

Architecture (ASR1 et ASR2)

TP N° 8

1 Objectifs :

- Utilisation avancé de l'assembleur
- Observation et rédaction de code assembleur moderne
- Intégration de code assembleur dans un programme C/C++
- Observation du gain de performance lors de l'utilisation d'opérateurs évolués
- Mémoire cache

2 Organisation du TP :

Le travail effectué durant cette séance sera à rendre à votre responsable de TP par email impérativement à la fin de la séance. Le sujet de l'email sera [2012][Architecture][TP8] Nom Prénom .

Le TP comporte plus d'exercices que vous ne pourrez normalement en faire pendant la séance. Ces exercices supplémentaires-vous sont donnés à titre d'entraînement et votre responsable de TP pourra vous les corriger si vous les lui rendez. Attention, corriger vos exercices ne veut pas dire qu'il vous fournira une correction type des exercices ! Il vous indiquera des pistes pour résoudre vos problèmes et pour vous aider à trouver une solution correcte qui fonctionne.

3 Exercice 1 : Intégrer de l'assembleur dans du C/C++ (Obligatoire)

Saisissez le code ci-dessous dans un projet Visual C++ vierge (attention on utilisera Visual Studio 2008 ou Visual Studio 2010 en fonction de la version disponible sur votre machine). Ce code est une illustration de comment intégrer de l'assembleur dans un programme C/C++, il ne compile pas dû à une erreur volontaire.

```
#include <iostream>
using namespace std;
int main(int argc, char ** argv)
{
    int a = 4;
    int b = 3;
    __asm{
        mov ax, a;
        mov bx, b;
        add ax, bx;
        mov a, ax;
    };
    cout << a << " " << b << endl;
}
```

Question : Modifier le code suivant pour qu'il compile et que l'opération $a=a+b$ soit effective.

4 Cours sur les instructions MMX :

Les instructions MMX ne s'utilisent pas avec les registres standards (AX, BX ...) ni avec leur version étendu (EAX, EBX, ...). Elles s'utilisent avec les registres MMX qui s'appellent mm1, mm2 jusqu'à mm6. Il s'agit de registres 64 bits spécifiques aux unités de calcul MMX.

Les instructions MMX dont vous avez besoin pour résoudre cet exercice sont les suivantes.

Architecture (ASR1 et ASR2)

TP N° 8

4.1 L'instruction MOVQ :

Cette instruction copie le contenu d'un registre MMX (8 octets) à l'adresse fournie en argument. Par exemple, avec l'instruction suivante, les 8 octets contenus dans le registre `mm1` vont être mis dans la mémoire à l'adresse contenu dans le registre `eax`.

```
MOVQ [eax], mm1
```

Inversement, cette fonction peut remplir un registre MMX avec le contenu de la mémoire à l'adresse passée en argument comme le montre l'exemple suivant

```
MOVQ mm1, [eax]
```

Pour plus d'information, consulter la documentation sur cette instruction en annexe.

4.2 L'instruction PADDUSB et l'instruction PADDUSW:

Ces instructions est un peu plus compliquées qu'une simple addition entre deux registres MMX. Comme les autres instructions, elles sont de type *PADDUSB Destination, Source*. Leur fonction est d'ajouter les registres *Destination* et *Source* puis le sauvegarder le résultat dans le registre *Destination* en considérant ces registres comme 8 registres 8 bits pour PADDUSB ou 4 registres 16 bits pour PADDUSW.

Pour plus d'information, consulter la documentation sur cette instruction en annexe.

5 Exercice 2 : Traitement d'images (Obligatoire)

L'objectif de cet exercice est de mesurer l'amélioration des performances qu'apporte l'assembleur à un programme, à condition que l'assembleur soit bien utilisé.

Pour illustrer de manière simple le gain en temps de calcul que peut apporter l'assembleur, nous allons comparer 3 fonctions faisant la même chose. La première sera écrite en C, la deuxième en assembleur simple et la dernière en utilisant les « fonctions avancées » de l'assembleur, dans notre cas les unités de calcul MMX. Pour illustrer ces fonctions, nous allons travailler sur une opération qui consiste à éclaircir une image en niveaux de gris. Le choix de cette opération trouve trois justifications.

- Premièrement celle-ci est simple, elle consiste à ajouter une valeur fixe à la valeur de chacun des pixels. Un pixel, dans le cas d'une image noir et blanc classique, est codé sur un octet non signé soit un unsigned char en C/C++. On fera donc attention lors de l'opération d'éclaircissement de ne pas dépasser la valeur 255 pour ne pas avoir de débordement.
- Deuxièmement, il nous sera possible de vérifier visuellement que le résultat des 3 opérations est identique en visionnant les images résultantes des 3 opérations d'éclaircissement.
- Et enfin Troisièmement, si nous travaillons sur de grosses images, nous devrions « sentir » (sans même avoir besoin de mesurer le temps de calcul) que l'opération en assembleur avancé est plus rapide que les deux autres opérations.

Nous allons commencer par vérifier s'il y a réellement un gain de performance.

Copiez sur votre compte le fichier «TP8_VersionDeTest.7z». Exécutez le programme `TP_MMX.exe`, observez et commentez les résultats. Testez la programme avec les images `noir.jpeg`, `gris.jpeg` et `blanc.jpeg` en les renommant `Image.jpeg`.

Maintenant c'est à vous de réaliser ce programme. Pour vous aider, nous avons commencé à écrire un squelette du programme.

Architecture (ASR1 et ASR2)

TP N° 8

Copiez le fichier « TP8_Projet_Visual_Cpp_2008.7z » ou « TP8_Projet_Visual_Cpp_2010.7z » sur votre compte et ouvrez, avec Visual Studio 2008/2010, le projet TPMMX.sln qui se trouve dans le répertoire « Projet Visual C++\TPMMX\ »

Attention, la fonction main de ce programme ne doit pas être modifiée pour le moment.

1. Elle commence par lire une image sur le disque dur, puis elle crée une image vierge de même taille.
2. Ensuite à l'aide d'une fonction fournie par une bibliothèque bien connu dans le monde du traitement d'image ([OpenCV](#)), elle ajoute la valeur 50 à tous les pixels de l'image (fonction `cvAddS` de la bibliothèque OpenCV). En parallèle, elle mesure le temps qu'il a fallu à l'ordinateur pour effectuer cette opération (fonction `cvGetTickCount`).
3. Puis le programme appelle les fonctions que vous aurez à coder : `add_value_i386` (fonction en assembleur simple) et `add_value_MMX` (fonction en assembleur avancé).

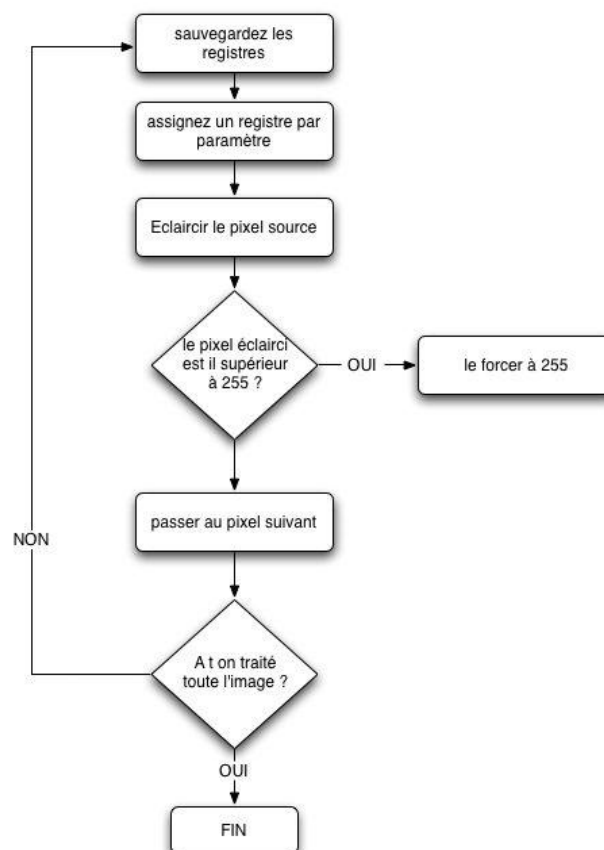
5.1 `add_value_i386` : la fonction en assembleur simple

Cette fonction prend comme argument :

- Un pointeur vers la zone mémoire contenant l'image à éclaircir.
- Un autre pointeur vers la zone mémoire contenant l'image vierge (que vous allez remplir), la valeur (entre 0 et 255) que vous voulez ajouter à tous les pixels de l'image source ainsi que la taille de l'image en pixel.

Compléter le code de la fonction `add_value_i386` en n'utilisant que les instructions du 386 (c'est à dire n'utilisant pas les instructions MMX).

Si vous le souhaitez, vous pouvez vous aider du schéma ci-dessous.



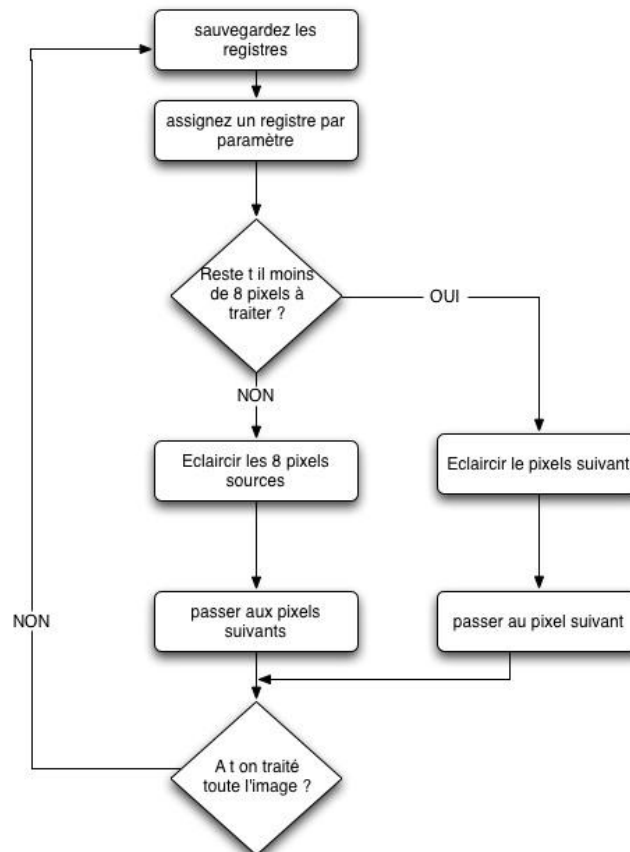
Architecture (ASR1 et ASR2)

TP N° 8

5.2 add_value_MMX : la fonction en assembleur avancé

A l'aide des fonctions MMX définie précédemment, complétez le code de la fonction `add_value_MMX`.

Pour pouvoir utiliser correctement la fonction `PADDUSB` il est nécessaire de remplir un registre MMX avec 8 fois la valeur que l'on désire ajouter à l'image. La petite boucle "for" en début de fonction permet de remplir la mémoire pointée par la variable `value64` de 8 valeurs les unes à la suite des autres. Ainsi, un appel de `MOVQ` permettra de remplir un des registres MMX avec la valeur à ajouter, répétée 8 fois.



Maintenant faite tourner votre programme et constatez les améliorations apportés par le MMX. A titre d'exemple, sur un portable à base de core 2 Duo P9500 à 2.53GHz, les résultats suivants ont été obtenus pour une image de 1000 x 1000 pixels.

Fonction	Temps moyen en microsecondes
<code>cvAddS</code>	1 440
<code>add_value_i386</code>	1 300
<code>add_value_MMX</code>	190

6 Exercice 3 : Traitement d'une image couleur (Conseillé)

Les fonctions que vous venez de coder ont l'inconvénient de ne fonctionner que sur des images en « niveaux de gris ». C'est à dire des images pour lesquelles chaque pixel est codé sur un octet non signé.

Architecture (ASR1 et ASR2)

TP N° 8

Une image en couleur, quant à elle, a plusieurs façons possibles d'être codée. Quand on charge une image couleur avec la fonction OpenCV (`cvLoadImage(const char fname, CV_LOAD_IMAGE_COLOR)`) elle est codée en BGR24. Ce qui signifie que chaque pixel est codé par sa composante Bleu, puis Verte (Green), puis Rouge, la somme des trois composantes occupant une taille de 24 bits (3*8 bits). Le type de codage employé pour décrire une image est appelé « l'espace de couleur » dans lequel est exprimé l'image.

6.1 Eclaircir une image en couleur

Proposez une évolution du programme réalisé au cours de l'exercice précédent permettant d'éclaircir une image couleur chargée avec `cvLoadImage`. Pour vous mettre sur la voie, sachez que l'image couleur sera chargée en BGR24 dans la mémoire, et qu'il faudra trois fois plus d'octets pour décrire l'image que dans le cas d'une image en niveau de gris.

6.2 Une image en couleur ou en niveaux de gris

D'après vous, comment openCV gère-t-il le chargement en mémoire d'une image en niveau de gris si l'on utilise le paramètre `CV_LOAD_IMAGE_COLOR` ?

Que faudrait-il changer dans le programme de l'exercice 3.a pour que celui-ci soit générique, c'est à dire qu'il fonctionne à la fois avec des images couleurs et des images en niveau de gris.

7 Rappel de cours : Cache inclusif ou exclusif

Les caches L1 et L2 fonctionnent en mode :

- **inclusif** : une donnée peut être stockée à la fois dans L1 et L2. L'espace cache total est alors : $\text{Taille}(L_1 + L_2) - \text{Taille}(L_1)$. Cette méthode n'a d'intérêt que si le cache L2 est plus grand que le cache L1. On considère que $\text{Taille}(L_2) \geq 4 \times \text{Taille}(L_1)$ pour que L2 soit efficace.
- **exclusif** : une donnée est stockée soit dans L1, soit dans L2. L'espace cache total est alors : $\text{Taille}(L_2) + \text{Taille}(L_1)$. L'espace cache est ici plus grand que dans la méthode précédente mais il faut faire en sorte que les données ne soient pas stockées dans les 2 caches à la fois.

8 Exercice 4 : Mémoires caches (Obligatoire)

Sachant que le Pentium III (Tualatin 1,4 Ghz) a les caractéristiques suivantes :

	Cache L1-D	Cache L2
Taille du cache	16 ko	256 ko
Associativité	4-way	8-way
Taille d'une ligne	32 octets	32 octets

Donnez le nombre d'ensembles du cache, le nombre de lignes par ensemble et la description de l'étiquette de chaque ligne du cache.

9 Exercice 5 : Mémoires caches (Obligatoire)

Nous allons maintenant observer la mémoire et les différents caches du processeur. Pour cela nous allons utiliser différents outils.

Architecture (ASR1 et ASR2)

TP N° 8

9.1 Lavalys EVEREST

Everest de la société Lavalys est un logiciel de diagnostic (informations très complètes sur les différents éléments de votre PC, mesures des performances...). Ce logiciel ne couvre pas seulement la partie matérielle de votre ordinateur mais analyse également la couche logicielle et notamment le système d'exploitation et ses modules importants comme Direct X et les pilotes réseaux & ODBC.

Ce logiciel n'étant pas gratuit, nous ne pouvons le mettre à votre disposition pour le TP, vous pouvez cependant télécharger une version d'évaluation chez vous, pour tester cela sur votre machine.

Pour le TP nous utiliserons des captures d'écran présenté ci-dessous :

	Read	Write	Copy	Latency
Memory	8266 MB/s	7058 MB/s	5896 MB/s	65.6 ns
L1 Cache	47974 MB/s	47851 MB/s	95700 MB/s	1.0 ns
L2 Cache	13666 MB/s	5445 MB/s	9490 MB/s	7.2 ns
L3 Cache				
CPU Type	DualCore Intel Core 2 E8400 (Wolfdale, LGA775)			
CPU Clock	3000.0 MHz (original: 3000 MHz)			
CPU FSB	333.3 MHz (original: 333 MHz)			
CPU Multiplier	9.0x		CPU Stepping	C0
Memory Bus	400.0 MHz		DRAM:FSB Ratio	6:5
Memory Type	Dual Channel DDR2-800 SDRAM (5-5-5-18 CR2)			

	Read	Write	Copy	Latency
Memory	5975 MB/s	5916 MB/s	5603 MB/s	81.3 ns
L1 Cache	42434 MB/s	42326 MB/s	84632 MB/s	1.1 ns
L2 Cache	18117 MB/s	10303 MB/s	19948 MB/s	7.6 ns
L3 Cache				
CPU Type	Mobile DualCore Intel Core 2 Duo T9400 (Penryn, Socket 479)			
CPU Clock	2653.3 MHz (original: 2533 MHz, overclock: 5%)			
CPU FSB	279.3 MHz (original: 266 MHz, overclock: 5%)			
CPU Multiplier	9.5x		CPU Stepping	C0
Memory Bus	349.1 MHz		DRAM:FSB Ratio	10:8
Memory Type	Dual Channel DDR2-698 SDRAM (5-5-5-15 CR2)			

	Read	Write	Copy	Latency
Memory	8332 MB/s	6755 MB/s	9950 MB/s	45.8 ns
L1 Cache	83328 MB/s	41685 MB/s	43415 MB/s	1.1 ns
L2 Cache	20857 MB/s	16603 MB/s	19118 MB/s	3.5 ns
L3 Cache	8432 MB/s	9545 MB/s	10450 MB/s	9.1 ns
CPU Type	QuadCore AMD Phenom II X4 810 (Deneb, Socket AM2+)			
CPU Clock	2608.8 MHz (original: 2600 MHz)			
CPU FSB	200.7 MHz (original: 200 MHz)			
CPU Multiplier	13x		CPU Stepping	RB-C2
Memory Bus	668.9 MHz		DRAM:FSB Ratio	20:6
Memory Type	Unganged Dual Channel DDR3-1338 SDRAM (7-7-7-20 CR1)			

	Read	Write	Copy	Latency
Memory	8028 MB/s	6579 MB/s	10021 MB/s	54.5 ns
L1 Cache	83055 MB/s	41547 MB/s	55372 MB/s	1.1 ns
L2 Cache	20789 MB/s	16532 MB/s	18837 MB/s	3.6 ns
L3 Cache	8237 MB/s	9542 MB/s	10485 MB/s	7.7 ns
CPU Type	QuadCore AMD Phenom II X4 810 (Deneb, Socket AM2+)			
CPU Clock	2600.2 MHz (original: 2600 MHz)			
CPU FSB	200.0 MHz (original: 200 MHz)			
CPU Multiplier	13x		CPU Stepping	RB-C2
Memory Bus	533.4 MHz		DRAM:FSB Ratio	16:6
Memory Type	Unganged Dual Channel DDR2-1066 SDRAM (5-5-5-15 CR2)			

	Read	Write	Copy	Latency
Memory	16225 MB/s	13427 MB/s	19446 MB/s	34.5 ns
L1 Cache	52787 MB/s	52762 MB/s	105493 MB/s	1.2 ns
L2 Cache	35150 MB/s	32025 MB/s	40755 MB/s	3.1 ns
L3 Cache	25557 MB/s	19676 MB/s	26314 MB/s	3.9 ns
CPU Type	QuadCore Intel Bloomfield (Bloomfield / Gainestown, LGA1366)			
CPU Clock	3300.0 MHz (original: 2933 MHz, overclock: 13%)			
CPU FSB	150.0 MHz			
CPU Multiplier	22x		CPU Stepping	C0/C1
Memory Bus	750.0 MHz		DRAM:FSB Ratio	5:1
Memory Type	Triple Channel DDR3-1500 SDRAM (8-8-8-24)			

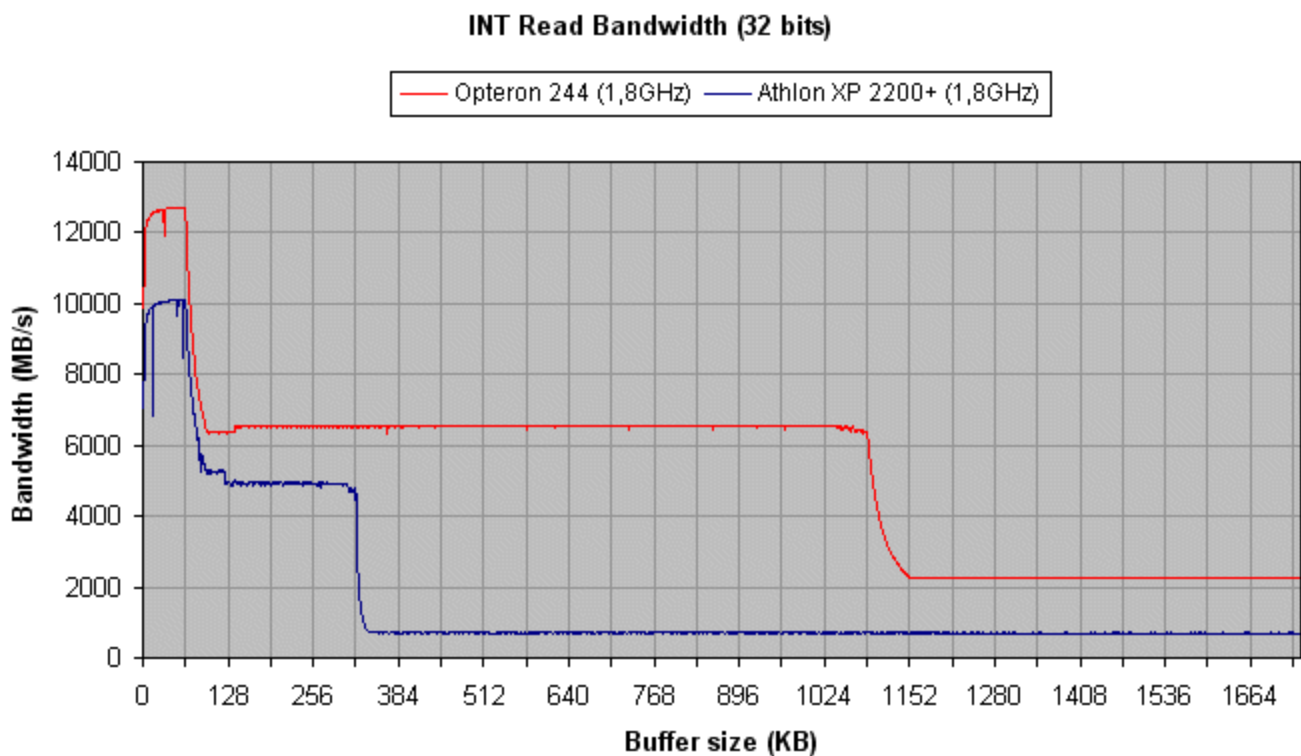
Architecture (ASR1 et ASR2)

TP N° 8

- En examinant les résultats du premier benchmark, que dire des performances entre la mémoire, le cache L2 et la cache L1 ?
- Calculez la bande passante théorique du « Core 2 E8400 », du Phenom II X4 810 et du « Core i7 ». Que dire de l'efficacité en lecture de ces processeurs ?
- En comparant les 2 benchmarks sur le Phenom II X4 810, est-il intéressant d'utiliser une mémoire plus rapide ?
- Calculez la latence (en cycle) du cache L1 et L2 du core i7, du core 2 duo (penryn) ? Intel a-t-il amélioré les caches de ces processeurs ?
- Comparez les caches L1 des processeurs Intel et ceux d'AMD. Qu'en déduisez-vous ?

9.2 Article x86-secret.com

Démarré en 2001, x86-secret.com avait pour but de vous proposer des articles indépendants et techniques sur le matériel PC. Le site étant actuellement fermé, voici quelques captures d'écran qui nous permettront de travailler sur les caches.



Quels sont les tailles (en Ko) des caches L1 et L2 de ces processeurs ? Que pouvons-nous en déduire d'autre concernant les caches de ces processeurs ?

9.3 SiSoftware Sandra

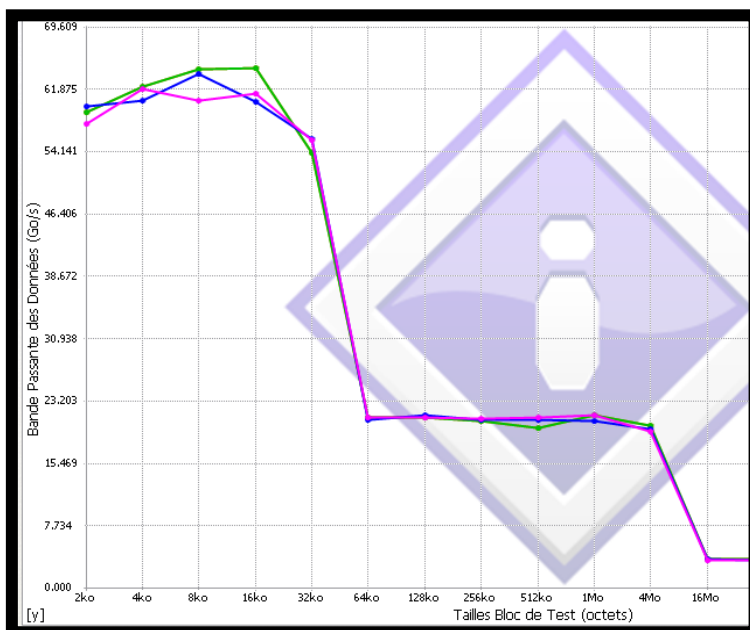
SANDRA est un des logiciels de références depuis plusieurs années, il propose de nombreux outils vous permettant d'analyser la configuration de votre ordinateur dans ses moindres détails : processeur, carte mère, ports, bus, périphériques audio, carte graphique ainsi que des outils (analyse du matériel, des logiciels installés, benchmarks des différents composants). Les outils de benchmark vous permettent également, en plus d'analyser les

Architecture (ASR1 et ASR2)

TP N° 8

performances de votre processeur ou de votre disque dur, de le comparer à plusieurs composants de référence à sélectionner dans une liste.

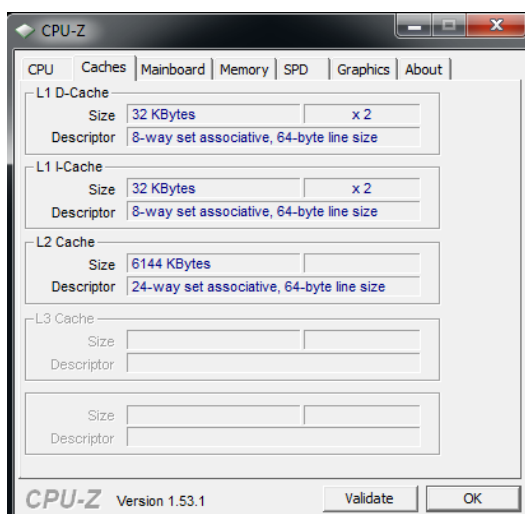
Voici 3 tests effectués sur la même machine. Les résultats vous donnent la vitesse de transfert (en Go/s) en fonction de la taille des blocs de données transférés.



Indiquez les tailles des caches du processeur testé. Que pouvez-vous en déduire sur le fabricant de ce processeur et sur la famille du dit processeur ?

10 Exercice 3 : Mémoires caches (Conseillé)

En utilisant les données de CPU-Z, répondez aux mêmes questions que pour l'exercice 1. A votre avis de quel processeur s'agit-il ?



Architecture (ASR1 et ASR2) TP N° 8

11 Annexe



INSTRUCTION SET REFERENCE

MOVQ—Move 64 Bits

Opcode	Instruction	Description
0F 6F <i>ir</i>	MOVQ <i>mm</i> , <i>mm/m64</i>	Move quadword from <i>mm/m64</i> to <i>mm</i> .
0F 7F <i>ir</i>	MOVQ <i>mm/m64</i> , <i>mm</i>	Move quadword from <i>mm</i> to <i>mm/m64</i> .

Description

This instruction copies quadword from the source operand (second operand) to the destination operand (first operand) (refer to Figure 3-47). A source or destination operand can be either an MMX™ technology register or a memory location; however, data cannot be transferred from one memory location to another memory location. Data can be transferred from one MMX™ technology register to another MMX™ technology register.

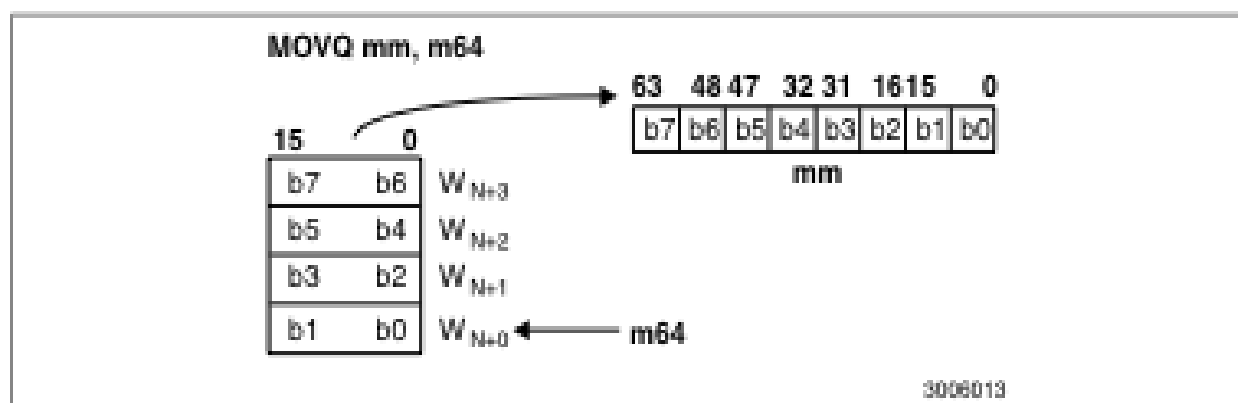


Figure 3-47. Operation of the MOVQ Instructions

Operation

DEST ← SRC;

Flags Affected

None.

Architecture (ASR1 et ASR2) TP N° 8



INSTRUCTION SET REFERENCE

PADDUSB/PADDUSW—Packed Add Unsigned with Saturation

Opcode	Instruction	Description
0F DC <i>ir</i>	PADDUSB <i>mm</i> , <i>mm/m64</i>	Add unsigned packed bytes from <i>mm/m64</i> to unsigned packed bytes in <i>mm</i> and saturate.
0F DD <i>ir</i>	PADDUSW <i>mm</i> , <i>mm/m64</i>	Add unsigned packed words from <i>mm/m64</i> to unsigned packed words in <i>mm</i> and saturate.

Description

These instructions add the individual unsigned data elements (bytes or words) of the packed source operand (second operand) to the individual unsigned data elements of the packed destination operand (first operand) (refer to Figure 3-57). If the result of an individual addition exceeds the range for the specified unsigned data type, the result is saturated. The destination operand must be an MMX™ technology register; the source operand can be either an MMX™ technology register or a quadword memory location.

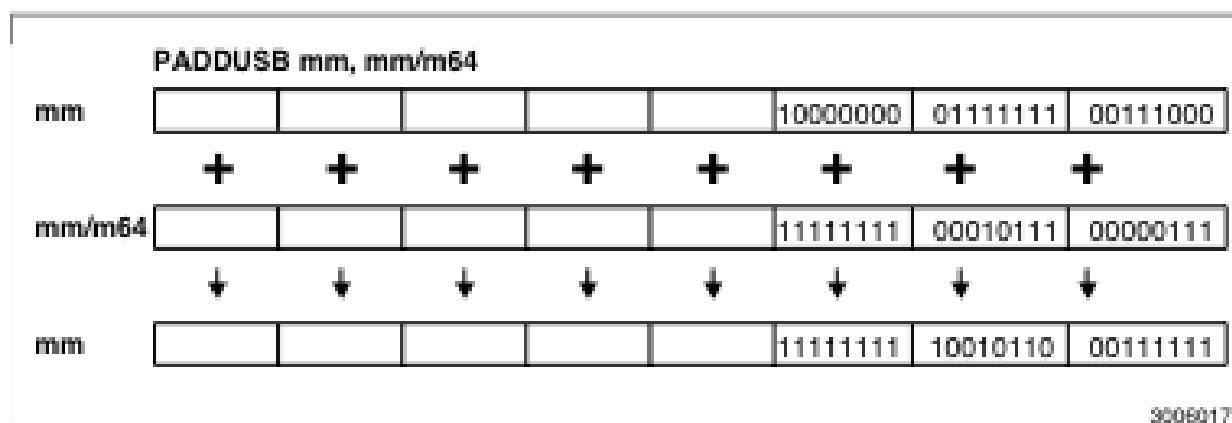


Figure 3-57. Operation of the PADDUSB Instruction

The PADDUSB instruction adds the unsigned bytes of the source operand to the unsigned bytes of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of an unsigned byte (that is, greater than FFH), the saturated unsigned byte value of FFH is written to the destination operand.

The PADDUSW instruction adds the unsigned words of the source operand to the unsigned words of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of an unsigned word (that is, greater than FFFFH), the saturated unsigned word value of FFFFH is written to the destination operand.