Assembleur

Malek.Bengougam@gmail.com

Rappel: numération

- **Chiffres**: ensemble des valeurs numériques dénombrables.
  - 0 à 1 en binaire, 0 à 7 en octal, 0 à 9 en décimal, 0 à 15 en hexadécimal etc..

- **Nombre**: Chaque chiffre dans un nombre représente un multiple de la base. En décimal (base 10) nous avons les unités (0 à 9), les dizaines, les centaines, les milliers etc…

- Il s'agit en fait des puissances successives de 10: $10^0 = 1$, $10^1 = 10$, $10^2 = 100$, $10^3 = 1000$ …

- 1978 est ainsi $1*10^3 + 9*10^2 + 7*10^1 + 8*10^0$.

Rappel : binaire

- Un chiffre binaire est appelé « **bit** » qui est la contraction de « *binary digit* ».

- Une valeur numérique peut être formée par la concaténation de bits successifs:
  - Un groupe de 8 bits est ainsi appelé **octet**.
  - Attention, le terme anglais « **byte** » (morceau) est souvent utilisé comme synonyme d'octet mais ce n'est pas forcément le cas sur toutes les architectures.

  - Vous allez parfois trouver le terme « **nibble** » qui représente généralement un demi octet soit un nombre de 4 bits.

Numération (2)

- Le **binaire** est un système dont la base est 2.
- Les nombres sont formés à partir de chiffres booléens dont les deux seules valeurs sont 0 et 1.
- On peut former n'importe quel nombre par la succession de valeurs binaires,
- ex: 5 en binaire se représente ainsi 101.
  - Soit $1*2^2 + 0*2^1 + 1*2^0 = 4 + 1 = 5$.
- Par la suite on distinguera les nombres binaires par un $_b$ final: 101_b

Numération (3)

- **Octal** : il s'agit de chiffres en base 8 (octo). 8 étant une puissance de 2 on peut aisément représenter un nombre octal par un groupe de 3 bits.

- **Hexadécimal** : un chiffre hexadécimal représente 16 valeurs différentes : 0 à 15.

- De manière analogue à l'octal, 16 est une puissance de 2 et on peut donc représenter un nombre hexadécimal par un nombre binaire composé de 4 bits, c'est donc un « nibble ».

  - De 0 à 9, les chiffres sont les mêmes qu'en décimal, ce qui facilite la lecture par rapport au binaire ou à l'octal.

  - De 10 à 15, un chiffre hexadécimal va de A à F, A valant 10 en décimal, B valant 11, etc… jusqu'à F valant 15.

ARCHITECTURE

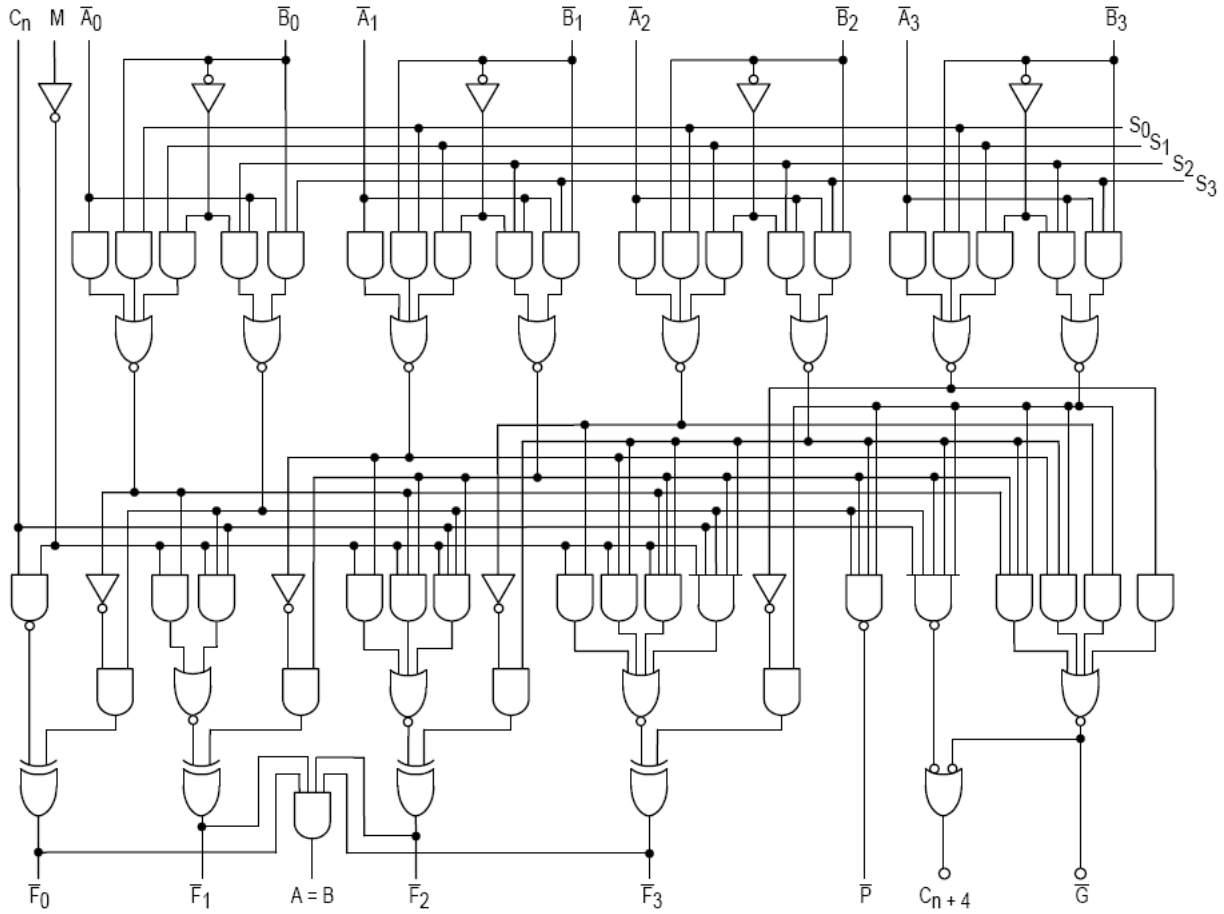
Microprocesseur

- A la base, les micro-processeurs sont des circuits intégrés. C'est-à-dire qu'ils sont un agrégats de transistors et de porte logiques (ET, OU, XOR, NOT etc…).

- C'est d'ailleurs également le cas pour la mémoire (bascules/latches) et autres stockages non mécanique/magnétique.

- Cependant ils sont au cœur d'une notion fondamentale: la possibilité d'exécuter un programme.
  - C'est l'idée principielle de Charles Babbage, d'Ada Lovelace, et des premiers calculateurs d'Alan Turing, Von Neumann etc…

Exemple d'une ALU 4 bit simplifiée



Code Opératoire

- Les instructions du CPU sont définies via un code binaire permettant d'identifier l'opération ainsi que les opérandes à utiliser.

- Ce Code Opératoire (*Operation Code* – ***Opcode***) correspond à la forme de programmation la plus bas niveau.

- Un exemple d'Opcode du Motorola 68000: **$1100\ 000\ 0\ 01\ 000\ 001_b$**

  - Les 4 bits de poids fort désignent l'instruction à proprement parler, ici, AND, opérateur logique.
  - Les 3 bits suivants indiquent le registre source, opérande de gauche, ici le registre D0.
  - Le bit suivant indique si l'instruction peut potentiellement écrire en mémoire (ici non).
  - Les 2 bits suivants précisent la taille des données.
    Pour les opérations non signées on a 00b=8-bit, 01b=16-bit, 10b=32-bit.
  - Finalement les deux groupes de 3 bits suivants indiquent l'endroit où sont stockées les données.
    Ici 000 indique qu'il s'agit d'un registre et 001 est le numéro du registre.

  On a donc l'opération suivante: D0 = D0 & D1, ce qui donne en assembleur 68000: **AND D0, D1**

Assembleur

- Aussi appelé langage d'assemblage (*assembly language*) car il sert à assembler facilement des instructions pour former un exécutable.

- Langage de très bas niveau, chaque instruction assembleur est directement liée à un OpCode particulier.

- Ces instructions sont appelées **mnémoniques**.

- Certaines mnémoniques assembleurs effectuent des opérations plus complexes que d'autres et génèrent en fait un groupe d'instructions. On dit alors que ces mnémoniques sont « micro-codées ».

L’assembleur comme langage

- Le premier langage de programmation fut celui développé en 1833 par **Ada Lovelace** afin de programmer *l’Analytic Engine* de **Charles Babbage**.

- La première machine programmable fut l’ENIAC, en 1945, qui se programmait en câblant manuellement les entrées/sorties.

- La première machine disposant d’un langage d’assemblage fut l’EDSAC en 1949.

- Le premier langage assembleur fut développé par Stan Poley en 1955 pour l’IBM 650.

Lexique des langages d'assemblage

- Une **instruction**: est une commande CPU primitive. L'ensemble des instructions (jeu d'instructions) défini une **ISA** (*Instruction Set Architecture*).
  - Les opérandes: paramètres d'une instruction
  - Les registres: les registres généraux (General Purpose Registers) servent de stockage temporaire
  - Les valeurs immédiates: valeur numérique directement stockée dans l'instruction.
- Les **mnémoniques**: ce sont les instructions du langage qui sont ensuite traduis en instruction CPU.
- Le **cycle** d'instruction: temps d'exécution d'une instruction par le CPU

Complex Instruction Set Computer (CISC)

- La plupart des processeurs des années 70 aux années 90 ont essayé de supporter un maximum de possibilités au niveau des instructions ce qui les rend très puissant, surtout avec le micro codage d'instructions.

- Cependant cela conduit à plusieurs formes de difficultés:
  - Les instructions sont complexes à décoder: en fonction du type d'opérande (immédiate, registre, mémoire…) une même instruction peut avoir plusieurs opcodes différents et s'exécuter en un nombre de cycle différent selon le cas.
  - Pour des raisons de gain de place en mémoire certaines instructions génèrent des opcodes avec des tailles différentes en fonction du type de donnée.

- Tout ceci peut sembler légitime mais le design des CPUs s'en trouve fortement complexifié et cela pose de nombreux problèmes au niveau du parallélisme des instructions (cf. pipeline et superscalaire).
- Les compilateurs de langages de haut niveau s'avèrent du coup plus difficile à écrire, notamment les optimisateurs de code.

Reduced Instruction Set Computer (RISC)

- Par opposition à CISC, une architecture RISC utilise un jeu d'instruction avec un code opératoire beaucoup plus simple et homogène.
  - Les instructions ont désormais la même taille en mémoire.

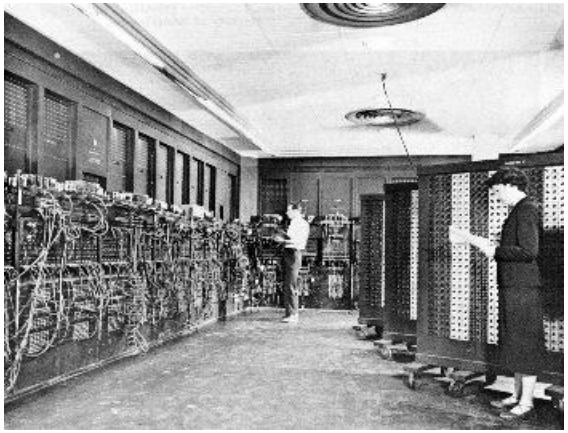
- La méthode RISC définie également des règles à suivre ou des composants qui doivent être obligatoirement présents.

- Certaines architectures modernes sont désormais hybrides.
  - Depuis le Pentium les architectures x86 ont conservé en apparence le même jeu d'instruction pour des questions de rétro compatibilité, mais en interne le fonctionnement est totalement RISC.

- Les architectures ARM offrent un mode de fonctionnement CISC (instructions THUMB sur 16 bits, plus compactes) et un mode de fonctionnement RISC.

ENIAC

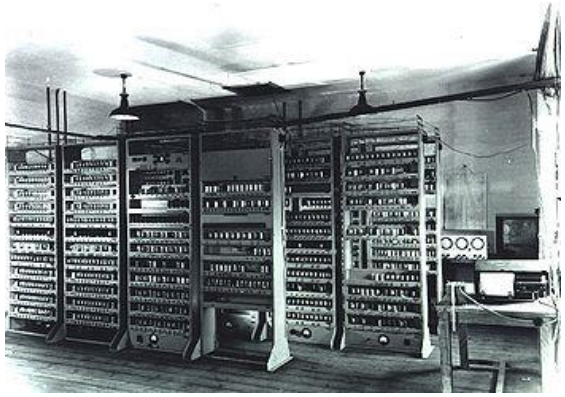
- Pour *Electronic Numerical Integrator And Computer* (1946-1955).
- Premier calculateur électronique à portée générale (Turing-complet).
- Initialement prévu pour le calcul des trajectoires de tirs d'artillerie, mais servi rapidement pour la recherche sur les armes thermo nucléaires.



- 1 cycle de 200 microsecondes soit 5000 cycles par secondes
  - $(1 / 5000 = 200*10^{-6})$

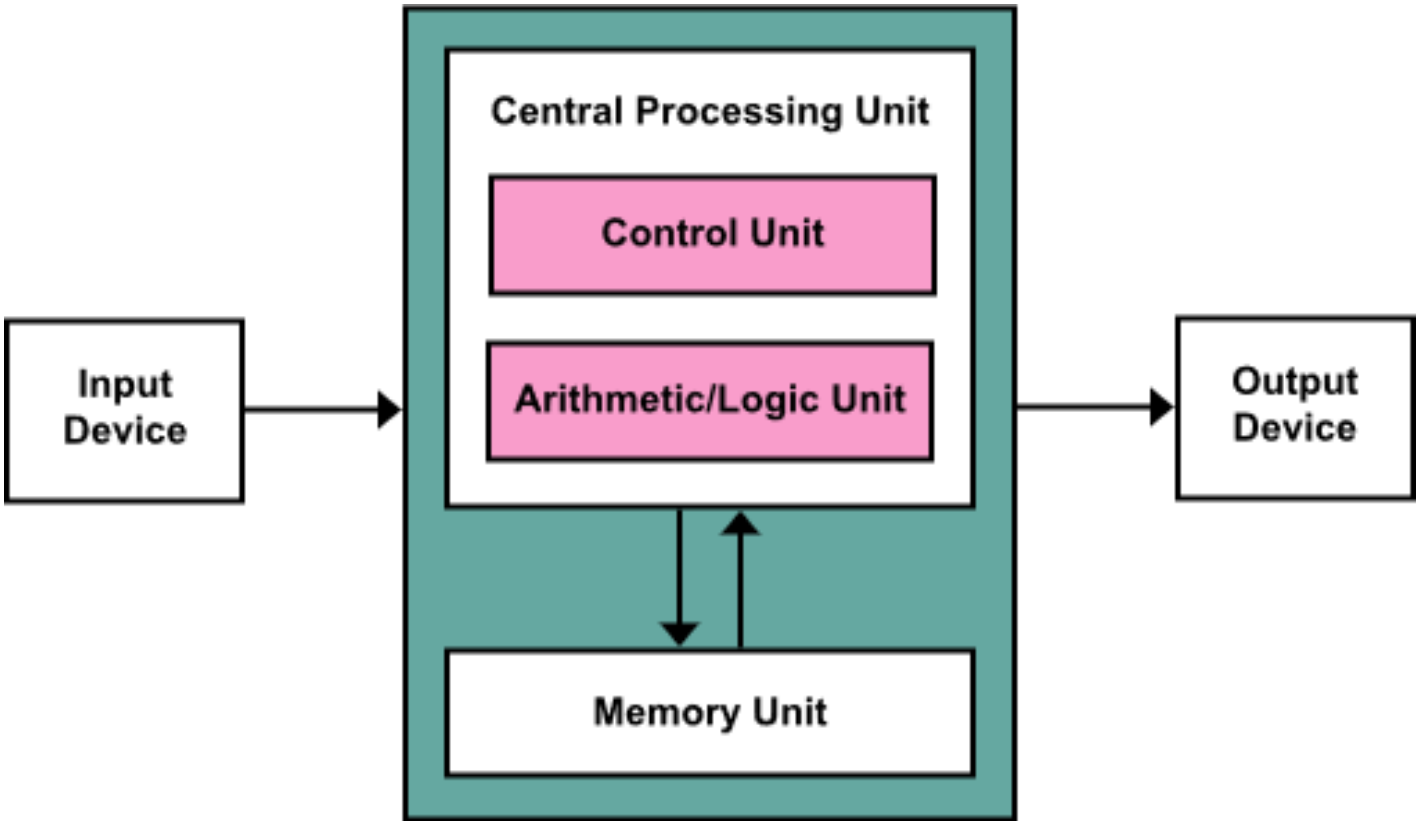
Stored-Program Concept

- Concepts issus de l'Universal Turing Machine, et l'Automatic Computing Engine et développés par Von Neumann.
  - Principale limite de l'ENIAC : la difficulté de programmation qui nécessite des manipulations de type branchement de cables etc..



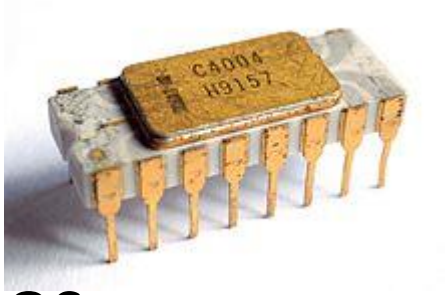
- On retrouve cette notion de programme stockable dans l'EDVAC premier ordinateur à utiliser des cartes perforées.
- Autre aspect important: la notion de « *Turing Completeness* » c'est-à-dire la possibilité d'exécuter un programme de manière séquentielle et structurée.
  - L'Analytic Engine de Charles Babbage aurait pu être *Turing Complete*

Architecture Von Neumann



Intel 4004

- Premier microprocesseur de l'histoire (1971)



- 2300 transistors MOS
- 16 registres 4 bits

- Largeur de bus adresse de 10 bits, 2^{10} = 1 kio
  - seuls 640 octets de mémoires sont adressables

- Equivalent d'un ENIAC (66 m^3) dans 10 mm^2

Mos 6502

- CPU 8-bit avec adressage 16-bit inspiré du Motorola 6800.
- Le 6502 est capable d'adresser une plage mémoire allant de l'octet 0 à l'octet 65535 (2^{16}-1).
- Il faut donc nécessairement deux octets pour définir (adresser) cette plage mémoire.
  - Toutes les machines ne disposaient pas pour autant de 64ko, mais le plus souvent 16ko ou 32ko, cependant cela nécessite tout de même 2 octets dans tous les cas de figure.
- La Famicom / NES utilise une version light du 6502 produit par Ricoh. La PC Engine de Nec et d'autres ordinateurs comme le C64 sont également architecturés autour d'un 6502, parfois customisé.

« Endianness » (1)

- Le 6502 étant un processeur 8 bit il ne peut manipuler qu'un octet à la fois sur les 2 octets qui composent l'adresse mémoire.

- Exemple : soit la variable DATA stockée en mémoire avec la valeur 0x1978 et représentant l'adresse d'un « sprite » à afficher.
- On a donc deux octets qui décrivent cette adresse mémoire : 0x19 et 0x78.

- Cependant il est possible de stocker ces deux octets de deux manière: soit l'octet de poids fort en premier, soit l'octet de poids faible en premier.

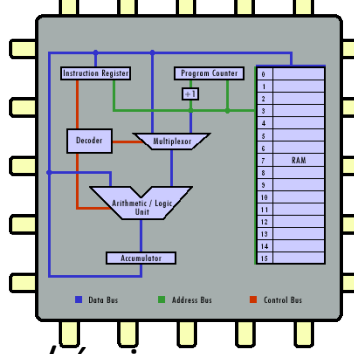
- La question de l'ordre des données stockées en mémoire est primordiale : doit on interpréter le premier octet comme étant l'octet de poids fort ou de poids faible ?

- En effet, si le processeur attend une donnée de poids faible en premier, et que le stockage est en poids fort en premier il interprétera l'adresse comme étant 0x7819 et non 0x1978 !

« Endianness » (2)

- On en déduit deux types de processeurs: ceux qui ont un fonctionnement en « **little-endian** » (par le petit bout) ou en « **big-endian** » (par le gros bout).
  - Cette terminologie est inspiré du livre satirique « **Les voyages de Gulliver** » qui décrit, entre autre, la guerre faisant rage dans le royaume de *Liliput* suite à un décret imposant de manger les œufs en commençant par le petit bout.
  - Les liliputiens rebelles au décret se nomment les « *big-endians* » (gros-boutiens) par opposition aux petits-boutiens (*little-endians*).
- Certains CPU sont Little-Endian (architectures x86/x64, …), d'autres Big-Endian (68000 et dérivés, SPARC, …) d'autres encore sont configurables (PowerPC, ARM, RISC) mais rarement à la volée.
  - La plupart des machines PowerPC (anciens Mac, Gamecube, Wii, PS3, Xbox 360 …) sont en Big-Endian, tandis que la plupart des machines ARM actuelles (mobiles, boxes etc…) sont en Little-Endian.

Quelques définitions



- Un CPU doit au minimum être capable d'exécuter des instructions et lire / écrire des données. C'est le rôle de l'unité de contrôle (***control unit - CU***)

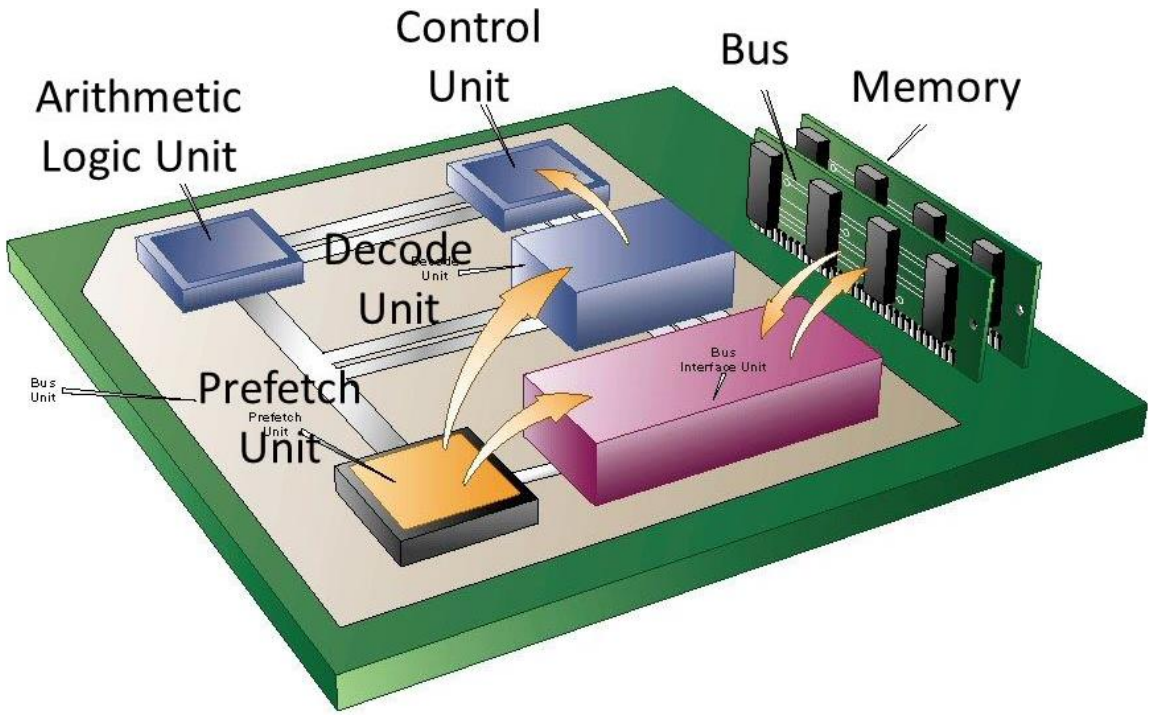
- Selon la taxonomie de Flynn, un processeur est dit **SISD** (*single instruction, single data*) lorsqu'une instruction n'affecte qu'une donnée à la fois

- Un CPU traite un flux d'instructions qu'il doit rapatrier (***fetch***) et interpréter (***decode***)
  - Les instructions d'un CPU sont de différents types : les macro-instructions sont composées de plusieurs micro-instructions (***µ-ops***) formant une **ISA** (*Instruction Set Architecture*)
    - Les µ_ops sont éventuellement subdivisées en nano-instructions.

- Ce flux (fetch->decode->execute->store) est appelé un **cycle**

- Les instructions ont des tailles différentes, il est courant qu'un CU charge plus de données que nécessaire, mais c'est volontaire. On appelle cela le ***prefetch***. Depuis la génération des processeurs 16-bits les CPU implémentent une ***prefetch queue***.

Une vision plus élargie



Bus

- Un bus permet de véhiculer des informations. Il est constitué de lignes de bus véhiculant chacun un bit.
- Plus le bus est large plus la bande passante est importante.

- Sur les architectures PC Intel des années 90 à nos jours il y'a plusieurs bus tel le « *Front-Side Bus* (FSB) » interfaçant le CPU avec un contrôleur mémoire appelé « *North Bridge* » et un autre contrôleur d'I/O appelé « *South Bridge* »
  - Le « *Back-Side Bus* » relie lui les caches et les cœurs.

- Le FSB agit sur la fréquence du CPU à travers un multiplicateur (par ex. fréquence CPU = 8 * fréquence FSB) ce qui ouvre la voie à l'overclocking / underclocking
  - De base, la fréquence du FSB devrait être celle de la mémoire, mais il existe également des mémoires au fonctionnement asynchrones

Fréquence du CPU

- La fréquence de fonctionnement d'un CPU indique le nombre de cycles, ou tic d'horloge, qu'il peut générer par seconde.
  - De nos jours, du fait de la complexité des architectures (pipeline, super-scalaire, multi-cœur, etc…) ce n'est pas un indicateur absolu de la puissance du CPU
  - Certains CPU, généralement embarqués ou prévu pour les pc portables, ajustent leur fréquence en fonction de la charge

- Le cycle d'horloge est généré par un circuit produisant une résonnance qui est ensuite amplifiée.
  - La résonnance est produite par un oscillateur piézo-électrique (qui transforme une impulsion en signal électrique) appelé cristal (souvent à base de Quartz)

Instructions par seconde

- Le nombres d'instructions par seconde (**IPS**) dépend tout à la fois de l'architecture interne (CISC, RISC, pipelines ..) et de la fréquence du CPU.
  - La comparaison en **MIPS** (millions d'IPS) d'une même architecture reste pertinente

- Il existe des méthodes tenant compte du fait qu'une architecture (RISC par ex.) nécessite plus d'instructions pour une tâche mais s'exécute plus rapidement.
  - Historiquement **Dhrystone** a été un instrument de mesure pionnier, mais son défaut est de ne pas représenter des calculs réels
  - **Geekbench** de nos jours est plus moderne et utilise plusieurs situation de code réels.
  - Cela dit, dans les deux cas, les effets de bord, notamment des caches d'instruction, ne permettent toujours pas une comparaison exacte.

Architecture 8086

INTEL X86

x86 – 16 bits

- Les registres du 80086 au 80286 sont sur 16 bits. Il y'a 3 classes de registres:

- Les registres standard: AX, BX, CX, DX
  - A = Accumulateur, B = Base, C = Compteur, D = Data
  - Ces registres peuvent être décomposés en deux sous registres 8 bits, high et low, ex: Ah et Al

- Les registres d'adressage: SI, DI, IP, BP, SP
  - SI = Source Index, DI = Destination Index,
  - IP = Instruction Pointer, BP = Base Pointer, SP = Stack Pointer (pile)

- Les registres de segment: CS, DS, SS, ES
  - CS = Code Segment, DS = Data Segment, SS = Stack Segment, E = Extra

Premières instructions

- Le CPU ne connait que l'OpCode de l'instruction.
  - Le langage assembleur est un langage bas niveau plus compréhensible pour le programmeur.
  - L'instruction **NOP** (No Operation) a pour OpCode 0x90
  - C'est en fait un alias de **XCHG (E)AX, (E)AX**
  - http://x86.renejeschke.de/html/file_module_x86_id_217.html

- Nous allons suivre la syntaxe Intel avec:
  - L'opérande source à droite, l'opérande destination à gauche
  - Le nom des registres tels que défini dans le manuel Intel (EAX, AX, AH, AL etc…)
  - '[]' pour déréférencer
  - Les instructions n'ont pas de suffixes (sauf dans certains cas)

- Ceci par opposition à la syntaxe AT&T utilisée par Gas (Gcc)
- http://www.imada.sdu.dk/Courses/DM18/Litteratur/IntelnATT.htm

Premières instructions

- **MOV** : permet de stocker une valeur définie dans l'opérande de droite (source) dans l'opérande de gauche (destination)

- MOV AX, 42
- MOV BL, AL

- **XOR** : effectue le Ou Exclusif des opérandes et stocke le résultat dans l'opérande de gauche.
  - Souvent utilisée pour mettre un registre à 0

- XOR AX, AX ; équivalent plus optimal de MOV AX, 0

Premières instructions

- **INC** / **DEC** permettent respectivement d'incrémenter / décrémenter la valeur d'un registre

- Elles sont similaires à **ADD** reg, 1 et **SUB** reg, 1

- INC AX ; similaire à ADD AX, 1
- DEC AX ; similaire à SUB AX, 1

- En pratique ADD et SUB sont plus optimales, et donc préférables à INC et DEC
  - Cependant l'opcode de INC est DEC est plus petit ce qui peut être utile si on optimise pour la taille

Segments

- Du fait que les premiers x86 (8088, 8086) étaient de processeurs 16 bits les registres ne peuvent qu'adresser 64kio (65536 octets).
  - Mais ces processeurs ont un bus d'adresse de 20 bits donc 1 Mio
- Un segment est un bloc de 64kio. Afin d'adresser toute la mémoire disponible les ingénieurs d'Intel ont mis en place l'astuce suivante:
  - Adresse = Segment * 16 + Décalage
- L'adresse mémoire est alors au maximum de 0xFFFF << 4 + 0xFFFF = 0xFFFF0 + 0xFFFF = 0X10FFEF
  - Ce qui est plus grand que 2^{10}-1 (1.062 Mio)

Adressage segmenté

- L'adresse d'un octet (byte) est composée de deux parties: le numéro de segment et un décalage (offset) par rapport au début du segment, ceci sous la forme

  Segment:Offset.

- Les registres de segment (CS, DS, SS, ES) permettent de spécifier les différentes zones mémoires correspondant au code, aux données ou à la pile.
- Les registres de déplacement (SI, DI, IP, BP, SP) servent à se placer en mémoire.

Mémoire adressable

- Le mode d'adressage standard du x86 est appelé « mode réel » (real mode)
- Le 80286 introduit le mode protégé (protected mode), extension du mode réel pour supporter un espace d'adressage virtuel avec un bus de 24 bits (16 Mio).
- En mode protégé les registres de segments sont interprétés comme des indices dans une table de descripteurs de segment (Look-up table).
  - Un descripteur de segment est une structure 64 bits stockant les informations relatives à un segment: adresse, taille, privilèges etc…
- Les processeurs Intel/AMD du Pentium à nos jours fonctionnent sur un modèle mémoire dit « flat model ».
- C'est-à-dire que contrairement aux processeurs 16 et 32 bits de la génération précédente, il n'est plus nécessaire de segmenter le code entre code, données, stack… on peut désormais accéder à l'ensemble de la mémoire de n'importe où.
- Note: encore de nos jours les (IBM) PC / Bios bootent en mode réel!

Architecture IA-32

INTEL X86 - 32 BITS

x86 – 32 bits

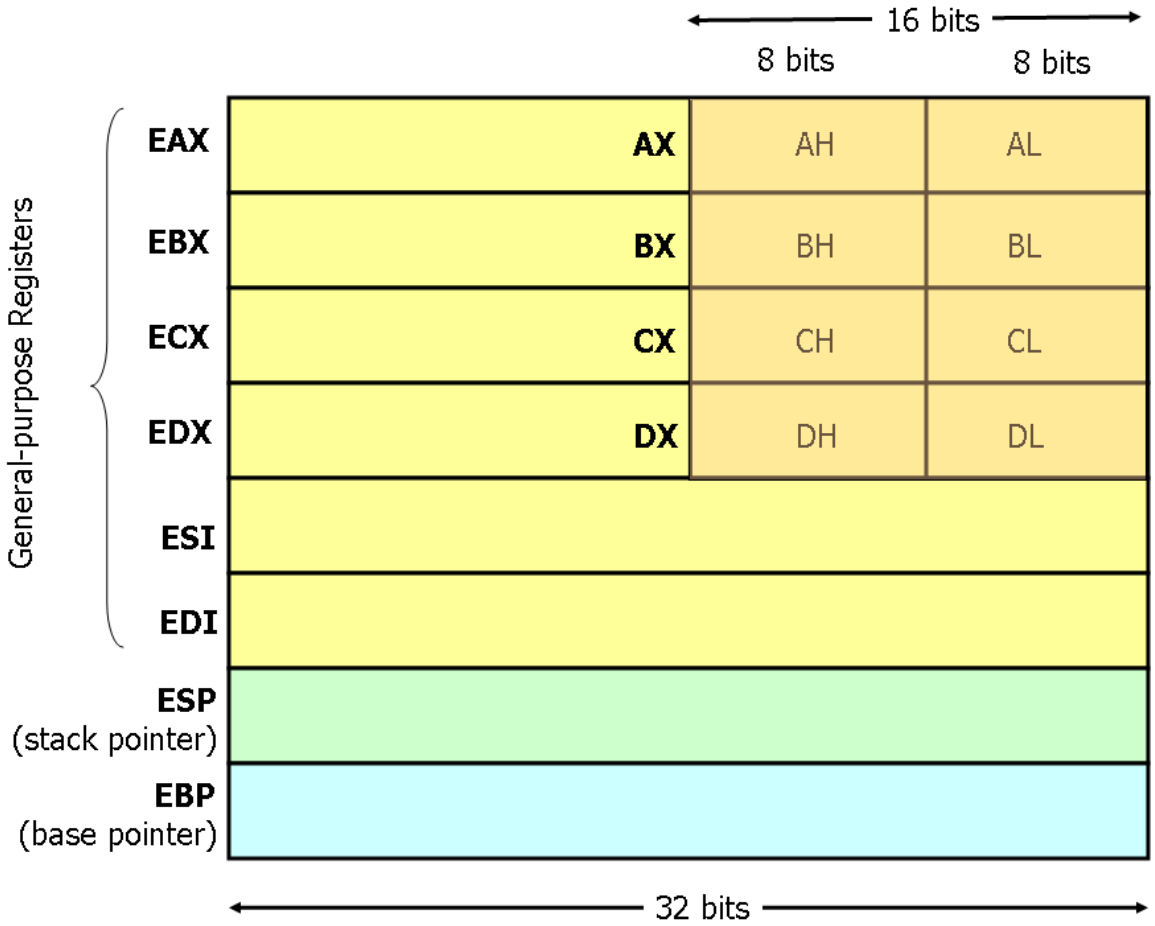
- Depuis le 80386 les registres ont été étendus à 32 bits, soit 4 octets.
  - Les registres 32 bits contiennent les registres 16 bits dans les bits de poids faible.

- Les noms des registres 32 bits sont préfixés par E pour « extended », çàd EAX, ECX, ESI, ESP, EIP etc…

- On peut désormais adresser jusqu'à 4Gio (2^{32} octets) de mémoire.

- De nouveaux registres de segments ont fait leur apparition avec le Pentium: FS et GS
  - Ils sont principalement utilisés en rapport avec le multithreading (thread local storage et pile locale)

Registres 32 bits



Dépréciation des anciens modèles

- Dans 99,99% des cas vous allez vous limiter au modèle mémoire ‘flat’.

- De ce fait les registres CS, DS, SS, ES voire FS sont totalement libres et ne servent plus de registres de segment.
  - Normalement forcés à zéro par le compilateur
  - GS est quant à lui utilisé dans un cas spécifique
  - Idem pour FS en 64 bits

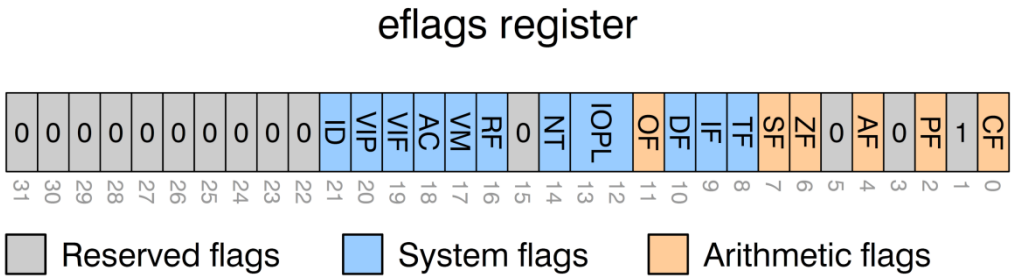
- De même, on utilisera toujours les variantes étendues des registres non généraux (EIP, ESP, EBP, ESI, EDI)

Flags

- La plupart des CPUs disposent d'un registre spécialement dédié à la gestion des « flags ». Pour les x86 ce registre est appelé EFLAGS.

- Il s'agit de bits indicateurs renseignant sur l'état courant du calcul mais également sur l'état du système (mode virtuel, trap, interruptions, privilèges, Identification...)

- Les indicateurs les plus utilisés sont
  - CF = carry (retenue), bit 0
  - PF = parité, bit 2
  - ZF = zéro, bit 6
  - SF = signe, bit 7
  - DF = direction (lecture en avant ou à rebours des chaînes de caractère), bit 10
  - OF = overflow (dépassement), bit 11



- Certaines instructions modifient l'état de ce registre tandis que d'autres le lisent.
  - **IMPORTANT**! Il faudrait spécialement faire attention au contenu de ce registre lors des tests, mais aussi dans des cas particuliers comme le calcul signé.

Récupérer la valeur d'un flag

- Les instructions **SETxx** dest permettent de stocker la valeur d'un flag dans un registre ou en mémoire
  - La valeur récupérée est 0 ou 1

- 'xx' peut être le nom d'un flag 'C' pour 'carry' et 'NC' pour 'no carry', 'Z' pour 'zero' et 'NZ' pour 'not zero' etc..

- Il existe des alias pour faciliter la lecture comme SETE (set if equal) equivalent de SETZ (set if zero)

- Ou encore SETLE (set if lower or equal) équivalent de SETNG (set not greater).
  - Renvoi 1 si ZF = 1 ou si CF (Carry) != OF (Overflow)

Zero

- Il est souvent utile de savoir si le résultat d'une opération vaut zéro, c'est pourquoi le CPU dispose d'un tel flag.
- Une opération arithmétique (comme une soustraction **SUB**) ou logique (**AND**)

```
SUB AX, AX ; le résultat de la soustraction est zéro, le flag ZF est levé
AND AX, AX; AX and AX égal zéro ssi AX = 0, donc dst = src
```

- Le flag peut être activé par une comparaison explicite avec **CMP** (qui fait un SUB en interne)

```
CMP AX, 42 ; le flag ZF est activé si AX vaut 42
```

- Les instructions de boucle comme **LOOPE/LOOPZ LOOPNE/LOOPNZ** lisent ce flag pour boucler
- Il est souvent plus judicieux d'utiliser l'instruction **TEST** qui simule un AND

```
TEST AX, AX; idem AND AX, AX mais ne modifie pas AX
TEST AX, BX; le flag ZF est levé si AX = BX = 0 ou si AX = ~BX
```

Carry (retenue) / Borrow (emprunt)

- Le flag ‘carry’ est activé lorsque le résultat d’une opération dépasse les limites du type d’opération en arithmétique **non-signée**.
  - Notez que le processeur ne sait pas explicitement si le nombre est signé ou non, il propose des flags en fonction de l’arithmétique mais c’est le rôle du programmeur de définir cet aspect

- Par exemple, dans le cas de nombres signés (8 bits) :

  255+2= 1 (257&255) et le flag carry est activé

- De même lorsque le nombre est trop petit

  1-2 = 255 et le flag carry est activé indiquant un emprunt (borrow)

  Le bit de signe est également activé

- Certaines instructions telles **ADC** (add with carry), **SBC** (sub with carry), **ADB** (add with borrow), **SBB** (sub with borrow) permettent de tenir compte de la valeur du flag
  - SBB dst, src permet de calculer dst-src-CF = dst-(src+CF)
  - si dst = src on récupère le complément à 2 du flag (0 ou -1) dans le registre dst
    - **SETC dst** fait la même chose en plus simple

Overflow

- Le flag ‘overflow’ est activé lorsque le résultat d’une opération dépasse les limites du type d’opération en arithmétique **signée**.
  - Cela se caractérise souvent par le fait qu’une opération avec deux opérandes de même signe produit un résultat du signe opposé
    - Comme vu précédemment, le CPU ne distingue pas ici les nombres signés ou non-signés, il active les flags CF ou OF en conséquence

- Par exemple, dans le cas de nombres signés 8 bits :

  127+1 = +128 sort du rang [-128;+127], le flag overflow est activé

- De même lorsque le nombre est trop petit

  -128-1 = -129

Bases de l’assembleur

En suivant la syntaxe de MASM

Modes d'adressage simple

- Adressage immédiat: stocke une constante dans un registre

  MOV EAX, 42

- Adressage registre: copie le contenu d'un registre dans un autre

  MOV EAX, EDX

- Adressage implicite: certaines instructions ne prennent pas d'opérandes mais utilisent des registres bien définis
  - RET // dépile la valeur de retour dans EIP
  - XLAT // look-up d'une table dans EBX à l'aide de AL, AL=[EBX+AL]
  - MOVS // copie le contenu de ESI dans EDI (cf. manip. de chaînes)

Modes d'adressage mémoire

- Tous les CPUs offrent différentes possibilités d'accéder à la mémoire.
  - Le but principal des modes d'adressage est de faciliter la réalisation d'opération d'indexation de haut niveau (simuler un tableau à 1, 2, 3 dimensions, gérer un sizeof struct etc…).
- L'architecture Intel défini les modes de base suivants:
- Adressage direct: stocke le contenu d'une variable dans un registre

```
MOV EAX, var
```

- Adressage indirect: copie le contenu de la mémoire référencée par le registre source (droite) dans le registre destination (gauche)

```
MOV EAX, [ESI]        ; attention, pas correct en model flat
```

  - L'opérateur [] indique le déréférencement

Modes d’adressage mémoire avancés

- Adressage indirect avec offset (valeur immédiate quelconque)

  MOV EAX, [EBX + 1024]

- Adressage indirect avec registre index (et facteur 2, 4, 8, 16)

  MOV EAX, [EBX*n + ESI] équivalent à
  MOV EAX, n[EBX][ESI]

- Adressage indirect avec registre index et offset

  MOV EAX, [EBX*n + ESI + 1024]

Opérandes de MOV

- R, R
- R, MEM
- MEM, R
- R, imm
- MEM, imm
- ~~MEM, MEM~~ aucune instruction ne peut avoir deux opérandes mémoires!

- R, [R] et [R], R
- R, [MEM] et [MEM], R
- R, offset variable

- offset est le contraire de [], offset calcul l'adresse d'une variable tandis que [] déréférence la variable. Ainsi, [offset variable] = variable.

Correction pour MASM en flat model

- On l'a vu, en modèle 'flat', les registres de segments sont supposés initialisés à zéro.
- Ce qui conduit la syntaxe MASM à suivre la règle suivante dans ce cas:
  - MOV EBX, [EAX] ; lit la valeur à l'adresse stockée dans eax
  - MOV EBX, [42] ; stocke 42 dans ebx malgré les []
  - MOV EBX, ES:[42] ; lit la valeur à l'adresse 42

LEA

- LEA, pour Load Effective Address, copie l'adresse d'une variable (généralement un tableau) dans un registre.

```
LEA ESI, variable          ; ESI contient l'adresse de variable
```

- C'est l'équivalent de

```
MOV ESI, offset variable
```

- LEA peut également s'utiliser avec une forme arithmétique (type adressage indirect avec registre index et offset)

```
LEA EAX, [ESI + EBX*2]     ; ESI = 0x1000, EBX = 0X42
```

- Attention, ici EAX contient 0x1042 et non la valeur à l'adresse 0x1042 !
  - LEA est le seul cas où les [] servent juste à utiliser la syntaxe [base + index*facteur + offset]

Types de données mémoire

- Dès que l'on adresse la mémoire il y'a ambigüité quant à la taille des données à lire ou écrire:

```
MOV [EBX], 0        ; doit-on écrire 1, 2 ou 4 octets ?
```

- On dispose de qualificateur permettant d'indiquer la taille des données:
- BYTE PTR, WORD PTR, DWORD PTR

```
MOV WORD PTR[EBX], 0    ; écrit deux octets

MOV AL, BYTE PTR[EBX]    ; lit 1 octet de poids faible de EBX dans AL

MOVZX EAX, BYTE PTR[EBX] ; idem mais altère le registre EAX entier.
```

Extension de donnée

- Parfois il peut être utile de copier une opérande source d'une taille inférieur à l'opérande destination tout en remplissant totalement ce dernier.
  - On parle alors d'extension, qui peut être non signée ou signée.

- MOVZX effectue une extension non signée
- MOVSX effectue une extension signée

- Exemple: BL contient 0xF0
- MOVZX AX, BL ; AX = 0x00F0
- MOVSX AX, BL ; AX = 0XFFF0
- MOVZX EAX, BL ; EAX = 0X000000F0
- MOVSX EAX, BL ; EAX = 0xFFFFFFF0

Test et comparaison

- **CMP** permet de comparer deux opérandes (immédiates, registres, mémoire). CMP agit comme une soustraction mais ne modifie aucune des opérandes.

- Le principal intérêt de CMP est de forcer une mise à jour des flags nous renseignant sur la différence entre les opérandes.
  - Si ZF est mis à true (1) cela indique que les valeurs sont identiques, ZF à 0 indique donc une différence.
  - Les autres flags comme SF, CF, OF peuvent nous renseigner plus précisément (ex: SF = 1 si l'opérande de gauche est plus petite que l'opérande de droite …).

- **TEST** à le même but que CMP sauf que cette fois-ci l'opération sous-jacente est un Et Logique (and). L'opérande de droite agit comme un masque et le résultat du masquage va lever/baisser des flags.

Branchements conditionnels

- L'instruction JMP peut être couplée avec un test ou une comparaison afin de déterminer si le saut doit être pris.

- E = EQUAL, similaire à Z

- Branchement Simples:

- JE ou JZ
- JNE ou JNZ

- JCXZ, si le registre CX == 0
- JECXZ, lorsque le registre ECX == 0

Branchements conditionnels

- Non signé:

  JA ou JNBE (Not Below or Equal)
  JNA ou JBE
  JAE ou JNB
  JB ou JNAE

- Signé:

  JG ou JNLE
  JNG
  JGE ou JNL
  JL ou JNGE
  JLE

Instructions modifiant les flags

- Les instructions de transfert ne modifient pas les flags: mov, movs, push, pop, stos, xchg…
- De même que les instructions de saut et de boucle, jmp et ses variantes, loop, rep etc…
- Les instructions arithmétiques, de comparaison et de déplacement modifient les indicateurs OF, SF, ZF, AF, PF, CF: add, adc, sub, sbb, mul, div, cmp, cmps, scas, shl, sal …
  - dec et inc ne modifient pas CF
- De même que les instructions logiques sauf que OF = 0 et CF = 0
- Les instructions de rotation modifient CF et OF

Décalages binaires

- Les processeurs utilisent un mécanisme appelé « *barrel shifter* » ou « décaleur » afin d'optimiser les calculs arithmétiques basiques.

- Au cœur de ce mécanisme on trouve le décalage binaire. Cela consiste à décaler la représentation binaire vers la droite ou vers la gauche.

- On doit nécessairement introduire un 0 à la place du bit manquant suite au décalage.
  - Un bit est également perdu, selon le sens du décalage cela sera le bit de poids fort (vers la gauche) ou le bit de poids faible (vers la droite).

Décalages arithmétiques

- Les décalages que nous avons vu jusqu'à présent sont dits « logiques » car il ne s'intéressent pas à la valeur du nombre représenté.
  - Ainsi, les nombres relatifs perdent leur propriétés lors des décalages car le signe n'est pas préservé.

- Les processeurs disposent d'instructions de décalages spéciales, dites de décalage arithmétiques, qui permettent de préserver le signe en le réinjectant lors des décalages.

- En C/C++ c'est le type de donnée (unsigned ou signed) qui permet de faire la différence entre décalage logique ou arithmétique

Astuce : masque binaire de signe

- On peut utiliser les décalages arithmétiques afin de générer un masque binaire

- Exemple permettant de représenter un test de signe sans pour autant coder un test.
  - On représente ici le décalage avec les doubles chevrons (>>), on suppose des registres 8 bits

    10100001 >> 8 = 11111111

    00011111 >> 8 = 00000000

- En appliquant un ET logique on obtient zéro si le nombre est positif

Exercice: fonction abs

- Calculer la valeur absolue d'un entier peut se faire assez facilement en utilisant des comparaisons

  If (x < 0) return -x;

  return x;

- On sait que -x est le complément à 2 de x c'est-à-dire le complément à 1 plus 1 : not(x) + 1
- -(-x) = x on peut donc appliquer la même logique
  - …tout comme l'addition + 1 est identique à une soustraction - (-1)
  - mais il ne faut le faire que dans le cas où la valeur est négative
- On sait que -1 en complément à 2 = tout les bits à un (0xffffffff)
- Un xor d'un entier avec 0xffffffff donne le complément à 1
- Un xor d'un entier avec 0 donne l'entier

Décalages logiques

- Le principal avantage des décalages (shifts) est la représentation naturelle des multiplications et divisions par des nombres puissances de 2.
  - Il suffit d'appliquer un décalage d'un nombre de bits équivalent à la valeur de la puissance.

- Pour diviser un nombre par 4, 4 étant égal à 2^2 on décale les bits de 2 rangés vers la droite.

- Pour multiplier un nombre par 16, 16 étant égal à 2^4 on décale les bits de 4 rangs vers la gauche.

Rotations

- Une rotation consiste en un décalage vers la gauche ou la droite dont les valeurs sortantes (carry) sont réinjectées dans l'octet opposé.

- Ainsi **ROL**, *rotate left*, déplace des bits vers la gauche. Le msb est donc au fur et à mesure de la rotation sorti en tant que retenue mais se trouve réinjecté dans le lsb

- **ROR**, *rotate right*, effectue l'inverse: décalage à droite et réinjection du lsb dans le msb.

- L'opérande source ne peut être que 1 ou valeur immédiate 5 bits, ou le registre CL.
- Le flag CF contient la valeur du msb ou lsb selon le sens de la rotation.

- **RCL** et **RCR** sont des variantes permettant d'ajouter le contenu de CF dans la rotation.

Manipulation de chaînes de caractère

- La plupart des mnémoniques manipulant des chaînes de caractère ont un adressage implicite.

- Très souvent la source doit être dans DS:[ESI] tandis que la destination doit être spécifiée dans ES:[EDI]
  - DS et ES étant dépréciés en mode ‘flat’ la plupart du temps ces registres sont mis à zéro.

- EFLAGS dispose d’un flag spécial indiquant la direction vers laquelle procède la manipulation.
  - Va-t-on incrémenter ESI et EDI, DF (direction flag) = 0, ou bien décrémenter ESI et EDI, DF = 1 ?

- Les mnémoniques **STD** (Set DF) et **CLD** (Clear DF) permettent de forcer la valeur du flag
  - Attention! Il faut impérativement rétablir la valeur précédente en fin de proc. !

Manipulation de chaînes de caractère

- **MOVS** transfère la donnée pointée par ESI dans celle pointée par EDI
- **CMPS** compare les données pointées par ESI et EDI
- **SCAS** compare la donnée pointée par EDI avec EAX
- **LODS** charge la donnée pointée par ESI dans EAX
- **STOS** écrit la valeur de EAX dans la donnée pointée par EDI

- Ces mnémoniques existent en version typée avec le suffixe B, W, D.
  - Dans le cas de SCAS, LODS, STOS on utilisera le sous registre de EAX adapté au type, ex: LODSB stocke dans AL, STOSW écrit AX…

Manipulation de chaînes de caractère

- Toutes les fonctions précédentes incrémentent ESI et EDI de 4, 2 ou 1 octet suivant le type de donnée spécifié.

- Il est possible de répéter automatiquement l'opération à l'aide des méta-instructions **REP**
  - Elles nécessitent une valeur > 0 dans ECX

- **REP** s'utilise avec STOS et MOVS et répète tant que ECX != 0
- **REPE** et **REPNE** répètent CMPS ou SCAS tant ECX != 0
  - REPE s'arrête si [ESI] != [EDI], donc si ZF = 0
  - REPNE s'arrête si [ESI] == [EDI], donc si ZF = 1

LA PILE

La Pile (stack)

- C’est un élément fondamental qui permet des mécanismes tels les appels de fonctions, la récursivité voire le multitâche.
  - Nous verrons plus tard la notion importante de « stack frame ».

- L’empilement s’effectue à l’envers, c’est-à-dire que le sommet de la pile à une adresse plus petite que la base.

- Le registre ESP pointe sur la position actuelle dans la pile.
  - Pour des raisons d’adressabilité (SP, en mode 16 bits, n’était pas utilisable dans beaucoup de modes d’adressage), un autre registre EBP est utilisé conjointement à ESP.

- EBP peut être utilisé comme base ou offset de certains modes d’adressage ce qui permet de lire et/ou écrire dans la pile.

Instructions de pile

PUSH opérande

- Empile l'opérande (contenu d'un registre, variable, constante, etc…) sur la pile.
- Ceci a pour effet de décrémenter ESP de 4 octets.

POP opérande

- Dépile et stocke la valeur dépilé dans l'opérande.
- Ceci produit l'incrémentation d'ESP de 4 octets.

PUSHFD / POPFD

- empile/dépile le registre EFLAGS

Sauts

- Comme il n'est pas possible de modifier directement le contenu de CS ou la valeur de (E)IP, la seule façon déplacer le compteur ordinal est d'effectuer un saut à l'aide de l'instruction

  JMP adresse

- Saut relatif, encodé comme un déplacement
  - « court », -128 à +127 octets
  - « proche » (NEAR), dans le même segment de 64kio, ou bien dans le même module
  - « long » (FAR), en dehors du segment courant et/ou du module

- Saut absolu (direct ou indirect) : l'adresse est encodée dans l'instruction

Fonctions et procédures

- La possibilité de sauter d'un emplacement mémoire à un autre permet la mise en place de procédures et fonctions.

  CALL opérande

- L'instruction CALL pousse l'adresse de l'instruction suivante (EIP++) sur la pile –qui servira donc d'adresse retour- et saute à l'adresse spécifiée par l'opérande.

- **RET**, quant à elle, dépile l'adresse de retour et place la valeur dans EIP.
  - Il existe d'autres variantes de RET, telle RET n où n indique le nombre d'octets supplémentaires à dépiler

- Si la procédure est une fonction, la valeur de retour sera écrit dans EAX (ou EDX et EAX si la valeur de retour est un quad word).

- La méta-fonction **PROC**, qui suit un label, permet de définir le début d'une procédure, **ENDP** la fin du sous programme.

Paramètres de sous programme

- Les paramètres d'une fonction doivent être fourni avant l'appel de la fonction. Il existe plusieurs conventions:
  - L'ordre et la méthode de passage dépendent de la convention déterminée par l'OS et le CPU (c'est ce que l'on appele l'ABI, Architecture Binary Interface).
- Passage total par registre (méthode RISC), rare en x86 à cause du faible nombre de registre.
- Passage partiel par registre, fastcall (Windows seulement)
  - Les deux premiers paramètres sont passés par registre, les autres par la pile, de la droite vers la gauche
- Passage par la pile, cdecl ou stdcall (Windows seulement)
  - Les paramètres sont empilés de la droite vers la gauche
  - En cdecl c'est la fonction appelante qui s'occupe de nettoyer la pile (cf. le slide stack frame), en stdcall / fastcall c'est la fonction appelée (callee)

Conventions

- Toujours en fonction de l'ABI, certains registres doivent être préservés / rétablis obligatoirement lors des appels de procédure.
- Certains registres sont considérés comme temporaire (scratch) et ne sont pas sauvegardés par les procédures. On dit que ces registres sont volatiles.
  - Aucun rapport direct avec le mot clé volatile du C
- EAX, ECX, EDX et ESP sont considérés comme volatile.
  - Tout autre registre modifié par une procédure **devra être préservé** puis rétabli (via la pile) !!!
  - Bien que ESP soit volatile, il faut tout de même veiller à ce que les frames soient correctement dépilées.
- Lorsque vous appelez une fonction de convention C (cdecl) c'est à vous qu'il incombe de dépiler la stack et de rétablir les registres.
- Dans les autres conventions Windows (sdtcall, etc…) c'est la fonction appelée qui doit rendre une stack propre.

Stack Frame

- Du fait que la pile est un élément central et que le contenu de la pile peut être modifié dans une procédure il nous faut un moyen de retrouver l'état originel de la pile avant l'appel.

- Ceci se fait à l'aide d'un cadre de pile (stack frame) qui consiste à
  - Stocker la valeur précédente de EBP sur la pile
  - Stocker la valeur de ESP (sommet de la pile) dans EBP

- On peut ainsi modifier à notre convenance ESP (allouer des variables, etc…).
- Sauvegarder ESP dans EBP nous permet d'accéder aux paramètres (offsets positifs) et variables locales (offsets négatifs)

- En quittant la stack frame
  - on rétabli ESP, ESP = EBP
  - Puis on récupère le précédent EBP sur la pile

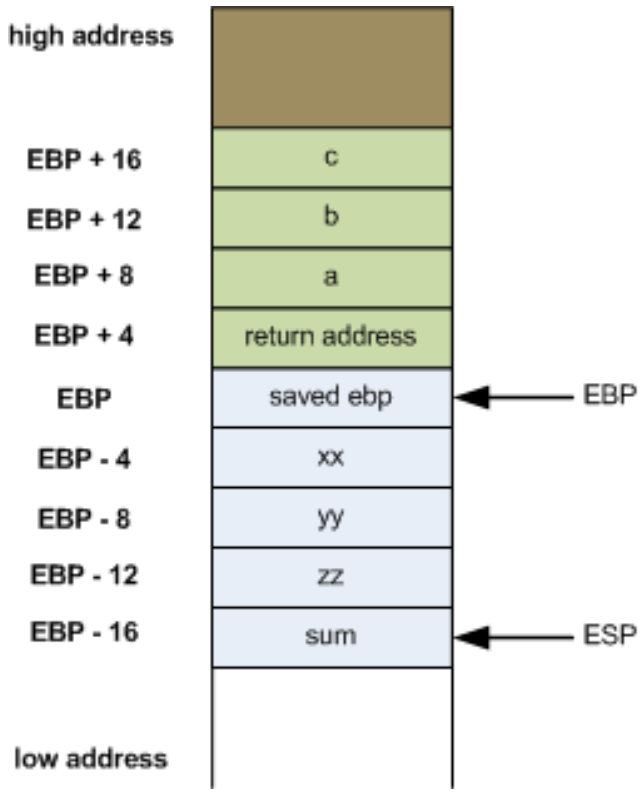
Stack Frame

- La méta-fonction ENTER est équivalente à:
  PUSH EBP
  MOV EBP, ESP
  - On parle également de « prologue » (prolog)

- La méta-fonction LEAVE est équivalente à:
  MOV ESP, EBP
  POP EBP
  - On parle également d'épilogue (epilog)

- Notez que ce n'est obligatoire, les compilateurs offrent une option pour désactiver la stack frame et libérer l'usage du registre EBP.
- L'omission d'une stack frame peut rendre le code moins lisible, mais cependant légèrement plus efficace pour les fonctions courtes.

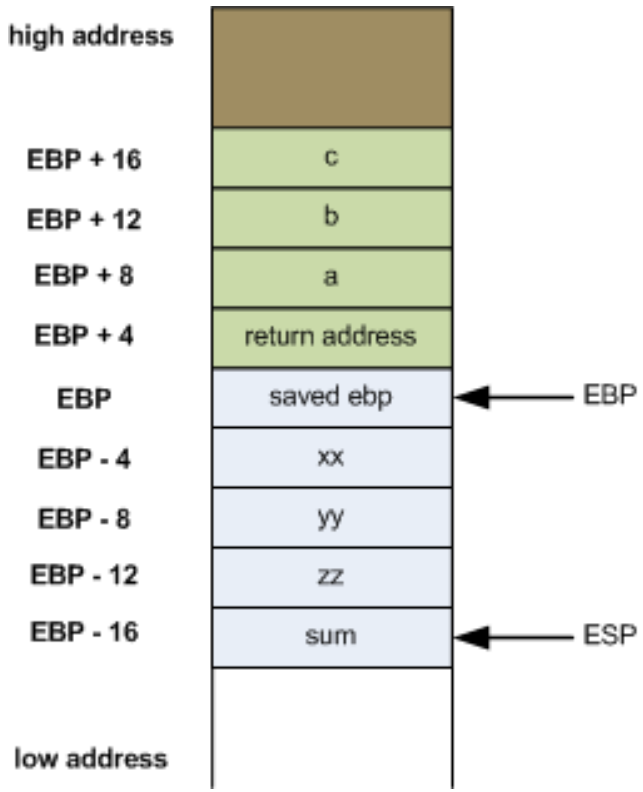
Stack Frame: exemple




```
int function(int a, int b, int c)
{
    int xx, yy, zz;
    int sum = ….;
    return sum;
}
```

- Les données en vert (offsets positifs) ont été empilé avant de rentrer dans la fonction.
- Et celles en bleu clair (offsets négatifs) sont des variables locales.
- EBP pointe toujours sur la valeur empilée avant l'appel de fonction, qui est ici EBP.

Stack Frame: retour




```
int function(int a, int b, int c)
{
    int xx, yy, zz;
    int sum = ….;
    return sum;
}
```

- L'appel de fonction nécessite les instructions

```
push c
push b
push a
call function
```

- Le retour nécessite de dépiler les arguments

```
add esp, 0Ch        ; 12 octets
```

Quelques bouts de code usuels

- manipulation du pointeur de pile sp

```
sub     esp,0C0h        ; réserve 192 octets sur la pile
add     esp,0C0h        ; « libère » 192 octets sur la pile
```

- En debug, marquage du zone mémoire autour de la pile pour détecter les dépassements. Visual Studio initialise cette zone avec 0xCCCCCCCC.
  - **0xCC** est l'opcode de **INT 3**, interruption logicielle qui correspond à un breakpoint

```
lea     edi,[ebp-0C0h]          ; adresse destination
mov     ecx,30h                 ; nombre de boucles
mov     eax,0CCCCCCCCh          ; valeur à copier
rep stos    dword ptr es:[edi]  ; répète la copie et dec. Ecx
                                ; es est initialisé à zéro ici
```

Cas du C++ et des classes

- En C++ on a un pointeur (implicite) supplémentaire: **this**
- Selon l'OS on aura:
  - Windows: this stocké dans ecx / rcx
  - autres: this est stocké sur la pile
- Test t; t.function(…);

```
lea      ecx, [t] ; adresse de t dans ecx
```

- Test *t = new T; t.function(…);

```
mov     ecx, dword ptr [t] ; t est un pointeur ici
```

Instructions avancées

- Un des principaux intérêt de l'assembleur comparativement aux langages de haut niveau réside dans la possibilité d'utiliser des instructions bas niveau qui ne sont pas utilisées par les compilateurs.
  - C'est en partie le cas avec ROL, RCL, ROR, RCR

- La mnémonique **CMOV** effectue un MOV conditionnel, en fonction du résultat d'un test.

- Comme JMP, CMOV se combine avec des conditions
  - On aura donc CMOVG, CMOVNE etc…

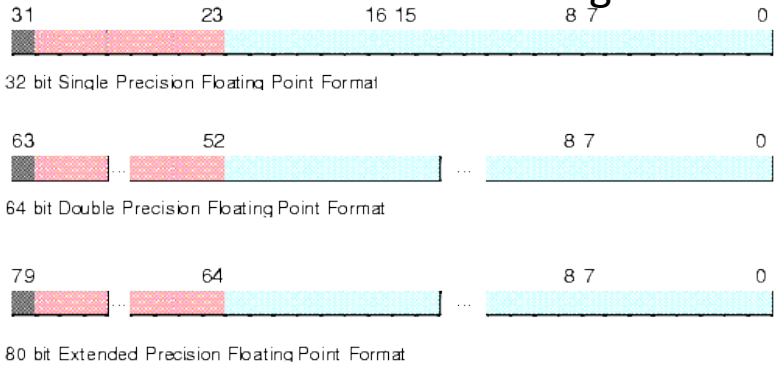
- Cette instruction permet la programmation sans branchement (branchless) ce qui peut améliorer la vitesse d'exécution du programme lorsque la prédiction de branchement rate souvent.

En plus de(s) l’ALU entière les CPUs modernes disposent de plusieurs autres unités de calculs dédiées à certains types d’opérations

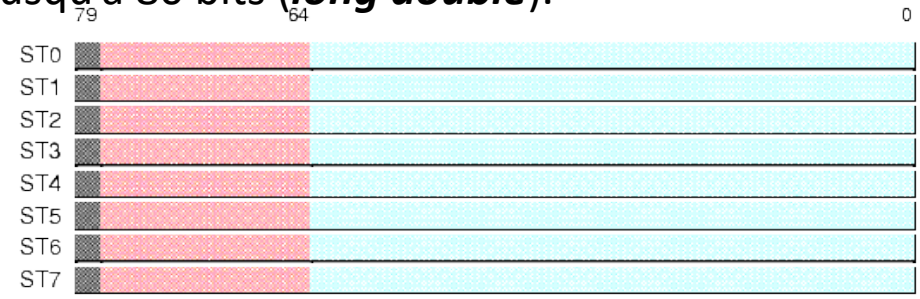
UNITES DE CALCULS

Floating Point Unit (FPU)

- Le x86 dispose d'une unité de calcul de réels à virgule flottante nommée x87



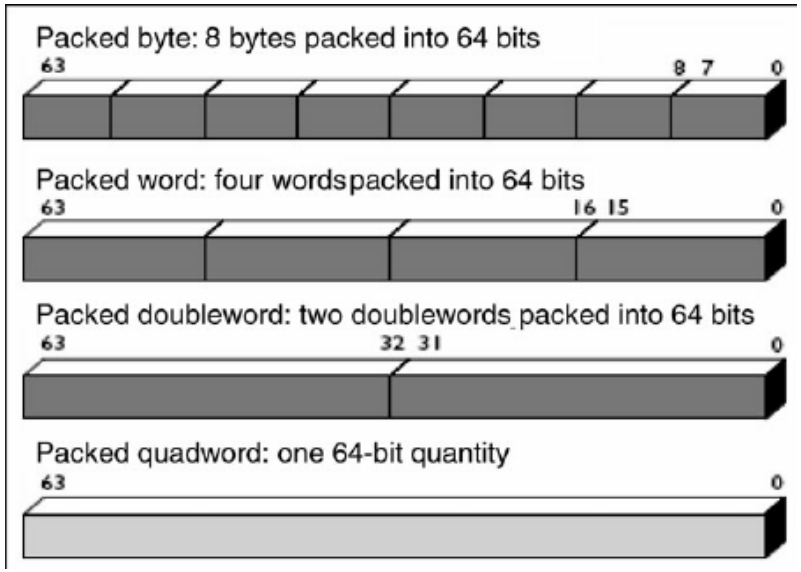
- Une des grandes spécificités du x87 est la large précision offerte par les 8 « registres » , jusqu'à 80 bits (***long double***).



- Le x87 ne dispose pas de registres à proprement parler mais d'une pile de 8 éléments, ce qui rend sa programmation légèrement complexe.

MMX

- Depuis le Pentium MMX les registres du FPU sont partagés par des instructions vectorielles (SIMD) sur 64 bits
  - les registres sont splittés en sous parties de 8, 16 ou 32 bits.
  - Ces « packed values » représentent des valeurs distinctes et séparées mais pas un registre
  - Il convient de noter que les instructions MMX sont des instructions agissant sur des **entiers** et non des réels!



- Plus de détails ici:
- http://softpixel.com/~cwright/programming/simd/mmx.php

SSE: SIMD des réels

- L'ajout d'une unité de calcul vectoriel spécifique au Pentium III, donc séparé du FPU, a permis d'introduire des fonctionnalités puissantes pouvant traiter jusqu'à **4 réels simple précision (32 bits) en parallèle**.
  - les registres du SSE sont au nombre de 8 (XMM0-7) et ont une taille de **128 bits**.

- On retrouve des instructions agissant sur un float isolé ou un paquet de 4 float-s.

- La plupart des instructions accédant à la mémoire sont réparties existent en deux versions
  - une pour les accès mémoire alignés (multiple de 16) et une autre pour les accès mémoire non alignés.

SSE2

- Le Pentium IV introduit le jeu d'instruction SSE2 qui étend le MMX en offrant la possibilité d'utiliser les registres SSE en entier de 8, 16, 32, 64 bits.

- Le SSE2 étend également les instructions SSE aux réels double précision.
  - Ajout également de fonctions de conversion de/vers les double-s

SSE3 et SSSE3

- Le principal ajout du SSE3 est l'opération horizontale, c'est-à-dire la possibilité de calculer l'addition ou la soustraction des sous-parties float ou double d'un registre SSE.

- Les premiers Core2 ajoutent des fonctionnalités horizontales supplémentaires (Supplemental SSE3) de type MMX entier

SSE4

- SSE4.1 ajoute des fonctions de comparaison, arrondi, mix (blend) d'entiers et de réels
- SSE4a est une extension spécifique à AMD
- SSE4.2 ajoute des fonctions de manipulation de chaînes de caractère

AVX, AVX2, AVX-512 et autres

- AVX supporte les registres réels 256 bits
- AVX2 ajoute les instructions pour les entiers
- AVX-512 supporte les registres 512 bits

- Le SSE5 proposé par AMD n'a jamais vu le jour officiellement, mais on retrouve plusieurs extensions:
  - F16C : fonctions de conversions de/vers float16 à float32
  - FMA3 : implémente des opérations à 3 opérateurs dites « fused multiply-add » de type a = a*b + c
  - FMA4 : spécifique à AMD pour le moment, 4 opérateurs de type d = a*b + c

Dépréciation du FPU

- Depuis Visual Studio 2008 le code utilisant le FPU x87 a été peu à peu remplacé par le SSE.
  - Initialement optionnel, c'est la règle depuis Visual Studio 2012.

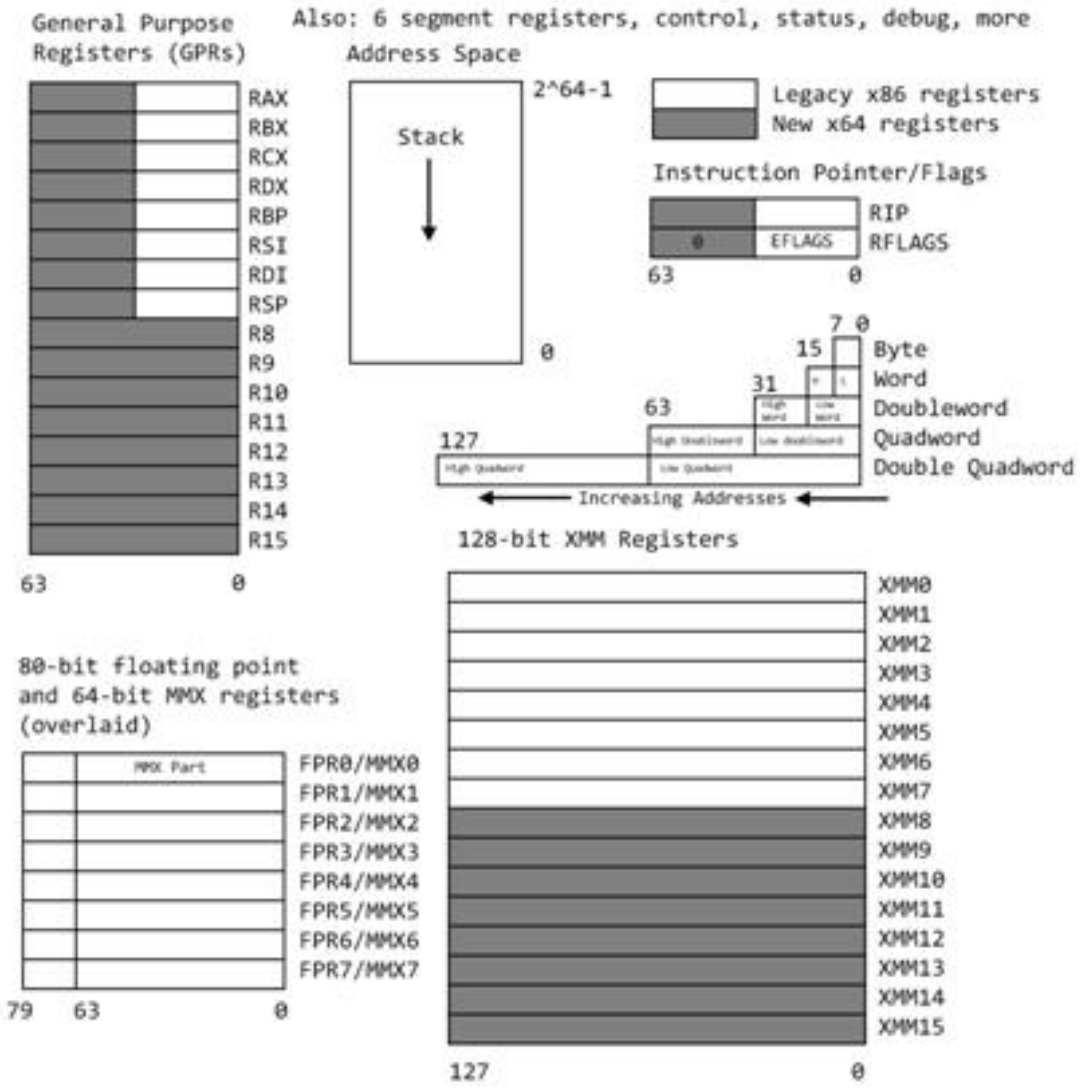
- Les six premiers registres SSE (XMM0-5) sont utilisés pour stocker les float-s.
  - Cela implique la sauvegarde de ces registres sur la pile lors des appels de fonctions / changement de contexte
  - A noter que seuls les 4 premiers registres SSE sont utilisables comme paramètres de fonctions
    - Si besoin de passer plus de 4 paramètres vectoriels il faut les empiler

Architecture x86-64

Attention cette architecture est différente de IA-64 (Itanium …)

X86-64 / AMD64

Les registres x86 et x64 en x86_64



Registres x64

- Du fait que les pointeurs mémoires sont désormais sur 8 octets (64 bits) les registres sont donc également sur 64 bits.

- Ils sont désormais préfixés par ‘R’, ex: RAX extension 64 bits de EAX, RFLAGS extension de EFLAGS etc…
  - On peut aussi accéder à l’octet de poids faible de n’importe quel registre (DIL, SIL, BPL, SPL …)

- PUSH et POP décrémente/incrémente RSP de 8 octets

- La stack est également alignée par défaut sur 16 octets.

  - plusieurs règles spécifiques pour rendre la stack compatible avec l’usage intensif du SSE en calcul float

Nouveaux registres

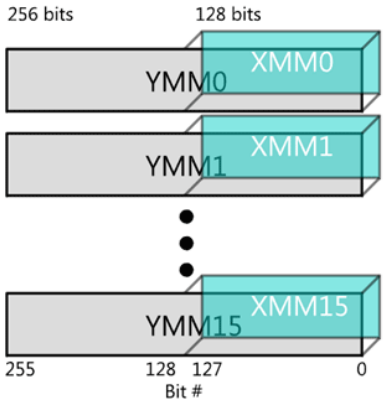
- Nouveaux registres généraux R8 à R15
  - Les fractions 32 bits sont suffixées par D
  - Les fractions 16 bits sont suffixées par W
  - Les fractions 8 bits sont suffixées par B

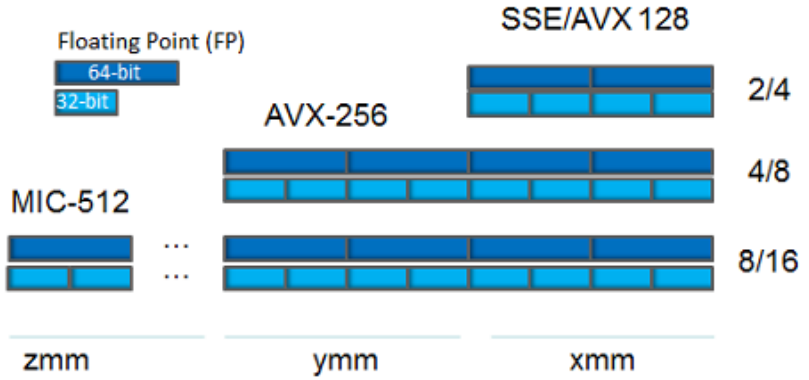
- RAX, RBX, RCX, RDX, RSI, RDI, RSP et RBP sont également accessibles via les alias respectifs:
  - R0, R1, R2, R3, R4, R5, R6, R7

- Pour réserver un espace mémoire sur 8 octets on utilisera DQ (Q pour Quad Word, 4 * 2 octets):
  - var64 dq 0X123456789ABCDEF

SSE et AVX

- 8 nouveaux registres SSE: XMM8 à XMM15





- Lorsque AVX est supporté les registres SSE sont étendus à 256 bits nommés YMM0 à YMM15.
  - XMM0-15 correspondent alors aux 128 bits de poids faibles de YMM0-15.

- Lorsque AVX2 est supporté les registres AVX sont étendus à 512 bits et nommé ZMM0-15

Conventions d'appel x64

- Il n'y a plus qu'une seule convention d'appel : cdecl … mais les règles sont différentes selon l' OS.

- Windows (MS64):
  - les **4 premiers** paramètres sont passés par registres sur RCX, RDX, R8, R9 dans cet ordre, les autres paramètres doivent être empilés
  - En C++, this est le premier paramètre
  - RAX, RCX, RDX, R8, R9, R10, R11 sont volatiles, les autres doivent être préservés
  - Les 4 premiers paramètres virgule flottante sont passés via XMM0 à XMM3
    - Si les paramètres sont un mix d'entier/pointeur et float les registres sont alternés, ex: RCX (1er), XMM1 (2ème), XMM2, R9
- Vector Call : convention spécifique à Windows
  - https://docs.microsoft.com/en-us/cpp/cpp/vectorcall?view=vs-2019
  - Les 6 premiers paramètres SSE/AVX sont passés par les registres XMM0 à XMM5

- System V AMD64:
  - Suivie par la plupart des OS dérivés d'Unix (FreeBSD, Linux, MacOS,…)
  - les **6 premiers paramètres** sont passés par registres sur RSI, RDI, RCX, RDX, R8, R9 dans cet ordre, les autres paramètres doivent être empilés
  - En C++, this est le premier paramètre
  - RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11 sont volatiles
  - Les paramètres virgule flottante sont passés via XMM0 à XMM7, sur la pile si plus.

Règles de la fonction appelante

- Lors d'un CALL en 64 bits il faut s'assurer que l'adresse de la pile est bien multiple de 16.

- De plus il faut également réserver un espace sur la pile appelé « shadow space » de 32 octets (zone miroir pouvant contenir RCX, RDX, R8, R9).
  - En System V / AMD64 une « red zone » de 128 octets est requise à la place, avec l'avantage de disposer de cette zone pour les variables locales

- Suivant la convention C standard, c'est le rôle de la fonction appelante de nettoyer la pile.

- https://software.intel.com/en-us/articles/introduction-to-x64-assembly

- Note: en 64 bits il n'est plus possible de faire de l'assembleur inline.
- On doit forcément utiliser un logiciel assembleur ou utiliser les intrinsics.

Modèle de donnée 64 bits

- Le « Data Model » caractérise la façon dont un OS gère les types standard à savoir les short-s (S), int-s (I), long-s (L) et les pointeurs (P).

- MS-64 se caractérise par un modèle LLP64 c'est-à-dire que seuls les types « long long » et les pointeurs sont sur 64 bits.

- La plupart des OS 64 bits dérivés d'Unix suivent le modèle LP64, ce qui indique les types « long » (implicitement « long long » également) et les pointeurs sont sur 64 bits.
  - Cela complexifie le portage du code utilisant long

Espace d'adressage

- Les OS modernes utilisent l'adressage virtuel (pagination mémoire) dans le cadre de l'exécution d'un processus via un mécanisme de la MMU (Memory Management Unit) du CPU nommé TLB (Translation Lookaside Buffer).

- Les CPU x64/AMD64 ne permettent qu'un espace d'adressage physique de 48 bits au mieux, soit au maximum 256 Tio, contre 16 Eio (exa octets binaires).

- C'est d'ailleurs préférable pour les OS car cela réduit le nombre de pages virtuelles à gérer.
  - Windows fractionne la mémoire en pages de 4 kio
  - Les versions de Windows avant 8.1 limitaient encore l'espace d'adressage virtuel physique à 44 bits soit 64 Tio (48 bits logiques)
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/virtual-address-spaces

MASM

Syntaxe fichier assembleur MASM

- Il faut d'abord indiquer le processeur

```
.686       ; ou .286, .586 … 586=Pentium, 686 si supérieur à PentiumIV
; indique donc un commentaire
```

- Viennent ensuite le modèle mémoire et en option la convention d'appel

```
.model flat, c       ; ou stdcall ou fastcall
```

- La directive « option » permet de customiser à votre goût certains aspects (respect de la casse, etc...)

- Les directives include et includelib permettent d'indiquer respectivement le fichier include à charger et le fichier lib à linker.

Sections et procédures

- .data variables globales initialisées
- .data? Variables globales non initialisées (aussi appelé bss)

- .code début de la section de code
  - Equivalent de: _Text SEGMENT
- end <label> indique la fin de la section de code, ainsi que le label représentant le code.
  - Equivalent de: _Text ENDS

- Pour définir une procédure

  _MaFonction proc

  …

  _MaFonction endp

  - Notez que la procédure commence par un '_' mais sera visible sans le '_'

Spécificité de MASM

- On peut déclarer le prototype d'une fonction:
- MaFonction PROTO: param:TYPE, ….

- INVOKE est une pseudo fonction similaire à CALL mais s'occupant automatiquement de l'empilement/dépilement

- LOCAL nom:TYPE permet de créer facilement des variables locales

Constantes

- Il est possible de définir des valeurs constantes.

- La plupart du temps ces constantes sont insérées dans le code de l'instruction, on parle alors de **valeur immédiate**.

- SoixanteDixHuit EQU 78
- SeptanteHuit EQU SoixanteDixHuit

- Les constantes peuvent aussi être stockées sous forment de données en lecture seule dans l'exécutable.

Variables

- Une variable se caractérise par un espace réservé pour le stockage de la donnée.

- var1 DB 42 ; variable de type BYTE, (unsigned) char
- var2 DW 42 ; variable de type WORD, 1 mot de 2 octets, (unsigned) short
- var3 DD 42 ; variable de type DWORD, 2 mots, (unsigned) int

- Une variable peut être de longueur quelconque mais toujours fixe en taille.

- Var4 DB 40, 41, 42, 43, 44 ; tableau de 5 bytes
- Var5 DW 5 dup(42) ; tableau de 5 words valant tous 42

- Dans le cas où l'on souhaite que le contenu soit non initialisé il suffit de remplacer la valeur par le symbole '?'

- Temp DW ?
- Buffer DD 512 Dup(?)

Chaînes de caractères

- Les chaînes de caractères sont une forme spéciale de variable.

- String DB ‘chaine de caractere’,
  - Autrefois, en MS-DOS ‘$’, marquait la fin de chaîne. On s’en sert surtout pour calculer la longueur de la chaine:
    - StringLen EQU $-String

- On rencontre principalement trois formes :

  - Une forme de taille fixe, un ou plusieurs octets stockent alors la longueur de la chaîne.
  - Une forme de taille variable, où, en plus de la longueur réelle de la chaîne de caractère on stocke la taille du buffer (méthode Pascal).
  - Un forme de taille variable dont la particularité est d’être terminée par le caractère ‘null’ (0) – les fameuses null-terminated strings du C/C++.

Conventions d’appel

- On l’a vu les fonctions doivent être préfixées par ‘_’ mais dans le cas où la convention d’appel est différente de ‘c’ (cdecl) on doit ajouter un suffixe selon le type:

- Suffixe de stdcall : ‘@0’

  En c++ : extern « C » void _stdcall maFonction();
  En asm: _maFonction@0 proc

- Pour fastcall c’est le même suffixe que stdcall mais le préfixe n’est plus ‘_’ mais ‘@’

  En c++ : extern « C » void _fastcall maFonction();
  En asm: @maFonction@0 proc

Fonctions intrinsèques

- Visual Studio expose les mnémoniques assembleurs sous la forme de fonctions.
- Ces fonctions sont déclarées dans « intrin.h »
- On va retrouver la majeure partie des instructions à quelques petites exceptions
  - Pas de fonction intrinsèque pour CMOVcc par exemple ce qui est dommage
  - Cela dit le compilateur VC++ a encore du mal a générer cette instruction même avec les optimisations

Appendices

REPRÉSENTATION DES NOMBRES

Entiers négatifs

- Nous savons qu'avec n bits nous pouvons représenter 2^n valeurs différentes, de zéro à n-1 inclus.
  - Par exemple avec 8 bits on dispose de 256 valeurs différentes : [0-255]

- Cependant nous n'avons à faire qu'à des entiers naturels (positifs). Se pose la question de la représentation des nombres négatifs.

- Si l'on compare deux chiffres opposés, -7 et +7 par exemple, on se rend compte que le signe ne prend que deux valeurs, + et -, ce qui colle bien avec un bit.
- Il est assez tentant de dédier un bit, le bit de poids fort, pour désigner le signe.
  - Cependant en faisant cela on se retrouve avec une valeur neutre, zéro, ayant deux représentations différentes !

- Nous avons déjà vu l'opérateur logique NOT qui produit l'inverse (appelé complément) d'un chiffre binaire.
- Si l'on complémente (inverse) les n bits d'un entier on obtient le complément à un de cet entier, que l'on peut voir comme son opposé.

Problème avec l'addition

- La représentation du signe comme bit de poids fort complexifie les opérations arithmétiques.

- L'addition binaire de deux nombres sur 4 bits valant 1 (0010_b en binaire) donne 0010_b par l'addition de la retenue vers la gauche.

- Prenons les nombres 3 et -4 ainsi représenté :
- 0011_b
- 1100_b (0100_b avec le bit de poids fort à 1)

- L'addition de 3 + (-4) donne ici -7 (0111_b avec le bit de poids fort à 1) au lieu de -1.

Complément à un

- Toujours en se limitant pour notre exemple à n=4 soit 4 bits on peut calculer le complément à un de chaque chiffre:
- 0000_b => 1111_b
- 0001_b => 1110_b
- 0010_b => 1101_b
- 0011_b => 1100_b etc…

- Le complément pouvant être vu comme l'opposé on interprète alors les valeurs complémentaires comme des valeurs négatives.
  - Ceci implique que le plus grand entier positif représentable n'est plus 2^n-1 mais 2^{n-1}-1 !

- Le principal défaut de ce système est que le zéro est représenté deux fois: +0 (0000_b) et -0 (1111_b).
  - Dans le calcul on veille également à ne pas reporter la retenu sur le bit de poids fort.

Complément à deux

- En ajoutant 1 au résultat du complément à un on obtient une forme intéressante.

- Pour le chiffre 0001_b, le complément à un donne 1110_b, si l'on ajoute 1 on obtient -1 $\Leftrightarrow$ 1111_b.

- Dans le cas de zéro, 0000_b, on a 1111_b pour le complément à un, puis $(1)0000_b$ en complément à deux. La retenue (1) est ignorée et on a bien +0 et -0 qui est représenté par une seule valeur 0000_b.

- Dans le cas de la somme 3 + (-4) on avait pour le complément à un : $0011_b + 1011_b$. En ajoutant 0001_b à 1011_b on obtient 1100_b. Cette fois-ci la somme de 0011_b et 1100_b donne 1111_b soit -1 !

REPRESENTATION A VIRGULE FIXE

Arithmétique à virgule fixe

- Lorsque des processeurs ne disposent que de registres et d'instructions sur des entiers il existe une technique permettant de simuler un nombre réel.

- Pour le cas de la base 2, cela consiste à placer une virgule fixe dans la représentation binaire d'un nombre en répartissant les bits à gauche et droite de la virgule.

- Les bits précédents la virgule vont alors indiquer la partie entière du nombre tandis que les bits suivants la virgule vont représentés la partie fractionnelle.

Exemple sur 8 bits

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
128 64 32 16 8 4 2 1

- Voici une répartition avec 4 bits entiers et 4 bits fractionnels

2^3 2^2 2^1 2^0 | 2^{-1} 2^{-2} 2^{-3} 2^{-4}
8 4 2 1 | 1/2 1/4 1/8 1/16

- Les notations usuelles sont M.N, ici 4.4 ou alternativement QN, ici Q4.
  - M indique le nombre de bits entiers, N le nombre de bits fractionnels

Domaine et précision

- Le domaine (*range*) est défini par la partie entière. Par exemple avec une partie entière sur 16 bits on aura les domaines suivants:
  - [0, 65536] pour un nombre non signé
  - [-32768, 32767] pour un nombre signé.

- La précision correspond au plus petit écart binaire-ment représentable entre deux nombres successifs.

- Pour un nombre à virgule fixe elle est de $1/2^N$.
  - 16.16: $1/2^{16}$ = 1/65536 (~0,000015258789)

Conversion vers/depuis virgule fixe

- Conversion depuis / vers un entier. Décaler de N bits:

  F = I << N

  I = F >> N

- Conversion de réel vers virgule fixe. Multiplier par 2^N and arrondir à l'entier le plus proche:

  F = Round [R * (1<<N) + (R>=0 ? 0.5 : -0.5]

- Conversion de virgule fixe vers réel. Diviser par 2^N:

  R = F / (1<<N)

Opérations arithmétiques

- L'addition et la soustraction s'opèrent comme en complément à 2.
- La multiplication est plus particulière. Il faut d'abord se souvenir que chaque nombre sous forme M.N consiste en une partie entière M décalée de N bits vers la gauche.
- En vertu des propriétés des décalages arithmétiques, la partie entière de chaque nombre est donc multipliée par un facteur 2^N.
- Une solution possible serait alors de multiplier les deux nombres ensemble et de diviser par 2^N.
  - Pour rappel diviser par une puissance de deux est équivalent à décaler les bits d'un nombre de N bits vers la droite.

(a * b) >> N

Dépassement (overflow)

- La principale problématique de la forme précédente est le dépassement de capacité.

- Si a et b dans l'exemple précédent sont de type virgule fixe 16.16 et valent tous les deux 2,0 (donc 2 << 16, soit 131072), la multiplication produit le nombre 17179869184.
  - Soit 0x4 0000 0000 en hexadécimal

- Ce nombre, qui dépasse la capacité de la partie entière (16 bits), nécessite en fait 34 bits soit plus grand que la taille standard d'un entier 32 bits (valeur max. 0xFFFFFFFF).

Multiplication: façon correcte

- Le résultat intermédiaire d'une multiplication nécessite au plus un stockage intermédiaire de 2M:2N (Q2N).

- Dans le cas d'une multiplication d'un nombre à virgule fixe de format 16.16, il nous faut exprimer nos deux nombres a et b avec 32 bits entiers et 32 bits fractionnels soit 64 bits pour que l'équation (a * b) >> N (ici N = 16) soit correcte.

- Pour le cas des divisions c'est relativement similaire. Pour diviser le numérateur a par le dénominateur b, il faut d'abord exprimer le numérateur en double précision, décaler de N bits vers la gauche puis diviser par le dénominateur:

- Il faut tenir compte du fait que a et b sont déjà multiple de 2^N. Si l'on divise a par b directement, les facteurs 2^N s'annulent et l'on n'obtient plus un nombre à virgule fixe. Il faut au préalable multiplier a par $2^{N:}$

$$(a << N) / b$$

Virgule fixe: avantages et inconvénients

- Les nombres à virgule fixe sont extrêmement pratique et selon le cas peuvent proposer une précision relativement importante d'autant que la précision est répartie de manière égale.

- De plus ils offrent une représentation exacte des nombres réels tant qu'il n'y a pas d'*overflow* ou d'*underflow*.

- Cependant, il n'est pas toujours aisé de manipuler des nombres ayant des rangs ou précisions différentes et de les utiliser ensemble dans des calculs.
  - Ex. additionner un nombre 24.8 avec un nombre 16.16.

REPRESENTATION A VIRGULE FLOTTANTE

Problématique de la représentation des nombres réels

- Infinité de nombre réels
- Représentation CPU finie (nombre de bits limités)
- Précision ne signifie pas pour autant justesse
  - 3,123456789 est certes précis mais non correct pour représenter PI (3,14159…)

Notation scientifique

- La notation scientifique utilise une forme normalisée décimale.
  - Cela signifie que l'on ramène un nombre à une partie comprise en valeur absolue entre [1, 10[et que l'on appelle coefficient. C'est la partie normalisée.

- Les nombres sont alors exprimés comme des multiples des puissances de 10:

$8174{,}12 \Leftrightarrow 8{,}17412 \cdot 10^{3}$

$-12{,}345 \Leftrightarrow -1{,}2345 \cdot 10^{1}$

$0{,}0000724 \Leftrightarrow 7{,}24 \cdot 10^{-5}$

Virgule flottante binaire

- Dans l'écriture scientifique nous avons un coefficient qui est un réel normalisé, avec une partie décimale et une partie fractionnelle, qui se trouve être multiplié par une puissance de 10.

- En binaire, on utilise logiquement des multiples de sur une base de puissances de 2 pour la partie **exponentielle**.

- Un nombre binaire peut se normaliser en un coefficient binaire, que l'on nomme **mantisse**, avec une partie décimale, qui sera toujours à 1, et une partie **fractionnelle**.

- Ceci prend la forme suivante: $+/-(1.F)*2^{E}$

Exemple

- Le nombre -8,75 en décimal se traduit par $-1{,}00011*2^3$ en virgule flottante binaire.

- Le signe identique, il est négatif ici
- L'exposant est de 3. Comme en notation scientifique cela indique le multiplicateur
  - Rappel: multiplication binaire = décalage vers la gauche

- En décalant de 3 bits vers la gauche on obtient
- Une partie décimale qui vaut 1<<3 = 8
- La partie fractionnelle vaut ½ + ¼ = 0,75

Avantages (1)

- Les principaux avantages du format virgule flottante que l'on vient de voir sont les suivants:

- 1. Les représentations des nombres réels sont uniques. En notation scientifique on peut représenter 100 par 10.10^{1}, 1.10^{2}, $0{,}1.10^{3}$ …

- 2. Comme la mantisse (le coefficient binaire) est toujours >= 1 en valeur absolue, l'exposant suffit pour représenter des petits nombres type 0.00001 en décimal.

- 3. Corollaire de 2, comme le premier bit est toujours à 1 dans la mantisse il n'a pas besoin d'être stocké, il devient donc implicite.

inconvénients (2)

- 4. La densité des nombres représentables varie avec l'exposant. Par exemple, dans le format simple précision IEEE que l'on va voir par la suite on a autant de précision (de valeurs différentes) entre 1.0 et 2.0 qu'entre 256.0 et 512.0.

- 5. Plus on monte dans l'exposant, plus l'écart se creuse. Toujours en IEEE Il y'a donc beaucoup plus de nombres entre 10.0 et 11.0 (1048575) qu'entre 10000.0 et 10001.0 (seulement 1023!).

- 6. Il est possible qu'un nombre ne soit plus représentable parce qu'il est trop grand (*overflow*) ou trop petit (*underflow*) pour le format.

- 7. Du fait qu'un nombre fixe de bit est réservé pour l'exposant cela rend le format moins adaptatif que le format virgule fixe.

- 8. Certains nombres ne sont pas exactement représentable tel 0,10

Format IEEE-754

- Le format IEEE-754 est un standard, défini à partir de 1985, et qui est aujourd'hui *de facto* la forme standard du format virgule flottante.

- Tous les CPUs et GPUs modernes implémentent des réels en utilisant ce format, cependant toutes les implémentations ne sont pas 100% standard.

- Le format spécifie deux représentations binaires basiques: simple précision et double précision.

Formules IEEE-754

- ‘s’ ⇔ signe, ‘1.F’ ⇔mantisse, ‘E’⇔exposant biaisé

- En simple précision on peut représenter des nombres allant de 10^{-38} à 10^{38} avec une précision de six à neuf chiffres décimaux:

$$V = (-1)^s * (1.F) * 2^{E-127}$$

- En double précision on peut représenter des nombres allant de 10^{-308} à 10^{308} avec une précision de quinze à dix-sept chiffres décimaux:

$$V = (-1)^s * (1.F) * 2^{E-1023}$$

Représentation binaire

Réel simple précision sur 32 bits:

Signe (1)	Exposant (8)	Fraction (23)
bit 31	bits 30 à 23	bits 22 à 0

Réel double précision sur 64 bits:

Signe (1)	Exposant (11)	Fraction (23)
bit 63	bits 62 à 52	bits 51 à 0

Virgule flottante simple précision

- Le format IEEE-754 reprend plusieurs points que l'on a abordé précédemment:

- La mantisse est de la forme +/-1.F, le signe est stocké dans le bit de poids fort, tandis que 'F', partie fractionnelle, est stockée dans les 23 premiers bits (0-22).
  - Le bit de poids fort, toujours à 1, n'est pas stocké par le format car considéré comme implicite, ce qui libère 1 bit de stockage pour la mantisse.

- L'exposant est sur 8 bit (bits 23 à 30) et il a plusieurs particularité.

- D'une part, pour le cas simple précision, l'exposant est stockée avec une valeur biaisée par +127. D'autre part les valeurs 0 (tout à zéro) et 255 (tout à un) sont des valeurs réservées.
  - Théoriquement un byte signé propose un rang de [-128, +127]mais du fait des valeurs réservés l'exposant pratique est limité à [-126,+127].

Exemples

Valeur	Hexadécimal	S E	Mantisse binaire
0.0f	0x00000000	0 0	(0.)000 0000 0000 0000 0000 0000
0.1f	0x3dcccccd	0 123	(1.)100 1100 1100 1100 1100 1101
1.0f	0x3f800000	0 127	(0.)000 0000 0000 0000 0000 0000
2.0f	0x40000000	0 128	(0.)000 0000 0000 0000 0000 0000
1/3	0x3eaaaaab	0 125	(0.)010 1010 1010 1010 1010 1011
-10.0f	0xc1200000	1 130	(1.)010 0000 0000 0000 0000 0000
Epsilon 2^{-23}	0x34000000	0 104	(1.)000 0000 0000 0000 0000 0000
Plus petit positif 2^{-126}	0x34000000	0 1	(1.)000 0000 0000 0000 0000 0000
1000000.0f	0x49742400	0 146	(1.)111 0100 0010 0100 0000 0000

- Rappelez vous que l'exposant se calcul en soustrayant le biais qui vaut +127 pour un réel simple précision.
- Ainsi pour l'epsilon $2^{-23} \Leftrightarrow 2^{127-23}$ donc E = 104
- Idem pour le plus petit entier positif $2^{-126} \Leftrightarrow 2^{127-126}$ donc E = 1 ici.

Valeurs spéciales

- Certaines valeurs comme zéro ne sont pas représentable par la formule IEEE-754 mais lorsque l'exposant vaut 0 ou 255 il faut comprendre qu'il s'agit de valeurs spéciales.

E	**Fraction**	**S**	**Valeur V**
0	0	0	V = 0
0	0	1	V = -0 (0 == -0)
0	!= 0		$V = (-1)^s * (1.F) * 2^{E-126}$
255	0	0	V = +INFini
255	0	1	V = -INFini
255	!= 0		V = NaN (Not a Number)

- Lorsque E=0 et que F!=0 on a un nombre particulier dit « dé-normalisé » (ou sub-normal). Il s'agit d'un mode spécial qui permet de représenter les valeurs plus petites que 2^{E-126} soit des valeurs extrêmement petites.

Float 16 bits

- Développé en software et hardware par de nombreux acteurs de l'imagerie (SGI, Nvidia et le format 'half' du CG, les formats d'images RGBE et OpenEXR…)
- IEEE float16 : 1 bit de signe, 5 bits d'exposant et 10 bits de partie fractionnelle
- Le format est maintenant connu par la dénomination binary16
  - les CPU récents disposent de l'extension F16C pour les conversions 32 bits ⇔16 bits
- Google a développé une alternative pour TensorFlow nommée BFloat16 où l'exposant est similaire à celui d'un float 32 bits (8 bits) ce qui laisse 7 bits pour la partie fractionnelle.
  - Conserve le même espacement des nombres qu'en float mais la précision est fortement réduite
  - Support hardware dans les processeurs Intel et ARM les plus récents
- IBM DLfloat16 (DL=Deep Learning) 1 bit, 6 bits d'exposant et 9 bits fractionnels

Notations et mesures

unités de mesure des octets

- Le symbole de l'octet est la lettre « o » minuscule.
  - Un B majuscule désigne un byte de 8 bits (pourrait être confondu avec le symbole B désignant un Bel, mesure de la puissance d'un son, mais fort heureusement cette mesure est plus souvent en décibels, soit dB)
  - tandis qu'un b minuscule désigne un bit.

- Pendant plusieurs années il y'a eu une confusion concernant la mesure des octets.
- En effet, par soucis pratique, 1024 octets étant proche de 1000, soit 1 kilo, les informaticiens ont pris pour acquis qu'un 1 kilo-octet (abréviation ko,) équivaut à 1024 octets.
  - Ce qui conduit naturellement à considérer que 1 méga octet (Mo) était équivalent de 1024 * 1024 octets.
- Tout ceci est un abus de langage qui est définitivement faux.

Kilo standard et Kilo binaire

- Le Système International (SI) considère qu'un kilo est toujours équivalent à 10^3.

- En 1998, afin de résoudre définitivement cette confusion, ce qui n'est pas toujours pas évident tant l'abus de langage est largement répandu, le SI a défini une norme afin de distinguer le kilo standard du kilo binaire (appelé **kibi**).

- On a alors

  1 **ko** = 10^3 soit 1000 octets

  1 **kio** = 2^{10} soit 1024 octets (**kio** désignant le kilo binaire)

Poids en octets

- kilo-octet (ko/kB) 10^3 kibi-octet (kio) 2^{10}
- méga-octet (Mo/MB) 10^6 mébi-octet (Mio) 2^{20}
- giga-octet (Go/GB) 10^9 gibi-octet (Gio) 2^{30}
- tera-octet (To/TB) 10^{12} tibi-octet (Tio) 2^{40}

- La standardisation porte également sur les bits
  - on a ainsi 1 kilo-bit (kb) = 1000 bits
  - tandis que 1 kibi-bit (kib) = 1024 bits

Vitesse, Taux et Fréquence

- La **vitesse**, ou **taux**, de **transfert** est exprimée en **octets par seconde**, ceci lorsque les données transitent via des bus locaux (vers la RAM, disque durs etc…).

- Lorsque le transfert s'effectue vers l'extérieur (réseaux, haut-parleurs etc…) on exprime le taux de transfert en bits par seconde, voire comme autrefois en bauds, mesure exprimant le nombre de symboles transférés par seconde.
  - Le terme **baud** provient d'Emile Baudot inventeur du code Baudot utilisé en télégraphie.

- La **fréquence** est exprimée en **hertz** (hz), et désigne le nombre d'événements par seconde. Pour un processeur il s'agit de la fréquence à laquelle son horloge interne émet un signal de synchronisation (appelé cycle). La vitesse d'un processeur s'exprime plus généralement en instructions par secondes (IPS).
  - Il existe des processeurs dits asynchrones car non synchronisé sur l'horloge