur la figure 5), les autres bits n'étant fournis qu'en forme de somme et retenue. Le bit faible disponible dès le début du cycle courant permet de choisir 64 quotients parmi les 128 sortant de la table ($q_0 ... q_{127}$). En parallèle, la somme et la retenue formant le préfixe sont additionnées (deux additionneurs 7 bits, l'un effectuant $s + c$ et l'autre $s' + c' + 1$, avec s'=somme($\neg s + \neg c + 1$) et c'=retenue($\neg s + \neg c + 1$), calculés par CSA au cycle précédent; on obtient ainsi les deux expressions possibles de la valeur absolue du reste). Les bits de résultat sont utilisés un à un au fur et à mesure de leur arrivée pour affiner le choix. Après 7 étages de multiplexeurs, une dernière sélection, basée sur le signe du nouveau reste, a lieu entre le quotient choisi par les sorties du premier additionneur (reste positif) et celui choisi par celles du second additionneur (reste négatif). La valeur absolue du quotient est alors prête. Pendant le même temps, les 16 restes possibles ($r_i = r_{i+1} - |q_i|b$, avec $0 \le q_i \le 15$ si $r_{i+1} \ge 0$ ou $r_i = r_{i+1} + |q_i|b$, $-15 \le q_i < 0$ si $r_{i+1} < 0$) sont calculés en parallèle (par exemple, $r_{i+1} - 7b$ est obtenu en retranchant d'abord $4b$, puis $2b$ et enfin b par trois étages de CSA; voir la moitié basse de la figure 5). Le quotient choisi sert ensuite à déterminer le reste en notation somme et retenue (4 niveaux de multiplexeurs). Il n'y a plus qu'à calculer la valeur exacte du bit faible du préfixe (additionneur un bit) et les termes s' et c' (additionneur CSA 8 bits) pour avoir tout ce qui est nécessaire à l'étape suivante pour son propre calcul de quotient.

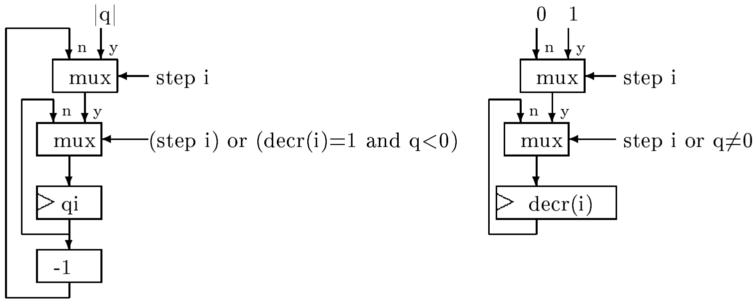

Figure 6: ajustement du quotient

Le calcul des 16 restes nécessite au total 16 rangs de 64 additionneurs CSA en 3 étages maximum (environ 25000 transistors). C'est le prix à payer pour fournir 4 bits de quotient dans le temps d'une addition 16 bits. Le troisième et le quatrième cycles (voir figure 4) calculent le reste par une addition 64 bits du même type que celle du multiplieur (addition par tranches de 8 bits puis sélection des tranches). En fonction du signe obtenu (signe vrai du reste), on ajuste le quotient. Si le reste obtenu au cycle 4 est positif, le quotient n'est pas modifié. Mais s'il est négatif, alors le préfixe de quotient déjà calculé (chiffres hexadécimaux q_{15} à q_{i+1}, $0 \leq i < 15$) doit être décrémenté et prolongé du chiffre de quotient q obtenu au cycle 2. Cette décrémentation est réalisée selon le procédé présenté en figure 6: les bascules formant les 16 chiffres q_i du quotient reçoivent en entrée un choix entre leur contenu antérieur ($q_i := q_i$ à l'étape j, $j \neq i$, si décrémentable(i)=0 ou $q \geq 0$), celui-ci décrémenté ($q_i := q_i - 1$ à l'étape j, $j \neq i$, si décrémentable(i)=1 et $q < 0$) ou le chiffre issu du cycle 2 ($q_i := q$ à l'étape i). Un chiffre est décrémentable s'il n'est suivi que de 0. Seize bascules conservent l'état décrémentable ou non de chacun des 16 chiffres du quotient (à l'étape j, décrémentable(j):=1, décrémentable(i):=0 si $q \neq 0$, $j \neq i$, décrémentable(i) n'est pas modifié si $q = 0$). Quant au reste, on lui ajoute le diviseur b pour le rendre positif. Pour cela, un second additionneur 64 bits à sélection de retenue est employé (cycles 5 et 6). Ainsi, la latence d'un quotient de 64 bits est de 20 cycles (rappelons que l'envoi par tranche ne peut pas s'appliquer au quotient puisqu'il est obtenu par ses chiffres de poids fort en premier) et celle du reste est de 21 cycles. Le débit maximum est d'un quotient et d'un reste de 64 bits tous les 16 cycles. Le coût est d'environ 60000 transistors (y compris 1500 bits mémorisés le long des 6 étages du pipeline).

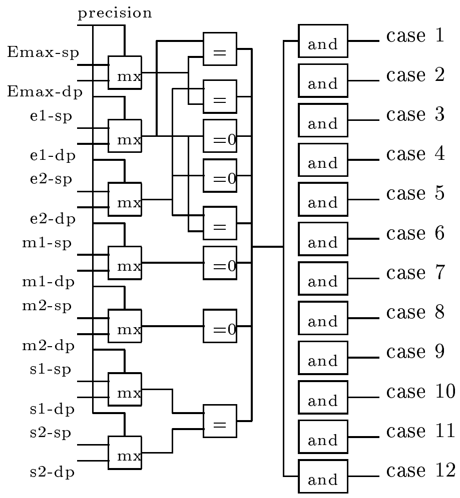

Figure 7: cas spéciaux des opérations flottantes

Ces opérateurs entiers représentent une amélioration très significative des opérateurs disponibles dans les différents processeurs commercialisés: le 21164 ne calcule un produit de même taille qu'en 14 cycles et le PPC604 effectue un produit 32*32 bits en 4 cycles, soit respectivement 28 et 20 de nos cycles. La division entière n'est disponible que sur le PPC604 et un quotient 32 bits n'y est délivré qu'en 20 cycles (soit 100 de nos cycles). Cette amélioration tient, il ne faut pas le cacher, aux gros moyens matériels mis en oeuvre pour calculer vite (l'unité entière, comprenant deux additionneurs à anticipation de retenue -deux pour accélérer l'addition flottante- deux décaleurs variables, un multiplieur et un diviseur est estimée à 200000 transistors, y compris les spécificités du flottant, essentiellement le traitement des exposants et des cas spéciaux). Mais cet investissement vaut tant pour le calcul entier que pour le calcul flottant. Et tous comptes faits, l'unité de calcul unique ne devrait pas être plus coûteuse qu'une unité flottante moins performante doublée d'une unité entière opérant par étapes et nécessitant deux ensembles de registres (le Weitek3364 contient 165000 transistors et le TI8847, 180000). Pour terminer la présentation matérielle de cette unité de calcul, voici la partie (figure 7) reconnaissant les différents cas spéciaux (étape 1 de chaque algorithme). Son traitement n'excède pas un cycle et sa taille n'est que de 2000 transistors environ. En entrée, on fabrique, à partir de la précision demandée (simple *sp* ou double *dp*), deux exposants de 16 bits cadrés à droite -ce sont des entiers- et complétés à gauche par des 0 et deux mantisses de 64 bits, cadrées à gauche cette fois -ce sont des nombres fractionnaires-, avec en tête le 1 implicite des nombres normalisés ou le 0 des nombres dénormalisés et complétées à droite par des 0. L'arrondi des mantisses pour toutes les opérations flottantes se fera en multiplexant la retenue du bit 40 de l'additionneur en simple précision et celle du bit 10 en double précision. La retenue choisie est soit celle sortant du rang précédent -addition ordinaire- soit le bit d'arrondi formé -arrondi d'une opération flottante-.

Conclusion.

L'idée d'une unité flottante micro-commandée apparait dans [DAL89]. Cependant, elle emploie un matériel séparé de celui réservé à l'unité entière. L'accent est mis sur le mécanisme de micro-commande en faisant remarquer qu'il offre de nombreuses opportunités d'optimisation de la part du compilateur comme par exemple certains enchaînements normalisation-alignement.
Les nouvelles possibilités d'intégration font qu'il devient possible de disposer d'opérateurs entiers très performants: un produit 128 bits par cycle, un quotient 64 bits en 16 cycles, avec un cycle réduit à une addition 16 bits. Ces opérateurs étant en général peu employés par les programmes (peu de divisions entières et la plupart des multiplications entières mettent en oeuvre une constante, souvent de faible valeur ce qui fait que la multiplication peut soit être réalisée par un opérateur à étapes soit par décalages et additions, le tout en peu de cycles), leur place dans un processeur RISC ne peut se justifier que s'ils servent en même temps au calcul flottant. Nous avons montré dans cet article que cela permet même d'améliorer la latence des opérations flottantes par rapport aux dispositifs usuels. De plus, la fusion des unités entières et flottantes devrait accroître la vitesse d'exécution des programmes faisant beaucoup de conversions, comme Spice [HEN90], en supprimant les transferts entre registres flottants et entiers. Enfin, l'uniformisation de la longueur des pipelines devrait simplifier grandement le contrôle des processeurs superscalaires.

Références:

[ATK68] D. E. Atkins: Higher Radix Division Using Estimation of the Divisor and Partial Remainders. IEEE Trans. Comp., C-17 925-934 1968
[DAL89] W. J. Dally: Micro-optimization of Floating Point Operators. 3rd ASPLOS, in Sigplan Notices Vol. 24 283-289 1989
[EDM95] J. H. Edmondson, P. Rubinfield, R. Preston, V. Rajagolapan: Superscalar Instruction Execution in the 21164 Alpha Microprocessor. IEEE Micro April 1995
[GOO95-1] B. Goossens, D. T. Vu: Data Slicing Arithmetic Operators. Real Numbers and Computer Conference, St-Etienne April 1995
[GOO95-2] B. Goossens, D. T. Vu: Further Pipelining and Multithreading to Improve RISC Processor Speed. A Proposed Architecture and Simulation Results. PaCT-95 Conference, St-Petersburg September 1995
[HEN90] J. L. Hennessy, D. A. Patterson: Computer Architerture. A Quantitative Approach. Morgan Kaufmann 1990
[KNU68] D. E. Knuth: The Art of Computer Programming. Seminumerical Algorithms. Vol. 2 p 238-247 1968
[MAG88] D.J. Magenheimer, L. Peters, K.W. Pettis, D. Zuras: Integer Multiplication on the HP Precision Architecture. IEEE Trans. Comput., 37-8 1988
[SON94] S. P. Song, M. Denman, J. Chang: The Power PC604 RISC Microprocessor. IEEE Micro October 1994