Langage Assemblage et Jeu d'Instructions

Cours (9x2h00) :

- Frédéric Mallet - fmallet@echo.unice.fr

TP (9x2h00 - 1 groupe) :

- Frédéric Mallet - fmallet@echo.unice.fr

http://deptinfo.unice.fr/~fmallet/laji/

Structure du cours

- Performances : évolution et comparaison
- Codage de l ’information
- Fonctions logiques et éléments mémoires
- Systèmes à microprocesseurs
- La famille Intel - 80x86
- La famille IBM - UltraSPARC
- Éléments avancés (parallélisme, mémoire, ...)
- Conception d'architectures numériques - VHDL

Intel 80x86 : Historique (1/7)

- 1970 : 4004 (2300 transistors)
  - Processeur 4-bit pour Busicom : calculatrice électronique
- 1972 : 8008 (3500 transistors)
  - Premier microprocesseur 8-bit, mémoire de 16ko
- 1978 : 8086/8088 (29000 transistors)
  - Architecture 16 bits avec registres **16 bits**
  - Architecture à *accumulateur étendu* : registres 'spécialisés'
    - Si aucune destination n'est fournie, les calculs sont accumulés dans un accumulateur
    - On peut utiliser le contenu de l'accumulateur à la place d'un registre
  - Mode réel à segments pour un adressage sur **20 bits** (1Mo)

La segmentation de la mémoire

- Avec un registre de 16 bits il est impossible d'adresser un espace de 2^{20} bits.
- Le 8086 utilise des registres de segments (16 bits):

  CS, DS, ES, SS

  - Un segment est une zone de 64Ko (16 bits), une adresse se réfère à un segment donné : adresse relative
  - Les segments sont espacés à intervalles de 16 octets
  - L'adresse physique est calculée à partir de l'adresse logique et le numéro de segment :
    - Physique = numéro_de_segment * 16 + adresse_logique

Exemple d'utilisation des segments

- L'adresse d'un octet se note XXXX:YYYY
  - où XXXX est le numéro de segment et YYYY l'offset
  - L'adresse 0000:0010 dénote l'adresse physique 0x10
  - L'adresse 0000:0100 dénote l'adresse physique 0x100
  - L'adresse 0001:0000 ne dénote PAS l'adresse physique 0x10000 mais l'adresse 0x10 également (octet 0 du segment 1)
- Pas d'unicité de la représentation d'une adresse
  - Octet 66 = adressé par 0000:0042, 0001:0032, 0002:0022, 0003:0012, 0004:0002
- Les bouts de 16 octets s'appellent les paragraphes

Historique (2/7)

- 1980 : 8087 – coprocesseur (Floating-Point Unit)
  - Le jeu d'instructions est étendu de 60 instructions sur réels
  - Architecture à pile étendue pour les instructions sur les réels
    - 'faux registres' de 36 bits
    - Un opérande optionnel détermine la position de l'opérande par rapport au sommet de pile
  - Exemple :
    - FADD 5.0 ; additionne 5.0 au sommet de pile et empile le résultat
      ; ST(0) = ST(0) + 5.0
    - FADD ST2 ; additionne la troisième donnée de la pile avec le
      ; sommet de pile, et empile le résultat
      ; ST(0) = ST(0) + ST(2)

Historique (3/7)

- 1982 : 80286 – mode protégé (134000 transistors)
  - Espace d'adressage de 24 bits
  - Les registres de segment deviennent des sélecteurs
    - Pointeurs dans une table de description (base,limite,droits)
  - Des protections au niveau des segments sont offertes pour l'exécution sécurisée multi-tâches
    - Le code n'a le droit d'accéder à tous les segments que sous certaines conditions (4 modes de protection)
    - Une vérification de la limite d'adressage est effectuée.
  - Le mode protégé est le mode par défaut
    - Mode recommandé et mode le plus performant

Historique (4/7)

- 1985 : 80386 – processeur 32 bits (275000 transistors)
  - Les registres 16 bits sont étendus sur 32 bits
  - Espace d'adressage de 32 bits : 4Go
  - Les segments ne sont utilisés que pour la protection
  - Des instructions et modes d'adressages sont ajoutés pour en faire une architecture à registres généraux

x86-32
IA-32

- 1989 : 80486 (1.2M transistors)
  - Intègre le FPU dans le processeur
  - Intègre un cache de niveau 1 (8Ko)
  - Pipeline RISC
  - Support multi-processeurs *built-in*

Historique (5/7)

- Compatibilité ascendante avec 8086 : mode réel
- Pas de *Trademark* sur les chiffres
- 1993 : Pentium (3.1M transistors)
  - Seules 4 instructions ont été rajoutées, co-processeur MMX
  - mais l'architecture physique évolue énormément pour les performances : pipeline, cache, prédiction, ...
- 1995 : Pentium Pro, AMD K5 (5.5M transistors)
  - Intègre un cache de niveau 2
  - Architecture complètement différente : 5 instructions à la fois
- 1997 : Pentium II, AMD K6 (7.5M transistors)
  - Technologie MMX intégrée : Accélération pour le multimédia

Historique (6/7)

- 1998-? : variantes
  - Celeron : Ligne *budget*, moins chère et moins performante
  - Xeon : cache plus grand, bus plus rapide
- 1999 : Pentium III (9.5M transistors)
  - SSE (Streaming SIMD Extensions) : graphisme 3D (128bits)
  - SIMD : Single Instruction Multiple Data
- 2000 : Pentium 4 (42M transistors) (115W à 3.6GHz)
  - Hyperthreading : 2 taches de contrôle simultanées, 2 jeux de registres, SSE2
- 2003 : Pentium M (Centrino)
  - + petit, + léger, basse consommation, avec WiFi *built-in*

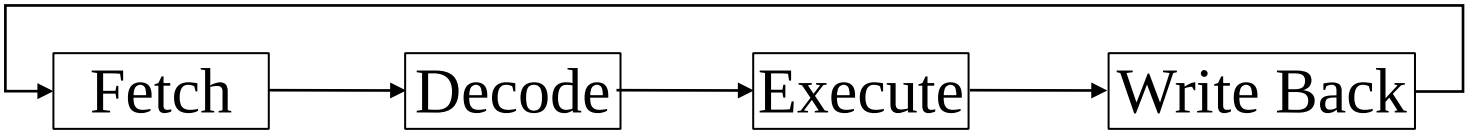
Historique (7/7)

- 2003 : L'Opteron de AMD
  - AMD transforme x86-32 en x86-64 (x64)
  - Implémenté dans proc. Intel, AMD, Cyrix, VIA, …
- 2004 : SSE3
- 2005/2006 : Dual core – Inter Core 2
  - SSE4, Itanium 2 (1,7 milliards de transistors)
- 2007 : quad-core, FPU 128 bits
- 2008 : Intel Core i7
  - L3 cache, in-order highly pipelined, very low power
  - VIA Nano: out-of-order, superscalaire.

Familles d'architectures (1/3)

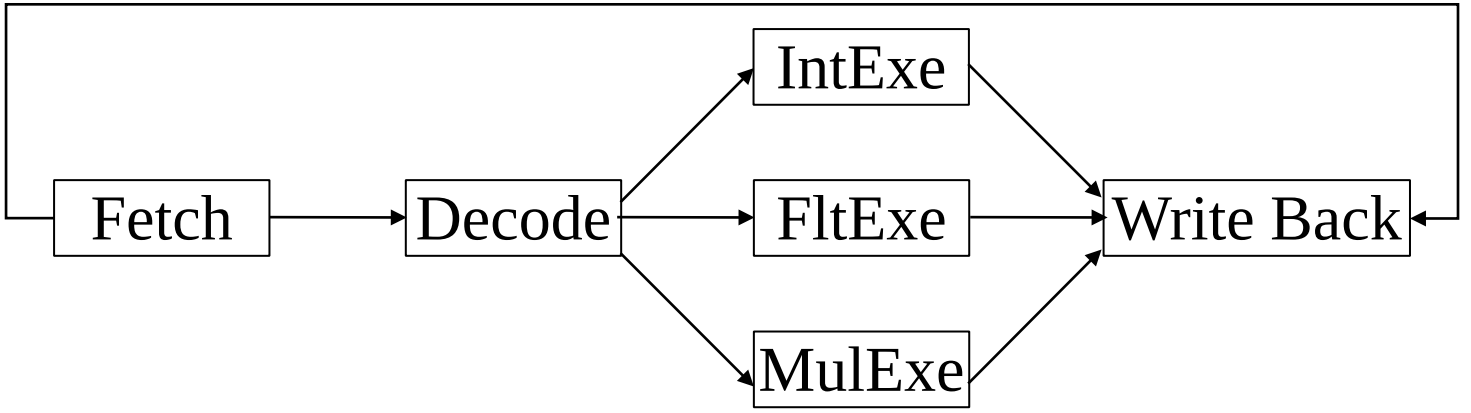
- RISC – Reduced Instruction Set Computer
  - Toutes les instructions ont la même taille
  - Que des modes d'adressage simples (immédiat, registre)
- CISC – Complex Instruction Set Architecture
  - Instructions de taille et de temps d'exécution variable
  - Modes d'adressage complexes
  - Il faut plusieurs instructions RISC pour 1 instruction CISC
- DSP – Digital Signal Processor
  - Mixte CISC/RISC : optimisés pour une application donnée

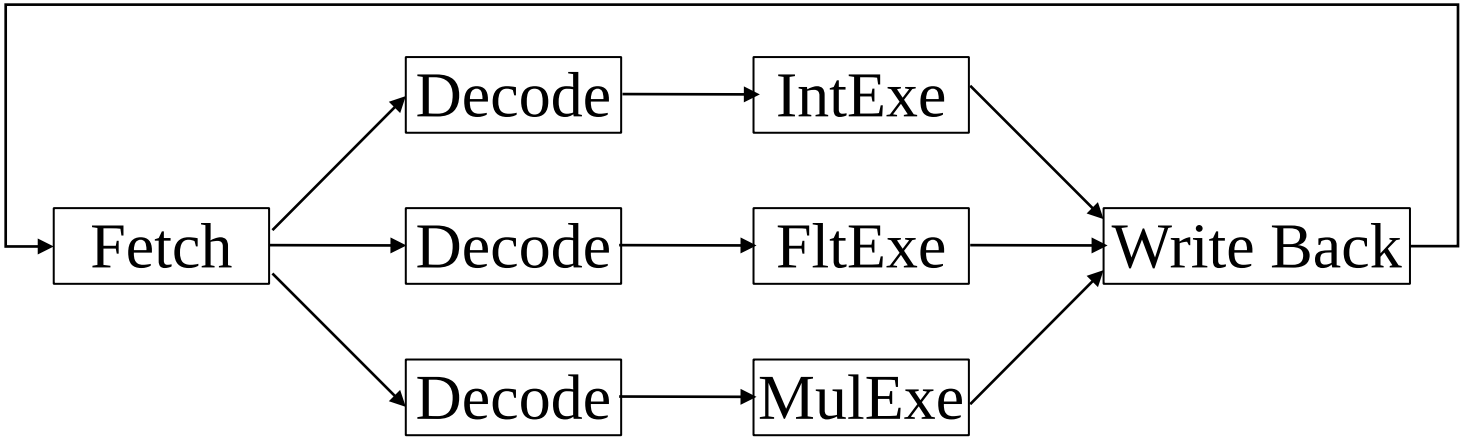
CISC/RISC



Super scalaires

- Plusieurs unités fonctionnelles (2 UAL, 2 multiplieurs, etc.)
- 1 seule instruction est décodée à la fois

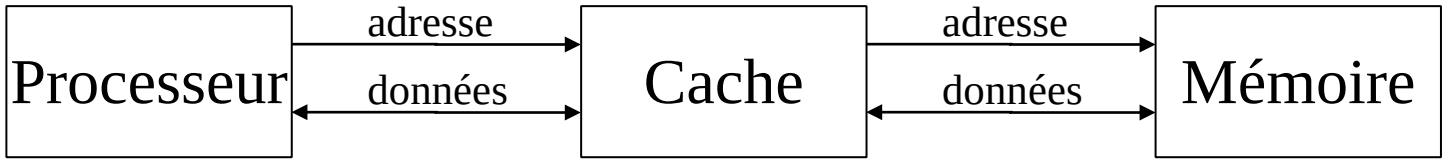




- VLIW – Very Long Instruction Word
  - Plusieurs unités fonctionnelles
  - 1 instruction est composée de plusieurs mini-instructions RISC indépendantes qui sont toutes exécutées en même temps
- EPIC – Explicit Parallel Instruction Computing
  - VLIW + mais il peut y avoir des dépendances entre les mini-instructions

Mémoire cache (aperçu)

- Il s'agit d'interposer une mémoire plus petite, donc plus rapide (faible latence) entre le processeur et la mémoire : mémoire cache
- Le cache étant plus petit, il ne peut pas contenir toute la mémoire :
  - Il faut associer plusieurs adresses mémoire à la même ligne du cache



L’assembleur NASM : pourquoi?

- Les outils d'assemblage transforment du code assembleur en langage machine
  - De `add r0,r1,r2` vers `0x0C010103`
  - Certains outils (MASM, GAS, TASM) utilisent une syntaxe assez différente de l'assembleur/langage machine réel
  - Ce n'est pas le cas de NASM qui est 'dédié' à x86
- De plus NASM fonctionne sous plusieurs OS
  - Windows et Linux
- La configuration de NASM est minimale

Cycle de développement

- La commande *nasm* permet d'assembler un programme assembleur et de créer un fichier objet
- Sous Linux on utilisera l'option *-f elf* pour générer des fichiers objets 32 bits
  - `nasm -f elf file.asm` *crée le fichier file.o*
- Ensuite il faut lier le fichier objet pour en faire un exécutable : commande *ld*
  - `ld -o file file.o` *crée le fichier exécutable file*
- Pour déboguer, on utilise la commande *ald*

Interface avec du C

- Rien n'empêche de lier le fichier généré par *nasm* avec d'autres fichiers objets ou bibliothèques extérieures
- e.g.
  - `nasm -f elf fichier1.asm` → fichier1.o
  - `gcc -c fichier2.c` → fichier2.o
  - `gcc fichier1.o fichier2.o -o fichier` → fichier

Les sections de NASM

- On utilise des sections différentes pour placer le code et les données (section ≠ segment)
- 3 sections importantes :
  - .text : code assembleur
  - .data : données initialisées
  - .bss : données non initialisées (allocation de mémoire)
- On utilise le mot clé section pour indiquer le début de la section
  - e.g. `section .text`

_start : Point d'entrée

- Le label _start est le point d'entrée par défaut d'un programme assemblé par NASM
- C'est l'équivalent de la fonction *main* en C
- Commande *global* : rend un label visible de l'extérieur et utilisable par d'autres fichiers objets,
  - e.g. `global _start`
  - Obligatoire pour créer un fichier exécutable
- Attention, déclarer *_start* global empêche de lier avec un fichier C qui contient un *main*.

Zone de code (.text)

- Une ligne de la zone de code ressemble à :

  label : instruction opérandes ; commentaire

- Une zone de code standard est :

```
section .text
   global _start
_start :
   .... ; commentaires

   ; instructions pour rendre la main à l'OS
```

Zone de données initialisées (.data)

Pseudo instructions DB, DW, DD, DQ, DT

```
db 0x55 ; juste l'octet 0x55
db 0x55,0x56,0x57 ; 3 octets consécutifs
db ’a’,0x55 ; caractère constant
db ’hello’,13,10,’$’ ; chaîne de caractères constantes
dw 0x1234 ; 0x34 0x12
dw ’a’ ; 0x61 0x00 (juste un chiffre)
dw ’ab’ ; 0x61 0x62 (caractère constant)
dw ’abc’ ; 0x61 0x62 0x63 0x00 (string)
dd 0x12345678 ; 0x78 0x56 0x34 0x12
dd 1.234567e20 ; floating-point constant
dq 1.234567e20 ; double-precision float
dt 1.234567e20 ; extended-precision float
```

Pseudo instructions EQU

```
message db ’hello, world’
msglen equ $-message
```

Zone de données initialisées (.data)

- Pseudo instructions TIMES

```
zerobuf: times 64 db 0
```

  - L'argument de *times* peut être une expression numérique

```
buffer: db 'hello, world'
times 64-$+buffer db ' '
```

  - Permet de créer un emplacement exactement de 64 octets, quelle que soit la taille de *buffer*,
  - `TIMES` peut s'appliquer à des instructions

```
times 100 movsb
```

Zone de données non initialisées (.bss)

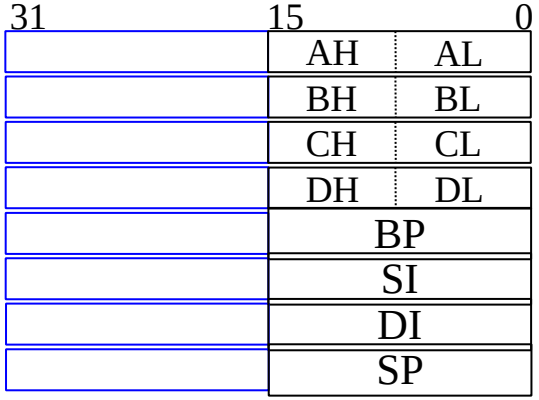
- Pseudo instructions RESB

```
buffer: resb 64 ; réserve 64 octets
wordvar: resw 1 ; réserve 1 mot de 16 bits
realarray: resq 10 ; tableau de 10 nombres réels
```

  - Tous les types sont utilisables :
    - RESB, RESW, RESD, RESQ, REST
- Allocation de mémoire sans donner de valeur

- Pour plus d'informations :
  - Documentation en ligne à http://nasm.sourceforge.net

Le Jeu d'instructions : les registres



16 bits	32 bits	Fonction privilégiée
AX	EAX	Accumulateur
BX	EBX	Pointeur vers les données (segment DS)
CX	ECX	Compteur de boucle et d'opérations sur les chaînes
DX	EDX	Pointeur pour les entrées/sorties
BP	EBP	Pointeur vers les données de la pile
SI	ESI	Pointeur 'SOURCE' vers données du segment DS
DI	EDI	Pointeur 'DESTINATION' vers données du segment DS
SP	ESP	Pointeur de pile : sommet de la pile (segment SS)

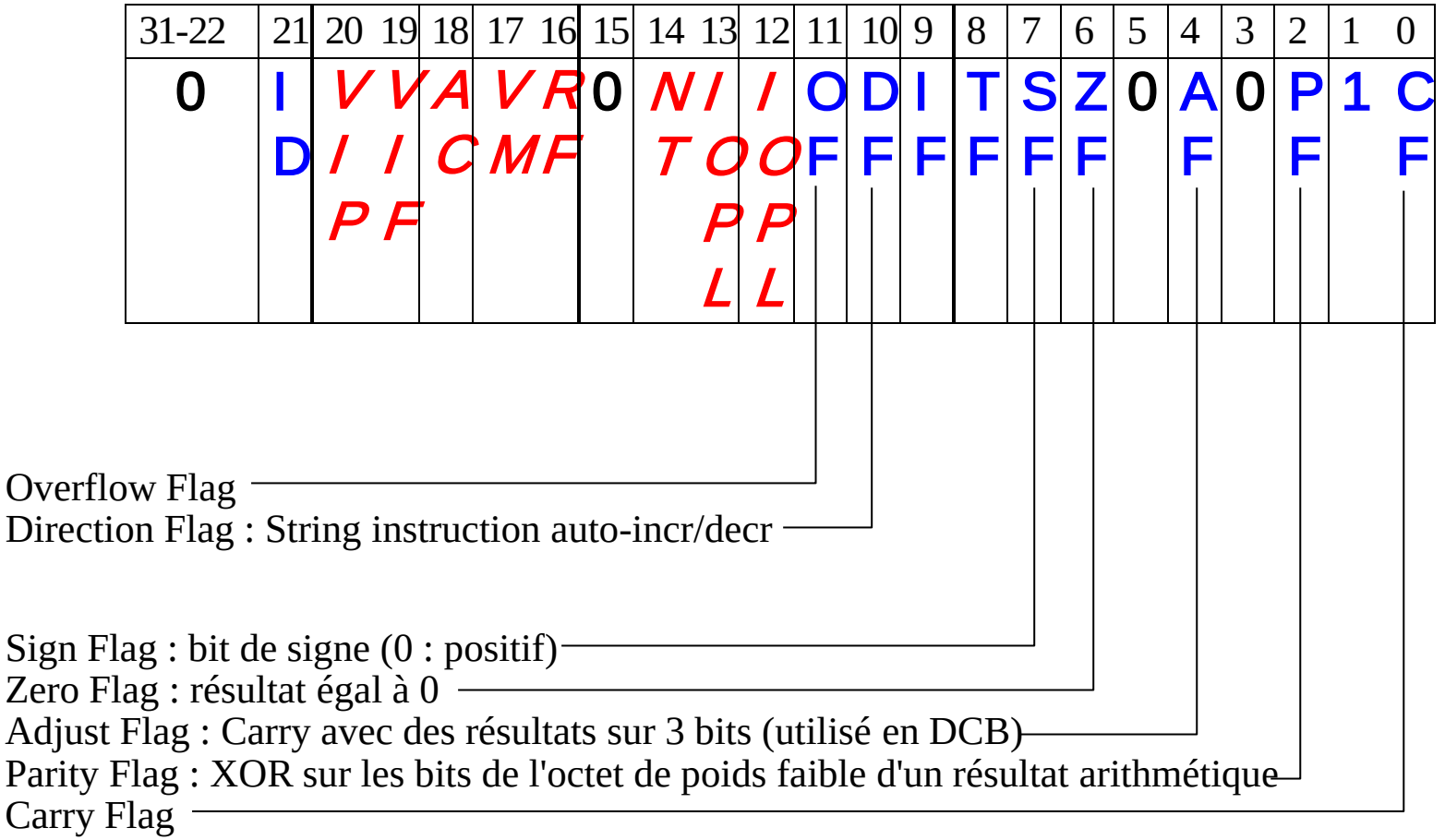
Registres 'généraux' - 32 bits

	Mémoire segmentée (obligatoire en mode réel)	Mémoire à plat – non segmentée
CS	Segment de code (programme)	Segment de code (programme)
DS	Segment de données	Segment de données et pile
SS	Segment de pile	Segment de données et pile = DS
ES	Segment de données additionnel	Segment de données et pile = DS
FS	Segment de données additionnel	Segment de données et pile = DS
GS	Segment de données additionnel	Segment de données et pile = DS

Registres segments - 16 bits

Les registres (suite)

- Pointeur d'instruction EIP : équivalent au PC
- Registre de contrôle et d'état (EFLAGS)



Le mode 16-bit ou 32-bit

- En mode 16-bit
  - seuls les registres 16 bits sont accessibles : segment et données
  - Les opérandes peuvent être sur 8 ou16 bits
- En mode 32-bit
  - Les registres 16 et 32 bits sont accessibles
    - Segment : 16-bits, données : 32-bits
  - Les opérandes peuvent être sur 8 ou 32 bits
- Il est interdit de mélanger les modes

Petits et gros boutiens / little and big endian

- Désigne la manière dont un ordinateur organise les octets qui composent un mot.
- Gros boutiens : octet de poids fort en premier
  - Le nombre (valeur, adresse ou instruction) $1A2B3C4D_h$
    - Est codé en mémoire 1A 2B 3C 4D (octet de poids fort 1A)
  - Exemples : Motorola 68K, PowerPC, MIPS, SPARC, ...
- Petits boutiens : octet de poids faible en premier
  - Le nombre (valeur, adresse ou instruction) 0x1A2B3C4D
    - Est codé en mémoire 4D 3C 2B 1A (octet de poids faible 4D)
  - Exemples : Intel, ARM, Alpha, Vax

Les familles d'instructions (1/4)

- 4 grandes familles d'instructions
  - Instructions entières (plusieurs sous-familles)
    - Instructions de déplacement : MOV, XCHG, PUSH, POP, IN/OUT
    - Arithmétique binaire : ADD,SUB,MUL/IMUL,INC,DEC,NEG,CMP
    - Arithmétique décimale : DAA, DAS (Décimal codé binaire)
    - Logique et décalage : AND, OR, XOR, NOT, SHL, SAR, ROL, ROT
    - Bit et octets : BT, BTS, BTR, BTC (set the CF flag)
    - Contrôle : JMP, CALL, RET, INT ; conditionnels JG, JL, JE, JC, JS
    - Chaînes : MOVS, CMPS
    - ENTER/LEAVE : pour faciliter l'appel de procédures
    - Instructions sur EFLAGS ou sur les registres de segments, ...

Instructions de déplacement

Modes d'adressage	Source → Destination
Mémoire vers registre	Mémoire → registre général Mémoire → registre de segment
Registre vers mémoire	registre de segment → mémoire Registre général → mémoire
Registre vers registre	Registre général → registre général Registre général → registre segment Registre segment → registre général Pas de segment vers segment
Immédiate vers registre	Valeur immédiate → registre général
Immédiate vers mémoire	Valeur immédiate → mémoire
Mémoire vers mémoire	Utiliser MOVS (MOV String)

☆ **MOV Dest, Source**

Réalise : Dest := Source

Ex: MOV CL, AL
MOV AX, 12
MOV EAX, 10101010h

☆ **XCHG D1, D2**

Réalise : D1:= D2 || D2:= D1

Ex: XCHG EBX, ECX

B.4.156 MOV: Move Data

```
MOV r/m8,reg8
MOV r/m16,reg16
MOV r/m32,reg32
MOV reg8,r/m8
MOV reg16,r/m16
MOV reg32,r/m32

MOV reg8,imm8
MOV reg16,imm16
MOV reg32,imm32
MOV r/m8,imm8
MOV r/m16,imm16
MOV r/m32,imm32

MOV AL,memoffs8
MOV AX,memoffs16
MOV EAX,memoffs32
MOV memoffs8,AL
MOV memoffs16,AX
MOV memoffs32,EAX

MOV r/m16,segreg
MOV r/m32,segreg
MOV segreg,r/m16
MOV segreg,r/m32
```

Adressage de la mémoire

- Une adresse effective (éventuellement relative à un registre de segment) est une combinaison de :
  - Un Déplacement (absent, 8 bits, 16 bits, 32 bits)
  - Une base (EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI)
  - Un indexe (EAX,EBX,ECX,EDX,EBP,ESI,EDI)
  - Un facteur d'échelle (1,2,4 ou 8)
- Adresse effective = base + indexe*facteur + déplacement

Exemples avec NASM

- Si *wordvar* et *offset* sont déclarées dans *.data*

```
mov ax,[wordvar]
mov ax,[wordvar+1]
mov ax,[es:wordvar+bx]
```

- **Adressages plus complexes**

```
mov eax,[ebx*2+ecx+offset]
mov ax,[bp+di+8]
```

- NASM peut simplifier ou convertir des expressions qui semblent illégales

```
mov eax,[ebx*5] ; assemble comme [ebx*4+ebx]
mov eax,[eax*2] ; assemble comme [eax+eax] car + court
                ; 2 octets au lieu de 4
```

Instructions sur la pile

- PUSH, POP
  - Utilise le pointeur de pile ESP
  - Adressage immédiat, par registre (tous), direct (@)
  - PUSH : pointeur de pile décrémenté de 4, puis empile
  - POP : dépile, puis pointeur de pile incrémenté de 4
- PUSHA, POPA
  - Empile et dépile tous les registres dans l'ordre
    - EAX, ECX, EDX, EBX, Vieux ESP, EBP, ESI, EDI

Instructions arithmétiques binaires

Incrémentation, décrémentation

- **INC Registre**

  DEC Registre

  Réalise : Registre := Registre +/- 1

Addition/soustraction avec ou sans retenue

- **ADD Registre, Valeur**

  ADC Registre, valeur (utilise CF)

  Réalise : Registre := Registre + valeur

- **SUB Registre, Valeur**

  SBB Registre, valeur (utilise CF)

  Réalise : Registre := Registre - valeur

```
ADD r/m8,reg8
ADD r/m16,reg16
ADD r/m32,reg32

ADD reg8,r/m8
ADD reg16,r/m16
ADD reg32,r/m32

ADD r/m8,imm8
ADD r/m16,imm16
ADD r/m32,imm32

ADD r/m16,imm8
ADD r/m32,imm8

ADD AL,imm8
ADD AX,imm16
ADD EAX,imm32
```

Modes d'adressage pour ADD

Codage	Instruction	Description
04 ib	ADD AL, imm8	Ajoute imm8 à AL
05 iw	ADD AX, imm16	Ajoute imm16 à AX
05 id	ADD EAX, imm32	Ajoute imm32 à EAX
80 /0 ib	ADD r/m8, imm8	Ajoute imm8 à r/m8
81 /0 iw	ADD r/m16, imm16	Ajoute imm16 à r/m16
81 /0 id	ADD r/m32, imm32	Ajoute imm32 à r/m32
83 /0 ib	ADD r/m16, imm8	Ajoute imm8 à r/m16 (extension)
83 /0 ib	ADD r/m32, imm8	Ajoute imm8 à r/m32 (extension)
00 /r	ADD r/m8, r8	Ajoute r8 à r/m8
01 /r	ADD r/m16, r16	Ajoute r16 à r/m16
01 /r	ADD r/m32, r32	Ajoute r32 à r/m32
02 /r	ADD r8, r/m8	Ajoute r/m8 à r8
03 /r	ADD r16, r/m16	Ajoute r/m16 à r16

ib : octet

iw : mot (16 bits)

id : double (32 bits)

/0 : reg OU mem

/r : reg ET (reg ou mem)

r/m8 : reg ou mem 8-bit

r/m16 : reg ou mem 16-bit

r/m32 : reg ou mem 32-bit

Pas de différences entre les opérations signées et non signées : utilise SF

Multiplication signée ou non signée

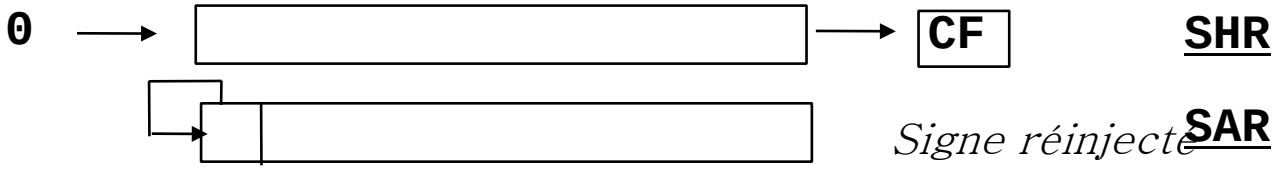
- IMUL : multiplication signée
- MUL : multiplication non-signée
  - Multiplication 8-bit, résultat 16-bit dans AX
  - Multiplication 16-bit, résultat 32-bit dans DX:AX
  - Multiplication 32-bit, résultat 64-bit dans EDX:EAX
- Modes d'adressage :

```
MUL r/m8
MUL r/m16
MUL r/m32
```

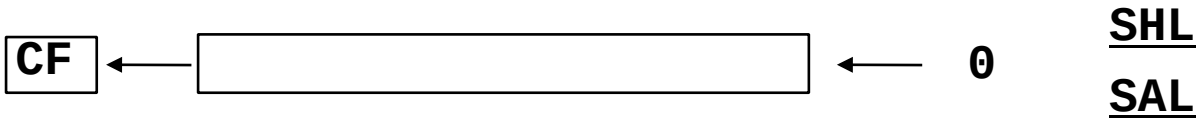
```
IMUL r/m8
IMUL r/m16
IMUL r/m32

IMUL reg16,r/m16
IMUL reg32,r/m32

IMUL reg16,imm8
IMUL reg16,imm16
IMUL reg32,imm8
IMUL reg32,imm32

IMUL reg16,r/m16,imm8
IMUL reg16,r/m16,imm16
IMUL reg32,r/m32,imm8
IMUL reg32,r/m32,imm32
```

Opérations de décalage

- Décalage logique (SHx) : pas de signe
- Décalage arithmétique (SAx) : conserve le signe
- Décalage à droite : division par 2^i



- Décalage à gauche : multiplication par 2^i



☆ **SXX Opérande, nombreDeBits**

Exemples de décalage à droite

- Décalage logique (SHR)

  $7_{10} = 0111_2$ $\quad\quad$ $-7_{10} = 1001_2$

  SHR 7,1 = $0011_2 = 3_{10}$ et CF=1

  SHR -7,1 = $0100_2 = 4_{10}$ et CF = 1

- Décalage arithmétique (SAR)
  - SAR 7,1 = $0011_2 = 3_{10}$ et CF=1
  - SAR -7,1 = $1100_2 = -4_{10}$ et CF = 1

Instructions de branchement

- Branchements : JMP, CALL, RET, INT
- Branchements conditionnels : JS, JC, ...
- Boucles matérielles : LOOP, LOOPNE, LOOPE
  - Utilise ECX comme compteur de boucle
  - LOOP label : ECX--, saut si ECX!=0
  - LOOPNE label : ECX--, saut si ECX!=0 et SF!=0
  - LOOPE label : ECX--, saut si ECX!=0 et SF==0
  - ECX-- ici ne modifie pas le registre d'état

Exemple d'utilisation de LOOPNE

- Calcule la taille d'une chaîne de caractères qui a au maximum 80 caractères

```
   mov ESI, Chaine  ; début de la chaîne de caractères
   mov ECX, 80      ; 80 caractères max
   mov AL, 0        ; caractère de fin de chaîne
do:                 ; étiquette
   cmp AL, [ESI]    ; comparaison
   inc ESI          ; caractère suivant
   loopne do        ; ECX--, boucle si ECX != 0 ET ZF=0
```

- *Le résultat est ESI - Chaine*

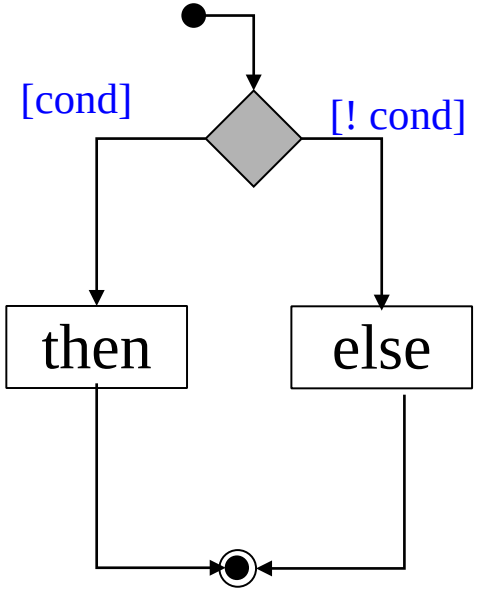
Sous-programme : CALL et RET

- CALL adresse : appel d'un sous-programme
  - Adresse de retour (instruction suivante) est empilée
  - Le pointeur de pile est décrémenté (ESP = ESP-4)
  - Le branchement est effectué
  - L'appelant et l'appelé doivent se mettre d'accord sur un protocole pour passer les paramètres éventuels (dans les registres : nombre limité, dans la pile)
- RET : retour d'un sous-programme
  - L'adresse de retour est dépilée puis branchement
  - La pile DOIT avoir été vidée : responsabilité du sous-programme

Structures de contrôles

- Un processeur ne connaît que les structures de contrôle simples
  - Séquence :
    - comportement par défaut
  - Branchement : rupture de séquence
    - Branchements inconditionnels : JMP, CALL, INT
    - Branchements conditionnels : Jcc, Loop, Loopxx
- Pas de structure de contrôle de haut niveau
  - if/else, while, do/while, for, switch

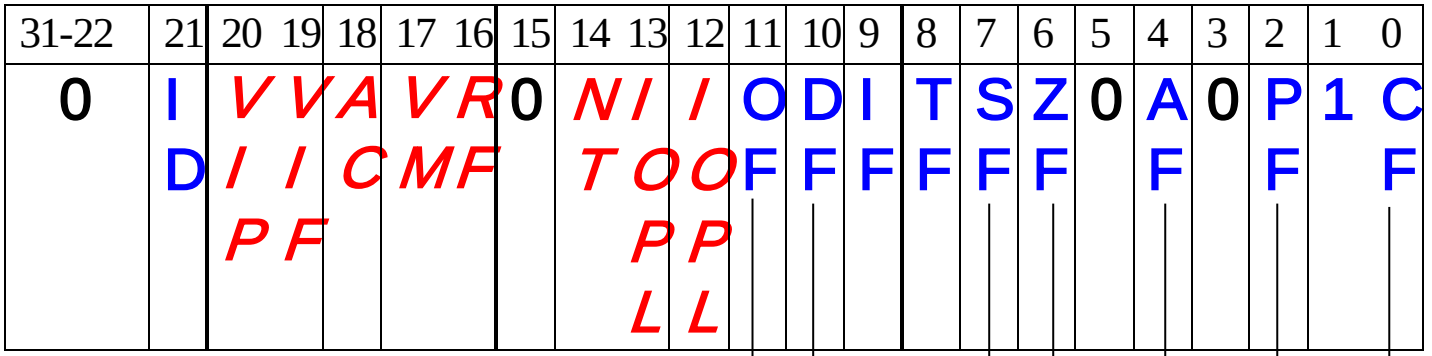
If/else



- Il y a 2 cas (1 bit pour le codage)
  - Soit la condition est vraie, alors on exécute l'instruction *then*
  - Soit la condition est fausse, alors on exécute l'instruction *else*
- En assembleur, les ruptures de séquences sont conditionnelles à la valeur des bits du registre EFLAG
  - Calcul qui positionne 1 bit de EFLAG à 1 (ou 0) si la condition est vraie, à 0 (ou 1) sinon.
  - Utiliser l'instruction *jcc* adéquate

Le registre EFLAGS

- Registre de contrôle et d'état (EFLAGS)



Overflow Flag
Direction Flag : String instruction auto-incr/decr

Sign Flag : bit de signe (0 : positif)
Zero Flag : résultat égal à 0
Adjust Flag : Carry avec des résultats sur 3 bits (utilisé en DCB)
Parity Flag : XOR sur les bits de l'octet de poids faible d'un résultat arithmétique
Carry Flag

Les sauts conditionnels : Jcc

- Jo : saut si OF = 1; Jno : saut si OF = 0
- Jb,Jc,Jnae : sauf si CF = 1; Jae, Jnb, Jnc : saut si CF=0
- Je, Jz : saut si ZF = 1; Jne, Jnz : saut si ZF = 0
- Jbe, Jna : si ZF+CF = 1; Ja, Jnbe : ZF+CF = 0
- Js : saut si SF = 1; Jns : saut si SF=0
- Jl, Jnge : saut si SF⊕OF = 1; Jge, jnl : si SF⊕OF = 0
- Jle, jng : saut si ZF+(SF⊕OF) = 1; jnle, jg : contraire
- Jp : saut si PF = 1; Jnp : saut si PF = 0

If/else - exemple

```
if (a==5)
   b = 6;
```

```
.if  cmp eax, 5
     jnz .finsi
.then mov ebx, 6
.finsi
```

```
.if  cmp 5, eax
     jz .finsi
.then mov ebx, 6
.finsi
```

```
if (a % 2 == 1)
   b = 6;
else
   b = 2;
```

```
.if  shr eax, 1
     jnc .else
.then mov ebx,6
jmp  .finsi
.else mov ebx, 2
.finsi
```

```
.if  bt eax, 0
     jnc .else
.then mov ebx,6
jmp  .finsi
.else mov ebx, 2
.finsi
```

```
if (a > 5) b=6;
else b=2;
```

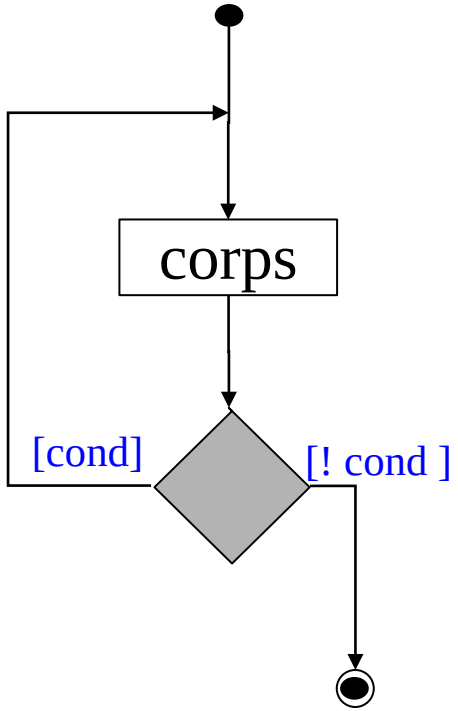
Calculs NON Signés

```
.if  cmp 5, eax
  js .then
.elsemov ebx, 2
jmp .finsi
.thenmov ebx,6
```

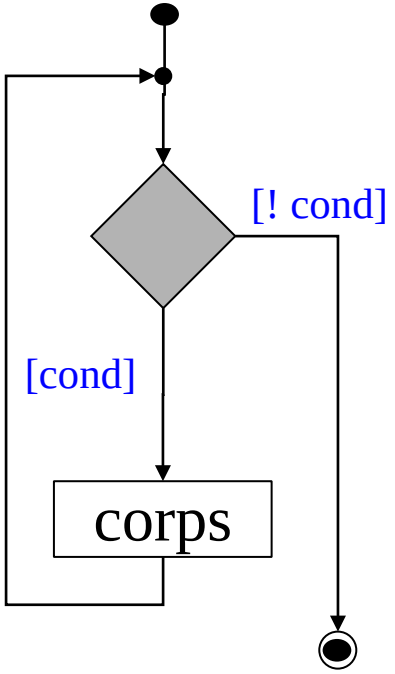
```
.if  cmp eax,5
  jg .then
```

```
.if  cmp 5,eax
  jge .else
```

do/while

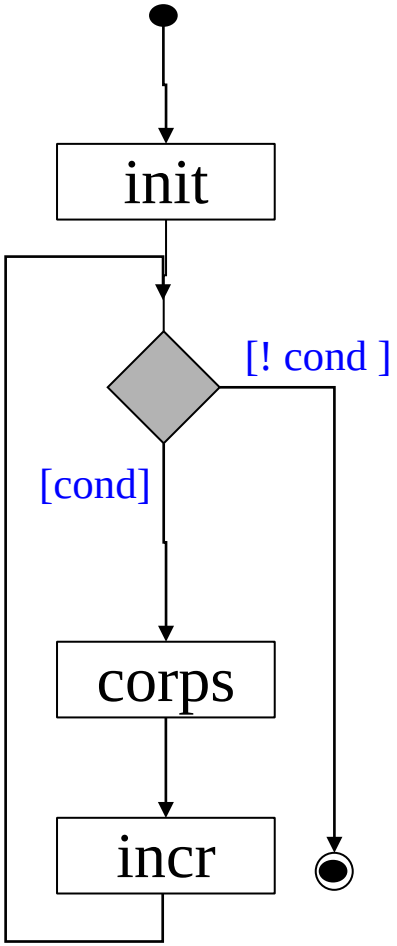



```
.do
  ; corps de boucle

  cmp {condition}
  Jcc .do
```




```
.while
  cmp {condition}
  jcc .while_end
  ; corps de boucle

  jmp .while
```

for (init; cond; incr)



- Le *for* est un cas particulier du *while*
- Cas spécial : for (i=n; i>0; i--); n>0
  - *Loop* ressemble plus à do/while

```
mov ecx, n    ; init
.for
   ; corps de la boucle

   loop .for
```

La pile et les sous-routines

Exemple

```
int ajoute (int a, int b) {
   return a + b;
}

int main(void) {
  int x = 5, y = 3;
  int c = ajoute(x, y);
}
```

```
ajoute:
    mov eax, [esp+8]
    mov ebx, [esp+4]
    add eax, ebx
  ret

_start:
  mov ecx, 3
  push ecx
  mov ebx, 5
  push ebx
  call ajoute
  add esp,8
```

```
ajoute:
   push ebx
   mov eax,[esp+12]
   mov ebx, [esp+8]
   add eax, ebx
   pop ebx
   ret
```

Que se passe-t-il si *ajoute* utilise la pile ? (e.g. pour sauver ebx)

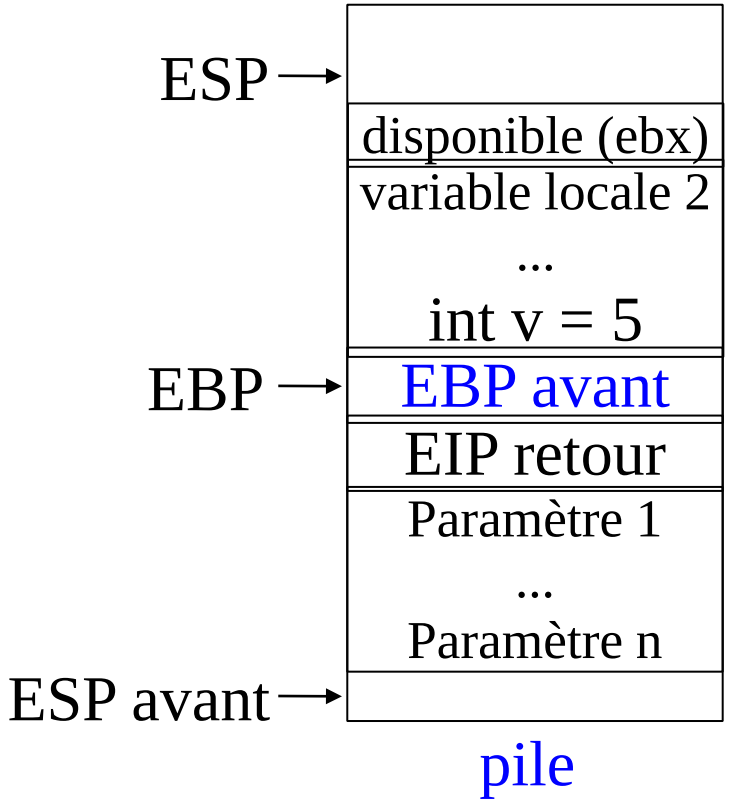
Paramètres et pile

```
ajoute:
  push ebp
  mov ebp, esp
  mov eax,[ebp+12]
  mov ebx, [ebp+8]
  add eax, ebx
  pop ebp
  ret
```

```
ajoute:
  push ebp
  mov ebp, esp
  push ebx (pusha, ...)
  mov eax,[ebp+12]
  mov ebx, [ebp+8]
  add eax, ebx
  (popa, ...) pop ebx
  pop ebp
  ret
```

Variables locales

```
push ebp
mov ebp, esp
mov eax, 5          ; int v = 5
push eax
push ebx
...
mov ebx, [ebp-4] ; ebx = v
...
pop ebx
add esp, 4    ; libère la variable locale
pop ebp
ret
```



Les interruptions : le polling

- Les périphériques (disques, port série, imprimante) sont lents par rapport au processeur :
  - UART (port série) : 130k baud → 16Ko/s
  - Processeur 1GHz, 5 cycles par OUT → 200Mo/s
  - Si le processeur attend l'UART, perte énorme de temps
- Polling : “Attente active”
  - Le processeur teste si l'UART est prête pour la suite
  - Rien d'autre n'est fait pendant ce temps
  - C'est un “téléphone sans sonnerie”

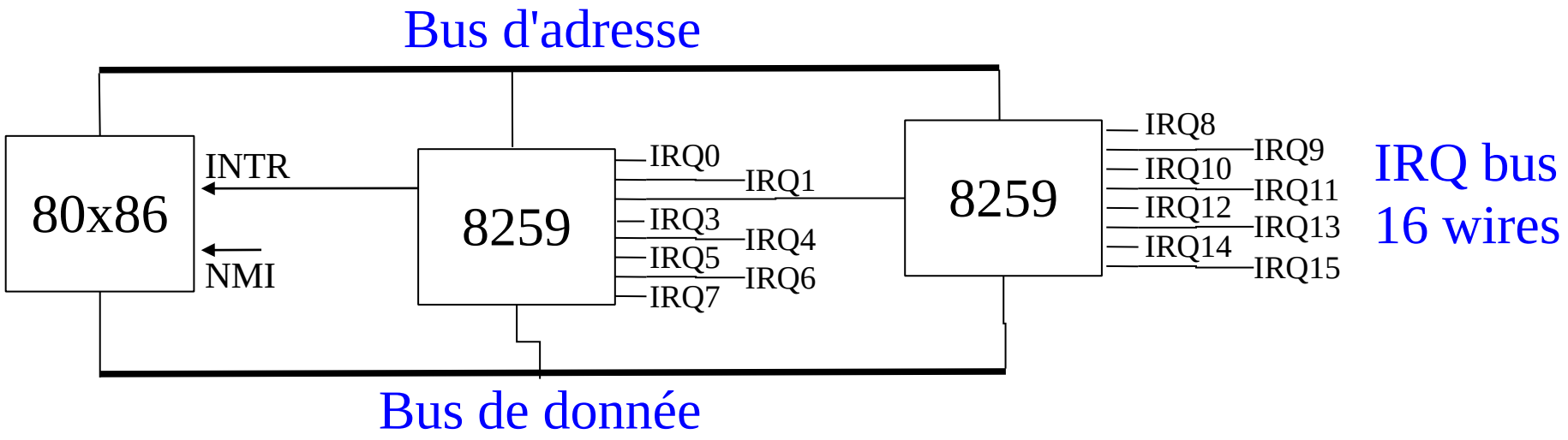
Interruptions et Exceptions

- Interrompre le traitement est 'nécessaire' dans deux situations
- Interruption :
  - Événement indépendant de l'exécution en cours (e.g. Le port série est prêt, l'horloge s'écoule, ...)
- Exception :
  - Circonstances particulières dans l'exécution d'un programme (e.g. Division par zéro)

Interruptions externes

- Broche INTR (Interrupt Request : IRQ) :
  - Interruptions masquables contrôlées par le contrôleur d'interruptions programmable (8259)
  - Chaque interruption a une priorité
  - Chaque interruption est associée à un sous-programme
- Broche NMI (Non Masquable Interrupt) :
  - Signale des événements catastrophiques (erreur mémoire, erreur de parité sur le bus)
  - Chaque interruption est associée à un sous-programme fixe

Le contrôleur d'interruptions



- Quand INTR est activé, le x86
  - Finit l'exécution de l'instruction en cours
  - lit sur le 8259 le numéro d'interruption (≠ IRQ num)
  - Branche vers le sous-programme associé (table)
- La priorité des IRQ dépend de leur numéro
  - 0>1>8>9>10>11>12>13>14>15>3>4>5>6>7

IRQ standards (ISA)

IRQ	Allocation	N° Interruption
IRQ 0	System timer	0X8
IRQ 1	Keyboard	0X9
IRQ 3	Serial port #2	0xB
IRQ 4	Serial port #1	0xC
IRQ 5	Parallel port #2	0xD
IRQ 6	Floppy Controller	0xE
IRQ 7	Parallel port #1	0xF
IRQ 8	Real Time Clock	0x70
IRQ 9	Available	0X71
IRQ 10	Available	0X72
IRQ 11	Available	0x73
IRQ 12	Mouse	0x74
IRQ 13	87 ERROR line	0X75
IRQ 14	Hard Drive Controller	0x76

- IRQ 0 est associée à l'interruption 8
- Sous linux : **`cat /proc/interrupts`**

Interruptions internes et logicielles

- Interruptions internes : détectées par le processeur
  - Faute : détectée avant l'exécution (violation de segment)
  - Trappe : résultat d'une instruction (division par zéro)
  - Abandon : protection générale
- Interruptions logicielles : instructions INT et IRET
  - Permet à un programme d'accéder à un sous-programme d'interruption
    - Pour dialoguer avec des périphériques
    - Pour dialoguer avec le système d'exploitation

Interruptions standards

N° Interruption	Description	Type
0x0	Erreur de division	Faute
0x1	Exception pas à pas	
0x2	NMI	
0x3	Point d'arrêt	Trappe
0x4	Dépassement (overflow)	Trappe
0x5	Teste limite (Bound check)	Faute
0x6	Code opération erroné	Faute
0x7	Coprocesseur non disponible	Faute
0x8	Double faute	Abandon
0x9	Dépassement segment coprocesseur	Abandon
0xA	Task State Segment Erronée	Faute
0xB	Segment absent	Faute
0xC	Exception pile	Faute
0xD	Protection générale	Faute
0xE	Défaut de page	Faute
0x10	Erreur coprocesseur	Trappe

- Certains numéros sont communs aux IRQs
  - Le sous-programme doit faire la différence

Appel d'une procédure d'interruption

- INT - Similaire au CALL
  - Faute : l'adresse de l'instruction qui cause la faute est empilée
    - L'instruction sera re-exécutée après l'interruption
  - Autre : l'adresse de l'instruction suivante est empilée
  - Le registre EFLAGS est empilé
  - Saut à l'adresse de la procédure d'interruption
    - Déterminée en utilisant une table de description (IDT)
- Retour : IRET
  - EFLAGS puis l'adresse de retour sont dépilés

Table des descripteurs d'interruption (IDT)

- En mode réel (8086) :
  - Une table de vecteur se situe à l'adresse 0
  - A chaque numéro d'interruption est associé un vecteur – CS:IP
- En mode protégé :
  - Le registre IDTR contient l'adresse de la table de descripteurs
  - Lors du boot, l'OS charge la table et positionne IDTR
  - Les instruction LIDT et SIDT permettent de manipuler le registre IDTR (Load/Store)

Les portes (gates) de l'IDT

- L'IDT est une table qui contient des entrées appelées portes de 8 octets chacune
  - IRQ 8 (Real Time Clock) = Interruption 0x70
    - Porte de l'IRQ 8 à l'adresse : IDTR + 0x70*8
- Une porte contient en particulier :
  - Un sélecteur : indexe dans la table de descripteurs de segments
  - Un déplacement
  - La combinaison des deux donne l'adresse de la procédure d'interruption à exécuter

Les interruptions imbriquées

- Rien n'empêche *a priori* une interruption externe d'être levée lors de l'exécution d'une procédure d'interruption
  - On utilise la priorité de l'interruption pour autoriser l'appel imbriqué
  - Les interruptions peuvent être masquées : sauf NMI
    - Indicateur IF du registre EFLAGS
    - Les instructions STI et CLI permettent de modifier IF
    - C'est pour cela qu'il faut empiler la valeur de EFLAGS avant l'appel d'une interruption
    - Très utile quand on change le code d'une procédure d'interruption

Les interruptions systèmes

- Sous DOS :
  - Interruption logicielle 0x21
- Sous Linux :
  - Interruption logicielle 0x80
  - Plusieurs sous-fonctions : /usr/include/asm/unistd.h
    - Avant l'appel de l'interruption, le registre EAX doit contenir le numéro de la sous-fonction
    - Les registres EBX, ECX, EDX, EDI et ESI contiennent la valeur des 5 paramètres potentiels
    - Après l'appel, EAX contient le code d'erreur

Exemples d'appels systèmes POSIX

- Linux et WinNT sont conformes à la norme POSIX
  - Exit (sortie de programme) : fonction 1
    - `void _exit (int code);`

```
MOV EAX, 1    ; fonction système numéro 1 (exit)
MOV EBX, 0    ; le code de retour est 0 (paramètre code)
INT 0x80      ; appel de l'interruption système
```

  - Write (Écriture dans un fichier) : fonction 4
    - `ssize_t write(int fd, const void *buf, size_t count);`
  - Manuel :
    - man 2 exit pour avoir la description de la fonction système

Les fonctions systèmes

- L'interruption système nous donne accès à toutes les fonctions de base du système d'exploitation
  - `exit, read, write, chdir, utime, setitimer, reboot`
- Beaucoup de fonctions de plus haut niveau existent
  - Par exemple **`printf`** et **`scanf`** dans la bibliothèque C : libc
  - Ces fonctions sont accessibles comme des sous-routines standards (CALL) si la bibliothèque a été liée

Les fonctions de bibliothèque : libc

- Contrairement aux interruptions
  - Les paramètres sont empilés dans l'ordre inverse
  - L'appel à la fonction C se fait avec **CALL**
- La valeur de retour est retournée dans EAX
- Histoire de convention
  - Fortran, Basic, Pascal : les sous-routines appelées sont responsables de dépiler les paramètres
  - C : Le code appelant doit dépiler les paramètres

Exemple : tolower

- `int tolower (int c);`
  - Converti un caractère c en minuscule
- En assembleur :

```
car DD 'A'
extern tolower ; tolower est défini à l'extérieur
PUSH dword [car] ; empile ce caractère (int)
CALL tolower     ; appel la fonction tolower
ADD ESP, 4       ; dépile le paramètre
```

- Pour compiler : **`nasm -f elf tolower.c`**
- Pour lier :

```
ld -dynamic-linker /lib/ld-linux.so.2 tolower.o -lc
```

Exemple : C vers assembleur

- En C:

```
#include <stdio.h>
extern int toMin(int c);

int main() {
    int c = 'A', m;
    m = toMin(c);
    printf("%c %c\n", c, m);
    return 0;
}
```

- En assembleur:

```
section .text
    global toMin
toMin:
    MOV EAX, [ESP+4]
    ADD EAX, 'a'-'A'
    RET
```

- **gcc -c main.c** **nasm -f elf toMin.asm**
- **gcc main.o toMin.o -o toMin**

Les arguments de la ligne de commande

- En C :
  - `int main(int argc, char *argv[])`
- En assembleur :
  - C'est identique
  - Les arguments (argc, argv) sont empilés
  - Les éléments d'un tableau sont empilés en séquence
  - Un pointeur sur **`char`** est une adresse (32 bits) vers un octet (char) de la mémoire :
    - le premier élément de la chaîne de caractères; terminé par \0.
  - argv[0] est le nom de l'exécutable

Exemple : auto

- On compile un programme avec le nom **`auto`**

ald>exam ESP

```
ald> exam esp
Dumping 64 bytes of memory starting at 0xBFFFFA60 in hex
BFFFFA60:  01 00 00 00 6E FB FF BF 00 00 00 73 FB FF BF    ....n.......s...
BFFFFA70:  80 FB FF BF 90 FB FF BF A9 FB FF BF F2 FB FF BF    ................
BFFFFA80:  0F FC FF BF 1E FC FF BF 3D FC FF BF 50 FC FF BF    ........=...P...
BFFFFA90:  5B FC FF BF 63 FC FF BF 8E FC FF BF B5 FC FF BF    [...c...........
ald>
```

exam 0xBFFFFB6E

```
ald> exam 0xBFFFFB6E
Dumping 64 bytes of memory starting at 0xBFFFFB6E in hex
BFFFFB6E:  61 75 74 6F 00 55 53 45 52 3D 66 6D 61 6C 6C 65    auto.USER=fmalle
BFFFFB7E:  74 00 4C 4F 47 4E 41 4D 45 3D 66 6D 61 6C 6C 65    t.LOGNAME=fmalle
BFFFFB8E:  74 00 48 4F 4D 45 3D 2F 75 2F 64 65 70 74 69 6E    t.HOME=/u/deptin
BFFFFB9E:  66 6F 2F 66 6D 61 6C 6C 65 74 00 50 41 54 48 3D    fo/fmallet.PATH=
ald>
```

Les variables d'environnement

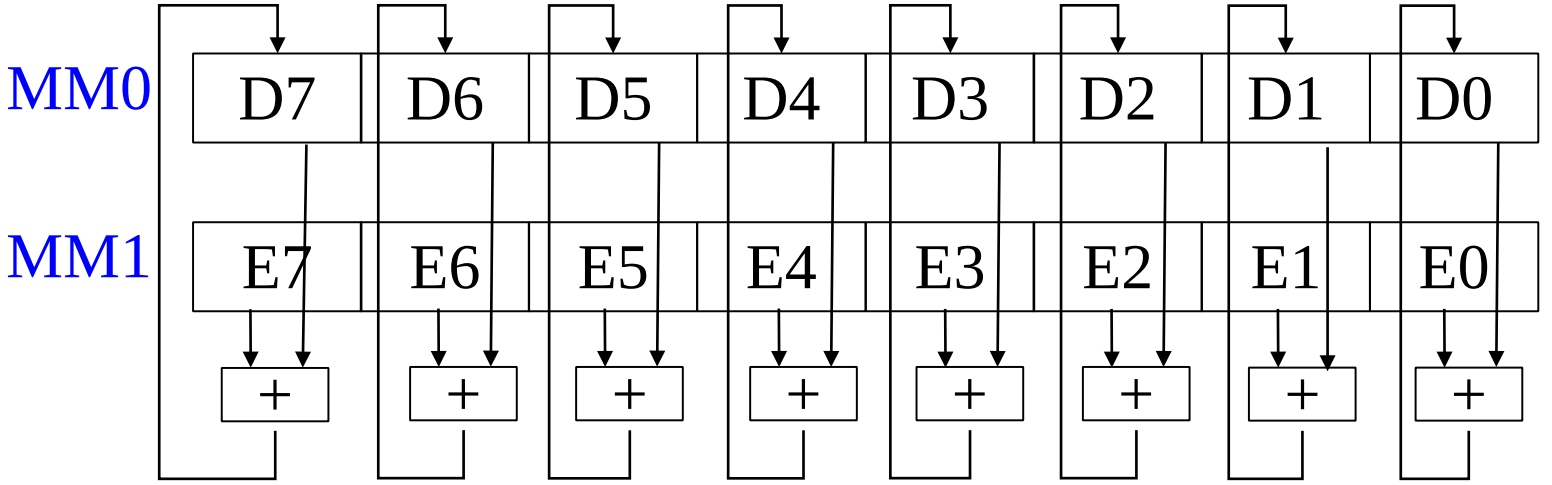
- En fait, la signature complète de **`main`** est :
  - `int main(int argc, char *argv[], char *env[]);`
- Le tableau *env* contient les variables d'environnements :
  - `SHELL = zsh`
  - `PATH = .:/usr/bin:/home/fmallet/bin`
  - `USER = fmallet`
  - `PWD = /home/fmallet/asm`
- Attention, modifier la pile ne modifie pas les variables

Les familles d'instructions (2/4)

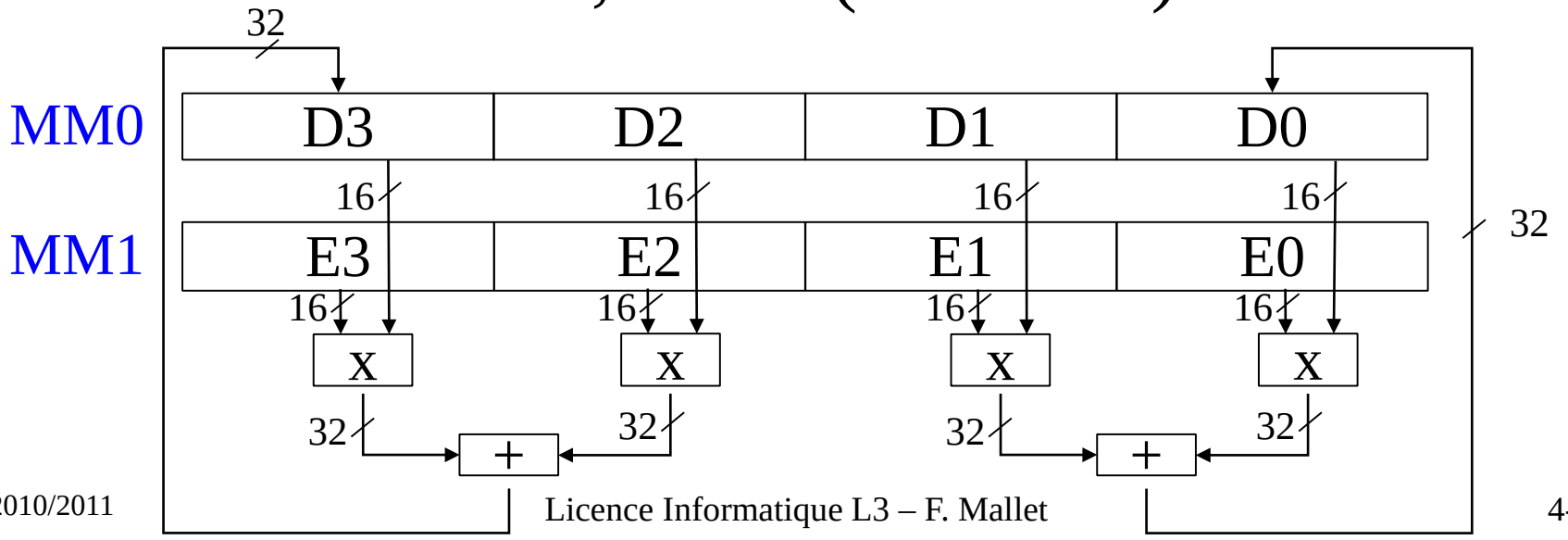
- 4 grandes familles d'instructions
  - Instructions MMX (Extension multimédia)
    - 8 registres de 64 bits : MM0, ..., MM7
    - Opérations arithmétiques sur 8,16,32 ou 64 bits
    - Les applications multimédias (2D et 3D) ont tendance à répéter plusieurs fois la même opération (instruction) sur des tableaux de données : modèle SIMD (Single Instruction Multiple Data)
    - e.g. PADDSB MM0, MM1 effectue 8 additions signées 8-bit sur les octets des registres MM0 et MM1, le résultat est dans MM0
      - Pas de carry ou overflow, mais saturation
    - e.g. PMADDWD effectue 4 multiplications signées 16-bit puis somme les résultats 2 par 2 pour produire 2 résultats sur 32 bits

Exemples d'instructions MMX

- PADDSB MM0, MM1 (S:saturation, B:byte)



- PMADDWD MM0, MM1 (Mul-Add)



Les familles d'instructions (3/4)

- 4 grandes familles d'instructions
  - Instructions sur les nombres réels (Floating-Point Unit)
    - 8 registres de 80 bits pour codage des nombres réels double précision sans perte d'information sur les calculs intermédiaires (ST0 ... ST7)
    - 3 registres spéciaux :
      - FPU Status Register : Registre d'état pour les opérations du FPU
      - FPU Instruction Pointer : PC de la prochaine instruction FPU
      - FPU Operand Pointer : Pointeur de pile sur les registres ST0,...,ST7
    - Exemple de programme : (5.6 x 2.4) + (3.8 x 10.3)
      - FLD 5.6 // empile 5.6 à l'adresse donnée par FPU_OP
      - FMUL 2.4 // multiplie le sommet de pile par 2.4 et empile le résultat
      - FLD 3.8 // empile 3.8
      - FMUL 10.3 // multiple le somme de pile par 10.3 et empile le résultat
      - FADD ST(1) // ajoute les 2 valeurs du sommet de pile et empile le résultat

Les familles d'instructions (4/4)

- 4 grandes familles d'instructions
  - Instructions sur les nombres réels (Floating-Point Unit)
    - Instructions spéciales pour opérations trigonométriques
      - FLDPI : empile la valeur de PI
      - FSIN, FCOS, ... : opérations trigonométriques standards
    - Opérations logarithme et exponentielle
      - FYL2X : $y * \log_2 x$
      - F2MX1 : $2^x - 1$
    - Diverses : FSQRT (racine carrée)
    - Le FPU doit être initialisé : FINIT/FNINIT
  - Instructions système
    - Pour les systèmes d'exploitation: configure le cache, manipule les tables de descriptions des segments, ...