ARCHITECTURE VIRGULE FLOTTANTE DE L'ITANIUM

Jonathan Protzenko

3 novembre 2008

Table des matières

PARTIE I

PRÉFACE

L'introduction d'une nouvelle architecture de processeurs est une événement rare : au cours des vingt dernières années, peu de *challengers* sont venus concurrencer la suprématie du jeu d'instructions x86. C'est pourtant le cas avec l'architecture Itanium : d'abord projet de recherche démarré chez HP au début des années 90, Itanium devient un projet mené conjointement par Intel et HP en 1994. La première génération d'Itanium sort en 2001, suivie en 2002 par la deuxième génération (appelée Itanium 2).

L'Itanium incorpore nombre de concepts novateurs, et cherche à rompre avec les erreurs de conception du passé. On peut citer comme grands principes fondateurs de l'architecture Itanium :

- EPIC (explicit parallel instruction computing) : la parallélisation des instructions n'est pas gérée par un microcode dédié au sein du processeur mais par le compilateur au moment de la génération du code ;
- gestion innovatrice de la prédiction et des branchements ;
- calcul performant en virgule flottante, notamment avec l'insertion de la fameuse instruction FMA (fused multiply and add) qui ouvre la voie à de nouveaux algorithmes de division et de racine carrée par exemple.

Cette initiative tout à fait louable visant à libérer les utilisateurs de l'ISA x86 est pourtant un échec[1]. Une nouvelle architecture aurait été la bienvenue, mais à l'heure actuelle, l'architecture x86 est plus populaire que jamais (en témoigne la nouvelle ligne de processeur Atom introduisant le x86 jusque dans les appareils embarqués), alors qu'elle hérite de plusieurs dizaines d'années de défauts accumulés. Comment se fait-il qu'avec ses concepts novateurs, son unité virgule flottante incorporant le fruit de plusieurs années de recherche, l'Itanium n'ait connu qu'un succès d'estime ? Comment se fait-il que le x86, vieux, mal conçu, formé d'une accumulation successives de jeux d'instructions (MMX1 et 2, SSE1 jusqu'à 5…) connaisse un succès grandissant ?

Il y a donc des failles dans la conception de l'Itanium. En se reposant sur les compilateurs, en demandant un changement au niveau des utilisateurs, en supposant enfin qu'une FPU performante suffirait à vendre le processeur, les concepteurs de l'Itanium ont choisi la mauvaise stratégie.

Plutôt que de louer les mérites (indéniables) de l'Itanium comme le font les auteurs de l'article étudié [CHT], nous pensons qu'il sera plus instructif de critiquer les choix effectués dans l'Itanium. En mettant en perspective les solutions retenues par l'Itanium et celles retenues par d'autres architectures (Power notamment), nous pensons qu'il est possible d'effectuer une étude intéressante de cet article.

La première partie sera consacrée à la conception générale de l'Itanium : choix des types de données, paral-

[1] Dans les milieux bien informés, l'Itanium est surnommé Itanic…

lélisme, conception du processeur... Une deuxième partie s'intéressera à la division dans l'Itanium : la fameuse instruction FRCPA et les algorithmes qu'il est nécessaire d'implémenter constituent une illustration instructive des concepts de l'Itanium. Une comparaison des divisions en fonction du compilateur, du type de données sous-jacent sera effectuée. Quelques tests permettront d'effectuer des comparaisons relatives de performance.

Une troisième et dernière partie s'intéressera plus particulièrement à l'instruction FMA, à son support dans les différents langages, et à ses apports sur le plan pratique lors du développement (et non pas ses bénéfices sur le plan théorique, que nous connaissons bien).

PARTIE II

PRÉSENTATION GÉNÉRALE DE L'ITANIUM

1 L'architecture Itanium dans les grandes lignes

a. CISC, RISC, VLIW et EPIC

L'architecture x86 (également connue sous le nom de IA-32) est de type CISC[2]. Les instructions sont incohérentes, de longueur variable, et peuvent opérer sur les registres ou sur la mémoire. L'ISA (jeu d'instructions) est le résultat d'une suite d'ajouts d'instructions, ainsi que des multiples extensions (MMX, SSE 1,2,3,4,5...). Ceci explique le succès commercial de cette famille de processeurs.

À l'opposé, la famille des RISC[3] s'efforce de garder un jeu d'instructions cohérent, orthogonal, n'opérant que sur les registres. Traditionnellement, les RISC représentent un segment limité du marché : serveurs haut de gamme, comme le PA-RISC de HP par exemple.

Les VLIW sont des mots d'instruction comprenant plusieurs sous-instructions. Ces sous-instructions sont exécutées en parallèle : le parallélisme est donc contenu dans les instructions. Cependant, cette famille résiste mal aux évolutions des processeurs, et nécessite une recompilation du code pour tirer parti des nouvelles capacités des processeurs.

Bien que les VLIW permettent de se débarasser du silicium nécessaire pour microcoder l'ordonnancement des instructions, comme dans les architectures out of order traditionnelles, ils n'ont jamais vraiment gagné en popularité.

L'Itanium se propose de résoudre les problèmes inhérents aux VLIW tout en gardant l'idée du parallélisme explicité par le compilateur : c'est l'EPIC. De manière intuitive, les instructions ne sont pas groupées par 3, 4, ou plus, de manière séquentielle comme avec le VLIW : les dépendances entre instructions sont indiquées sous formes de blocs d'instructions. Le processeur se charge ensuite d'en exécuter le plus possible en même temps. Ainsi, si jamais la génération suivante de processeurs permet d'exécuter trois instructions virgule flottante au lieu de deux simultanément, il n'est pas nécessaire de recompiler comme avec un VLIW.

b. Lenteur de la mémoire

Si les processeurs suivent la loi de Moore depuis bon nombre d'années, la mémoire, quant à elle, est toujours à la traîne. La toute dernière DDR3 (qui n'est même pas largement adoptée !) culmine, dans ses versions les plus rapides, à 800Mhz... ce qui est la moitié de la fréquence des derniers Itanium 2 (1.6Ghz). La mémoire est donc bien souvent le facteur limitant. Des systèmes de cache L1, L2, puis L3 ont été successivement mis en place, mais pour peu que les données ne soient pas mises en cache, le processeur peut « attendre » plusieurs cycles que les

[2] Complex Instruction Set Computing, où « Complex » est un euphémisme

[3] Reduced Instruction Set Computing

données arrivent ou soient mises en mémoire, avant de continuer. En effet, les instructions de chargement ou de sauvegarde marquent le début ou la fin d'un bloc, et leur succès conditionne la suite de l'exécution.

L'architecture Itanium incorpore différentes fonctionnalités visant à limiter au mieux les effets néfastes de la lenteur de la mémoire :

- 128 registres permettant d'exécuter la plupart des algorithmes sans avoir besoin de stocker quoi que ce soit en mémoire (voir à ce propos le schéma page 1) ;
- instructions de *prefetching*, qui permettent de suggérer au processeur de commencer à rapatrier de la mémoire certaines données, en prévision des instructions futures ;
- *speculative load* : dans le cas d'un branchement, commencer à exécuter les instructions de chaque branchement, et si l'une de ces instructions est un `load`, les exceptions sont supprimées et ne surgissent que quand le branchement est effectif (voir paragraphe suivant).

c. Prédiction, branchement...

Des instructions peuvent être exécutées tout en étant liées à un *branch register* : au moment de la compilation, le résultat d'un test peut être inconnu. Le compilateur peut cependant paralléliser en exécutant simultanément les deux branches. Chacun des branches est liée au résultat du test, et au moment de l'exécution, l'une des deux branches devient une série de `nop`, permettant plus de parallélisme.

Comme expliqué ci-dessus, si une instruction `load` dépend du résultat d'un test, mais que l'on souhaite néanmoins la lancer en avance pour pallier la lenteur de la mémoire, il est possible de supprimer les exceptions : ainsi, si l'instruction génère une erreur, mais ne sera jamais exécutée (justement parce que le test `if (ptr != NULL)` a échoué par exemple), on perd juste un chargement. Si en revanche l'instruction est utilisée, les exceptions sont levées au moment où l'on connaît le résultat du branchement.

2 À propos de la partie virgule flottante

Ce qui suit ainsi que la partie précédente ne constitue qu'un aperçu extrêmement rapide des différents concepts incorporés dans l'architecture Itanium. Pour avoir une vision plus détaillée, il est possible de se référer à [CHT02] et [Mar00].

a. Unités FPU

L'Itanium comporte deux unités virgule flottante (FPU), permettant d'exécuter deux opérations en parallèle. 128 registres virgule flottante sont mis à disposition de l'utilisateur afin, et c'est la philosophie, de pouvoir effectuer tous les algorithmes dans les registres.

Ces registres sont tournants (*rotating registers*) : dans le cas d'une boucle, ou de l'exécution concurrente de plusieurs procédures, il est possible d'effectuer une demande d'« allocation de registres » : le processeur se charge alors d'effectuer une conversion entre les registres que voit le code et les registres réels. Cela permet de lancer plusieurs boucles simultanément, opérant sur des registres différents, de manière à maximiser le parallélisme.

Les registres 2 à 32 sont fixes, tandis que les registres 33 à 127 peuvent être utilisés dans le cadre de ce mécanisme. C'est le RSE (Register Stack Engine) qui se charge de gérer ces mécanismes (voir schéma).

À titre de comparaison, le Power6 possède deux BFU (Binary Floating-Point Unit). Chacune peut traiter deux threads en parallèle, et chacune possède sa propre copie des registres flottants (voir [TSSK07]).

b. Types de données

Pour pouvoir exécuter tous les algorithmes dans les registres, il faut pouvoir disposer d'une bonne précision pour les calculs intermédiaires. Le format interne de registre virgule flottante possède 82 bits : 64 bits de mantisse, 17 pour l'exposant, et 1 bit de signe. Il est à remarquer que le format interne n'utilise pas la convention du premier bit implicite : on ne suppose pas que la mantisse commence par un 1. À cause de cette non-convention, un même nombre peut avoir plusieurs représentations.

L'Itanium implémente naturellement les types de données IEEE-754 simple et double précision. Ceci porte le nombre de formats à 3.

Cependant, l'Itanium possède un mode de compatibilité IA-32 : il implémente donc aussi le double étendu cher à Intel. Quatre formats.

Le lecteur averti se souviendra de l'existence d'un format supplémentaire pour les registres de la FPU 8087 : dans cette famille de coprocesseurs (et pour tous les x86 de manière générale), les calculs qui étaient effectués au sein de la FPU en simple et double précision bénéficiaient de deux bits d'exposant supplémentaires, toujours dans l'esprit de garder une précision supplémentaire pour les calculs intermédiaires. Toujours pour la prise en charge de l'IA-32, l'Itanium implémente ces formats FPU[4].

Appliquons cet esprit de précision supplémentaire mais pour le mode 64 bits, et l'on obtient des types simple et double précision, mais avec 4 bits d'exposant supplémentaire, lorsqu'ils sont stockés dans les registres. Huit formats au total.

Ces différents formats sont à cauchemar à gérer. Ils ont tous une bonne (ou presque) intention justifiant leur existence, mais les inconvénients sont nombreux :

- pas de contrôle sur le type de format utilisé : je code mon algorithme en C, comment être certain que mes données intermédiaires restent dans les registres de la FPU ?
- problèmes de conversion subtils lorsque le contenu des registres est remis dans la mémoire : certains formats sont stockés tels quels, d'autres sont mis au format IEEE...

Le Power6, pour sa part, ne possède qu'un seul format pour les registres flottants, de 73 bits. 64 sont consacrés aux données, 8 sont des bits de parité et 1 est implicite. En ne traitant que les nombres sur 64 bits (format double précision IEEE), le Power assure une simplicité certaine.

[4] Appelés dans l'article « stack IA-32 » car les registres de la FPU ne se manipulent pas comme les `eax`, `ebx`, `ecx`, `edx`, `efp`, `esp` généraux mais sous forme de stack.

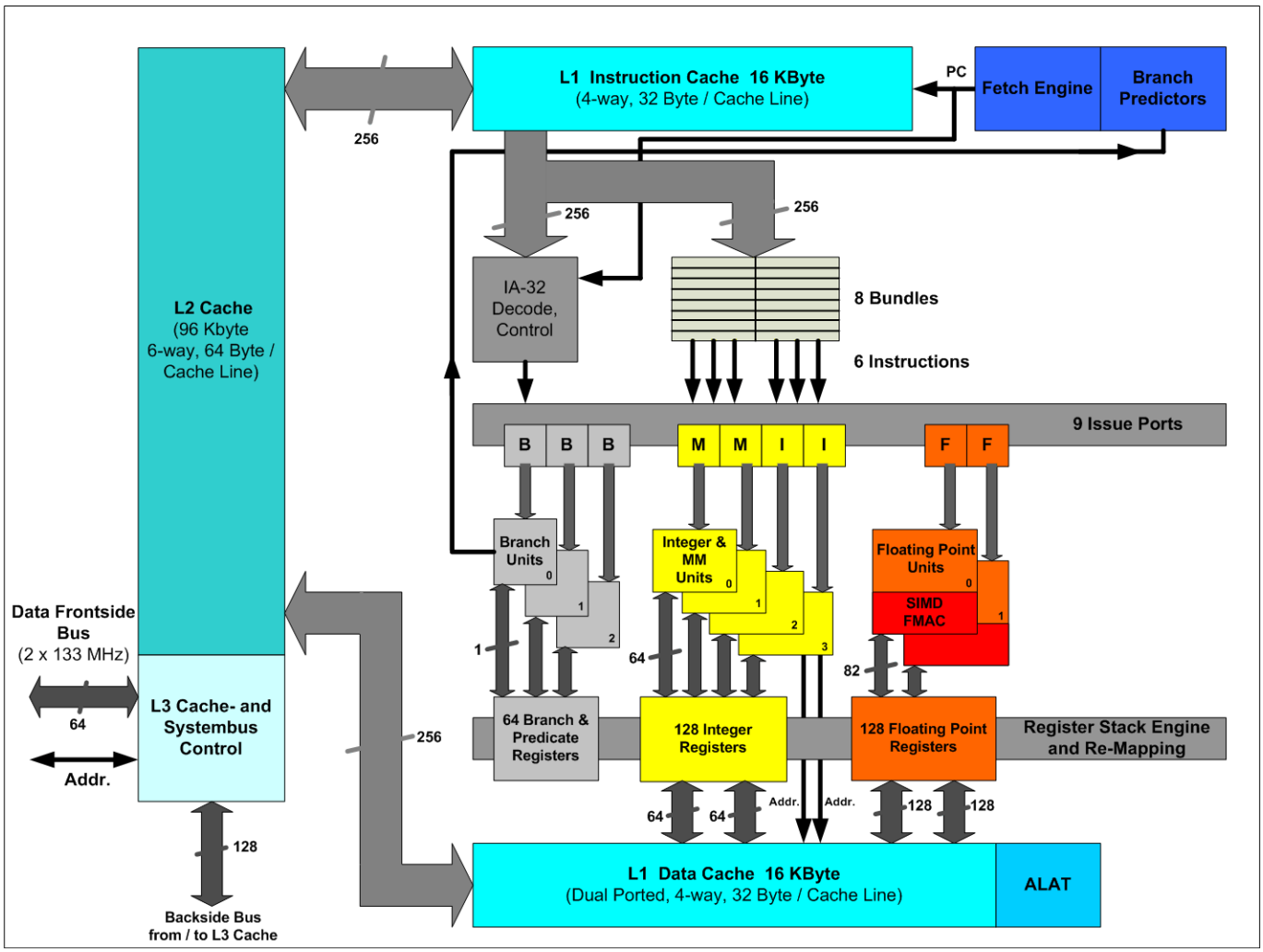

FIG. 1 – Architecture de l'Itanium

Précisons à sa décharge qu'il ne porte cependant pas l'héritage du passé inhérent aux processeurs Intel.

3 L'Itanium, une fausse bonne idée ?

L'Itanium a « essuyé les plâtres » : partant de bonnes idées, la mise en œuvre n'a pas suivi. Aucun processeur EPIC n'avait jamais été conçu auparavant. Les retards se sont accumulés, et à vouloir trop bien faire, des lourdeurs ont été incorporées.

Du point de vue marketing, l'Itanium n'a attaqué qu'un segment extrêmement limité du marché. L'engouement qui a suivi son annonce s'est rapidement volatilisé au vu des premiers retards, et après la première déception, il a été difficile de regagner les cœurs des entreprises.

Il aurait peut-être été possible d'intéresser un autre segment du marché avec l'Itanium. Les joueurs ont un profil auquel l'Itanium aurait pu s'adapter. Les jeux effectuent de nombreux calculs graphiques souvent parallélisables. Les opérations de division et de racine carrée, souvent utilisée dans les calculs graphiques, se seraient largement accomodées des rapides `frcpa` et `fsqrta`. La philosophie d'ensemble, en s'adaptant un petit peu, aurait pu viser un segment du marché plus large et peut-être amener l'Itanium vers une carrière plus prometteuse. Cela n'a malheureusement pas été le cas.

L'Itanium a cependant fortement influencé les nouvelles générations d'architecture. Citons par exemple le Power ([TSSK07, LSF+07]) qui reprend dans sa version 6 nombre de concepts de l'Itanium :

- le FMA,
- tables d'approximation pour la division et la racine carrée correctes à 8 bits pour la division (voir ci-dessous, c'est exactement ce que fait l'Itanium) : les ingénieurs de chez IBM préconisent l'utilisation d'une solution logicielle pour ces deux opérations,
- mécanismes de branchements et de *prefetching* avancés…

L'Itanium est donc plus une preuve qu'il est possible d'intégrer des concepts puissants dans une « vraie » architecture qu'un réel succès. Cependant, l'Itanium s'améliore peu à peu, et ses performances sont devenues acceptables. Il ne connaîtra cependant probablement jamais de réel succès.

PARTIE III
DIVISION SUR L'ITANIUM

Nous allons pour illustrer un peu plus les concepts de l'Itanium nous intéressons à la division. Non pas du point de vue de son élégance théorique, mais du point de vue pratique : puisque l'Itanium incorpore une division novatrice et performante, le code devrait en bénéficier directement…

La division est extrêmement représentatrice de la réalité « Itanesque » en général. Elle constitue l'essentiel de notre exposé.

1 Philosophie

La division sur les processeurs Itanium rompt avec les traditions en vigueur alors dans le monde des microprocesseurs[5] : en fait, il n'y a pas de division dans l'Itanium, seulement une instruction `frcpa`, pour floating-point reciprocal approximation :

```
frcpa q,p = n, d
```

Cette instruction se comporte de la façon suivante, nous indique [Mar00] : si `n` et `d` sont des valeurs acceptables (pas d'infini ni de NaN, pas de division par 0), alors le registre de prédiction `p` est mis à vrai et `q` contient une approximation de $1/d$ avec au moins 8 bits valides. Si la division n'est pas valide, alors l'issue du calcul se détermine aisément grâce au standard IEEE-754 : `p` est mis à faux et `q` contient n/d. Par ailleurs, $8 * log(2)/log(10) \approx 2$, on a donc un résultat bon à 2 chiffres significatifs au moins en décimal (pour se représenter la qualité de l'approximation).

Intel fournit un PDF d'une centaine de pages où des algorithmes recommandés sont présentés : divisions simple, double, et double étendue. Différentes versions sont proposées : optimisées pour le débit, optimisées pour la latence. Ce manuel à l'usage des concepteurs de compilateurs est plus ou moins respecté par eux, comme nous le verrons.

2 Que nous donne le compilateur ?

Considérons le programme C suivant.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
#if 0
  double a,b,c;
  scanf("%lf", &a);
  scanf("%lf", &b);
#else
  float a,b,c;
  scanf("%f", &a);
  scanf("%f", &b);
```

[5] Sur x86 par exemple, le programme ci-dessous se traduit, compilé par GCC avec ou sans `-O3`, par une instruction `fdivs` ou `fdivrs` qui sont en fait les divisions standard en virgule flottante, le `r` échangeant simplement l'ordre des opérandes

```
#endif
  c = a/b;
  printf("a/b=%1.20f\n", c);

  return EXIT_SUCCESS;
}
```

Ce programme sera toujours lancé sur l'entrée $a = 355$ et $b = 113$ au cours de l'exposé.

Les différents compilateurs utilisés (ICC, GCC, OpenCC) utilisent des approches différentes. Tous cependant commencent naturellement par une instruction `frcpa`. Celle-ci renvoie[6], pour 113, la valeur $0.008842\cdots$ pour une valeur exacte de $\frac{1}{113} = 0.00884955\cdots$[7].

a. GCC

GCC nous génère l'assembleur suivant :

```
     frcpa.s0   f8, p6 = f7, f6 ;;
(p6) fnma.s1    f9  = f6, f8, f1 ;;
(p6) fma.s1     f9  = f9, f9, f9 ;;
(p6) fma.s1     f8  = f9, f8, f8 ;;
(p6) fmpy.s.s1  f9  = f7, f8 ;;
(p6) fnma.s1    f10 = f6, f9, f7 ;;
(p6) fma.s      f8  = f10, f8, f9 ;;
     mov        f6  = f8 ;;
```

`f6` contient `f7/f6` à l'issue de l'algorithme. Deux choses sont à constater :

- c'est très précisément l'algorithme décrit dans l'article pour la division en simple précision optimisée pour le débit ;
- il aurait été plus judicieux d'utiliser une division optimisée latence.

Les suffixes `.s` accolés au nom des instructions indiquent un calcul en simple précision. Toute la multiplication effective dépend du registre de prédiction `p6`, ce qui permet d'avoir un résultat correct même lorsque la division n'est pas possible.

b. ICC

ICC génère l'assembleur suivant :

```
      frcpa     f32,p8=f6,f7
(p8)  fma.s1    f33=f6,f32,f0
(p8)  fnma.s1   f34=f32,f7,f1
(p8)  fma.s1    f35=f33,f34,f33
(p8)  fma.s1    f36=f34,f34,f0
(p8)  fma.s1    f37=f35,f36,f35
(p8)  fma.s1    f38=f36,f36,f0
(p8)  fma.d.s1  f39=f37,f38,f37
(p8)  fma.s     f32=f39,f1,f0
      mov       f40=f32
```

Le suffixe `.d` représente une opération en double précision.

Après décodage de l'algorithme, on obtient les opérations suivantes, pour la division de a par b.

$y = \text{frcpa}(a, b)$ $(y \approx 1/b)$

$q = ay$	$e = 1 - by$
$y_1 = qe + q$	$e_1 = e \times e$
$y_2 = y_1 e_1 + y_1$	$e_2 = e_1 \times e_1$
$y_3 = y_2 e_2 + y_2$	

Le lecteur rôdé aura reconnu des itérations de Markstein. Il est intéressant de remarquer que cette fois-ci, c'est l'algorithme optimisé pour la latence qui a été utilisé (c'est donc plus approprié). C'est l'algorithme proposé par Intel dans son PDF mais pas l'algorithme proposé par Markstein dans [Mar00].

La facilité d'utilisation de ces itérations réside dans le fait que les calculs sont effectués dans des registres avec 80 bits de précision (registres IA-64) : il n'y a pas de propagation d'erreur significative. Nous ne recopierons pas la preuve de correction de ces deux algorithmes, que l'on trouvera dans [Mar00].

c. Que retenir ?

GCC, par défaut, met des `;;` entre les paquets d'instructions, sans expliciter les *bundles* (voir [CHT02]). En revanche, dans ICC, les instructions sont groupées dans des *bundles* explicites de type MFI[8] où seule une instruction de type F (float) est présente, les instructions M et I étant de simples `nop`.

En passant un argument `-O3`, la situation change un peu avec GCC : les *bundles* sont explicites, et la division est une suite de *bundles* de type MMF avec deux `nop` et une instruction F. Cependant, l'algorithme n'est pas meilleur dans GCC…

Il n'existe par ailleurs aucun support dans le langage pour choisir le type de division que l'on préfère (*high-throughput* ou *low-latency*). Le choix peut se révéler heureux ou malheureux, on ne peut que se remettre au compilateur.

Par ailleurs, on peut imaginer que pour une application non critique, deux décimales suffisent pour calculer une division : on touche ici à une question fondamentale. Comment contrôler les possibilités offertes par l'Itanium ? L'Itanium s'est énormément reposé sur le compilateur mais il apparaît que les compilateurs n'ont absolument pas suivi. Comme nous allons le voir dans la partie suivante, les lacunes de GCC étaient jusqu'à récemment énormes concernant la prise en charge des particularités de l'Itanium, et ce n'est que depuis récemment que ces lacunes ont été partiellement comblées.

3 Divisions en rafale

Regardons ce qu'il est possible de faire en lançant des divisions en rafale à partir de nombres aléatoires. Voici le programme utilisé pour les tests :

```
static int g_nmax = 512244;
```

[6] Valeur trouvée en éditant l'assembleur généré (option `-S`) pour ne garder que le résultat du `frcpa`

[7] Les valeurs exactes ont été calculées dans cet article avec Maple

[8] Les instructions qui font 41 bits sur l'Itanium sont regroupées par groupes de 3 (à peu-près 128 bits) qui peuvent êtres exécutées de manière parallèle (avec des restrictions sur les dépendances de lecture/écriture entre les blocs). Les blocs sont de différents types (voir [CHT02]), par exemple MII, où M est une instruction de type `load` ou `store`, et I une instruction sur les entiers. Des `;;` marquent la fin d'un bloc, un bloc pouvant contenir plusieurs *bundles*.

```
typedef struct {
  unsigned int W[2];
} _FP64;

inline void dbl_div_max_thr(_FP64*,
  _FP64*, _FP64*);

int main() {

  int i;
  for (i = 0; i < g_nmax; ++i) {
    _FP64 a,b,q;
    a.W[0] = rand();
    a.W[1] = rand();
    b.W[0] = rand();
    b.W[1] = rand();
    dbl_div_max_thr(&a, &b, &q);
  }

  return EXIT_SUCCESS;
}
```

En effectuant une série de divisions à l'aide de l'algorithme `dbl_div_max_thr`, les performances sont les suivantes. Le fichier externe `dbl_div_max_thr.s` contient l'algorithme préconisé par Intel pour la division de deux nombres double précision en virgule flottante, optimisée pour le débit.

	GCC	ICC
`-O0`	1.366s	0.696s
`-O3`	0.641s	0.477s

GCC n'aligne même pas la mémoire pour ces opérations avec `-O0`, ce qui donne des erreurs du type `a.out(15822): unaligned access to 0x60000ffffeaa38dc, ip=0x4000000000000b51`.

Le plus intéressant réside dans la comparaison avec le programme suivant :

```
static int g_nmax = 512244;

int main() {

  int i;
  for (i = 0; i < g_nmax; ++i) {
    double a,b,q;
    a = rand();
    a += rand();
    b = rand();
    b += rand();
    q = a/b;
  }

  return EXIT_SUCCESS;
}
```

A priori, le même nombre d'opérations est effectué dans les deux programmes.

	GCC	ICC
`-O0`	0.544s	0.559s
`-O3`	0.441s	0.435s

GCC se comporte clairement mieux qu'avec le programme précédent. Les différences ne sont pas significatives avec ICC.

Les valeurs exactes ne sont pas importantes, les différences sont trop faibles pour que l'on puisse affirmer quoi que ce soit sur un benchmark aussi peu évolué que celui-ci.

La première information importante à retenir est que l'emploi d'un algorithme dédié en assembleur n'apporte rien de significatif par rapport à la division donnée par le compilateur. Elle ralentit même dans la plupart des cas les calculs.

La seconde information importante est l'absence d'une fonctionnalité pourtant cruciale : la rotation des registres, qui permettrait de lancer en parallèle les calculs de plusieurs boucles, avec des résultats dans des registres différents…

Encore une fois, le parallélisme des instructions n'est aucunement exploité.

4 Conclusion sur la division

Si l'on veut tirer parti des capacités de l'Itanium, on ne peut pas réellement compter sur le compilateur. Des fonctionnalités majeures (rotation des registres) ne sont toujours pas prises en charge par GCC, compilateur majeur s'il en est. On ne peut pas non plus compter sur l'utilisateur. Le langage ne permet en général pas d'être suffisamment précis pour contrôler finement les algorithmes sous-jacents. Il est donc virtuellement impossible d'exploiter au mieux l'Itanium en produisant du bon assembleur.

Il faut donc recourir à des solutions non portables. On peut par exemple utiliser des fonctions définies uniquement avec le compilateur Intel, le compiler avec des options kabbalistiques, et arriver à manipuler l'Itanium comme souhaité. En voici un exemple.

```
#include <stdlib.h>
#include <stdio.h>

#define _SF1 1

int main() {
  __fpreg y0, a, b;

  float af, bf, y0f;
  scanf("%f", &af);
  scanf("%f", &bf);
  a = af;
  b = bf;
  _Asm_frcpa(&y0, a, b, _SF1);
  y0f = y0;
  printf("%f\n", y0f);

  return EXIT_SUCCESS;
}
```

On remarquera la lourdeur du code, le besoin de spécifier lesquelles des variables vont dans les registres flottants, et l'instruction `_Asm_frcpa`. Sachant que tout ceci ne fonctionne qu'avec le compilateur Intel propriétaire.

La solution la plus simple reste encore de coder ses propres algorithmes en assembleur, et de les appeler depuis le code C (en utilisant l'assembleur fourni par Intel dans son PDF). Pour l'avoir testée, c'est ce qu'il y a de plus simple en œuvre, mais la portabilité est mauvaise et les performances ne sont pas au rendez-vous comme nous l'avons vu…

Pour conclure cette partie, indiquons le lien suivant : `http://gcc.gnu.org/ml/gcc/2008-05/msg00229.html`. Il dresse un tableau de la prise en charge des fonctionnalités de l'Itanium dans GCC. On remarquera que la plupart des améliorations sont *récentes* (GCC 4.x) et que les registres tournants ne sont pas pris en charge : trois quarts des registres sont inutilisés…

PARTIE IV
PRISE EN CHARGE DU FMA : DE RÉELS AVANTAGES ?

La division a déjà largement illustré la différence entre la puissance théorique de l'Itanium et l'utilisation qui en est faite dans la vraie vie. Terminons rapidement par une petite étude la prise en charge du FMA.

1 Prise en charge par le compilateur

Le programme de test est dans la lignée des précédents :

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  float a,b,c,d;
  printf("d = a*b+c, a ?\n");
  scanf("%f", &a);
  printf("d = a*b+c, b ?\n");
  scanf("%f", &b);
  printf("d = a*b+c, c ?\n");
  scanf("%f", &c);
  d = a*b + c;
  printf("%f\n", d);

}
```

Voici ce que donnent les différents compilateurs :
- GCC n'utilise pas d'instruction FMA ;
- ICC non plus (!) ;
- OpenCC met plus de temps à compiler mais utilise un FMA.

2 Faire autrement ?

Le compilateur Intel possède une instruction `_Asm_fma` qui possède tous les inconvénients de son homologue pour le `frcpa`.

Les conclusions sont les mêmes que pour la partie précédente, et il n'y a rien à ajouter qui ne soit une copie de ce qui a été dit concernant la division.

Partie V
Conclusion

La traditionnelle conclusion est ici sans surprise. L'Itanium est un bien beau processeur, qui incorpore de superbes idées issues de la recherche. Néanmoins, un segment de marché extrêmement spécifique, des mésaventures lors du développement (trop novateur), une envie de trop bien faire, et surtout, une foi en la puissance du compilateur injustifiée rendent les performances de ce processeur médiocre car utilisé largement, très largement en deça de ses capacités.

Partie VI
Bibliographie

Références

[CHT] M. Cornea, J. Harrison, and P.T.P. Tang. Intel® Itanium® Floating-Point Architecture. *WCAE 2003*.

[CHT02] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.

[LSF+07] HQ Le, WJ Starke, JS Fields, FP O'Connell, DQ Nguyen, BJ Ronchetti, WM Sauer, EM Schwarz, and MT Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[Mar00] Peter Markstein. *IA-64 and Elementary Functions*. Hewlett-Packard professionnal books, 2000.

[TSSK07] Son Dao Trong, Martin Schmookler, Eric. M. Schwarz, and Michael Kroener. P6 binary floating-point unit. *Computer Arithmetic, IEEE Symposium on*, 0:77–86, 2007.