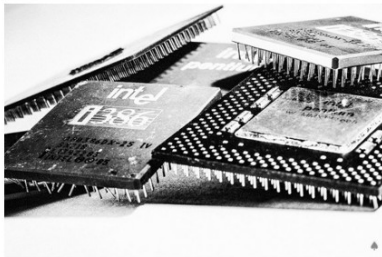
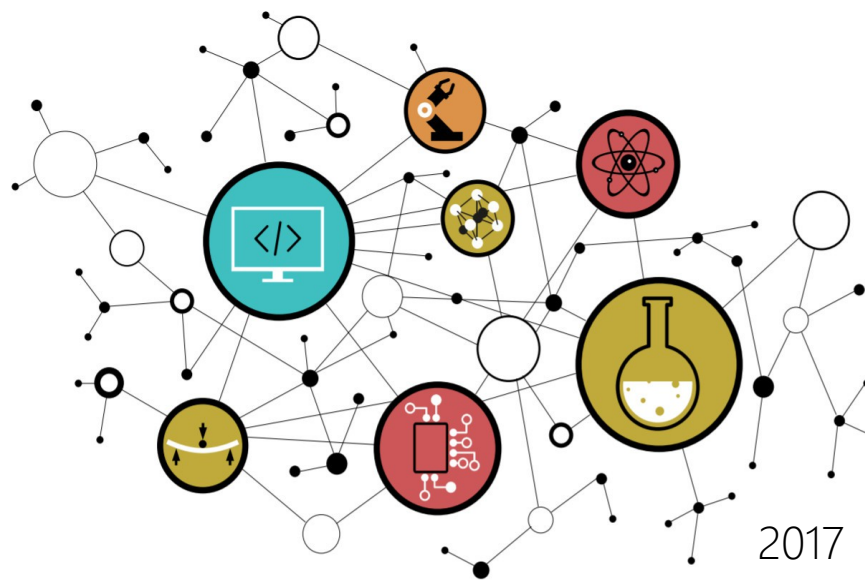
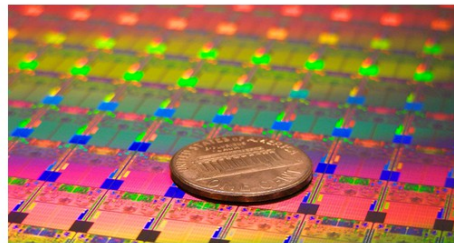
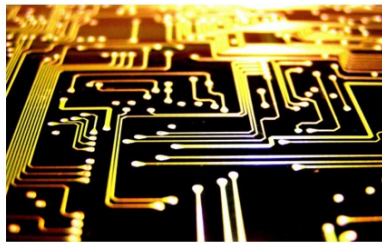


# ARCHITECTURES ET TECHNOLOGIES DES ORDINATEURS

---



GNU/Linux



## CONTACTS



### Établissement

ENSICAEN  
6 boulevard Maréchal Juin  
CS 45053  
14050 CAEN cedex 04

### Référent

hugo descoubes  
[hugo.descoubes@ensicaen.fr](mailto:hugo.descoubes@ensicaen.fr)  
+33 (0)2 31 45 27 61  
+33 (0)6 22 07 21 51  
 <https://fr.linkedin.com/in/hugo-descoubes-823a4268>

## RESSOURCES



Les différentes ressources numériques sont accessibles sur la plateforme pédagogique de l'ENSI CAEN (aucune authentification requise)

- ▶ Lien direct : <https://foad.ensicaen.fr/course/view.php?id=117>
- ▶ Lien indirect : <http://foad.ensicaen.fr/>
  - ▶ Formation Classique
  - ▶ Spécialité Électronique et Physique Appliquée
  - ▶ 2<sup>ième</sup> année
  - ▶ Processeurs Spécialisés
  - ▶ download
  - ▶ 2018
  - ▶ sate
  - ▶ *Ne pas oublier de s'inscrire au cours avant tout dépôt !*

## SEQUENCEMENT

Supports vidéos



[https://www.canal-u.tv/producteurs/centre\\_d\\_enseignement\\_multimedia\\_universitaire\\_c\\_e\\_m\\_u/ensicaen/architecture\\_et\\_technologie\\_des\\_ordinateurs](https://www.canal-u.tv/producteurs/centre_d_enseignement_multimedia_universitaire_c_e_m_u/ensicaen/architecture_et_technologie_des_ordinateurs)

Supports numériques

- ***gpp-cm-1-hétérogénéité*** : nouveau (sauf MCU)
- ***gpp-cm-2-motherboard*** : nouveau
- ***gpp-cm-3-toolchain*** : rappels + nouveau (p16 à p26)
- ***gpp-cm-4-assembly langage*** : rappels + nouveau (p11 à 31)
- ***gpp-cm-5-cpu*** : rappels + nouveau (p22 à p53)
- ***gpp-cm-6-sandy bridge*** : nouveau
- ***gpp-cm-7-memory*** : nouveau

## EVALUATION



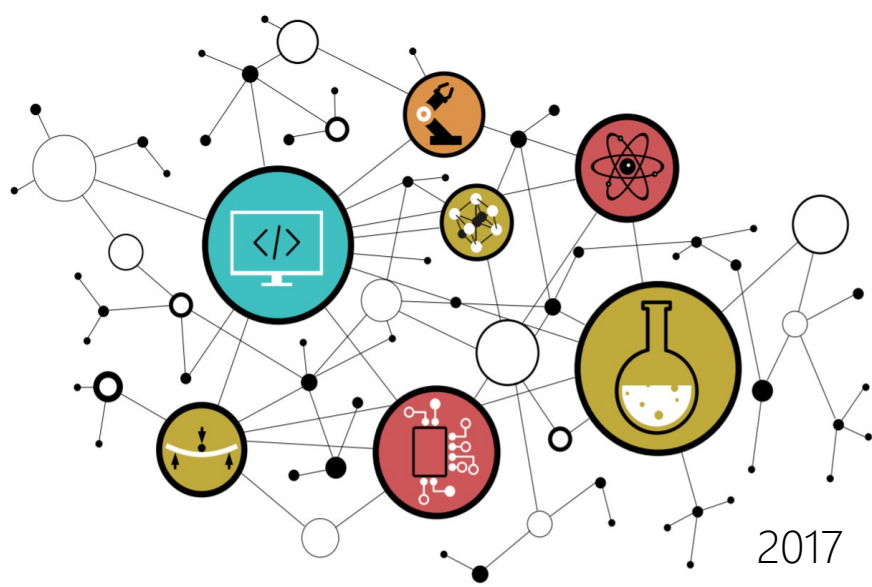
L'évaluation sur table (1h30) portera sur les séances de Cours Magistral (potentiellement sur tout point présent dans les supports ou présenté à l'oral) ainsi que sur la trame de Travaux Pratiques. Tous documents autorisés (polycopié, notes de cours, notes de TP, sauf la copie du voisin, pas d'appel à un ami, etc). L'évaluation sera découpée en deux parties indépendantes :

- **SAVOIR – 8pts** : Questions de culture générale pouvant traiter sur tout point abordé en séance de cours présentiel ou présent dans le support de travail. *Connaissances fondamentales et culture scientifique de l'ingénieur électronicien.*
- **ANALYSER – 12pts** : Exercice de traduction d'un programme assembleur vers un programme C et analyse des mécanismes de gestion de la pile conjointement réalisés par le système d'exploitation, la chaîne de compilation et le processeur. *Comprendre et maîtriser le travail d'un processeur numérique, d'une chaîne de compilation et des mécanismes de gestion de la mémoire. Adaptabilité de l'élève ingénieur.*

Il est possible de s'entraîner très simplement concernant l'exercice d'analyse de code. Éditer un programme C simple (1 à 2 appels de fonctions imbriqués, pas plus de 2 paramètres par fonction). Les fonctions réaliseront des traitements élémentaires. Générer le fichier assembleur 64bits x64 correspondant à votre programme et le fournir à un camarade de promotion sans lui donner le programme C équivalent. Que votre camarade réalise le même travail. En partant du fichier assembleur, essayer de proposer un programme C pouvant générer le même assembleur (plusieurs solutions possibles). De même, proposer le contenu exhaustif de la pile lorsque le pointeur de sommet de pile (SP) se trouve au plus haut sur la pile. Confronter par la suite les solutions proposées. Ne pas hésiter à répéter l'exercice avec une difficulté incrémentale (sans appel de fonction, puis 1 appel de fonction, etc).

# TRAVAUX PRATIQUES

---



# SOMMAIRE

## PREAMBULE

## 1. COMPILATION ET EDITION DES LIENS

- 1.1. Préprocesseur
- 1.2. Analyse et génération de code natif
- 1.3. Assembleur
- 1.4. Éditeur de liens
- 1.5. Exécutable minimal
- 1.6. Linker script minimal

## 2. ALLOCATIONS AUTOMATIQUES ET GESTION DE LA PILE

- 2.1. Fonction main
- 2.2. Variables locales non initialisées
- 2.3. Variables locales initialisées
- 2.4. Appel de fonction
- 2.5. Limites de la pile

## 3. ALLOCATIONS STATIQUES ET FICHIER ELF

- 3.1. Variables globales
- 3.2. Variables locales statiques
- 3.3. Chaînes de caractères

## 4. ALLOCATIONS DYNAMIQUES ET GESTION DU TAS

- 4.1. Gestion du tas
- 4.2. Limites du tas

## 5. EXCEPTIONS MATERIELLES ET SIGNAUX UNIX

- 5.1. Lecture seule
- 5.2. Défaut d'alignement
- 5.3. Division par zéro
- 5.4. Debug
- 5.5. Pointeur nul
- 5.6. Signal UNIX

## 6. UNITE DE PAGINATION

# SEQUENCEMENT

- **Travail préparatoire :** lire le préambule et réaliser l'exercice 1. Un rapport de synthèse de 3 pages sera à déposer sur la plateforme d'enseignement avant l'arrivée en première séance
- **Séance n°1 :** exercice 2 jusqu'au 2.3 inclus, le reste à la maison et préparer vos questions pour la séance suivante
- **Séance n°2 :** jusqu'au 2.4 inclu, le reste à la maison et préparer vos questions pour la séance suivante
- **Séance n°3 :** jusqu'au 3.2 inclu, le reste à la maison et préparer vos questions pour la séance suivante
- **Séance n°4 :** jusqu'au 4.2 inclu, le reste à la maison et préparer vos questions pour la séance suivante
- **Séance n°5 :** le plus loin possible !



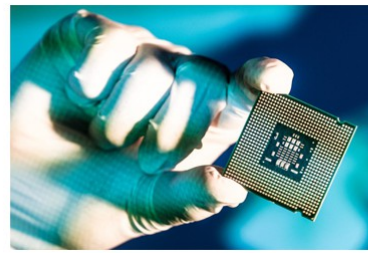
# PREAMBULE

---



### PREAMBULE

Cet enseignement ainsi que la trame de travaux pratiques associée a pour objectif de poser quelques bases dans compréhension des architectures matérielles actuelles réalisant l'exécution de programme quel qu'en soit le langage de programmation à la source. L'idée étant de comprendre plus finement les processus d'exécution de code et de gestion des ressources de stockage réalisés conjointement par l'architecture processeur, le système d'exploitation et les outils de compilation.



Cette trame s'appuie sur les enseignements de logiques, d'outils de développement logiciel et de programmation en langage C. Le langage C est favorisé dans cet enseignement car il reste suffisamment proche du matériel pour comprendre les liens entre langages de haut niveau et langages natifs bas niveaux supportés par les architectures processeurs cibles. Toutes les illustrations s'appuieront sur l'analyse de programmes simples écrits en C. Rappelons que bien que maintenant ancien, le langage C reste encore à notre époque un langage de programmation majoritairement rencontré dans les couches basses des systèmes et sur les architectures à ressources limitées voire spécialisées (noyaux, systèmes d'exploitations, bibliothèques spécialisées, systèmes embarqués ...). De plus, à travers ses différents standards, le langage C continu toujours d'évoluer à notre époque (normes C ANSI ou C89 en 1989, ISO C90 en 1990, C99 en 1999 et C11 en 2011).

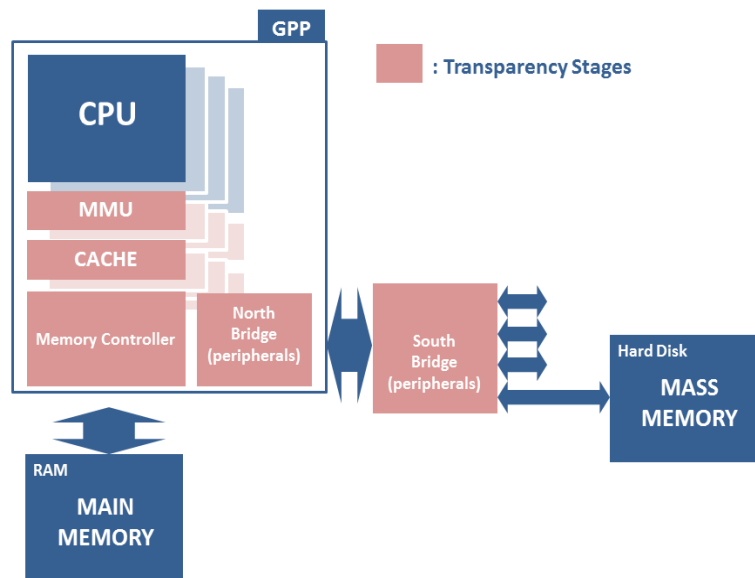
Cette trame de travaux pratiques est avant tout une séquence d'analyse de programmes. Nous nous intéresserons aux impacts de nos développements sur la qualité et le fonctionnement du code généré. L'objectif opérationnel étant d'élargir la capacité à durcir un programme quelque soit l'architecture cible voire à l'optimiser en tenant des contraintes imposées par celle-ci. Nous nous attarderons notamment sur les stratégies de gestion mémoire et une bonne compréhension et maîtrise des segments mémoire associés à notre applicatif :

- *Allocations automatiques et gestion de la pile*
- *Allocations statiques et fichier ELF*
- *Allocations dynamiques et gestion du tas*

Nous profiterons de cet enseignement pour découvrir des mots clés, qualificatifs de types et classes de stockage non vues en enseignement d'initiation au langage C. Nous aurons l'occasion d'observer leurs impacts à l'exécution et nous affinerons notre connaissance des outils de développement : *const*, *register*, *restrict*, *volatile*, *inline*, *static* ...

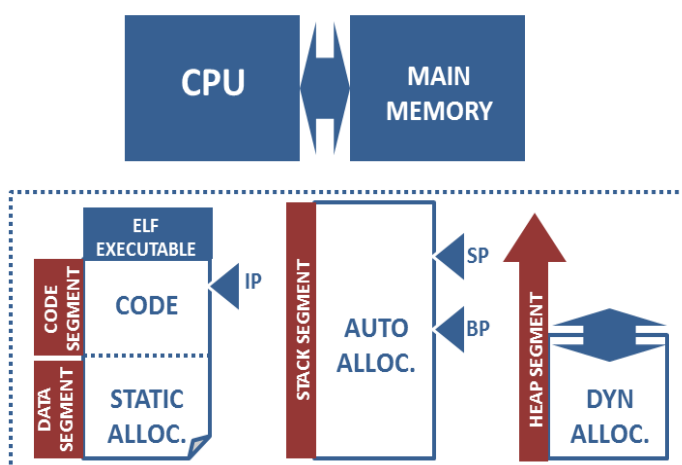
### Architecture matérielle réelle

Rappelons que l'architecture matérielle réelle sur laquelle nous travaillons est complexe. Néanmoins, la plupart des étages en jeu sont présents afin d'accélérer le mécanisme d'exécution ou de virtualiser l'accès aux ressources et sont dans tous les cas transparents pour le développeur.



### Architecture matérielle virtualisée

Une fois chargé en mémoire principale, virtualisé et lancé sur un CPU par le système, le modèle de la machine vu du programme et donc du développeur reste relativement simple. Modèle d'exécution de J. Von Neuman (1946). La compréhension d'une représentation en segments logiques de code et de données de notre applicatif permet alors une réelle expertise dans la réalisation de programmes durcis.



### Archive de travail

Avant de commencer la trame de travaux pratiques, télécharger sur la plateforme moodle de l'ENSICAEN dans la section *download* l'archive nommée *gpp* (arborescence de TP dans le répertoire *gpp/tp/x86x64*). Accès libre à la plateforme en lecture, néanmoins, bien penser à s'inscrire avant tout dépôt éventuel. Effectuer l'extraction sur votre bureau :

```
# tar -xvf gpp.tar -C <dir_path>
```



### Outils de développement

La trame de travaux pratiques sera réalisée sur système GNU/Linux, les différents sources seront donc compilés sous GCC (GNU Collection Compiler, <http://gcc.gnu.org/>). Les chaînes de compilation GCC-like restent majoritairement rencontrées, notamment dans le domaine de l'embarqué. Dans un premier temps, l'assembleur généré durant les phases d'analyse de code devra être compatible avec les architectures IA-32 (assembleur 32bits compatible x86). Puis, nous analyserons de l'assembleur 64bit compatible pour architectures x64.

```
# gcc -Wall -m32 [options] fileName.c      ----> assembleur 32bits x86
# gcc -Wall [options] fileName.c           ----> assembleur 64bits x64
```

Nous vous encourageons d'ailleurs à utiliser vos propres machines pour réaliser ces TP. Si vous souhaitez réaliser la trame sur vos machines personnelles 64bits, il faut installer les packages suivant, offrant notamment le support des bibliothèques système 32bits.

```
# sudo apt-get install gcc-multilib watch
```

### Documentation

Vous aurez probablement à vous documenter sur le fonctionnement de certaines instructions assembleur. Bien respecter la chronologie suivante. S'aider avant tout d'internet, puis rechercher vos réponses dans les documents de référence Intel (répertoire de TP */x86/doc/intel/*, jeu d'instructions dans le volume 2), et enfin demander à l'enseignant référent. Ne pas hésiter à s'aider de vos voisins de table afin de confronter vos idées. Liens officiels vers les ressources Intel ci-dessous :

<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

# COMPILATION

---

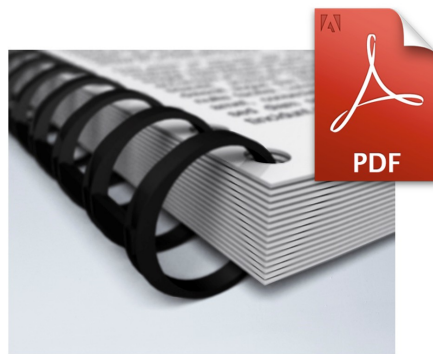
# EDITION DES LIENS

---



# 1. COMPILATION ET EDITION DES LIENS

Cette partie est à réaliser à la maison avant l'arrivée en première séance de TP. Il s'agit d'une trame d'analyse du processus de compilation ainsi que d'édition des liens. Elle sert de support à la séquence de cours traitant du sujet. A la fin de cet exercice (~1h), vous aurez à réaliser et à déposer sur la plate-forme, dans le dépôt associé à votre groupe de TP, un rapport de 2-3 pages expliquant le processus de compilation, d'édition des liens et de chargement d'un programme. S'aider de schémas afin d'illustrer vos propos. Que votre document soit progressif et pédagogique. Concernant le dépôt du fichier, respecter l'extension et le nommage suivant *gpp-homework-2016-<name>-<surname>.pdf* (tout en minuscule)



Se placer dans le répertoire */x86/toolchain/* pour l'application des commandes suivantes. Le fichier *README.txt* contient la séquence d'exécution complète. Analyser le fichier source présent dans le répertoire */x86/toolchain/src/hello.c*. Compiler le projet et analyser la sortie.

```
# gcc -Wall -m32 -I./inc/ ./src/hello.c -o ./build/bin/hello
# ./build/bin/hello
```

## 1.1. Préprocesseur

- Compiler à nouveau le même projet en s'arrêtant à l'étape de préparation du code pour la compilation (preprocessing). Analyser la sortie.

```
# gcc -E -Wall -m32 -I./inc/ ./src/hello.c > ./build/misc/hello.i
```

- Jouer sur la valeur de la macro *MINIMAL* présente dans le fichier d'en-tête */x86/toolchain/inc/hello.h* puis répéter l'opération précédente.
- Rappeler le travail du pré-processeur, et préciser les principales commandes élémentaires appliquées dans le cadre de cet exercice.



### 1.2. Analyse et génération de code natif

- Compiler le programme en partant du code déjà préparé par le préprocesseur et en s'arrêtant à l'étape d'assemblage. Analyser la sortie.

```
# gcc -S -Wall -m32 ./build/misc/hello.i -o ./build/misc/hello.s
```

- Bien observer qu'un fichier assembleur est avant tout un fichier texte. Nous pouvons développer en assembleur. Nous rencontrons ce type de développement à notre époque dans certains cas spécifiques (optimisation de code à l'exécution, diminution de l'empreinte mémoire d'applicatif sur système contraint, bibliothèques spécialisées, hacking ...)

### 1.3. Assembleur

- Assembler le fichier assembleur `/x86/toolchain/build/misc/hello.s` et achever ainsi le processus de compilation. Le fichier source résultant est un fichier binaire au format ELF 32bits. Nous ne pouvons plus utiliser un éditeur de texte afin d'afficher son contenu. Il nous faudra utiliser un utilitaire dédié (`objdump` ou `readelf`). Analyser la sortie. Bien avoir lu la partie du cours traitant des fichiers ELF.

```
# as --32 ./build/misc/hello.s -o ./build/obj/hello.o
# readelf -h ./build/obj/hello.o
# objdump -Sx ./build/obj/hello.o
```

- Le fichier `/x86/toolchain/build/obj/hello.o` est un fichier ELF 32bits. Préciser son type ainsi que l'architecture cible après lecture de son en-tête.

### 1.4. Édition des liens

- Finaliser la génération d'un fichier exécutable par l'édition des liens puis exécuter le programme. Analyser la sortie. Nous étudierons plus en détail l'édition des liens par la suite.

```
# gcc -Wall -m32 ./build/obj/hello.o -o ./build/bin/hello
# readelf -h ./build/bin/hello
# objdump -Sx ./build/bin/hello
# ./build/bin/hello
```

- Le fichier `/x86/toolchain/build/bin/hello` est un fichier ELF 32bits. Préciser son type.

### 1.5. Exécutable minimal

- Nous allons maintenant chercher à obtenir un programme binaire exécutable minimal. Pour ce faire, nous allons jouer avec l'étage d'édition des liens et réaliser celle-ci en appelant directement le linker. Donner la taille actuelle du fichier binaire de sortie alors que notre programme ne fait que quelques dizaines d'octets.

```
# ls -l build/bin
```

La fonction *main* n'est jamais le premier point d'entrée réel d'un programme. Tout programme C débute par un source générique d'amorçage obligatoirement chargé d'appeler la fonction *main*. L'entrée typique sur système GNU/Linux est la fonction ou label *\_start*. Nous verrons par la suite que cette entrée peut être renommée. Ce ou ces fichiers d'amorçage système restent toujours les mêmes utilisés à l'édition des liens et sont pré-compilés pour une architecture cible donnée.



```
1 .section .text
2 .global _start
3
4 _start:
5     mov     $0, %ebp
6     push    %ebp
7     mov     %esp, %ebp
8     call    main
9 _trap:
10    jmp     _trap
11    pop     %ebp
12    ret
```

- Nous allons utiliser un fichier d'amorçage minimal. Ouvrir le fichier assembleur */x86/toolchain/build/startup/crt0.s*, analyser puis expliquer son travail.

- Assembler le nouveau fichier d'amorçage et l'ajouter manuellement à l'édition des liens en appelant directement le linker. Valider l'exécution du programme. A ce stade là, nous ne pourrons plus utiliser de bibliothèques liées dynamiquement. Bien placer la macro MINIMAL à 1. Tous les programmes suivants de la trame de TP n'utilisant aucune fonction dépendante de bibliothèques dynamiques peuvent être compilés jusqu'à édition des liens en utilisant ce fichier d'amorçage.

```
# as --32 ./build/startup/crt0.s -o ./build/obj/crt0.o
# ld -melf_i386 ./build/obj/crt0.o ./build/obj/hello.o -o ./build/bin/hello
# readelf -h ./build/bin/hello
# objdump -Sx ./build/bin/hello
# ls -l build/bin
# ./build/bin/hello
```

- Observer la taille et le contenu du fichier ELF binaire de sortie. Préciser sa nouvelle taille.
- En analysant plus précisément les options passées à gcc, notamment concernant l'édition des liens, en déduire les noms des fichiers binaires d'amorçage utilisés par défaut par le système et préciser leurs noms (préfixés par *crt*).

```
# gcc -v -Wall -m32 ./build/obj/hello.o -o ./build/bin/hello
```

### 1.6. Linker script minimal

- Nous pouvons constater à cette étape que l'empreinte du programme binaire exécutable est optimale. Néanmoins, l'éditeur de liens continu à architecturer le fichier ELF de sortie avec un grand nombre de sections génériques maintenant inutiles. Le linker utilise un fichier texte nommé *linker script* (extension .ld), lui permettant de définir l'ossature du fichier ELF exécutable de sortie. Observer le linker script utilisé par défaut par le linker (riche en informations).

```
# gcc -m32 -Wl,--verbose
```

- Ouvrir maintenant le fichier `/x86/toolchain/build/script/linker_scrip_minimal.ld` et analyser sont contenu. Bien constater qu'il s'agit d'un élagage du linker script utilisé par défaut.

```

1 OUTPUT_FORMAT("elf32-i386")
2 OUTPUT_ARCH(i386)
3 ENTRY(_start)
4
5 SECTIONS
6 {
7     . = SEGMENT_START("text-segment", 0x400000) + SIZEOF_HEADERS;
8     .text :
9     {
10         *(.text)
11     }
12     .rodata :
13     {
14         *(.rodata)
15     }
16     .data :
17     {
18         *(.data)
19     }
20     .bss :
21     {
22         *(.bss)
23     }
24     /DISCARD/ : {
25         *(.comment)
26         *(.note.GNU-stack)
27         *(.eh_frame)
28     }
29 }
```

Ce nouveau script ne propose plus que les sections accessibles au développeur. S'aider de la documentation officielle et de la page wikipedia traitant des fichiers ELF ([https://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format) et <https://sourceware.org/binutils/docs/ld/Scripts.html>). Il s'agit d'ordres et d'une ossature spécifiée à l'éditeur de liens lui permettant de définir la segmentation logique du binaire exécutable de sortie.



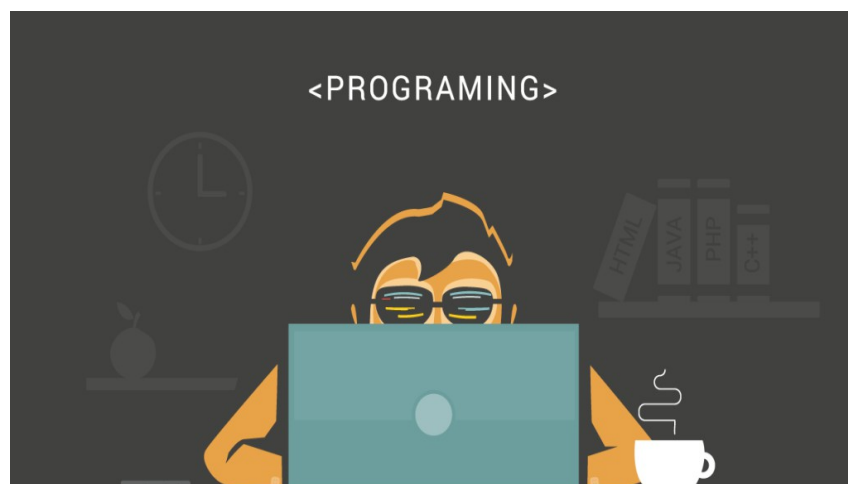
- Forcer le linker à utiliser ce script minimal et observer maintenant le contenu et la taille du binaire de sortie. Valider l'exécution du programme. Préciser la nouvelle taille du binaire exécutable.

```
# ld -melf_i386 -T ./build/script/linker_script_minimal.ld ./build/obj/crt0.o
./build/obj/hello.o -o ./build/bin/hello
# objdump -Sx ./build/bin/hello
# ls -l ./build/bin
# ./build/bin/hello
```

- Dernière rapide optimisation, nous allons nettoyer le fichier ELF (commande *strip*) notamment en concaténant les différentes sections de même nature afin de former des segments de code et de données. Observer maintenant le contenu et la taille du binaire de sortie. Valider l'exécution du programme. Préciser la nouvelle taille du binaire exécutable.

```
# strip ./build/bin/hello
# objdump -Sx ./build/bin/hello
# ls -l ./build/bin
# ./build/bin/hello
```

Voilà, l'exercice est maintenant terminé. Vous venez d'obtenir probablement l'un des programmes C les plus légers que l'on puisse faire tourner sur un système GNU/Linux 32bits et sur architecture x86. Rustique, mais il s'exécute !



# ALLOCATIONS AUTOMATIQUES

## GESTION DE LA PILE

---



## 2. ALLOCATIONS AUTOMATIQUES ET GESTION DE LA PILE

Dans cette nouvelle partie, nous allons nous intéresser aux mécanismes d'allocations mémoire sur la pile (variables locales hors qualifiées de static, paramètres de fonction, adresses de retour de fonction ...). Rappelons que la pile ou stack étudiée est propre au programme en cours d'analyse et a été allouée par le système au chargement du programme. Elle se situe en mémoire principale.

### 2.1. Fonction main

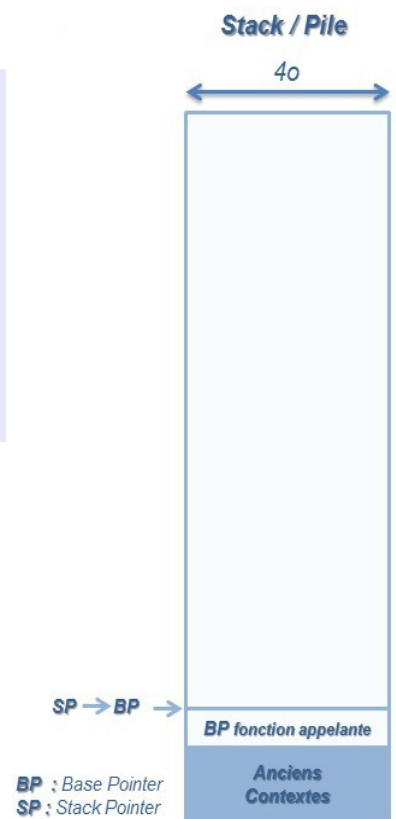
- Éditer puis compiler le programme présent dans le répertoire de TP `/x86x64/stack/main.c` (cf. ci-dessous) en s'arrêtant à la phase d'assemblage. Dans un premier temps, nous analyserons de l'assembleur 32bits x86 (option `-m32`) pour passer par la suite à de l'assembleur 64bits x64.

```
# gcc -Wall -m32 -S main.c
```

- Comme ci-dessous, insérer le code assembleur généré à votre fichier source C sous forme de commentaires puis Interpréter son fonctionnement.

```
int main (void) {
    /* pushl %ebp : sauvegarde le contexte de la fonction appelante
       movl %esp, %ebp : préparation à l'allocation de ressources mémoire
       pour le contexte de la fonction courante ou fonction main */
    return 0 ;
    /* movl $0, %eax : passage par registre de la valeur de retour
       popl %ebp : restaure le contexte de la fonction appelante
       ret : dépile l'adresse de retour de la fonction appelante, la charge dans
       le pointeur d'instruction puis quitte la fonction main */
}
```

- Analyser graphiquement l'usage fait de la pile par votre programme.
- Quel est le travail réalisé par l'instruction `push` ?
- Proposer une autre écriture des instructions `push` et `pop` via les instructions `sub`, `add` et `mov`



Les directives assembleur `.cfi` (Call Frame Information) sont des extensions proposées par GAS (GNU Assembler). Elles peuvent potentiellement être utilisées par les outils de debug dans une optique de trace et de déroulement de la pile en proposant un complément d'information sur le contexte d'exécution de la procédure courante. Analyse poussée du contenu de la pile. *Dans une optique d'analyse logique du code, ne pas s'y intéresser.* Pour les retirer à la compilation, ajouter l'option `-fno-asynchronous-unwind-tables`





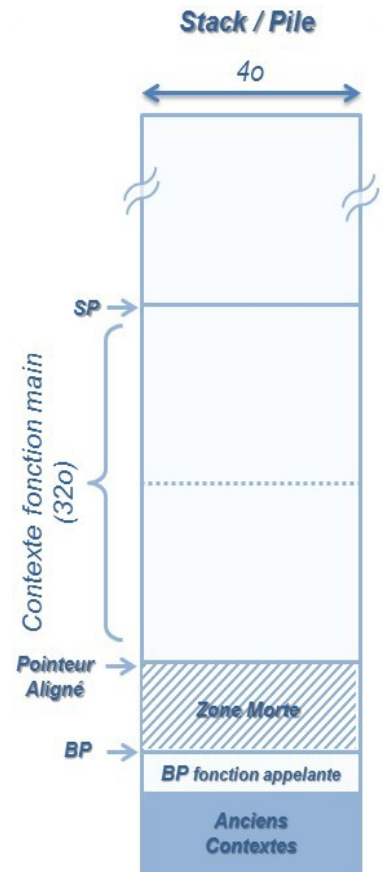
### 2.2. Variables locales non initialisées

- Éditer puis compiler le programme *local.c* en s'arrêtant à la phase d'assemblage.

```
int main (void){
/* pushl %ebp
   movl %esp, %ebp
   andl $-8, %esp : alignement mémoire du pointeur de sommet de pile.
   adresse générée de valeur multiple de la taille d'un path CPU
   subl $32, %esp : allocation de 32o sur la pile (multiple de 16o) */

    char foo;
    unsigned int bar;

    return 0;
/* movl $0, %eax
   popl %ebp
   ret */
}
```



- Que constatons-nous ?
- Ajouter le "qualifier" ou qualificateur de type *volatile* devant chaque déclaration, compiler le programme puis interpréter le résultat.
- Analyser graphiquement l'usage fait de la pile par votre programme.

Le "qualifier" ou qualificateur de type *volatile* force le compilateur à n'opérer aucune optimisation sur les variables ainsi déclarées et laisse alors la porte ouverte à des modifications potentielles de la ressource par d'autres entités de l'appliquatif (exemple des applications multi-tâches ou multi-threads). Ceci est extrêmement utilisé dès qu'il s'agit de manipuler des variables critiques (ressources partagées) et que nous sommes amenés à lever les options d'optimisation à la compilation. *Un qualificateur de type doit être vu comme une directive de compilation sciemment écrite par le développeur afin d'aiguiller voir forcer la chaîne de compilation à opérer des traitements privilégiés sur la variable.* De façon générale, il s'agit de stratégies de durcissement ou d'optimisation de code (vitesse ou taille).



### 2.3. Variables locales initialisées

- Éditer puis compiler le programme *init.c* en s'arrêtant à la phase d'assemblage.

```
int main(void){
    char foo = 1;
    unsigned int bar = 2;

    foo = (char) bar;
    return 0;
}
```

- Insérer le code assembleur généré à votre fichier source C sous forme de commentaires puis Interpréter son fonctionnement.
- Représenter graphiquement au *crayon gris* l'usage fait de la pile par votre programme. Faire évoluer le schéma précédent.
- Préciser les adresses relatives des variables locales
- Qualifier la variable *bar* de *const*. Compiler le programme puis interpréter le résultat.

- Le qualificateur de type *const* n'est qu'une protection à la compilation dédié à certaines variables ou pointeurs initialisés accessibles qu'en lecture seule. Qualifier une variable de *const* n'impacte en rien sa localisation en mémoire et permet de durcir l'applicatif en cours de développement en laissant le compilateur s'assurer qu'une ressource accessible en lecture seule le reste d'un point de vu logiciel.
- Constaté que durant un transtypage d'un format entier long vers un format entier plus court, la chaîne de compilation ne préserve que les bits les moins significatifs ou bits de poids faibles (Lsb ou Less Significant Bits). Ce point peut complexifier grandement l'implémentation d'algorithmes au format entier (format à virgule fixe).



Vous constaterez que la chaîne de compilation effectue à votre place des alignements mémoire. Vous ne rencontrerez ce type d'optimisation que sur processeur vectoriel. Ceci se manifeste par des espaces mémoires vides des quelques octets présents entre vos différentes allocations de structures de données en mémoire. *Tout pointeur aligné, possède une valeur multiple de l'alignement réalisé et peut ouvrir l'accès au compilateur à l'utilisation d'instructions de chargement et de sauvegarde mémoire plus rapide en temps d'exécution.*

Il s'agit de mécanismes d'optimisation matériels permettant au CPU de minimiser les accès mémoires aux données. Dans une optique d'optimisation, ceci peut éviter des défauts d'alignement mémoire (programmation vectorielle) et l'utilisation d'instructions dédiées à la lecture/écriture de données alignées (souvent plus lentes).

La taille de ces alignements mémoire est le plus souvent liée à la taille d'un mot "vectoriel" CPU. D'une architecture CPU à une autre, cette taille diffère. Sur architecture vectorielle x86/x64 récente, un mot CPU fait 128bits (16o) depuis l'arrivée des extensions SIMD SSE (chemins/bus larges vers la mémoire donnée L1D). Les alignements mémoire de structures de données se feront donc le plus souvent via des adresses de valeurs multiples de 16 (modulo 16o).

De quoi dépend la taille d'un mot CPU :

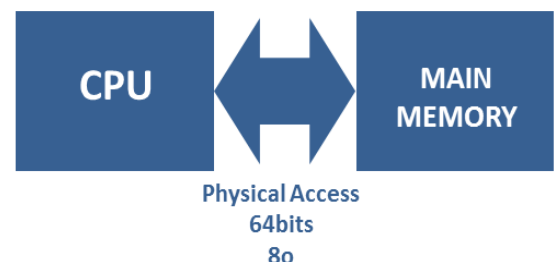
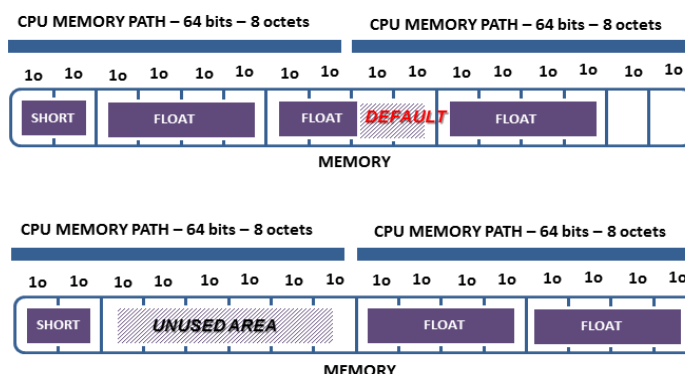
- *Taille des path vers la mémoire donnée de niveau 1 ou L1D (le plus souvent un cache) :*  
Exemple de la famille coreiX sandyBridge, 128bits vers le cache programme L1P, 2x128bits load et 1x128bits store vers le cache donnée L1D
- *Taille des lignes de cache :* tailles multiples d'un mot vectoriel CPU



Prenons ci-dessus un exemple de jeu de données, les premières non-alignées et les secondes alignées modulo 8 octets. Nous supposons que les bus ou path entre CPU et niveau mémoire L1D possèdent une taille de 64bits ou 8o :

```
short foo ;
float bar[3] ;
```

**NOK**



### 2.4. Appel de fonction

- A partir de maintenant et jusqu'à la fin du TP, nous analyserons de l'assembleur 64bits compatible architecture x64. Assembleur généré par défaut sur système GNU/Linux 64bits.

```
# gcc -Wall -S main.c
```

- Éditer puis compiler le programme *function.c* en s'arrêtant à la phase d'assemblage.

```
/* private prototypes */
int inc (int bar);

/* program entry point */
int main (void){
    int foo ;

    foo = inc (1);
    return 0;
}

/* function incrémentant la valeur d'entrée */
int inc(int bar) {
    return bar +1;
}
```

- Insérer le code assembleur généré à votre fichier source C sous forme de commentaires puis Interpréter son fonctionnement.
- Déclarer maintenant à la volée la variable locale *foo*. Que constatons-nous ?

```
int foo = inc (1);
```

Attention ce type de déclaration à la volée n'est pas supporté par toutes les chaînes de compilation C (exemple de la norme C89 également nommée C ANSI). Vous aurez d'ailleurs de mauvaises surprises durant les enseignements de systèmes embarqués si vous utilisez ce type de déclaration (exemple de la toolchain de Microchip pour architecture PIC18 ne supportant que les normes C89-ANSI et partiellement la norme C99). *Prenez dès maintenant pour habitude en langage C de gérer vos déclarations de variables locales en début de fonction, notamment dans l'embarqué et si vos projets futurs peuvent impliquer des portages sur de nouvelles architectures cibles*



- Observons le programme désassemblé (traduction binaire vers assembleur) correspondant au binaire exécutable après compilation et édition des liens. Intéressons-nous aux sections de code correspondant à notre programme d'analyse. Bien constater que les références symboliques (ex : *inc*) sont remplacées à l'édition des liens.

```

00000000004004d6 <main>:
4004d6: 55                push    %rbp
4004d7: 48 89 e5          mov     %rsp,%rbp
4004da: 48 83 ec 10       sub     $0x10,%rsp
4004de: bf 01 00 00 00    mov     $0x1,%edi
4004e3: e8 0a 00 00 00    callq   4004f2 <inc>
4004e8: 89 45 fc          mov     %eax,-0x4(%rbp)
4004eb: b8 00 00 00 00    mov     $0x0,%eax
4004f0: c9               leaveq  %eax
4004f1: c3               retq

00000000004004f2 <inc>:
4004f2: 55                push    %rbp
4004f3: 48 89 e5          mov     %rsp,%rbp
4004f6: 89 7d fc          mov     %edi,-0x4(%rbp)
4004f9: 8b 45 fc          mov     -0x4(%rbp),%eax
4004fc: 83 c0 01          add     $0x1,%eax
4004ff: 5d                pop     %rbp
400500: c3               retq
400501: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
400508: 00 00 00          nopl   0x0(%rax,%rax,1)
40050b: 0f 1f 44 00 00    nopl   0x0(%rax,%rax,1)

```

```
# gcc -Wall function.c -o function
# objdump -S function > disassembly.txt
```

- L'instruction *call* réalise deux opérations. Elle modifie le pointeur programme (écriture dans PC ou IP). Elle sauvegarde également l'adresse de retour sur la pile et modifie donc SP. Dans le listing ci-dessus, quelle est la valeur de l'adresse de retour ?
- Quelle est l'adresse de la fonction *inc* ?
- Bien observer le code binaire (opcode) équivalent à l'instruction *call inc*, que constatez-vous ? Quel est l'opcode de l'instruction *call* pour un adressage relatif 32bits (cf. doc Intel) ?
- Proposer une autre écriture des instructions *call* et *ret* via les instructions *push*, *pop* et *jump*.

- Représenter graphiquement au *crayon gris* l'usage fait de la pile par le programme. Compléter le schéma ci-dessous.
- Observer les mécanismes de gestion de la variable locale (ou paramètre, c'est la même chose !) présente dans la fonction *inc*. Critiquer les choix opérés par la chaîne de compilation.
- Écrire à la main une solution assembleur plus performante (sans toucher au code généré pour la fonction *main*).



- Modifier le prototype de la fonction par *char inc (register char bar)*. Interpréter le code généré et en déduire le rôle de cette classe de stockage.

Le mot clé *register* est une *classe de stockage* demandant (sans l'imposer) à la chaîne de compilation de manipuler, si les ressources matérielles le permettent, les variables ainsi qualifiées par registre CPU et non par la pile. Il s'agit d'un mécanisme simple d'optimisation permettant de limiter légèrement l'empreinte mémoire d'un programme mais pouvant augmenter significativement ses performances en évitant des allez/retours en lecture/écriture avec la pile présente en mémoire principale (DDR SRAM).



- Représenter graphiquement au *crayon gris* l'usage fait de la pile par le programme. Compléter le schéma ci-contre.
- Étudions maintenant un mot clé arrivé avec la norme *ISO C99* du langage C et pouvant être très puissant si bien utilisé, les fonctions *inline*. Modifier les prototypes à la déclaration et la définition de la fonction *inc* comme ci-dessous, compiler et analyser le code résultant ?

```
/* private prototypes */
inline int inc( int bar) __attribute__((always_inline));

...

inline int inc( int bar) {
    return bar+1;
}
```



- Quels sont les avantages et inconvénients de l'*inlining* de fonction ?
- Proposer une solution équivalente à l'aide de *macro fonction*.

Pour information, sachez qu'une fonction peut également être déclarée avec le mot clé *static*. Le cas échéant, si votre chaîne de compilation le supporte, elle implémentera les mécanismes suivants :

- Inlining éventuel de la fonction (élimine l'overhead inhérent à un appel de fonction)
- Utilise sinon des saut relatif courts plutôt que des sauts absolus longs.
- N'empile que les registres strictement nécessaires à la fonction (peut être différent de la convention par défaut utilisée par la toolchain)
- Limite la portée de la fonction au source courant
- Selon la chaîne de compilation, peut également supprimer toute trace de la fonction dans l'exécutable de sortie



### 2.5. Limites de la pile

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86x64/except/grow.c*

```
int main( void ){
    // allocations sur la pile jusqu'à débordement
    main();

    return 0 ;
}
```

- Que constatons-nous ? De quel segment logique s'agit-il ?

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/stackOverflow.c*

- Qu'elle est la taille actuelle de la pile associée au programme ?

- Interpréter le code source de ce fichier puis le modifier de façon à obtenir une pile de *16Mo*. Tester et vérifier les modifications apportées.



# ALLOCATIONS STATIQUES

---

## FICHIERS ELF

---



### 3. ALLOCATIONS STATIQUES ET FICHIER ELF

Dans cette nouvelle partie, nous allons nous intéresser aux allocations statiques de ressources, c'est à dire aux allocations réalisées à la compilation contrairement aux allocations automatiques (pile/stack) et dynamiques (tas/heap) qui sont allouées à la volée durant l'exécution. Nous nous focaliserons donc sur les sections et segments présents dans le fichier ELF binaire exécutable. Une allocation statique garantit la non modification de l'emplacement mémoire d'une ressource du début à la fin de l'exécution du programme courant puisque déjà présente dans le fichier exécutable de sortie. D'où la dénomination de *static*.

#### 3.1. Variables globales

- Éditer puis compiler le programme présent dans le répertoire de TP `/x86x64/static/global.c` en s'arrêtant à la phase d'assemblage.

```
/* private declarations */
static char foo[1000000] = {'a'};

/* program entry point */
int main(void){
    *foo = 'A';
    return 0;
}
```

- Insérer le code assembleur généré à votre fichier source C sous forme de commentaires puis Interpréter son fonctionnement.
- Compiler le programme précédent en allant jusqu'à l'édition des liens incluse, puis observer la taille du fichier exécutable de sortie. Comment expliquer le résultat obtenu ?
- En utilisant l'utilitaire *objdump* du package *binutils* du projet GNU, observer le contenu du fichier ELF exécutable après compilation et édition des liens. Observer, après désassemblage du binaire, l'adresse du premier élément du tableau présent dans la section *.text* label *main*. Pourquoi l'adresse générée n'est-elle pas une adresse absolue mais relative à IP (Instruction Pointer) ?

```
# gcc -Wall global.c -o global
# objdump -S global
```

- Durant le développement d'une application, en cas d'absolue nécessité concernant l'allocation de larges ressources mémoire, quelle solution est à préconiser afin de garder une gestion optimale des ressources mémoire de la machine ?
- Sur quels types de machines peut-on utiliser sans risque la fonction standard de libération de ressources mémoire *free* ?

Il faut savoir que durant l'édition des liens, toute chaîne de compilation C travaille de façon très ordonnée et structure l'espace mémoire virtuel/logique de l'application en segments (regroupement de sections) ne pouvant se chevaucher (aussi appelé Virtual Memory Area ou VMA sous Linux). Prenons l'exemple de quelques unes des principales sections : *.text* (pour le code), *.data*, *.bss*, *.rodata* ... Après compilation, nous pouvons observer la table des sections en utilisant l'utilitaire *objdump* de *binutils* (option *h*, comme header sections) :

- `gcc foo.c -o foo`
- `objdump -h foo`

Dans les langages compilés comme le langage C, une donnée est soit allouée statiquement à la compilation soit allouée dynamiquement/automatiquement à l'exécution. Par essence, les langages interprétés ne peuvent qu'allouer des ressources durant l'exécution (allocations dynamiques) :

- *Allocations statiques* : allocations à la compilation et structuration des familles de variables en sections (variables globales ou statiques initialisées, non-initialisées, constantes ...). Les allocations statiques de ressources (variables et codes) sont présentes dans le fichier exécutable de sortie (fichier ELF). Principales sections de données développeur. Adressage relatif à IP :
  - *.data* : section mémoire où se situe les données statiques initialisées
  - *.bss* : section mémoire où se situe les données statiques non initialisées
  - *.rodata* : données statiques initialisées en lecture seule (read-only)
- *Allocations automatiques* : allocations/libérations à l'exécution sur la pile ou stack : contextes d'exécution courant (encapsulé entre BP et SP, variables locales, paramètres de fonctions) et sauvegardes des contextes (adresses de retour des fonctions, contextes d'exécution des fonctions - BP)
- *Allocations dynamiques* : allocations/libérations à l'exécution sur le tas ou heap (malloc, realloc, free ... et variantes)

Pour information, ne jamais utiliser la fonction *free*, amenant une fragmentation de la mémoire physique, sur un processeur ne possédant pas de PMMU (unité de pagination). *Par exemple sur un MCU, nous n'effectuerons que des allocations statiques et automatiques.*



### 3.2. Variables locales statiques

- Éditer puis compiler le programme *static.c* en s'arrêtant à la phase d'assemblage.

```
int main(void){
    static char foo = 'a';
    foo++;
    return 0;
}
```

- Insérer le code assembleur généré au fichier source C sous forme de commentaires puis Interpréter son fonctionnement.
- La variable *foo* est-elle gérée par la pile ? Dans quelle section du binaire de sortie est rangée cette variable (changer sa valeur et effectuer plusieurs compilations afin de répondre à la question) ? S'aider de l'utilitaire *objdump* (*objdump -s <exec\_name>*)
- Pourquoi une variable locale statique peu préserver les valeurs qui lui ont été précédemment affectées durant les appels antérieurs de la fonction dans laquelle elle a été déclarée ?
- Nous allons maintenant utiliser l'utilitaire *strip* (décaper/enlever) de *binutils*, très utilisé dans le domaine de l'embarqué. Compiler votre fichier source jusqu'à l'obtention d'un fichier exécutable puis observer son empreinte mémoire. "Stripper" le fichier puis observer à nouveau son empreinte mémoire. Expliquer en quoi le processus d'exécution de ce binaire sera plus efficace ?

```
# gcc -Wall -m32 static.c -o static
# strip ./static
```

- Une variable locale *static* possède une portée limitée à la fonction courante. Ce point n'est géré qu'à la compilation. Physiquement et spatialement, ces variables sont très proches des variables globales.
- Le package *binutils* ou *GNU Binutils* est un ensemble d'outils GNU dédiés à l'analyse et la manipulation de fichiers binaires ELF et DWARF. Le service *strip* permet de nettoyer le contenu de fichier binaire en retirant notamment les informations inutiles (debug, références symboliques, concaténation de sections, etc) et en concaténant notamment certaines sections (table des sections allégée). *gcc* peut *strip*per à la compilation via l'option *-s*.



### 3.3. Chaînes de caractères

- Éditer puis compiler le programme *string.c* en s'arrêtant à la phase d'assemblage. Ajouter la commande *-fno-stack-protector* à la compilation, sans quoi le compilateur insérera du code pour la protection de débordement de buffer (*-fstack-protector*)

```
/* program entry point */
int main(void){
    char* pfoo = "foo";
    char bar[] = "bar";

    return 0;
}
```

- Insérer le code assembleur généré à votre fichier source C sous forme de commentaires puis Interpréter son fonctionnement.
- Comment expliquer la taille de la variable *pfoo* ?
- Comment est allouée la chaîne de caractères *"toto"* ?
- Comment est allouée la chaîne de caractères *"titi"* ? Ma phrase précédente a-t-elle un sens ?
- En utilisant l'utilitaire *objdump* du package *binutils* du projet GNU, observer le contenu du fichier ELF exécutable après compilation et édition des liens du fichier *string.c*. Dans quelle section a été rangée la chaîne de caractères *"toto"* ? Est-ce normal ? Justifier votre réponse.

```
# gcc -Wall -m32 string.c -o string
# objdump -s string
```

Dans le cadre d'allocations statiques, bien constater que toutes les allocations sont réalisées à la compilation et sont donc présentes dans le fichier exécutable de sortie (fichier ELF dans notre cas).



# ALLOCATIONS DYNAMIQUES

---

## GESTION DU TAS

---





### 4. ALLOCATIONS DYNAMIQUES ET GESTION DU TAS

Dans cette partie, nous allons nous intéresser aux variables allouées dynamiquement sur le tas ou heap. Cette solutions est à préconiser dès que votre applicatif nécessite de large ressources mémoires. *Ne jamais oublier de libérer les ressources allouées après usage.*

#### 4.1. Gestion du Tas

- Éditer puis compiler le programme `/x86x64/heap/heap.c` en s'arrêtant à la phase d'assemblage . Ce code étant très simple, analyser rapidement l'assembleur généré.

```
#include <stdlib.h>

/* program entry point */
int main(void){
    char *pfoo;

    pfoo = (char *) malloc (100 * sizeof(char));
    *(pfoo + 7) = 1;
    free (pfoo);

    return 0;
}
```

- Comparer une adresse allouée sur le tas à un adresse allouée sur la pile. Constaté qu'il s'agit d'espaces mémoire virtuels distincts (affichage via printf d'un pointeur sur la pile et sur le tas).

#### 4.2. Limites du Tas

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP `/x86x64/except/heapOverflow.c`. Ce programme permet de tester la taille limite du Tas.
- Qu'elle la taille limite théorique du Tas sous système Linux (information système) ?

# EXCEPTIONS MATERIELLES

---

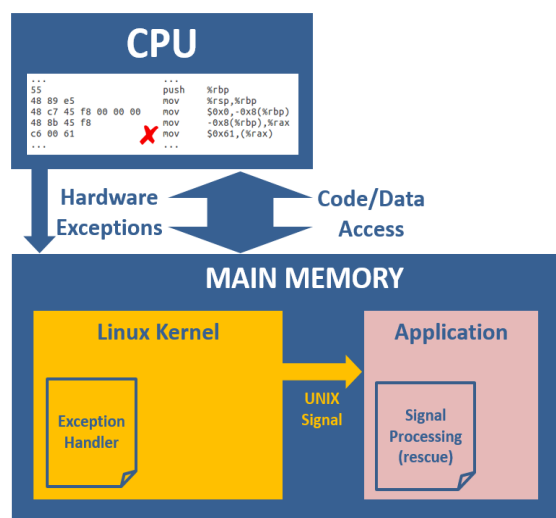
## SIGNAUX UNIX

---



### 5. EXCEPTIONS MATERIELLES ET SIGNAUX UNIX

Dans cette nouvelle partie, nous allons nous intéresser aux exceptions matérielles générées par l'ensemble CPU/MMU. Une exception est un événement matériel synchrone généré par le CPU (synchrone au regard du fonctionnement d'un CPU dont les traitements restent synchronisés sur une référence d'horloge, et non au regard de la probabilité d'occurrence). Ces événements sont relevés par le CPU lorsque celui-ci détecte une voire plusieurs conditions prédéfinies durant l'exécution d'une instruction (violation de privilège, division flottante par zéro, accès illégal en mémoire ...). Rappelons les traitements réalisés par l'ensemble CPU/MMU et le systèmes d'exploitation lorsque qu'une exception matérielle survient :



- Le programme en cours d'exécution génère une erreur connue du CPU. Les familles et types d'exceptions sont toutes prédéfinies et connues de l'architecture.
- Le CPU stoppe l'exécution du programme en cours et appelle une procédure enfouie et endormie du système. Une sauvegarde du contexte CPU (registres internes) est également réalisée et peut être accessible depuis un applicatif en espace utilisateur. Pour les curieux, il s'agit de la fonction `do_page_fault` dans le cas de Linux (présente dans le fichier `/arch/<cpu>/mm/fault.c` du système de fichier du kernel Linux).
- La procédure d'exception appelée (ou exception handler) est chargée de relever le type de défaut et de générer, si l'exception le permet, un signal logiciel UNIX à destination du processus ayant généré le défaut.
- Si le processus ne capte et ne traite pas le signal UNIX, il est alors tué par le système. Ceci assure un *cloisonnement des défauts et la stabilité du système*.
- Si le processus gère le signal UNIX, votre applicatif peut alors tenter d'acquiescer le défaut (restaurer un contexte CPU viable par exemple) ou de réaliser un traitement spécifique assurant une robustification ou durcissement de votre applicatif (message d'erreur, redémarrage de l'application, mode dégradé, fichier de log ...).

## 5.1. Lecture seule

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/readonly.c*. Ce programme force la génération d'une exception matérielle de type *Fault*.
- Interpréter et expliquer le résultat obtenu. Quel type de signal UNIX a été envoyé au processus à la source du défaut ? Préciser le segment concerné ?
- Durant les exercices précédents, vous avez été amenés à forcer des débordements de pile et de tas. Quelles types et familles d'exceptions et de signaux ont été levés dans les deux cas (s'aider du cours) ?

## 5.2. Défaut d'alignement

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/busererror.c*.
- Interpréter et expliquer le résultat obtenu. Quel type de signal UNIX a été envoyé au processus à la source du défaut ?

### 5.3. Division par zéro

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/divzero.c*.
- Interpréter et expliquer le résultat obtenu. Quel type de signal UNIX a été envoyé au processus à la source du défaut ?

### 5.4. Debug

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/sigtrap.c*. Ce programme force la génération d'une exception matérielle de type *Trap*. Dans certains cas, ce type d'exception peut-être utilisé par les outils de debug.
- Interpréter et expliquer le résultat obtenu. Quel type de signal UNIX a été envoyé au processus à la source du défaut ?

### 5.5. Pointeur nul

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP `/x86/except/nullptr.c`. Ce programme force la génération d'une exception matérielle de type *Fault*.
- Interpréter et expliquer le résultat obtenu. Quel type de signal UNIX a été envoyé au processus à la source du défaut ?

### 5.6. Signal UNIX

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP `/x86/except/ackdefault.c`. Ce programme force la génération d'une exception matérielle de type *Fault*.
- Analyser le résultat obtenu.
- Dé-commenter la partie propre à la configuration et la gestion de signaux UNIX pour votre application (gestion du signal UNIX SIGSEGV). Compiler, exécuter puis analyser le résultat obtenu.
- Sachant que le contexte CPU est sauvé dans la structure `context->uc_mcontext` déclarée dans le fichier d'en-tête `ucontext.h` (se documenter sur internet pour répondre à cette question) :
  - Identifier le registre à la source du défaut (désassemblage de votre fichier exécutable)
  - Observer puis restaurer un contexte CPU viable depuis le handler de signal
  - Acquitter le défaut puis rendre la main au processus
  - Proposer l'affichage suivant depuis le shell

*hardware exception has been generated*

*\*\*\* signal handler*

*\*\*\* current cpu context eax=0 (dereferenced pointer)*

*\*\*\* new cpu context eax=0x???? (referenced pointer)*

*\*\*\* resume processus*

*exception has been acknowledged, tmp = 'a'*

# UNITE DE PAGINATION

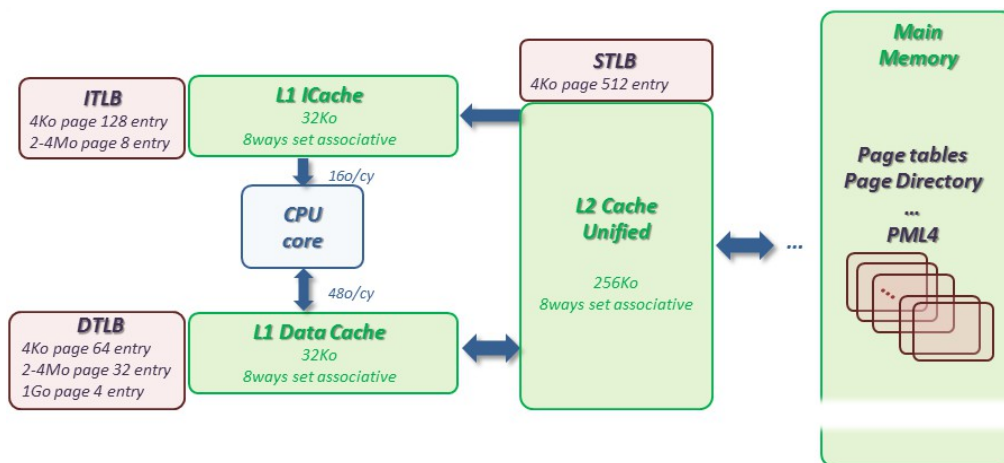
---





### 6. UNITE DE PAGINATION

Dans cette ultime partie, nous allons nous intéresser au travail réalisé par la MMU (Memory Management Unit). Rappelons que sur architecture x86-x64 la MMU réalise deux grands traitements sur les adresses virtuelles, la segmentation et la pagination. Néanmoins, nous avons pu voir en cours que l'étage de segmentation est transparent à notre époque (segmentation logique réalisée à la compilation par les outils et au chargement par le système). De ce fait, les MMU sont souvent nommées PMMU (Paged MMU) à notre époque.



Exemple d'architecture des TLB's  
sur la micro-architecture Intel Sandy Bridge

- Rappeler le travail réalisé par la PMMU ou l'unité de pagination ?
- Rappeler les granularités possibles de la mémoire physique ? Ces granularités sont des services proposés par le processeur et doivent ensuite être gérées par le système d'exploitation.
- Rappeler ce qu'est une TLB ?
- Analysons le travail réalisé par la fonction *malloc*. Analyser le projet présent dans le répertoire *mmu* puis le compiler. Sachant que cette application travail sur des tableaux de grande taille, nous allons constater qu'en appelant la fonction *malloc* le système prendra l'initiative d'allouer des *huge pages* de 2Mo plutôt qu'un grand nombre de pages de 4Ko.

Estimer par le calcul le nombre de huge pages de 2Mo nécessaires au stockage de nos différentes matrices et l'empreinte mémoire équivalente en Ko ? Conditions : SIZE = 2048 dans le fichier *bench.c*

- Le fichier */proc/meminfo* permet de visualiser quelques statistiques de l'usage fait de la mémoire par le système (fichier mis à jour en temps réel par le système). Lancer la commande *watch* ci-dessous permettant de répéter toutes les secondes l'affichage du contenu du fichier *meminfo* puis lancer le script *perf.sh*. Constaté l'allocation mémoire de huge pages (champ *AnonHugePages*, les autres champs sont obsolètes) et valider le calcul de la question précédente.

```
# watch -n 1 "cat /proc/meminfo | grep Huge"
```

- Pour les étudiants travaillant sur leur machine personnelle (droit *root* exigé), nous allons maintenant forcer le système à désactiver l'allocation de *huge pages* puis constater la différence de performance entre les deux stratégies d'allocation.
- Modifier la configuration par défaut du système.

```
# sudo chmod 666 /sys/kernel/mm/transparent_hugepage/enabled
# echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

- Lancer le script *perf.sh* et constater le nombre de dTLB-load-misses :
- Restaurer la configuration par défaut du système, lancer le script *perf.sh* et constater le nombre de dTLB-load-misses :

```
# echo always > /sys/kernel/mm/transparent_hugepage/enabled
```

- Comparer également les temps d'exécution du programme dans les deux cas de figure et constater une dégradation des performances liée à un mauvais usage de la PMMU par le système.

- La question qui suit est issue d'un problème rencontré par un industriel partenaire. En effet, pour certains de ses algorithmes, il pouvait constater un grand manque de déterminisme avec des écarts dans les temps d'exécution allant jusqu'à 200% après plusieurs exécutions successives et des temps d'attente de plusieurs minutes. Son programme implémenté initialement la première solution.

Les deux exemples qui suivent réalisent l'allocation dynamique de la même empreinte mémoire (à quelques % près).

### *Solution 1*

```
#define SIZE 1024

int **pVect;
int i ;

pVect = (int **) malloc(sizeof(int*)*SIZE);

for(i=0; i<SIZE; i++)
    pVect[i] = (int *) malloc(sizeof(int)*SIZE);
```

### *Solution 2*

```
#define SIZE 1024

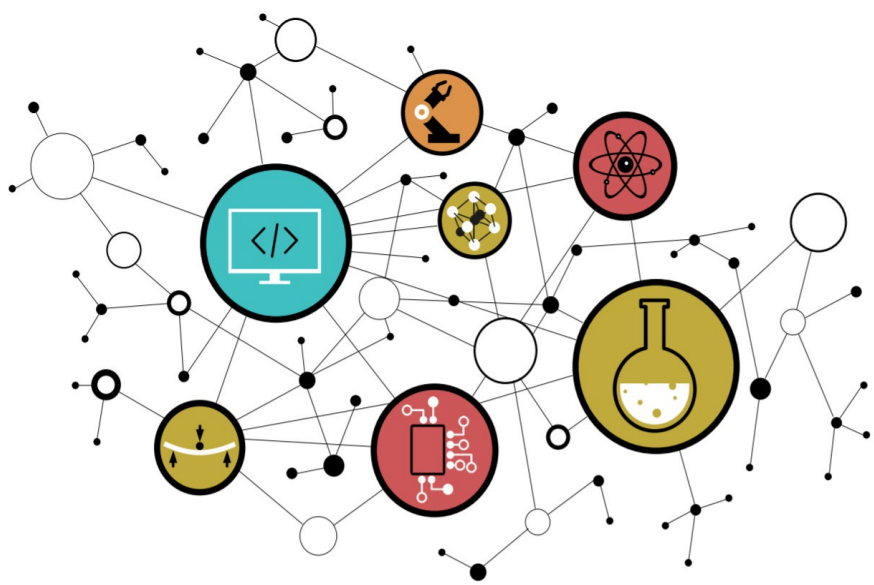
int *pMatrix;

pMatrix = (int *) malloc(sizeof(int)*SIZE*SIZE);
```

- En quoi la première solution est plus intéressante dans le cadre d'algorithmes implémentant du calcul matriciel ?
- Expliquer le travail réalisé par le système dans le premier cas et justifier en quoi la seconde solution est plus performante.

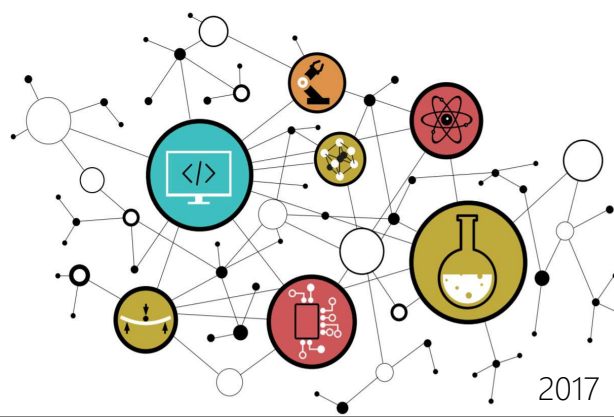
# ANNEXES

---



# ANNEXES

---



## SOMMAIRE

1. EXTRAITS – INTEL 64 AND IA-32 ARCHITECTURES – Software Developer's Manual – volume 1
2. JEU D'INSTRUCTIONS INTEL 8086
3. JEU D'INSTRUCTIONS INTEL SANDY BRIDGE – Agner Fog, Copenhagen University College of Engineering

GLOSSAIRE

## 1. EXTRAITS

Intel 64 and IA-32 Architectures – Software Developer's Manual – volume 1

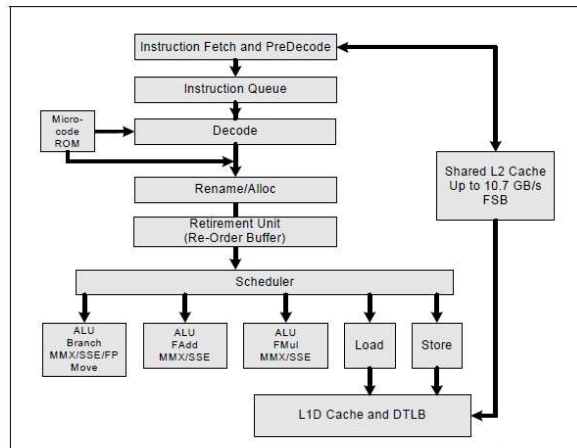


Figure 2-3. The Intel Core Microarchitecture Pipeline Functionality

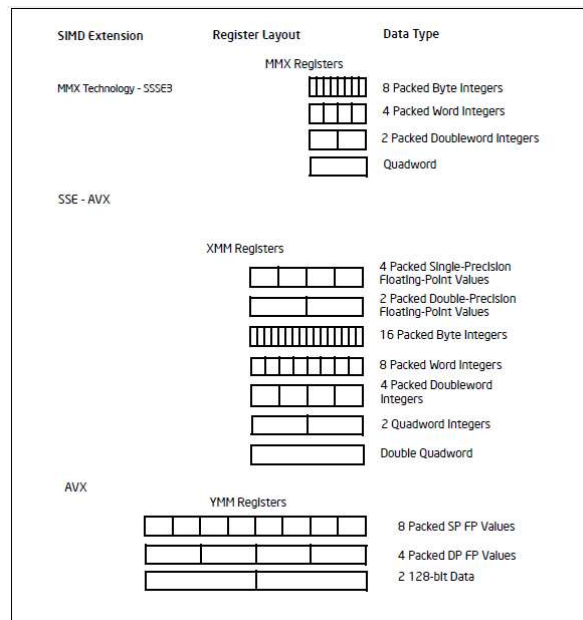


Figure 2-4. SIMD Extensions, Register Layouts, and Data Types

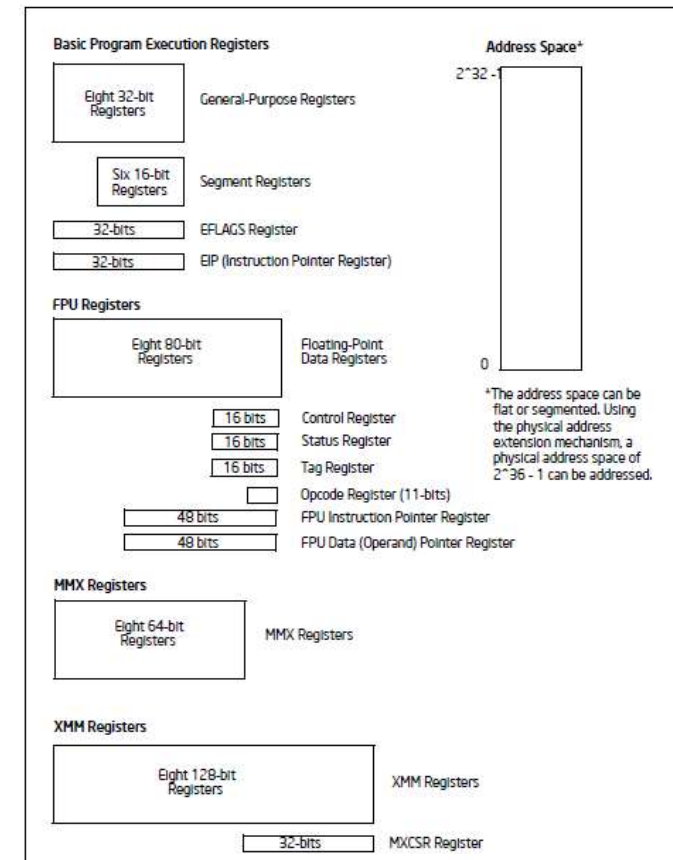


Figure 3-1. IA-32 Basic Execution Environment for Non-64-bit Modes

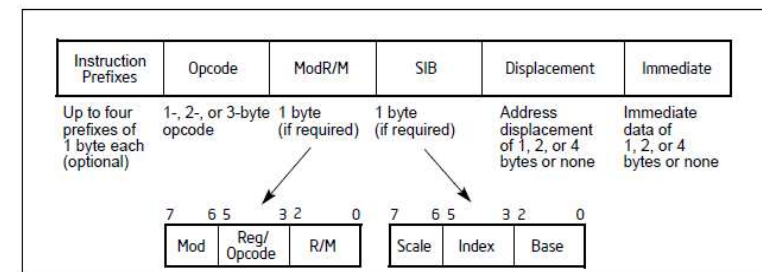


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

## 2. JEU D'INSTRUCTIONS INTEL 8086

<b>AAA</b>	ASCII adjust AL after addition	<b>HLT</b>	Enter halt state
<b>AAD</b>	ASCII adjust AX before division	<b>IDIV</b>	Signed divide
<b>AAM</b>	ASCII adjust AX after multiplication	<b>IMUL</b>	Signed multiply
<b>AAS</b>	ASCII adjust AL after subtraction	<b>IN</b>	Input from port
<b>ADC</b>	Add with carry	<b>INC</b>	Increment by 1
<b>ADD</b>	Add	<b>INT</b>	Call to interrupt
<b>AND</b>	Logical AND	<b>INTO</b>	Call to interrupt if overflow
<b>CALL</b>	Call procedure	<b>IRET</b>	Return from interrupt
<b>CBW</b>	Convert byte to word	<b>Jcc</b>	Jump if condition
<b>CLC</b>	Clear carry flag	<b>JMP</b>	Jump
<b>CLD</b>	Clear direction flag	<b>LAHF</b>	Load flags into AH register
<b>CLI</b>	Clear interrupt flag	<b>LDS</b>	Load pointer using DS
<b>CMC</b>	Complement carry flag	<b>LEA</b>	Load Effective Address
<b>CMP</b>	Compare operands	<b>LES</b>	Load ES with pointer
<b>CMPSB</b>	Compare bytes in memory	<b>LOCK</b>	Assert BUS LOCK# signal
<b>CMPSW</b>	Compare words	<b>LDSB</b>	Load string byte
<b>CWD</b>	Convert word to doubleword	<b>LODSW</b>	Load string word
<b>DAA</b>	Decimal adjust AL after addition	<b>LOOP/LOOPx</b>	Loop control
<b>DAS</b>	Decimal adjust AL after subtraction	<b>MOV</b>	Move
<b>DEC</b>	Decrement by 1	<b>MOVSb</b>	Move byte from string to string
<b>DIV</b>	Unsigned divide	<b>MOVSW</b>	Move word from string to string
<b>ESC</b>	Used with floating-point unit	<b>MUL</b>	Unsigned multiply

<b>NEG</b>	Two's complement negation	<b>SCASB</b>	Compare byte string
<b>NOP</b>	No operation	<b>SCASW</b>	Compare word string
<b>NOT</b>	Negate the operand, logical NOT	<b>SHL</b>	Shift left (unsigned shift left)
<b>OR</b>	Logical OR	<b>SHR</b>	Shift right (unsigned shift right)
<b>OUT</b>	Output to port	<b>STC</b>	Set carry flag
<b>POP</b>	Pop data from stack	<b>STD</b>	Set direction flag
<b>POPF</b>	Pop data from flags register	<b>STI</b>	Set interrupt flag
<b>PUSH</b>	Push data onto stack	<b>STOSB</b>	Store byte in string
<b>PUSHF</b>	Push flags onto stack	<b>STOSW</b>	Store word in string
<b>RCL</b>	Rotate left (with carry)	<b>SUB</b>	Subtraction
<b>RCR</b>	Rotate right (with carry)	<b>TEST</b>	Logical compare (AND)
<b>REPxx</b>	Repeat MOVs/STOS/CMPS/LODS/SCAS	<b>WAIT</b>	Wait until not busy
<b>RET</b>	Return from procedure	<b>XCHG</b>	Exchange data
<b>RETN</b>	Return from near procedure	<b>XLAT</b>	Table look-up translation
<b>RETF</b>	Return from far procedure	<b>XOR</b>	Exclusive OR
<b>ROL</b>	Rotate left		
<b>ROR</b>	Rotate right		
<b>SAHF</b>	Store AH into flags		
<b>SAL</b>	Shift Arithmetically left (signed shift left)		
<b>SAR</b>	Shift Arithmetically right (signed shift right)		
<b>SBB</b>	Subtraction with borrow		



### 3. JEU D'INSTRUCTIONS INTEL SANDY BRIDGE

#### Intel Sandy Bridge

##### List of instruction timings and $\mu$ op breakdown

Explanation of column headings:

**Operands:** i = immediate data, r = register, mm = 64 bit mmx register, x = 128 bit xmm register, (x)ymm = mmx or xmm register, y = 256 bit ymm register, same = same register for both operands. m = memory operand, m32 = 32-bit memory operand, etc.

**$\mu$ ops fused domain:** The number of  $\mu$ ops at the decode, rename, allocate and retirement stages in the pipeline. Fused  $\mu$ ops count as one.

**$\mu$ ops unfused domain:** The number of  $\mu$ ops for each execution port. Fused  $\mu$ ops count as two. Fused macro-ops count as one. The instruction has  $\mu$ op fusion if the sum of the numbers listed under p015 + p23 + p4 exceeds the number listed under  $\mu$ ops fused domain. A number indicated as 1+ under a read or write port means a 256-bit read or write operation using two clock cycles for handling 128 bits each cycle. The port cannot receive another read or write  $\mu$ op in the second clock cycle, but a read port can receive an address-calculation  $\mu$ op in the second clock cycle. An x under p0, p1 or p5 means that at least one of the  $\mu$ ops listed under p015 can optionally go to this port. For example, a 1 under p015 and an x under p0 and p5 means one  $\mu$ op which can go to either port 0 or port 5, whichever is vacant first. A value listed under p015 but nothing under p0, p1 and p5 means that it is not known which of the three ports these  $\mu$ ops go to.

**p015:** The total number of  $\mu$ ops going to port 0, 1 and 5.

**p0:** The number of  $\mu$ ops going to port 0 (execution units).

**p1:** The number of  $\mu$ ops going to port 1 (execution units).

**p5:** The number of  $\mu$ ops going to port 5 (execution units).

**p23:** The number of  $\mu$ ops going to port 2 or 3 (memory read or address calculation).

**p4:** The number of  $\mu$ ops going to port 4 (memory write data).

**Latency:** This is the delay that the instruction generates in a dependency chain. The numbers are minimum values. Cache misses, misalignment, and exceptions may increase the clock counts considerably. Where hyperthreading is enabled, the use of the same execution units in the other thread leads to inferior performance. Denormal numbers, NaN's and infinity do not increase the latency. The time unit used is core clock cycles, not the reference clock cycles given by the time stamp counter.

**Reciprocal throughput:** The average number of core clock cycles per instruction for a series of independent instructions of the same kind in the same thread.

The latencies and throughputs listed below for addition and multiplication using full size YMM registers are obtained only after a warm-up period of a thousand instructions or more. The latencies may be one or two clock cycles longer and the reciprocal throughputs double the values for shorter sequences of code. There is no warm-up effect when vectors are 128 bits wide or less.

##### Integer instructions

Instruction	Operands	$\mu$ ops fused domain	p015	p0	p1	p5	p23	p4	Latency	Reciprocal throughput	Comments
Move instructions											

MOV	r,ri	1	1	x	x	x			1	0.33	all addressing modes
MOV	r,m	1					1		2	0.5	
MOV	m,r	1					1	1	3	1	
MOVNTI	m,i	1					1	1	1	1	
MOVZX	r,r	2							~350	1	implicit lock
MOVXD	r,r	1	1	x	x	x			1	0.33	
MOVXD	r,m	1					1			0.5	
CMOVcc	r,r	2	2	x	x	x			2	1	
CMOVcc	r,m	2	2	x	x	x	1		2	1	not 64 bit
XCHG	r,r	3	3	x	x	x			2	1	
XCHG	r,m	8					2	1	25		
XLAT	r	3	2				1		7	1	
PUSH	i	1					1	1	3	1	not 64 bit
PUSH	m	1					1	1	1	1	
PUSHF(D/Q)		2					2	1	1	1	
PUSHA(D)		3	2	x	x	x	1	1	8	8	
POP	r	16	0				8	8		8	not 64 bit
POP	(E/R)SP	1					1		2	0.5	
POP	m	1	0				1		1	0.5	
POPF(D/Q)		2					2	1		1	
POPA(D)		9	8	x	x	x	1		18		not 64 bit
LAHF SAHF		18	10				8		9		
SALC		1	1						1	1	
LEA	r,m	3	3						1	1	
LEA	r,m	1	1	1	1				1	0.5	simple complex or rip relative
BSWAP	r32	1	1	1					3	1	
BSWAP	r64	1	1	1					1	1	
PREFETCHNTA	m	2	2						2	1	
PREFETCHQ1/2	m	1					1			0.5	
LFENCE		1					1			0.5	
MFENCE		2					1	1		4	
SFENCE		3	1				1	1		33	
		2					1	1		6	
<b>Arithmetic instructions</b>											
ADD SUB	r,ri	1	1	x	x	x			1	0.33	
ADD SUB	r,m	1	1	x	x	x	1			0.5	
ADD SUB	m,ri	2	1	x	x	x	2	1	6	1	
SUB	r,same	1	0						0	0.25	
ADC SBB	r,ri	2	2	x	x	x			2	1	
ADC SBB	r,m	2	2	x	x	x	1		2	1	
ADC SBB	m,ri	4	3	x	x	x	2	1	7	1.5	
CMP	r,ri	1	1	x	x	x			1	0.33	
CMP	m,ri	1	1	x	x	x	1		1	0.5	

Sandy Bridge										
INC DEC NEG NOT	r	1	1	x	x	x	1	0.33		
INC DEC NEG NOT	m	3	1	x	x	x	2	2		
AAA AAS		2	2				4		not 64 bit	
DAA DAS		3	3				4		not 64 bit	
AAD		3	3				2		not 64 bit	
AAM		8	8				20	11	not 64 bit	
MUL IMUL	r8	1	1	1			3	1		
MUL IMUL	r16	4	4				4	2		
MUL IMUL	r32	3	3				4	2		
MUL IMUL	r64	2	2				3	1		
IMUL	r,r	1	1	1			3	1		
IMUL	r16,r16,i	2	2				4	1		
IMUL	r32,r32,i	1	1	1			3	1		
IMUL	r64,r64,i	1	1	1			3	1		
MUL IMUL	m8	1	1	1	1		3	1		
MUL IMUL	m16	4	3		1	1	2			
MUL IMUL	m32	3	2		1	1	2			
MUL IMUL	m64	2	1		1	1	2			
IMUL	r,m	1	1	1	1	1	1			
IMUL	r16,m16,i	2	2		1	1	1			
IMUL	r32,m32,i	1	1	1	1	1	1			
IMUL	r64,m64,i	1	1	1	1	1	1			
DIV	r8	10	10				20-24	11-14		
DIV	r16	11	11				21-25	11-14		
DIV	r32	10	10				20-28	11-18		
DIV	r64	34-56					30-94	22-76		
IDIV	r8	10	10				21-24	11-14		
IDIV	r16	10	10				21-25	11-14		
IDIV	r32	9	9				20-27	11-18		
IDIV	r64	59-138					40-103	25-84		
CBW		1	1				1	0.5		
CWDE		1	1	1			1	1		
CDQE		1	1				1	0.5		
CWD		2	2				1	1		
CDQ		1	1				1	1		
CQO		1	1				1	0.5		
POPCNT	r,r	1	1	1			3	1	SSE4.2	
POPCNT	r,m	1	1	1	1			1	SSE4.2	
CRC32	r,r	1	1	1			3	1	SSE4.2	
CRC32	r,m	1	1	1	1			1	SSE4.2	
Logic instructions										
AND OR XOR	r,ri	1	1	x	x	x	1	0.33		
AND OR XOR	r,m	1	1	x	x	x	1	0.5		
AND OR XOR	m,ri	2	1	x	x	x	2	1		
XOR	r,same	1	0				0	0.25		
TEST	r,ri	1	1	x	x	x	1	0.33		
TEST	m,ri	1	1	x	x	x	1	0.5		
SHR SHL SAR	r,i	1	1	x			1	0.5		

Sandy Bridge										
SHR SHL SAR	m,i	3	1				2	1	1	2
SHR SHL SAR	r,d	3	3						2	2
SHR SHL SAR	m,d	5	3				2	1		4
ROR ROL	r,i	1	1						1	1
ROR ROL	m,i	4	3				2	1		2
ROR ROL	r,d	3	3						2	2
ROR ROL	m,d	5	3				2	1		4
RCR	r8,1	high							high	high
RCR	r16/32/64,1	3	3						2	2
RCR	r,i	8	8						5	5
RCR	m,i	11	7						6	6
RCR	r,d	8	8						5	5
RCR	m,d	11	7						6	6
RCL	r,i	3	3						2	2
RCL	r,i	8	8						6	6
RCL	m,i	11	7						6	6
RCL	r,d	8	8						6	6
RCL	m,d	11	7						6	6
SHRD SHLD	r,ri	1	1							0.5
SHRD SHLD	m,ri	3					2	1		2
SHRD SHLD	r,rd	4	4						2	2
SHRD SHLD	m,rd	5	3				2	1		4
BT	r,ri	1	1						1	0.5
BT	m,r	10	8				1			5
BT	m,i	2	1				1			0.5
BTR BTS BTC	r,ri	1	1						1	0.5
BTR BTS BTC	m,r	11	7				2	1		5
BTR BTS BTC	m,i	3	1				2	1		2
BSF BSR	r,r	1	1						3	1
BSF BSR	r,m	1	1			1	1			1
SETcc	r	1	1	x		x			1	0.5
SETcc	m	2	1	x		x	1	1		1
CLC		1	0							0.25
STC CMC		1	1	x		x			1	0.33
CLD STD		3	3							4
Control transfer instructions										
JMP	short/near	1	1			1			0	2
JMP	r	1	1			1			0	2
JMP	m	1	1			1	1		0	2
Conditional jump	short/near	1	1			1			0	1-2
Fused arithmetic and branch										
J(E)/CXZ	short	2	2	x		x	1			2-4
LOOP	short	7	7							5
LOOP(N)E	short	11	11							5
CALL	near	3	2			1	1	1		2
CALL	r	2	1			1	1	1		2

fastest if not jumping

Sandy Bridge									
FBSTP	m80	248						252	
FXCH	r	1	0				0	0.5	
FILD	m	1	1	1	1		8	1	
FIST(P)	m	3	1	1	1	1	7	2	
FISTTP	m	3	1	1	1	1	7	2	SSE3
FLDZ		1	1	1				2	
FLD1		2	2	1	1			2	
FLDPI FLDL2E etc.		2	2		2			2	
FCMOVoc	r	3	3				3	2	
FNSTSW	AX	2	2				2	1	
FNSTSW	m16	2	1			1	1	1	
FLDCW	m16	3	2				8		
FNSTCW	m16	2	1	1		1	1	5	1
FINCSTP FDECSTP		1	1	1			1	1	
FFREE(P)	r	1	1					1	
FNSAVE	m	143						166	
FRSTOR	m	90						165	
<b>Arithmetic instructions</b>									
FADD(P) FSUB(R)(P)	r	1	1	1			3	1	
FADD(P) FSUB(R)(P)	m	2	2	1	1			1	
FMUL(P)	r	1	1	1			5	1	
FMUL(P)	m	1	1	1		1		1	
FDIV(R)(P)	r	1	1	1			10-24	10-24	
FDIV(R)(P)	m	1	1	1		1		10-24	
FABS		1	1	1			1	1	
FCHS		1	1	1			1	1	
FCOM(P) FUCOM	r	1	1	1			3	1	
FCOM(P) FUCOM	m	1	1	1	1			1	
FCOMPP FUCOMPP		2	2	1	1			1	
FCOMI(P) FUCOMI(P)	r	3	3	1			4	1	
FIADD FISUB(R)	m	2	2	2	1			1	
FIMUL	m	2	2	1	1	1		1	
FIDIV(R)	m	2	2	1	1	1		1	
FICOM(P)	m	2	2	2	1			2	
FTST		1	1	1				1	
FXAM		2	2	1				2	
FPREM		28	28				21	21	
FPREM1		41-87					26-50	26-50	
FRNDINT		17	17				22		
<b>Math</b>									
FSCALE		27	27				12		
FXTRACT		17	17				10		
FSQRT		1	1	1			10-24		
FSIN		64-100					47-100		
FCOS		20-110					47-115		
FSINCOS		20-110					43-123		
F2XM1		53-118					61-69		
FYL2X		454	454				724		

Sandy Bridge									
FYL2XP1		484	484					726	
FPTAN		102	102					130	
FPATAN		28-91						93-146	
<b>Other</b>									
FNOP		1	1	1				1	
WAIT		2	2					1	
FNCLEX		5	5					22	
FNINIT		26	26					81	

# Integer MMX and XMM instructions

Instruction	Operands	uops fused domain	uops unfused domain						Latency	Reciprocal throughput	Comments
		p015	p0	p1	p5	p23	p4				
Move instructions											
MOVD	r32/64,(x)mm	1	1	x	x	x		1		0.33	
MOVD	m32/64,(x)mm	1					1	1	3	1	
MOVD	(x)mm,r32/64	1	1	x	x	x		1		0.33	
MOVD	(x)mm,m32/64	1					1		3	0.5	
MOVQ	(x)mm,(x)mm	1	1	x	x	x		1		0.33	
MOVQ	(x)mm,m64	1					1		1	0.5	
MOVQ	m64,(x)mm	1					1	1	3	1	
MOVQQA	x,x	1	1	x	x	x			1	0.33	
MOVQQA	x,m128	1					1		3	0.5	
MOVQQA	m128,x	1					1	1	3	1	
MOVQDU	x,m128	1	1				1		3	0.5	
MOVQDU	m128,x	1	1				1	1	3	1	
LDDQU	x,m128	1	1				1		3	0.5	
MOVQDQ2Q	mm,x	2	2						1	1	
MOVQDQ2Q	x,mm	1	1						1	0.33	
MOVNTQ	m64,mm	1					1	1	~300	1	
MOVNTDQ	m128,x	1					1	1	~300		
MOVNTDQA	x,m128	1					1			0.5	
PACKSSWB/DW PACK-USWB	mm,mm	1	1						1	1	
PACKSSWB/DW PACK-USWB	mm,m64	1	1		1		1				
PACKSSWB/DW PACK-USWB	x,x	1	1	x		x			1	0.5	
PACKSSWB/DW PACK-USWB	x,m128	1	1	x		x	1			0.5	
PACKUSDW	x,x	1	1	x		x			1	0.5	
PACKUSDW	x,m	1	1	x		x	1			0.5	
PUNPCKH/LBW/WD/DQ	(x)mm,(x)mm	1	1	x		x			1	0.5	
PUNPCKH/LBW/WD/DQ	(x)mm,m	1	1	x		x	1			0.5	
PUNPCKH/LQDQ	x,x	1	1	x		x			1	0.5	
PUNPCKH/LQDQ	x,m128	2	1	x		x	1			0.5	
PMOVSX/ZXBW	x,x	1	1	x		x			1	0.5	
PMOVSX/ZXBW	x,m64	1	1	x		x	1			0.5	

Sandy Bridge									
PMOVSX/ZXBD	x,x	1	1	x	x	1	0.5	SSE4.1	
PMOVSX/ZXBD	x,m32	1	1	x	x	1	0.5	SSE4.1	
PMOVSX/ZXBQ	x,x	1	1	x	x	1	0.5	SSE4.1	
PMOVSX/ZXBQ	x,m16	1	1	x	x	1	0.5	SSE4.1	
PMOVSX/ZXWD	x,x	1	1	x	x	1	0.5	SSE4.1	
PMOVSX/ZXWD	x,m64	1	1	x	x	1	0.5	SSE4.1	
PMOVSX/ZXWQ	x,x	1	1	x	x	1	0.5	SSE4.1	
PMOVSX/ZXWQ	x,m32	1	1	x	x	1	0.5	SSE4.1	
PMOVSX/ZXDQ	x,x	1	1	x	x	1	0.5	SSE4.1	
PMOVSX/ZXDQ	x,m64	1	1	x	x	1	0.5	SSE4.1	
PSHUFb	(x)mm,(x)mm	1	1	x	x	1	0.5	SSSE3	
PSHUFb	(x)mm,m	2	1	x	x	1	0.5	SSSE3	
PSHUFw	mm,mm,i	1	1	x	x	1	0.5		
PSHUFw	mm,m64,i	2	1	x	x	1	0.5		
PSHUFd	xmm,x,i	1	1	x	x	1	0.5		
PSHUFd	x,m128,i	2	1	x	x	1	0.5		
PSHUFb/HW	x,x,i	1	1	x	x	1	0.5		
PSHUFb/HW	x,m128,i	2	1	x	x	1	0.5		
PALIGNr	(x)mm,(x)mm,i	1	1	x	x	1	0.5	SSSE3	
PALIGNr	(x)mm,m,i	2	1	x	x	1	0.5	SSSE3	
PBLENDB	x,x,xmm0	2	2	1	1	2	1	SSE4.1	
PBLENDB	x,m,xmm0	3	2	1	1	1	1	SSE4.1	
PBLENDB	x,x,i	1	1	x	x	1	0.5	SSE4.1	
PBLENDB	x,m,i	2	1	x	x	1	0.5	SSE4.1	
MASKMOVQ	mm,mm	4	1	1	2	1	1		
MASKMOVQDU	x,x	10	4	1	4	x	6		
PMOVMsKB	r32,(x)mm	1	1	1	2	1	1		
PEXTRB	r32,x,i	2	2	x	x	2	1	SSE4.1	
PEXTRB	m8,x,i	2	1	x	x	1	1	SSE4.1	
PEXTRw	r32,(x)mm,i	2	2	x	x	2	1		
PEXTRw	m16,(x)mm,i	2	1	x	x	1	1	2	SSE4.1
PEXTRD	r32,x,i	2	2	x	x	2	1	SSE4.1	
PEXTRD	m32,x,i	3	2	x	x	1	1	1	SSE4.1
PEXTRD	r64,x,i	2	2	x	x	2	1	SSE4.1, 64b	
PEXTRQ	m64,x,i	3	2	x	x	1	1	1	
PINSRB	x,r32,i	2	2	x	x	2	1	SSE4.1	
PINSRB	x,m8,i	2	1	x	x	1	0.5	SSE4.1	
PINSRW	(x)mm,r32,i	2	2	x	x	2	1		
PINSRW	(x)mm,m16,i	2	1	x	x	1	0.5		
PINSRD	x,r32,i	2	2	x	x	2	1	SSE4.1	
PINSRD	x,m32,i	2	1	x	x	1	0.5	SSE4.1	
PINSRQ	x,r64,i	2	2	x	x	2	1	SSE4.1, 64b	
PINSRQ	x,m64,i	2	1	x	x	1	0.5		
Arithmetic instructions									
PADD/SUB(U,S)/B/W/D/Q	(x)mm, (x)mm	1	1	x	x	1	0.5		
PADD/SUB(U,S)/B/W/D/Q	(x)mm,m	1	1	x	x	1	0.5		
PHADD/SUB(S)/W/D	(x)mm, (x)mm	3	3	x	x	2	1.5	SSSE3	
PHADD/SUB(S)/W/D	(x)mm,m64	4	3	x	x	1	1.5	SSSE3	

Sandy Bridge									
PCMPEQ/GB/W/D	(x)mm,m	1	1	x	x	1	0.5		
PCMPEQQ	x,x	1	1	x	x	1	0.5	SSE4.1	
PCMPEQQ	x,m128	1	1	x	x	1	0.5	SSE4.1	
PCMPGTQ	x,x	1	1	1	1	5	1	SSE4.2	
PCMPGTQ	x,m128	1	1	1	1	1	1	SSE4.2	
PSUBxx, PCMPGTx	x,same	1	0			0	0.25		
PCMPEQx	x,same	1	1			0	0.5		
PMULL/HW PMULHUW	(x)mm,(x)mm	1	1		1	5	1		
PMULL/HW PMULHUW	(x)mm,m	1	1		1	1	1		
PMULHRSW	(x)mm,(x)mm	1	1	1		5	1	SSSE3	
PMULHRSW	(x)mm,m	1	1	1		1	1	SSSE3	
PMULLD	x,x	1	1	1		5	1	SSE4.1	
PMULLD	x,m128	2	1	1		1	1	SSE4.1	
PMULDQ	x,x	1	1	1		5	1	SSE4.1	
PMULDQ	x,m128	1	1	1		1	1	SSE4.1	
PMULUDQ	(x)mm,(x)mm	1	1	1		5	1		
PMULUDQ	(x)mm,m	1	1	1		1	1		
PMADDWD	(x)mm,(x)mm	1	1	1		5	1		
PMADDWD	(x)mm,m	1	1	1		1	1		
PMADDUBSW	(x)mm,(x)mm	1	1	1		5	1	SSSE3	
PMADDUBSW	(x)mm,m	1	1	1		1	1	SSSE3	
PAVGB/W	(x)mm,(x)mm	1	1	x	x	1	0.5		
PAVGB/W	(x)mm,m	1	1	x	x	1	0.5		
PMINMAXSB	x,x	1	1	x	x	1	0.5	SSE4.1	
PMINMAXSB	x,m128	1	1	x	x	1	0.5	SSE4.1	
PMINMAXUB	(x)mm,(x)mm	1	1	x	x	1	0.5		
PMINMAXUB	(x)mm,m	1	1	x	x	1	0.5		
PMINMAXSW	(x)mm,(x)mm	1	1	x	x	1	0.5		
PMINMAXSW	(x)mm,m	1	1	x	x	1	0.5		
PMINMAXUW	x,x	1	1	x	x	1	0.5	SSE4.1	
PMINMAXUW	x,m	1	1	x	x	1	0.5	SSE4.1	
PMINMAXU/SD	x,x	1	1	x	x	1	0.5	SSE4.1	
PMINMAXU/SD	x,m128	1	1	x	x	1	0.5	SSE4.1	
PHMINPOSUW	x,x	1	1	1		5	1	SSE4.1	
PHMINPOSUW	x,m128	1	1	1		1	1	SSE4.1	
PABSB/W/D	(x)mm,(x)mm	1	1	x	x	1	0.5	SSSE3	
PABSB/W/D	(x)mm,m	1	1	x	x	1	0.5	SSSE3	
PSIGNB/W/D	(x)mm,(x)mm	1	1	x	x	1	0.5	SSSE3	
PSIGNB/W/D	(x)mm,m	1	1	x	x	1	0.5	SSSE3	
PSADBw	(x)mm,(x)mm	1	1	1		5	1		
PSADBw	(x)mm,m	1	1	1		1	1		
MPSADBw	x,x,i	3	3			6	1	SSE4.1	
MPSADBw	x,m,i	4	3			1	1	SSE4.1	
only in some processors									
PCLMULQDQ	x,x,i	18	18			14	8		
AESDEC, AESDECLAST, AESENC, AESENC									



Sandy Bridge										
AESIMC	x,x	2	2					2	2	do.
AESKEYGENASSIST	x,x,i	11	11					8	8	do.
<b>Logic instructions</b>										
PAND(N) POR PXOR	(x)mm,(x)mm	1	1	x	x	x		1	0.33	
PAND(N) POR PXOR	(x)mm,m	1	1	x	x	x	1		0.5	
PXOR	x,same	1	0					0	0.25	
PTEST	x,x	1	1					1	1	SSE4.1
PTEST	x,m128	1	1				1		1	SSE4.1
PSLL/RL/RAW/D/Q	mm,mm/i	1	1		1			1	1	
PSLL/RL/RAW/D/Q	mm,m64	1	1		1		1		2	
PSLL/RL/RAW/D/Q	xmm,i	1	1		1			1	1	
PSLL/RL/RAW/D/Q	x,x	2	2					2	1	
PSLL/RL/RAW/D/Q	x,m128	3	2				1		1	
PSLL/RLDQ	x,i	1	1					1	1	
<b>String instructions</b>										
PCMPSTR	x,x,i	8	8					4	4	SSE4.2
PCMPSTR	x,m128,i	8	7			1		4	4	SSE4.2
PCMPSTRM	x,x,i	8	8					11-12	4	SSE4.2
PCMPSTRM	x,m128,i	8	7			1		4	4	SSE4.2
PCMPISTR	x,x,i	3	3					3	3	SSE4.2
PCMPISTR	x,m128,i	4	3			1		3	3	SSE4.2
PCMPISTRM	x,x,i	3	3					11	3	SSE4.2
PCMPISTRM	x,m128,i	4	3			1		3	3	SSE4.2
<b>Other</b>										
EMMS		31	31						18	

#### Floating point XMM and YMM instructions

Instruction	Operands	uops fused domain	uops unfused domain				Latency	Reciprocal throughput	Comments
			p0	p1	p2	p3	p4		
<b>Move instructions</b>									
MOVAPS/D	x,x	1	1					1	
MOVAPS/D	y,y	1	1					1	
MOVAPS/D MOVUPS/D	x,m128	1				1		3	0.5
MOVAPS/D	y,m256	1				1+		4	
MOVAPS/D MOVUPS/D	m128,x	1				1	1	3	
MOVAPS/D	m256,y	1				1	1+	3	
MOVSS/D	x,x	1	1					1	
MOVSS/D	x,m32/64	1				1		3	0.5
MOVSS/D	m32/64,x	1				1	1	3	
MOVHPS/D MOVLPS/D	x,m64	1	1			1		3	
MOVHPS/D	m64,x	1	1			1	1	3	
MOVLPS MOVHPS	x,x	1	1			1		1	

Sandy Bridge										
MOVMSKPS/D	r32,x	1	1	1				2	1	
MOVMSKPS/D	r32,y	1	1	1				2	1	
MOVNTPS/D	m128,x	1						1	1	
MOVNTPS/D	m256,y	1						1	4	~300
SHUFFPS/D	x,x,i	1	1					1	1	AVX
SHUFFPS/D	x,m128,i	2	1			1	1	1	1	
VSHUFFPS/D	y,y,i	1	1			1		1	1	AVX
VSHUFFPS/D	y,y,m256,i	2	1			1	1+	1	1	AVX
VPERMILPS/PD	x,x,i	1	1			1		1	1	AVX
VPERMILPS/PD	y,y,i	1	1			1		1	1	AVX
VPERMILPS/PD	x,x,m	2	1			1	1	1	1	AVX
VPERMILPS/PD	y,y,m	2	1			1	1+	1	1	AVX
VPERMILPS/PD	x,m,i	2	1			1	1	1	1	AVX
VPERMILPS/PD	y,m,i	2	1			1	1+	1	1	AVX
VPERM2F128	y,y,i	1	1			1		2	1	AVX
VPERM2F128	y,y,m,i	2	1			1	1+	1	1	AVX
BLENDPS/PD	x,x,i	1	1			1		1	0.5	SSE4.1
BLENDPS/PD	x,m128,i	2	1			1	1	1	0.5	SSE4.1
BLENDPS/PD	y,y,i	1	1			1		1	1	AVX
BLENDPS/PD	y,m256,i	2	1			1	1+	1	1	AVX
BLENDVPS/PD	x,x,xmm0	2	2			2		2	1	SSE4.1
BLENDVPS/PD	x,m,xmm0	3	2			2	1	1	1	SSE4.1
BLENDVPS/PD	y,y,y	2	2			2		2	1	AVX
BLENDVPS/PD	y,y,m,y	3	2			2	1+	1	1	AVX
MOVDDUP	x,x	1	1			1		1	1	SSE3
MOVDDUP	x,m64	1				1		3	0.5	SSE3
MOVDDUP	y,y	1	1			1		1	1	AVX
MOVDDUP	y,m256	1				1+		3	1	AVX
VBROADCASTSS	x,m32	1				1		1	1	AVX
VBROADCASTSS	y,m32	2	1			1	1	1	1	AVX
VBROADCASTSD	y,m64	2	1			1	1	1	1	AVX
VBROADCASTF128	y,m128	2	1			1	1	1	1	AVX
MOVSHLDUP	x,x	1	1			1		1	1	SSE3
MOVSHLDUP	x,m128	1				1		3	0.5	SSE3
MOVSHLDUP	y,y	1	1			1		1	1	AVX
MOVSHLDUP	y,m256	1				1+		4	1	AVX
UNPCKH/LPS/D	x,x	1	1			1		1	1	SSE3
UNPCKH/LPS/D	x,m128	1	1			1	1	1	1	SSE3
UNPCKH/LPS/D	y,y,y	1	1			1		1	1	AVX
UNPCKH/LPS/D	y,y,m256	1	1			1	1+	1	1	AVX
EXTRACTPS	r32,x,i	2	2			1		2	1	SSE4.1
EXTRACTPS	m32,x,i	3	2			1	1	1	1	SSE4.1
VEXTRACTF128	x,y,i	1	1			1		2	1	AVX
VEXTRACTF128	m128,y,i	2	1			1	1	1	1	AVX
INSERTPS	x,x,i	1	1			1		1	1	SSE4.1
INSERTPS	x,m32,i	2	1			1	1	1	1	SSE4.1
VINSERTF128	y,y,x,i	1	1			1		2	1	AVX
VINSERTF128	y,y,m128,i	2	1			1	1	1	1	AVX
VMASKMOVPS/D	x,x,m128	3	2			1		1	1	AVX
VMASKMOVPS/D	y,y,m256	3	2			1+		1	1	AVX

Sandy Bridge										
VMASKMOVPS/D	m128,x,x	4	2		1	1		1	AVX	
VMASKMOVPS/D	m256,y,y	4	2		1	1+		2	AVX	
Conversion										
CVTPD2PS	x,x	2	2	1	1			3	1	
CVTPD2PS	x,m128	2	2	1	1			1		
VCVTPD2PS	x,y	2	2	1	1			4	1	AVX
VCVTPD2PS	x,m256	2	2	1	1+			1		AVX
CVTSD2SS	x,x	2	2	1	1			3	1	
CVTSD2SS	x,m64	2	2	1	1			1		
CVTSP2PD	x,x	2	2	1	1			3	1	
CVTSP2PD	x,m64	2	2	1	1			1		
VCVTPS2PD	y,x	2	2	1	1			4	1	AVX
VCVTPS2PD	y,m128	3	3	1	1			1		AVX
CVTSS2SD	x,x	2	2	1	1			3	1	
CVTSS2SD	x,m32	2	1	1	1			1		
CVTDQ2PS	x,x	1	1	1	1			3	1	
CVTDQ2PS	x,m128	1	1	1	1			1		
VCVTDQ2PS	y,y	1	1	1	1			3	1	AVX
VCVTDQ2PS	y,m256	1	1	1	1+			1		AVX
CVT(T)PS2DQ	x,x	1	1	1	1			3	1	
CVT(T)PS2DQ	x,m128	1	1	1	1			1		
VCVT(T)PS2DQ	y,y	1	1	1	1			3	1	AVX
VCVT(T)PS2DQ	y,m256	1	1	1	1+			1		AVX
CVTDQ2PD	x,x	2	2	1	1			4	1	
CVTDQ2PD	x,m64	2	2	1	1			1		
VCVTDQ2PD	y,x	2	2	1	1			5	1	AVX
VCVTDQ2PD	y,m128	3	2	1	1			1		AVX
CVT(T)PD2DQ	x,x	2	2	1	1			4	1	
CVT(T)PD2DQ	x,m128	2	2	1	1			1		
VCVT(T)PD2DQ	x,y	2	2	1	1			5	1	AVX
VCVT(T)PD2DQ	x,m256	2	2	1	1+			1		AVX
CVTPI2PS	x,mm	1	1	1	1			4	2	
CVTPI2PS	x,m64	1	1	1	1			2		
CVT(T)PS2PI	mm,x	2	2	1	1			4	1	
CVT(T)PS2PI	mm,m128	2	1	1	1			1		
CVTPI2PD	x,mm	2	2	1	1			4	1	
CVTPI2PD	x,m64	2	2	1	1			1		
CVT(T)PD2PI	mm,x	2	2	1	1			4	1	
CVT(T)PD2PI	mm,m128	2	2	1	1			1		
CVTSI2SS	x,r32	2	2	1	1			4	1.5	
CVTSI2SS	x,m32	1	1	1	1			1		1.5
CVT(T)SS2SI	r32,x	2	2	1	1			4	1	
CVT(T)SS2SI	r32,m32	2	2	1	1			1		
CVTSI2SD	x,r32	2	2	1	1			4	1.5	
CVTSI2SD	x,m32	1	1	1	1			1		1.5
CVT(T)SD2SI	r32,x	2	2	1	1			4	1	
CVT(T)SD2SI	r32,m64	2	2	1	1			1		

Sandy Bridge										
ADDSS/D SUBSS/D	x,x	1	1	1				3	1	
ADDSS/D SUBSS/D	x,m32/64	1	1	1	1			1		
ADDPs/D SUBPs/D	x,x	1	1	1				3	1	
ADDPs/D SUBPs/D	x,m128	1	1	1	1			1		
VADDPs/D VSUBPs/D	y,y,y	1	1	1				3	1	AVX
VADDPs/D VSUBPs/D	y,y,m256	1	1	1	1+			1		AVX
ADDSUBPs/D	x,x	1	1	1				3	1	SSE3
ADDSUBPs/D	x,m128	1	1	1	1			1		SSE3
VADDSUBPs/D	y,y,y	1	1	1				3	1	AVX
VADDSUBPs/D	y,y,m256	1	1	1	1+			1		AVX
HADDPs/D HSUBPs/D	x,x	3	3	1	2			5	2	SSE3
HADDPs/D HSUBPs/D	x,m128	4	3	1	2	1		2		SSE3
VHADDPs/D	y,y,y	3	3	1	2			5	2	AVX
VHADDPs/D	y,y,m256	4	3	1	2	1+		2		AVX
MULSS MULPS	x,x	1	1	1				5	1	
MULSS MULPS	x,m	1	1	1	1			1		
VMULPS	y,y,y	1	1	1				5	1	AVX
VMULPS	y,y,m256	1	1	1	1+			1		AVX
MULSD MULPD	x,x	1	1	1				5	1	
MULSD MULPD	x,m	1	1	1	1			1		
VMULPD	y,y,y	1	1	1				5	1	AVX
VMULPD	y,y,m256	1	1	1	1+			1		AVX
DIVSS DIVPS	x,x	1	1	1				10-14	10-14	
DIVSS DIVPS	x,m	1	1	1	1			1		
VDIVPS	y,y,y	3	3	2	1			21-29	20-28	AVX
VDIVPS	y,y,m256	4	3	2	1	1+		20-28	20-28	AVX
DIVSD DIVPD	x,x	1	1	1				10-22	10-22	
DIVSD DIVPD	x,m	1	1	1	1			1		
VDIVPD	y,y,y	3	3	2	1			21-45	20-44	AVX
VDIVPD	y,y,m256	4	3	2	1+			20-44	20-44	AVX
RCPSS/PS	x,x	1	1	1				5	1	
RCPSS/PS	x,m128	1	1	1	1			1		
VRCPPS	y,y	2	3					7	2	AVX
VRCPPS	y,m256	4	3		1+			2		AVX
CMPPSS/D CMPPSS/D	x,x	1	1	1				3	1	
CMPPSS/D CMPPSS/D	x,m128	2	1	1	1			1		
VCMPSS/D	y,y,y	1	1	1				3	1	AVX
VCMPSS/D	y,y,m256	2	1	1	1+			1		AVX
COMISS/D UCOMISS/D	x,x	2	2					2	1	
COMISS/D UCOMISS/D	x,m32/64	2	2	1	1			1		
MAXSS/D MINSS/D	x,x	1	1	1				3	1	
MAXSS/D MINSS/D	x,m32/64	1	1	1	1			1		
MAXPS/D MINPS/D	x,x	1	1	1				3	1	
MAXPS/D MINPS/D	x,m128	1	1	1	1			1		
VMAXPS/D VMINPS/D	y,y,y	1	1	1				3	1	AVX
VMAXPS/D VMINPS/D	y,y,m256	1	1	1	1+			1		AVX

Sandy Bridge										
ROUNDSS/SD/PS/PS	x,m128,i	2	1	1	1		1	SSE4.1		
VROUNDSS/SD/PS/PS	y,y,i	1	1	1		3	1	AVX		
VROUNDSS/SD/PS/PS	y,m256,i	2	1	1	1+		1	AVX		
DPPS	x,x,i	4	4	1	2	12	2	SSE4.1		
DPPS	x,m128,i	6	5				4	SSE4.1		
VDPPS	y,y,y,i	4	4			12	2	AVX		
VDPPS	y,m256,i	6	5		1+		4	AVX		
DPPD	x,x,i	3	3			9	2	SSE4.1		
DPPD	x,m128,i	4	3		1		2	SSE4.1		
Math										
SQRTSS/PS	x,x	1	1	1		10-14	10-14			
SQRTSS/PS	x,m128	1	1	1	1		10-14			
VSQRTPS	y,y	3	3				21-28	AVX		
VSQRTPS	y,m256	4	3		1+		21-28	AVX		
SQRTSD/PS	x,x	1	1	1		10-21	10-21			
SQRTSD/PS	x,m128	2	1	1	1		10-21			
VSQRTPD	y,y	3	3			21-43	21-43	AVX		
VSQRTPD	y,m256	4	3		1+		21-43	AVX		
RSQRTSS/PS	x,x	1	1	1		5	1			
RSQRTSS/PS	x,m128	1	1	1	1		1			
VRSQRTPS	y,y	3	3			7	2	AVX		
VRSQRTPS	y,m256	4	3		1+		2	AVX		
Logic										
AND/ANDN/OR/XORPS/PS	x,x	1	1		1	1	1			
AND/ANDN/OR/XORPS/PS	x,m128	1	1		1	1	1			
VAND/ANDN/OR/XORPS/PS	y,y,y	1	1		1	1	1	AVX		
VAND/ANDN/OR/XORPS/PS	y,y,m256	1	1		1	1+	1	AVX		
(V)XORPS/PS	x/y,x/y,same	1	0			0	0.25			
Other										
VZEROUPPER		4				2	1	AVX		
VZEROALL		12					11	AVX, 32 bit		
VZEROALL		20					9	AVX, 64 bit		
LDMXCSR	m32	3	3		1	3	3			
STMXCSR	m32	3	3		1	1	1			
VSTMXCSR	m32	3	3		1	1	1	AVX		
FXSAVE	m4096	130					68			
FXRSTOR	m4096	116					72			
XSAVEOPT	m	100-168				60-500				

## GLOSSAIRE

### A

- **ABI** : Application Binary Interface
- **ADC** : Analog to Digital Converter
- **ALU** : Arithmetic and Logical Unit
- **AMD** : Advanced Micro Devices
- **ANSI** : American National Standards Institute
- **API** : Application Programming Interface
- **APU** : Accelerated Processor Unit
- **ARM** : société anglaise proposant des architectures CPU RISC 32bits
- **ASCII** : American Standar Code for Information Interchange

### B

- **BP** : Base Pointer
- **BSL** : Board Support Library

### C

- **CCS** : Code Composer Studio
- **CEM** : Compatibilité ElectroMagnétique
- **CISC** : Complex Instruction Set Computer
- **CPU** : Central Processing Unit
- **CSL** : Chip Support Library

### D

- **DAC** : Digital to Analog Converter
- **DDR** : Double Data Rate
- **DDR SDRAM** : Double Data Rate Synchronous Dynamic Random Access Memory
- **DMA** : Direct Memory Access
- **DSP** : Digital Signal Processor
- **DSP** : Digital Signal Processing

### E

- **EDMA** : Enhanced Direct Memory Access
- **EUSART** : Enhanced Universal Synchronous Asynchronous Receiver Transmitter
- **EMIF** : External Memory Interface
- **EPIC** : Explicitly Parallel Instruction Computing

## F

- **FPU** : Floating Point Unit
- **FLOPS** : Floating-Point Operations Per Second
- **FMA**: Fused Multiply-Add

## G

- **GCC** : Gnu Collection Compiler
- **GLCD** : Graphical Liquid Crytal Display
- **GNU** : GNU's Not UNIX
- **GPIO** : General Purpose Input Output
- **GPP** : General Purpose Processor
- **GPU** : Graphical Processing Unit

## I

- **IA-64** : Intel Architecture 64bits
- **I2C** : Inter Integrated Circuit
- **ICC** : Intel C++ Compiler
- **IDE** : Integrated Development Environment
- **IDMA** : Internal Direct memory Access
- **IRQ** : Interrupt ReQuest
- **ISR** : Interrupt Software Routine
- **ISR** : Interrupt Service Routine

## L

- **L1D** : Level 1 Data Memory
- **L1I** : Level 1 Instruction Memory (idem L1P)
- **L1P** : Level 1 Program Memory (idem L1I)
- **Lx** : Level x Memory
- **LCD** : Liquid Crytal Display
- **LRU** : Least Recently Used

## M

- **MAC**: Multiply Accumulate
- **MCU** : Micro Controller Unit
- **MIMD** : Multiple Instructions on Multiple Data
- **MIPS** : Mega Instructions Per Second

- **MMU** : Memory Managment Unit
- **MPLABX** : MicrochiP LABoratory 10, IDE Microchip
- **MPU** : Micro Processor Unit ou GPP
- **MPU** : Memory Protect Unit

## O

- **OS** : Operating System

## P

- **PC** : Program Counter
- **PC** : Personal Computer
- **PIC18** : Famille MCU 8bits Microchip
- **PLD** : Programmable Logic Device
- **POSIX** : Portable Operating System Interface, héritage d'UNIX (norme IEEE 1003)
- **PPC** : Power PC

## R

- **RAM** : Random Access Memory
- **RISC** : Reduced Instruction Set Computer
- **RS232** : Norme standardisant un protocole de communication série asynchrone
- **RTOS** : Real Time Operating System

## S

- **SDK** : Software Development Kit
- **SIMD** : Single Instruction Multiple Date
- **SOB** : System On Board
- **SOC** : System On Chip
- **SOP** : Sums of products
- **SP** : Stack Pointer
- **SP** : Serial Port
- **SPI** : Serial Peripheral Interface
- **SRAM** : Static Random Access Memory
- **SSE** : Streaming SIMD Extensions
- **STM32** : STMicroelectronics 32bits MCU

## T

- **TI** : Texas Instruments
- **TNS** : Traitement Numérique du Signal



- **TSC** : Time Stamp Counter
- **TTM** : Time To Market

U

---

- **UART** : Universal Asynchronous Receiver Transmitter
- **USB** : Universal Serial Bus

V

---

- **VHDL** : VHSIC Hardware Description langage
- **VHSIC** : Very High Speed Integrated Circuit
- **VLIW** : Very Long Intruction Word