Architecture des ordinateurs

par Christophe TOMBELLE

Télécom LILLE

Tous droits de traduction, de reproduction
et d'adaptation réservés pour tous pays.
© ENIC, Telecom Lille1, Telecom Lille, 2000

La loi du 11 mars 1957 n'autorisant, aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les "copies ou reproductions strictement réservées à l'usage du copiste et non destinées à une utilisation collective" et d'autre part, que les analyses et que les courtes citations dans un but d'exemple et d'illustration, "toute représentation intégrale ou partielle, faite sans le consentement de l'auteur ou de ses ayants droits, ou ayants cause est illicite" (alinéa 1er de l'article 40).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivants du code pénal.

Télécom Lille - Cité Scientifique
Rue G. Marconi – BP20145
59653 Villeneuve d'Ascq Cedex
Tél. : 03.20.33.55.77
http ://www.telecom-lille1.eu

Table des matières

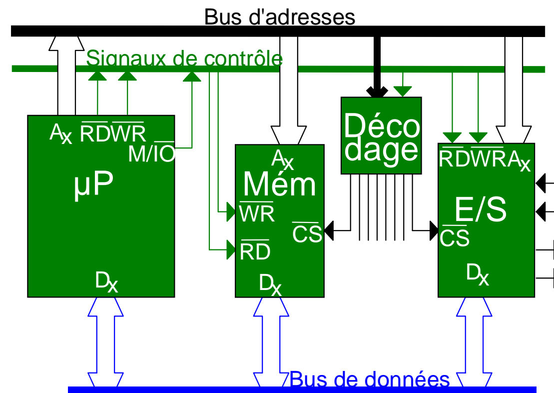
1. Le matériel ... 7
1.1. Système à microprocesseur ... 7
1.2. Les bus ... 8
1.3. Constitution interne du microprocesseur ... 13
1.4. Constitution interne de la mémoire ... 14
1.5. Entrées - sorties ... 15
1.6. Fonction décodage ... 18
1.7. Actions du microprocesseur ... 18
1.8. Le port série ... 18
1.9. Constitution ... 19
1.10. Brochage ... 20
1.11. Microcontrôleur ... 20
2. Le matériel : travail dirigé ... 23
2.1. Déroulement d'une séquence d'instructions ... 23
2.2. Chronogramme de lecture ... 24
2.3. Performances des processeurs ... 24
3. Le logiciel : modèle de programmation ... 27
3.1. Organisation de la mémoire ... 27
3.2. Le modèle de programmation ... 28
3.3. Segmentation du 8086 ... 30
3.4. Représentation à la Turbo-Debugger ... 32
3.5. Association par défaut des registres de segment ... 34
3.6. Méthodologie ... 35
4. Représentation des données ... 37
4.1. Les modes d'adressage ... 37
4.2. L'adressage immédiat ... 37
4.3. L'adressage direct ... 37
4.4. L'adressage indirect ... 38
4.5. L'accès aux variables ... 42
4.6. Plate-forme Turbo-C 8086 (taille des types de base) ... 45
4.7. Schémas de mémoire ... 45
4.8. Sujet du TP2 ... 46

5. Structuration des données 47
5.1. Les pointeurs 47
5.2. Les enregistrements 49
5.3. Tableau d'octets 50
5.4. Tableaux d'entiers 50
5.5. Tableau d'enregistrements 50
5.6. Tableaux à 2 dimensions 51
5.7. Tableau de pointeurs 51
5.8. Fin du TP2 52
6. Structuration du code 53
6.1. Les instructions de saut 53
6.2. Les structures de contrôle 55
6.3. Sujet du TP3 57
7. Structuration du code : les sous-programmes 61
7.1. Fonctionnement d'une pile 61
7.2. Représentation de la pile 61
7.3. L'appel de sous-programme 61
7.4. Retour de sous-programme 62
7.5. Imbrication des sous-programmes 63
7.6. Fonctions 63
8. Passage des arguments 65
8.1. Le passage des arguments en langage C 65
8.2. Passage des arguments par la pile 65
8.3. Point de vue de l'appelant 66
8.4. Point de vue de l'appelé 66
8.5. Exploitation du résultat de la fonction 67
8.6. Récursivité 68
8.7. Paramètres en nombre variable 68
9. Gestion des entrées-sorties 69
9.1. Introduction 69
9.2. Le polling 69
9.3. Le mécanisme d'interruption 69
9.4. Le contrôleur d'interruptions 70
9.5. Vectorisation des interruptions 70
9.6. Les interruptions logicielles 71
9.7. Mécanisme d'accès direct à la mémoire 71

10. Les logiciels d'exploitation *73*
10.1. Rôle d'un système d'exploitation **73**
10.2. La gestion de la mémoire **74**
10.3. Le parallélisme d'exécution **81**
10.4. La notion de protection **83**
11. Système d'exploitation : approfondissement *87*
11.1. Implantation du pseudo-parallélisme **87**
12. Evolution des performances *91*
12.1. Introduction **91**
12.2. Structure d'un ordinateur **91**
12.3. Progrès technologiques **91**
12.4. Amélioration des performances **91**
12.5. L'architecture RISC **93**
12.6. Architectures, super-pipeline et super-scalaire **94**
12.7. Processeurs CISC **94**
12.8. L'architecture VLIW **95**
12.9. Applications **95**
13. Bibliographie *97*

1. Le matériel

1.1. Système à microprocesseur



Le microprocesseur est capable d'exécuter des instructions qui agissent sur des opérandes. Les opérandes sont constitués d'une part des **registres** situés au cœur du processeur et d'autre part des variables situés dans la **mémoire**. Les **instructions** que le microprocesseur doit exécuter sont des *informations codées* qui résident également *en mémoire*. Le codage employé pour les instructions est propre au modèle de microprocesseur employé. Ce codage obéit à une certaine logique, logique que le microprocesseur emploie pour le décodage. L'ensemble des instructions qu'un microprocesseur est capable d'exécuter forme son **jeu d'instructions**.

Le processeur désigne les cases mémoires auxquelles il veut accéder en présentant leurs adresses sur le **bus d'adresses**. L'ensemble des signaux d'adresses d'un système s'appelle un **bus d'adresses**. Le nombre de signaux d'adresses détermine la **capacité d'adressage**. Avec 1 signal d'adresse supplémentaire, on peut désigner 2 fois plus de cases mémoire. Autrement dit, on double la capacité d'adressage. Ainsi, avec 1 seul signal d'adresses un processeur serait capable d'adresser 2 octets, avec 2 signaux d'adresses, il pourrait en adresser 4. Avec 16 signaux, le 8085 peut en adresser 2^{16} soit 64 ko, avec 20 signaux d'adresses, le 8086 peut en adresser 2^{20} soit 1 Mo, avec ses 32 signaux le 386 peut en adresser 2^{32}, soit 4 Go.

La mémoire échange des informations avec le processeur par le **bus de données** qui est bidirectionnel. Le processeur précise également l'instant et le sens du transfert grâce aux **signaux de contrôle** que sont les signaux de lecture et d'écriture.

Il existe de la mémoire accessible en lecture et en écriture (**RAM=Random Access Memory ou mémoire vive**) et de la mémoire accessible uniquement en lecture (**ROM=Read Only Memory ou mémoire morte**).

Le processeur comporte des registres non seulement pour les opérandes des opérations qu'il exécute mais aussi pour des usages plus techniques. Par exemple, l'adresse de la prochaine instruction à exécuter est contenue dans un registre du microprocesseur : le **pointeur d'instructions** (IP = Instruction Pointer), plus généralement appelé **compteur de programme** (PC = Program Counter). Le microprocesseur commence chacun de ses cycles de fonctionnement par un cycle de lecture en mémoire. Il va ainsi chercher le code de l'instruction à exécuter. Ce code transite sur le bus de données jusqu'au microprocesseur.

Celui-ci décode l'instruction et décide de la suite à donner à l'instruction. Eventuellement, des compléments d'instructions sont nécessaires pour que l'instruction soit complète. En effet, une **instruction** nécessite généralement un ou plusieurs opérandes; elle peut donc faire l'objet de **plusieurs cycles** de lecture en mémoire pour être complète.

La lecture des codes machine relève d'un comportement spontané du microprocesseur (que dois-je faire ?) l'exécution des codes machines définit le comportement programmé du microprocesseur : faire ceci, faire cela. Le fonctionnement d'un microprocesseur alterne entre comportement spontané et comportement programmé. Spontanément, le microprocesseur acquiert un code machine sous forme d'un ou plusieurs mots mémoire. Lorsque l'instruction est complète, il l'exécute. Cette exécution peut comporter une lecture ou une écriture en mémoire. Ensuite, le processeur

recommence avec l'instruction suivante. Dès qu'un processeur est sous tension, il acquiert spontanément des codes machine et les exécute. Les premiers codes machine proviennent d'une ROM car à la mise sous tension, le contenu d'une RAM est indéterminé tandis que celui d'une ROM, déterminé par le constructeur de l'ordinateur est maintenu en l'absence d'énergie.

1.2. Les bus

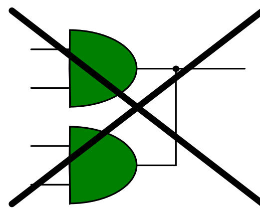
Les architectures à microprocesseur font appel à des voies de circulation des informations appelées bus. Un bus relie plusieurs dispositifs ensemble. Un câblage en bus est plus économique et plus évolutif qu'un câblage en réseau complètement maillé. Sur un bus, les informations ne proviennent pas toujours du même émetteur et n'ont pas toujours le même destinataire. Ainsi, une broche d'un dispositif peut :

- soit émettre un signal,
- soit ne pas émettre de signal mais au contraire écouter un signal émis par un autre dispositif,
- soit ne pas émettre de signal et ignorer un éventuel signal émis par un autre dispositif

La logique à 3 états autorise ce type de connexion.

1.2.1. Logique à 3 états

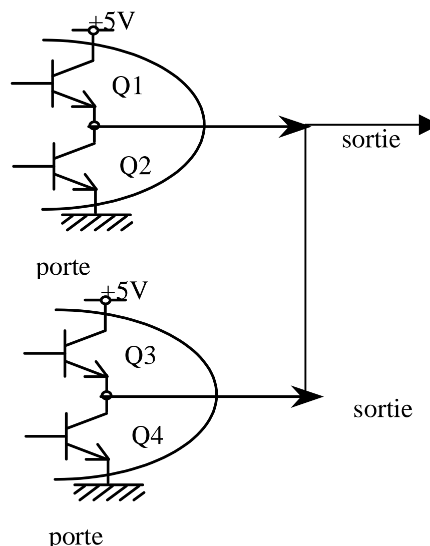
1.2.1.1. Le problème



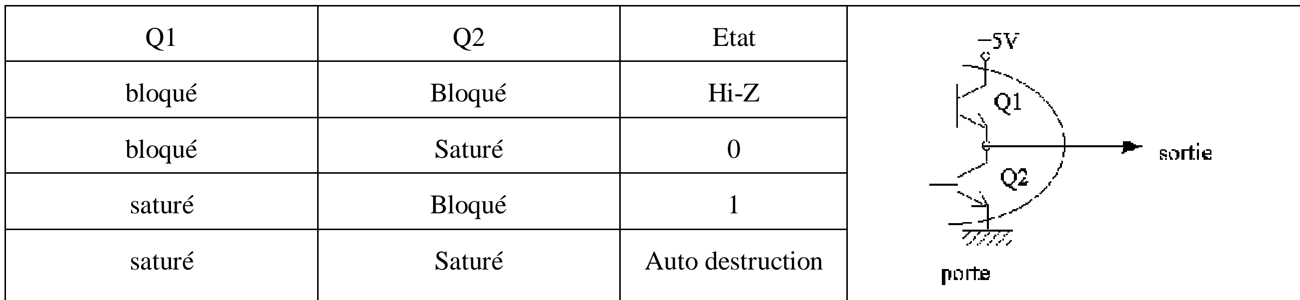
En logique classique à 2 états, les règles de conception interdisent de connecter deux sorties ensemble : un signal ne peut être émis que par une source unique, il ne peut en aucun cas venir "soit d'une porte, soit d'une autre".

1.2.1.2. Structure interne

Dans une porte classique à 2 états, l'un au moins des deux transistors de l'étage de sortie est passant. Si on relie deux sorties ensemble et que ces sorties cherchent à imposer des états logiques différents sur leur sortie respective, un transistor de l'un et un transistor de l'autre étant passants (Q1 et Q4 passants ou Q3 et Q2 passants), un court-circuit est établi entre l'alimentation et la masse, générant un courant destructeur pour les deux portes. En pratique, une résistance en série avec le collecteur de Q1 ou Q3 limite le courant de sortie de la porte pour éviter cette destruction. Il reste que la tension résultante à la sortie de cette porte ne correspond ni à un 1 ni à un 0.

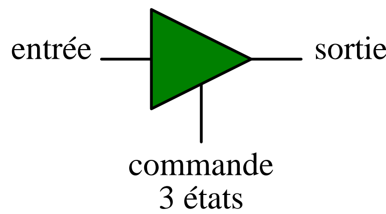


En plus des 2 états logiques habituels, la logique à 3 états fait intervenir un troisième état, technologique celui-là, qui correspond à l'état haute impédance. Dans l'état haute impédance, les deux transistors de sortie sont bloqués. Les états logiques 0 et 1 correspondant à un transistor passant, ce sont des états "basse impédance".



Q1	Q2	Etat
bloqué	Bloqué	Hi-Z
bloqué	Saturé	0
saturé	Bloqué	1
saturé	Saturé	Auto destruction

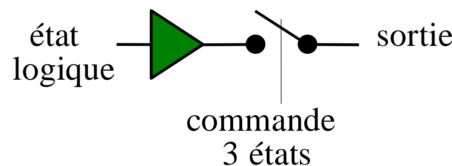
1.2.1.3. Porte à 3 états



Une porte à 3 états est capable de générer sur sa sortie les 3 états de la logique à 3 états : 0, 1 ou Z (haute impédance)

Ceci est une porte non inverseuse, c'est-à-dire un simple répéteur, mais comportant en plus une entrée de commande du 3ème état. Le fonctionnement est le suivant : soit la porte est active et la sortie recopie l'état de l'entrée, soit la porte est inactive et la sortie est en haute impédance.

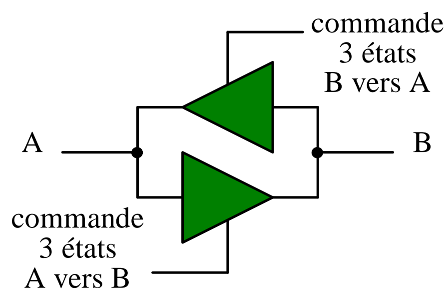
1.2.1.4. Schéma équivalent



Quand la sortie est en haute impédance, tout se passe comme si la sortie de la porte était coupée (interrupteur ouvert)

Une entrée "en l'air" est une entrée non pilotée par une sortie. Une entrée en l'air est dans l'état haute impédance. En effet, il n'y a pas de différence entre une entrée en l'air et une entrée reliée à une sortie dans l'état haute impédance (interrupteur ouvert).

1.2.1.5. Porte bidirectionnelle



Si on met deux portes 3 états en "parallèle-inverse", on réalise une barrière bidirectionnelle 3 états. Le processeur et les mémoires sont des composants capables de gérer la logique 3 états, c'est-à-dire qu'ils comportent des broches bidirectionnelles de ce type autorisant les fonctionnements suivants :

- fonctionnement A vers B (entrée côté A, sortie côté B, porte du bas en haute impédance)
- fonctionnement B vers A (entrée côté B, sortie côté A, porte du haut en haute impédance)
- fonctionnement A isolé de B (les deux portes en haute impédance)

Les côtés A et B sont bidirectionnels 3 états. Du côté A, on peut :

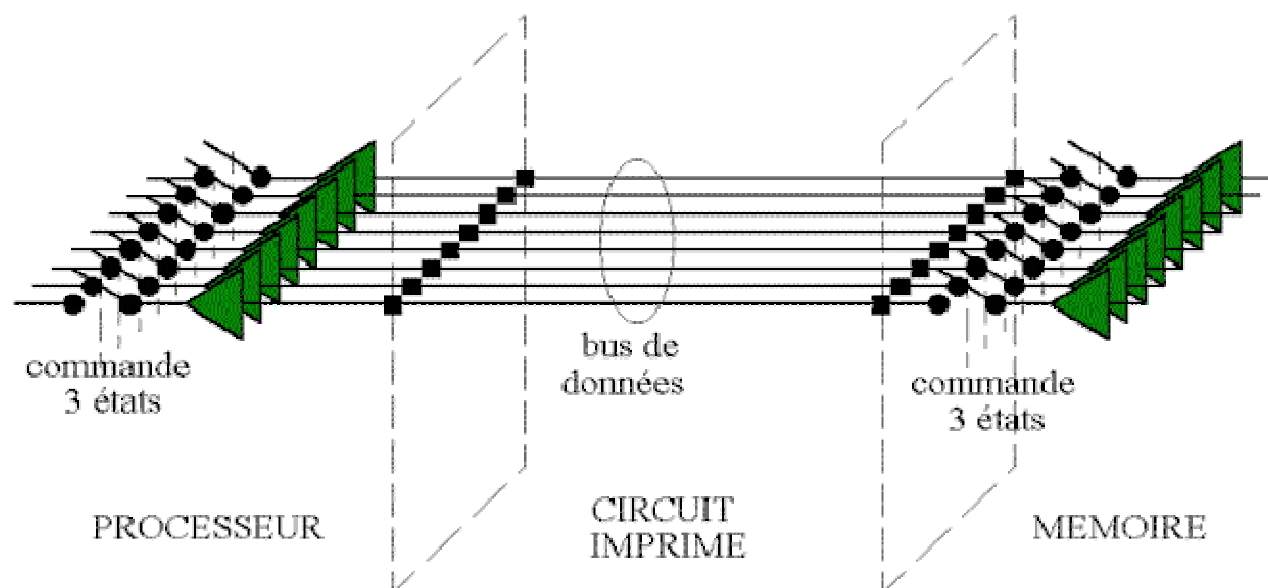
- soit émettre un signal,
- soit ne pas émettre mais recevoir un signal,
- soit ne pas émettre ni recevoir un signal.

A et B étant identiques, ce qui a été dit pour le côté A est vrai pour le côté B.

1.2.2. Bus de données et d'adresses

Un bus de données est un ensemble de signaux reliant des dispositifs à broches bidirectionnelles à 3 états. Le bus d'adresses relie les sorties à 3 états du bus d'adresses du processeur avec les entrées du bus d'adresses des autres dispositifs, mémoire, contrôleurs d'entrée-sortie, logique de décodage.

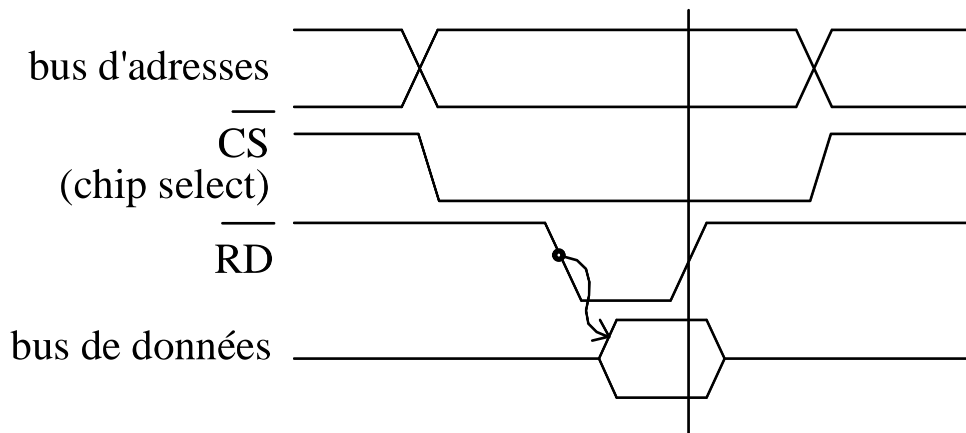
1.2.2.1. Cas de la lecture



En lecture, les informations circulant sur le bus de données vont de la mémoire vers le processeur. Le dispositif "**parleur**" (interrupteurs fermés sur les sorties) est la mémoire, le dispositif "**écouteur**" (interrupteurs fermés sur les entrées) est le processeur.

En l'absence d'échanges d'information entre la mémoire et le processeur, le bus de données est en haute impédance. Aucun dispositif ne présente d'état sur le bus de données. En revanche, lorsqu'un échange a lieu, le bus de données est en basse impédance. Il présente l'impédance de la sortie du composant qui "parle" sur le bus de données.

1.2.2.2. Cycle de lecture



Mémoire est validée => données apparaissent; sur le front montant de RD# le processeur prend en compte les données

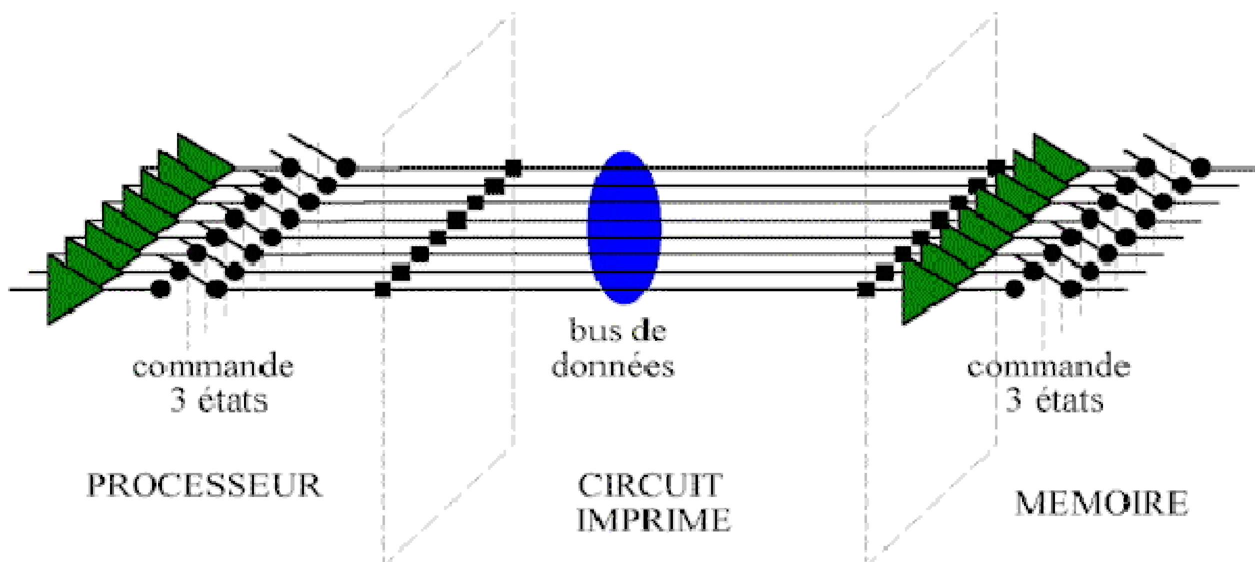
Un cycle de lecture est un transfert de données de la mémoire vers le processeur. Une lecture se produit spontanément lorsque le processeur a besoin de lire un code machine. Une lecture peut également résulter de l'exécution d'une instruction qui a besoin de la valeur d'une variable. Une lecture résulte d'un comportement spontané ou programmé du processeur.

L'adresse véhiculée sur le bus d'adresses provient du processeur. La donnée véhiculée sur le bus de données provient du bus de données. Le signal RD# est contrôlé par le processeur. Le signal de chip select (sélection de boîtier) est élaboré à partir des signaux d'adresses. Le signal RD# correspond sur la mémoire à une broche de même nom ou parfois appelée OE# (Output Enable). On peut retenir du signal RD# qu'il est une impulsion de lecture dont l'action reste conditionnée bien sûr par la sélection du boîtier.

Au départ, le bus de données est en haute impédance.

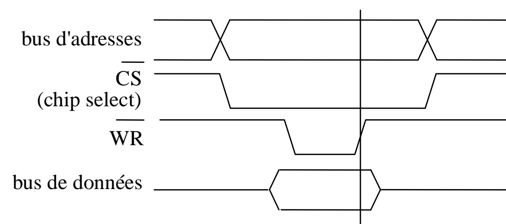
- le microprocesseur présente les adresses
- donc le CS# (chip select) de la mémoire concernée s'active
- le microprocesseur active le signal RD#
- la mémoire présente sur le bus de données le contenu de la case mémoire adressée
- le microprocesseur prend en compte les données présentes sur le bus de données
- le microprocesseur désactive le signal RD#
- la mémoire relâche le bus de données en haute impédance.

1.2.2.3. Cas de l'écriture



En écriture, les informations circulant sur le bus de données vont du processeur vers la mémoire. Le dispositif "**parleur**" (interrupteurs fermés sur les sorties) est le processeur, le dispositif "**écouteur**" (interrupteurs fermés sur les entrées) est la mémoire.

1.2.2.4. Cycle d'écriture

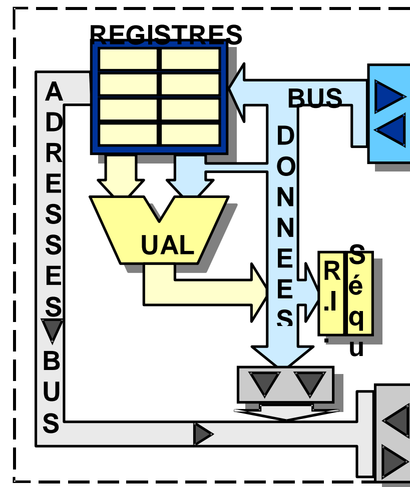


Un cycle d'écriture est un transfert de données du processeur vers la mémoire. Une lecture résulte de l'exécution d'une instruction de mise à jour d'une variable, donc d'un comportement programmé du processeur.

L'adresse véhiculée sur le bus d'adresses et la donnée véhiculée sur le bus de données proviennent du processeur. Le signal WR# est contrôlé par le processeur.

- le microprocesseur présente les adresses sur le bus d'adresses
- donc le chip select de la mémoire concernée s'active
- le microprocesseur présente sur le bus de données la donnée à écrire dans la case adressée
- le microprocesseur active le signal WR#
- le microprocesseur désactive le signal WR#
- le microprocesseur relâche le bus de données en haute impédance

1.3. Constitution interne du microprocesseur



Un microprocesseur fonctionne avec une horloge. Parfois l'oscillateur est interne, parfois externe. Le quartz est connecté aux bornes de l'oscillateur.

L'ALU sait effectuer des opérations arithmétiques (4 opérations) ou logiques (et, ou, non, ou exclusif, décalages et rotations) sur des entiers. Les opérations s'effectuent sur 1 ou 2 opérandes.

Les opérandes proviennent des registres du microprocesseur ou de la mémoire (externe) via le bus de données.

Il y a éventuellement un additionneur dédié à des calculs d'adresses.

Le registre instruction R.I. stocke le code machine de l'instruction à exécuter. Le séquenceur organise l'exécution des instructions. Par exemple, une instruction d'addition pourra spécifier qu'un des opérandes vient de telle case mémoire que l'autre vient de tel registre du microprocesseur. L'adresse de la case mémoire fait partie du code machine. Le séquenceur s'arrangera pour ouvrir le buffer bidirectionnel du bus de données en lecture, pendant qu'il génèrera un 0 sur le signal RD#. L'opération d'addition sera sélectionnée sur l'ALU et tel registre sera sélectionné parmi le jeu de registres du microprocesseur.

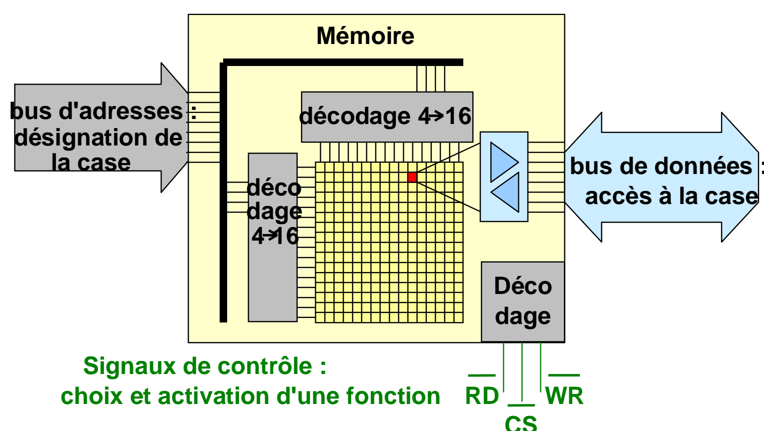
Les séquenceurs peuvent être câblés ou microprogrammés. De l'optimisation d'un microprogramme, dépend l'efficacité du processeur. Un microprogramme est constitué d'une succession de mots dont chaque bit commande un dispositif interne du processeur.

Les microprocesseurs les plus simples ne savent pas effectuer de multiplication ou de division en une seule instruction. Ils ont alors recours à un programme de multiplication ou de division. Les calculs effectués sur des nombres à virgule peuvent également faire l'objet de sous-programmes. Une autre solution consiste à recourir à un coprocesseur. C'est un processeur spécialisé dans les calculs des nombres à virgule.

Les processeurs puissants intègrent le "coprocesseur" sur leur puce. On ne parle plus de coprocesseur mais d'unité de calculs en virgule flottante. L'architecture matérielle des processeurs et des systèmes à hautes performances sera étudiée plus tard.

La logique d'interruption, non représentée sur ce schéma, sera étudiée ultérieurement avec les autres techniques de gestion des entrées-sorties.

1.4. Constitution interne de la mémoire



En dehors de ses registres, un microprocesseur ne contient pas de mémoire[1]. Un microprocesseur, notamment ne contient pas le programme qu'il exécute. La mémoire est donc un composant indispensable au microprocesseur, d'une part pour contenir le programme (RAM ou ROM), d'autre part pour contenir les variables (RAM)

Les cases mémoires sont repérées par des adresses, combinaisons binaires codées sur des signaux d'adresses. Un nombre n de signaux d'adresses disponibles sur une ROM détermine une capacité de stockage de 2^n cases mémoire. Plusieurs composants mémoire peuvent être raccordés aux mêmes signaux d'adresses, aux mêmes signaux de données et aux mêmes signaux de contrôle RD# et WR#. La broche CS# permet d'activer leur fonctionnement dans une plage d'adresses donnée qui leur est spécifique par conception de la logique de décodage.

L'adresse présente sur le bus d'adresses se sépare dans le composant en adresse ligne et adresse colonne. Le décodeur ligne décode l'adresse ligne et sélectionne une ligne de la matrice. Le décodeur colonne décode l'adresse colonne et sélectionne une colonne de l'adresse colonne. La case mémoire située à l'intersection de la ligne et de la colonne sélectionnée est ainsi adressée. Lorsque les signaux CS# d'une part et RD# ou WR# d'autre part sont actifs, un buffer 3 états bidirectionnel met en relation la case mémoire sélectionnée avec le bus de données, dans le sens approprié.

1.4.1. Les mémoires vives

Il existe 2 types de mémoire vive les mémoires statiques et les mémoires dynamiques.

La cellule de base (1 bit de mémoire) d'une RAM statique est un bistable. Une telle RAM peut être sauvegardée au moyen d'une batterie ou d'une pile. Certains composants sont même spécialement conçus à cet effet et consomment en stand-by sur leur broche d'alimentation un courant inférieur à 1 µA.

La cellule de base (1 bit) d'une RAM dynamique étant une capacité réalisée sur le silicium, la surface occupée sur le silicium est donc sensiblement moins importante que dans le cas d'une mémoire statique.

Les mémoires dynamiques sont donc intéressantes pour réaliser de gros plans mémoire.

En revanche, les courants de fuite des capacités les déchargent en quelques millisecondes. L'information qu'elles contiennent doit donc être lue et réécrite périodiquement avant le délai fatidique. Ces lectures-réécritures sont appelées "rafraîchissements" et sont effectuées pour une ligne complète de la matrice. En quelques millisecondes, c'est donc l'ensemble des lignes qui doit être rafraîchi, afin de revenir à temps à la 1ère ligne rafraîchie.

1.4.2. Les mémoires mortes

Une ROM est une mémoire accessible seulement en lecture. Elle ne comporte pas de broche WR#. Un signal RD# (parfois OE#) synchronise la sortie de la donnée mémorisée sur les signaux de données.

---

[1] Nous ne parlons pas des processeurs hautes performances qui intègrent sur la même puce un processeur et de la mémoire cache.

Plusieurs composants mémoire peuvent être raccordés aux mêmes signaux d'adresses, aux mêmes signaux de données et au même signal RD#. La broche CS# permet d'activer leur fonctionnement dans une plage d'adresses donnée qui leur est spécifique (voir logique de décodage).

1.5. Entrées - sorties

1.5.1. Activation des signaux selon l'instruction exécutée

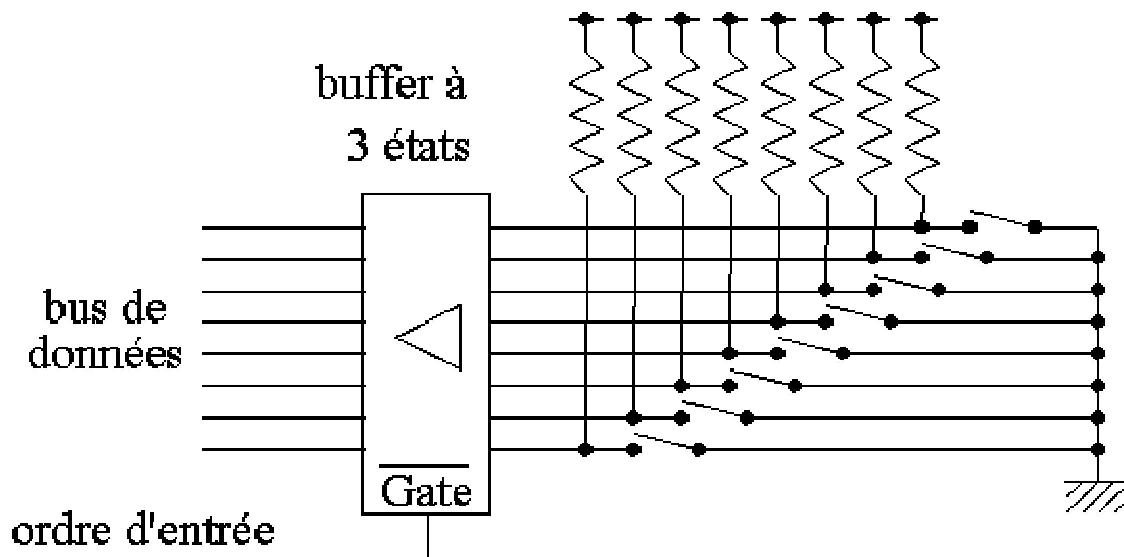
	$M/\overline{IO} = 0$	$M/\overline{IO} = 1$
$\overline{RD}$	**Entrée** : IN AL,adrIO	Tout accès en **lecture** à la mémoire.
$\overline{WR}$	**Sortie** : OUT adrIO,AL	Tout accès en **écriture** à la mémoire.

Des instructions d'entrée et de sortie existent parfois dans le jeu d'instructions d'un microprocesseur. IN et OUT sont les mnémoniques utilisés par Intel pour ses microprocesseurs. On parle alors d'espace adressable séparé pour les entrées-sorties ou d'adressage IO. Dans ce cas, le microprocesseur fournit et pilote un signal (M/IO#) qui indique à son environnement si l'adresse qu'il génère concerne la mémoire ou au contraire est une adresse IO, c'est-à-dire une adresse d'un registre d'entrée ou de sortie.

Parfois, on doit se servir des mêmes instructions que pour la mémoire. On parle alors d'espace adressable unique. Une partie de cet espace est alors réservé pour les dispositifs d'entrée-sortie, au détriment de l'espace mémoire disponible.

1.5.2. Un dispositif d'entrée de base

Ce dispositif élémentaire est un port parallèle très simple figé en entrée. Il donne au microprocesseur un accès à la combinaison binaire codée sur la série d'interrupteurs. Un interrupteur fermé code un 0 (masse), un interrupteur code un 1 logique (Vcc).

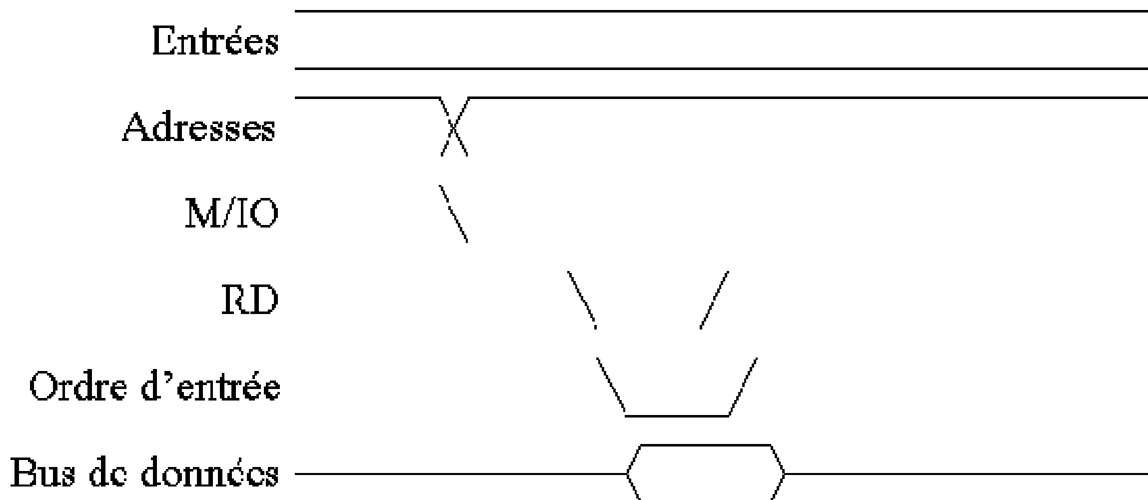


Lorsque le microprocesseur accède à ce registre d'entrée, il le fait à une adresse bien précise qu'on appellera ici « l'adresse des interrupteurs ». Le signal appelé ici « ordre d'entrée » provient d'une logique de décodage appropriée et est supposé s'activer (c'est-à-dire passer à 0) lorsque les conditions suivantes sont réunies :

- « l'adresse des interrupteurs » est présente sur le bus d'adresses
- le signal RD# est actif (dans l'état 0)
- le signal M/IO# indique un accès IO (dans l'état 0)

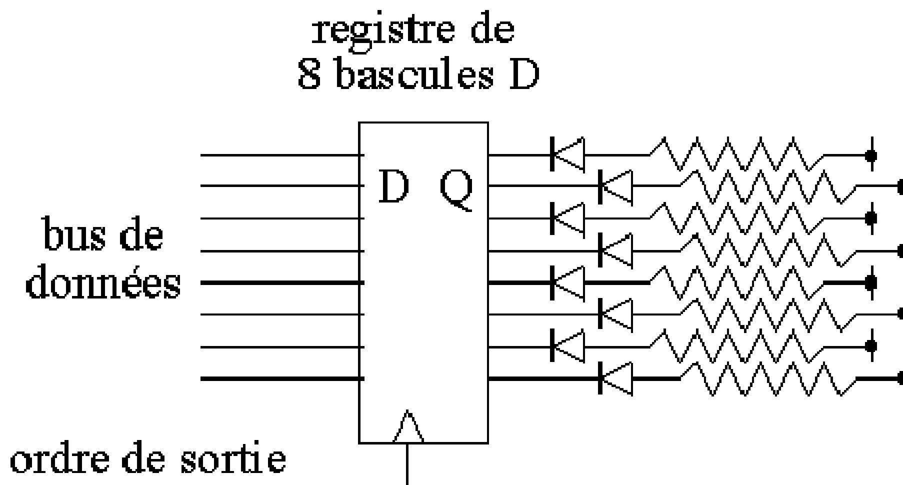
1.5.3. Chronogramme d'entrée

Quand le microprocesseur exécutera une instruction IN à l'adresse des interrupteurs, ces conditions seront réunies et l'ordre d'entrée passera à 0, le buffer 3 états propagera l'état des interrupteurs vers le bus de données. Le microprocesseur étant entrain d'exécuter cette instruction IN, un de ses registres est à l'écoute du bus de données et captera l'information. La combinaison binaire codée par les interrupteurs se trouvera donc mémorisée dans un des registres du microprocesseur, ouvrant la voie à un traitement approprié.



C'est ainsi que le microprocesseur peut **connaître son environnement**, qu'il s'agisse des boutons poussoirs d'une montre gérée par un microprocesseur embarqué, qu'il s'agisse de capteurs industriels reliés à un ordinateur de process ou qu'il s'agisse d'un périphérique d'entrée d'un ordinateur, comme un clavier ou une souris, on a recours à un mécanisme similaire.

1.5.4. Un dispositif de sortie de base



Ce dispositif élémentaire est un port parallèle très simple figé en sortie. Il permet au microprocesseur d'allumer des voyants à led selon une combinaison binaire de son choix. Un 0 (masse) sur une sotie Q du registre D-edge provoquera

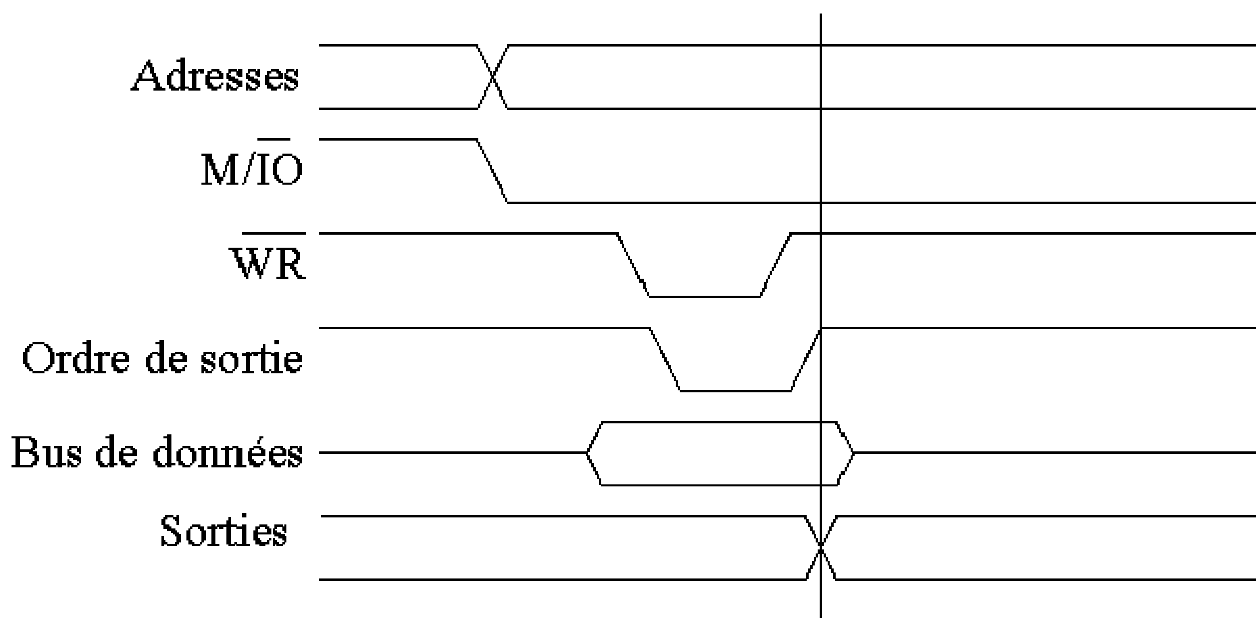
une ddp par rapport à Vcc, allumant ainsi le voyant. Un 1 sur une sortie Q annule la ddp relative à Vcc éteignant ainsi le voyant.

Lorsque le microprocesseur accède à ce registre de sortie, il le fait à une adresse bien précise qu'on appellera ici « l'adresse des voyants ». Supposons qu'une logique de décodage appropriée élabore un signal appelé ici « ordre de sortie » et que cette logique de décodage est conçue de telle façon que l'ordre de sortie s'active (c'est-à-dire passe à 0) lorsque les conditions suivantes sont réunies :

- « l'adresse des voyants » est présente sur le bus d'adresses,
- le signal WR# est actif (dans l'état 0)
- le signal M/IO# indique un accès IO (dans l'état 0)

1.5.5. Chronogramme de sortie

Quand le microprocesseur exécutera une instruction OUT à l'adresse des voyants, ces conditions seront réunies et **l'ordre de sortie passera à 0 et sur son front remontant**, l'information présente sur le bus de données sera mémorisée dans le registre D-edge. Le microprocesseur étant entrain d'exécuter cette instruction OUT, l'information présente sur le bus de données est la combinaison binaire d'allumage des voyants que le microprocesseur souhaite écrire.



C'est ainsi que le microprocesseur peut **agir sur son environnement**, qu'il s'agisse des commandes du moteur d'un magnétoscope géré par un microprocesseur embarqué, qu'il s'agisse d'actionneurs industriels reliés à un ordinateur de process ou qu'il s'agisse d'un périphérique de sortie d'un ordinateur comme une imprimante, on a recours à un mécanisme similaire.

Généralement, les dispositifs d'entrée-sortie se présentent plutôt sous forme de circuits intégrés appelés contrôleurs d'entrées-sorties. Ces dispositifs contiennent des registres d'entrée-sortie. **Il ne faut pas les confondre avec les registres du processeur.**

L'accès aux registres d'entrée-sortie est comparable à l'accès à la mémoire. Il y a cependant une différence importante entre une case mémoire et un registre d'entrée-sortie : quand on a écrit dans une RAM, on relit (normalement) la valeur qu'on vient d'écrire. Quand on écrit dans un registre de sortie à une adresse, **on ne relit généralement pas la même valeur** à cette même adresse. Il s'agit d'ailleurs conceptuellement de registres différents. Par exemple, le registre de données d'un port série est accessible à une même adresse en entrée et en sortie. Mais, en entrée, il contient un caractère que le port série a reçu, tandis qu'en sortie, il contient la donnée que le port série doit émettre.

1.6. Fonction décodage

La fonction décodage à pour rôle de sélectionner une barrette ou un boîtier mémoire, un contrôleur d'entrée-sortie parmi plusieurs. En effet, il y a plusieurs boîtiers mémoire (ou plusieurs banques) dans un système, qui sont tous câblé(e)s de la même façon à l'exception d'une broche spécifique : le chip select (ou chip enable). Pour les mémoires et les contrôleurs d'entrée-sortie, cette broche est une entrée qui active leur fonctionnement. Le but de la fonction de décodage est d'activer tel ou tel "chip select" selon la valeur de l'adresse présente sur le bus d'adresses.

La conception de la logique de décodage permet d'établir la **cartographie** d'un système, c'est-à-dire la **correspondance entre une plage d'adresses et une ressource** ou entre une adresse et un registre d'entrée ou de sortie.

L'espace adressable est l'ensemble des adresses qu'un microprocesseur peut générer sur son bus d'adresses. Dans l'espace adressable d'un système, il se peut qu'une plage d'adresses corresponde à de la RAM, de la RAM sauvegardée, de la Flash, à des registres d'entrée-sortie ou à **rien du tout**. Ce dernier cas se produit lorsque aucun dispositif ne voit son chip select activé pour une adresse ou une plage d'adresses donnée.

Il se peut aussi qu'une logique de décodage active une même ressource pour des plages d'adresses différentes.

La logique de décodage peut aussi parfois tenir compte de signaux de contrôle tels que RD#, WR# et M/IO# pour élaborer les chip select des contrôleurs d'entrée-sortie.

Une logique de décodage est généralement purement combinatoire. On peut l'implanter à l'aide de portes logiques, de circuits décodeurs, ou à l'aide de composants programmables.

Les contrôleurs d'entrée-sortie contiennent généralement plusieurs registres. Une logique de décodage intégrée exploitant quelques signaux d'adresses sélectionne les registres à l'intérieur de la plage d'adresses correspondant à leur chip select.

1.7. Actions du microprocesseur

M/IO#	RD#	WR#		Instruction exécutée	Sens de fonct^{t} du bus de données	Nom de l'opération
X	0	0		impossible		
0	0	1		IN	IO -> µP	entrée
1	0	1		fetch / MOV	mem -> µP	lecture
0	1	0		OUT	µP -> IO	sortie
1	1	0		MOV	µP -> Mem	écriture
X	1	1		travail interne	Hi-Z	repos (idle)

Ce tableau résume les comportements du microprocesseur et les différents usages du bus de données selon les différentes combinaisons binaires indiquées par le processeur sur ses signaux de contrôle.

1.8. Le port série

La notion de contrôleur d'entrée-sortie est illustrée ici par un port série. Les ports série sont parfois connus sous le nom d'UART ou USART (Universal Synchronous/Asynchronous Receiver Transmitter). Leur rôle est le suivant :

- Emission
    - conversion parallèle (bus données) - série (Tx)
- Réception
    - conversion série (Rx) - parallèle (bus données)
- Divers modes
    - synchrone : synchronisation à chaque trame
    - asynchrone : synchronisation à chaque caractère

Les contrôleurs d'entrée-sortie sont de natures très variées. Ils sont plus complexes que les dispositifs d'entrée et de sortie élémentaires étudiés précédemment.

Par exemple, pour une communication série asynchrone (modems) écrire dans le registre de donnée déclenche l'émission du caractère alors que la lecture du registre de donnée fournit le caractère reçu. Écrire dans le registre de configuration définit la vitesse de la transmission, le type de parité (pour la détection d'erreur), le nombre de bits d'arrêt, etc… Lire le registre de configuration donne accès à l'état du port série : un caractère reçu est disponible ou non dans le registre de donnée, une erreur de parité a été détectée, etc…

Dans une liaison série asynchrone, l'unité d'émission est le caractère qui peut être codé sur 5 à 8 bits utiles selon l'ampleur du jeu de caractères à transmettre. Un jeu de caractères à 8 bits autorisera 256 caractères différents.

L'horloge de transmission n'est pas émise sur un signal séparé. La fréquence des transitions est convenue entre l'émetteur et le récepteur et elle est générée par un oscillateur à quartz du côté émetteur comme du côté récepteur. Avant l'émission d'un caractère, il faut donc indiquer que la transmission va commencer, c'est le rôle du bit de start.

1. La ligne étant au repos à l'état logique 1, le bit de start sera un 0 à la sortie du port série qui indiquera au destinataire le début de la transmission et lui permettra de retrouver la phase de l'horloge.
2. Les bits utiles sont émis à la suite du bit de start. Un bit de parité peut être émis à titre de contrôle d'intégrité des données. Une parité paire est un 1 si le nombre de 1 dans les données utiles est pair. Dans le cas contraire, une parité paire sera un 0. Une parité impaire est un 1 si le nombre de 1 dans les données utiles est impair. Dans le cas contraire, une parité impaire sera un 0. L'émission d'une parité n'est pas obligatoire. L'émetteur et le destinataire doivent en convenir.
3. Avant d'émettre le caractère suivant, l'émetteur doit imposer un silence minimal sur la ligne appelé bit stop ou bit d'arrêt. L'état de silence de la ligne est l'état 1. Toujours par convention entre l'émetteur et le destinataire, ce silence aura une durée supérieur ou égal à 1 ou 1,5 ou 2 périodes de l'horloge de transmission. On parlera de 1 bit stop, 1.5 bit stop ou 2 bits stop. Si l'émetteur n'a rien à émettre, il peut maintenir un silence de plus longue durée sur la ligne.

A la sortie du port série, les niveaux sont TTL. Pour s'adapter à une norme électrique tels que V24 ou RS232C, il est nécessaire de recourir à des circuits "drivers" de ligne qui adaptent les niveaux TTL aux niveaux électriques requis par la norme. Les boucles de courant convertissent les niveaux TTL en passage ou non-passage d'un courant plutôt qu'un niveau électrique. Ce type de liaison assure généralement une meilleure immunité au bruit.

Les réseaux d'ordinateurs transmettent les informations de manière synchrone. L'opposition de vocabulaire "synchrone" ou "asynchrone" est assez discutable. La différence est l'unité de transmission et donc les moments où la synchronisation a lieu après un silence. La liaison asynchrone synchronise avant chaque caractère, la liaison synchrone synchronise avant chaque trame. Une trame a une taille de quelques dizaines à quelques milliers d'octets.

1.9. Constitution

Un port série est constitué de registres permettant au processeur non seulement d'émettre et de recevoir des données mais aussi de paramétrer son fonctionnement et de connaître son état.

- Registre de commande (write) et d'état (read)
- Registre de données
  - à émettre (write)
  - reçue (read)
- Registre de commande
  - rang de division
  - nb de bits de donnée
  - nb de bits d'arrêt
  - type de parité
  - génération d'interruptions
  - signaux de service
- Registre d'état
  - erreurs, indicateurs
  - signaux de service
- Emetteur
- Récepteur
- Pré diviseur

Afin de limiter les risques de perte de données, un double registre de données voir une petite file est parfois présente. Avec le fonctionnement en simple registre de données, une donnée est reçue dans le récepteur constitué d'un registre à

décalage. Quand la donnée est complète, elle est transférée dans le registre de donnée accessible en lecture par le processeur. Si le processeur ne la lit pas suffisamment rapidement, il se peut que le registre de donnée soit écrasé par la réception de la donnée suivante.

1.10. Brochage

- Côté microprocesseur
  - Quelques signaux d'adresses (sélection des registres)
  - chip select
  - bus de données
- Côté utilisation
  - Tx, Rx : émission, réception (niveau TTL)
  - TxClk, RxClk : horloges
  - RTS, CTS, DTR, DSR : signaux de service du modem

Ce port série est double : le signal A#/B, relié à un signal d'adresse permet de choisir entre les deux ports série. Le signal C/D# permet de choisir entre l'adresse du registre de commande-état d'une part et le registre de données d'autre part.

	$C/\overline{D} = 0$	$C/\overline{D} = 1$
$\overline{A}/B = 0$	Données canal A	Contrôle canal A
$\overline{A}/B = 1$	Données canal B	Contrôle canal B

Une petite logique de décodage interne élabore les ordres d'entrée et de sortie des registres présents dans un contrôleur d'entrée-sortie. Aussi son brochage comportera-t-il quelques signaux d'adresses (autant que nécessaire pour adresser le nombre de registres qu'il renferme) mais aussi les signaux RD# et WR# et bus de données. Il comportera également un ou plusieurs chip select. Selon le type d'accès (espaces adressables séparés ou espace unique) le chip select sera activé en tenant compte ou non d'un signal de type M/IO#.

Les contrôleurs d'entrée-sortie comportent aussi des signaux spécifiques à leur fonction : signaux d'émission et de réception pour un port de communication, détection de collision pour un port ethernet, etc…

1.11. Microcontrôleur

Un microcontrôleur, ou microprocesseur embarqué, associe sur une même puce, plusieurs dispositifs :

- Microprocesseur + RAM + ROM + dispositifs d'entrées-sorties
- Multiples entrées d'interruptions[2] internes et externes
- Choix selon ressources nécessaires
- Se programme comme une ROM
- Adapté aux "petits systèmes" 8, 16, 32 bits

Outre des dispositifs d'entrée-sortie, un microcontrôleur intègre un microprocesseur et de la mémoire RAM, ROM, éventuellement Flash ou EEPROM.

[2] Les interruptions seront étudiées dans le chapitre consacré à la gestion des entrées-sorties.

Un microcontrôleur est conçu pour exécuter des programmes orientés entrée-sortie, c'est-à-dire tenant compte des informations lues sur les registres d'entrée pour écrire dans les registres de sortie. Les premières applications des microcontrôleurs faisaient appel à des programmes simples. Ils pouvaient se passer de système d'exploitation et être écrits en assembleur. Mais comme toujours dans le domaine de la technologie, l'ambition des ingénieurs a augmenté avec les années, et le besoin de petits systèmes d'exploitation s'est fait sentir, notamment pour gérer l'exécution apparemment simultanée de plusieurs programmes. L'interface utilisateur fait de plus en plus facilement appel à un écran graphique, ce qui fait ressembler les programmes des microcontrôleurs à ceux des ordinateurs.

On parle d'informatique embarquée ou enfouie. C'est un domaine qui a le vent en poupe, avec l'explosion du marché des téléphones portables, de l'informatique nomade, et autres produits grand public à fort volume de vente.

Certaines réactions du microcontrôleur doivent intervenir dans un délai garanti, comme pour un système ABS de voiture. Dans ce cas, on a généralement recours à un système d'exploitation ou noyau temps-réel.

2. Le matériel : travail dirigé

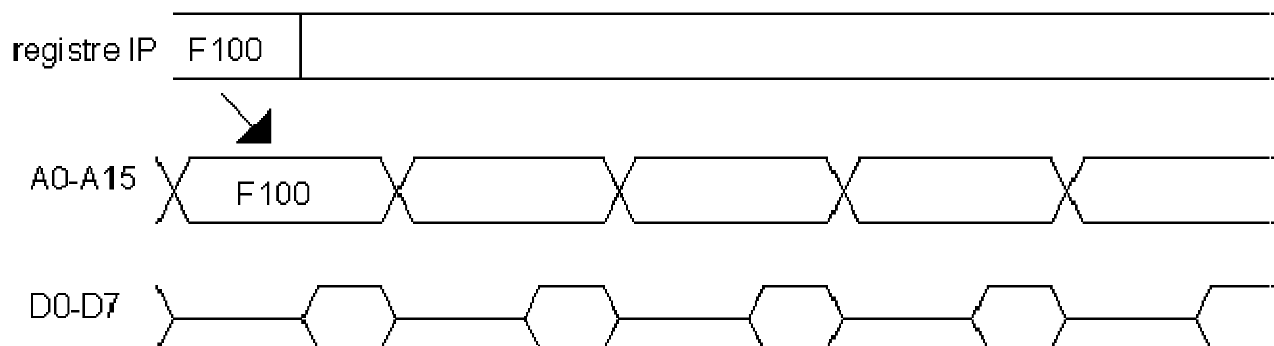
2.1. Déroulement d'une séquence d'instructions

Imaginez l'activité sur le bus d'adresses et le bus de données lorsqu'un processeur accède en lecture à une case mémoire en lecture. Pour fixer les idées, disons qu'il s'agit d'un processeur 8 bits ayant un bus d'adresses de 16 bits. L'instruction effectuée est la suivante.

Adresse	Code machine	Code assembleur	
F100	A07CF3	mov al,[F37C]	⇐ Adressage direct

On suppose également que les informations stockées en mémoire respectent l'ordre poids faible puis poids fort.

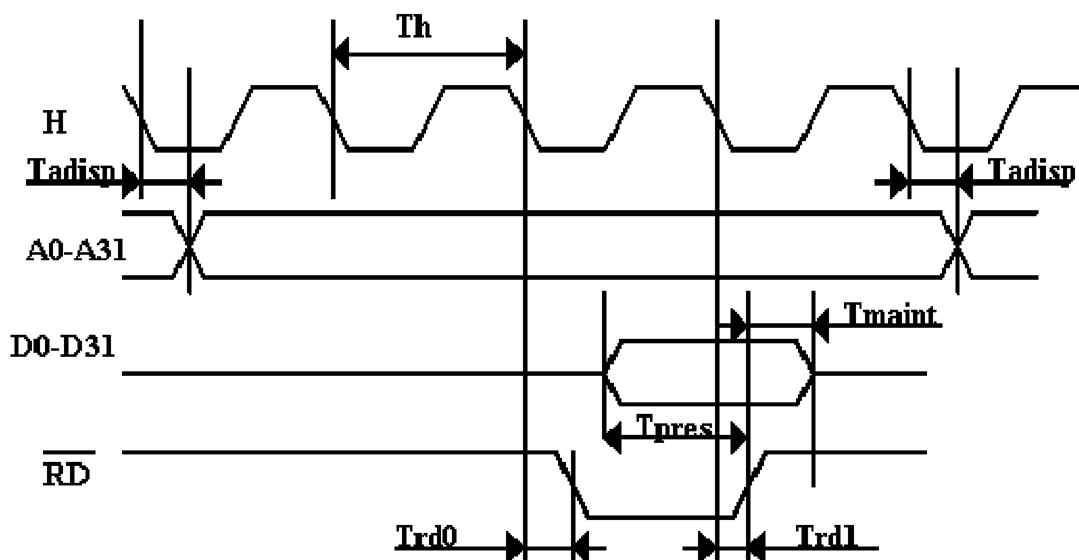
1. Complétez le chronogramme suivant.



2. Quelles sont les différentes informations qui circulent sur le bus de données ?

2.2. Chronogramme de lecture

La fiche technique d'un microprocesseur comporte le chronogramme de lecture suivant. Observer :



1. A quoi voit-on qu'il s'agit d'un chronogramme de lecture ?
2. Combien de périodes de l'horloge H y a-t-il pour le transfert d'un mot ?
3. Quelle est la taille de ce mot ?
4. Le constructeur précise également que deux octets consécutifs en mémoire sont distants de 1 dans l'espace adressable. Quelle est la taille de l'espace adressable ?

2.3. Performances des processeurs

La fiche technique fournit également les durées suivantes.

Temps	Signification	Min.	Max.
Th	Période d'horloge	50 ns	
Tadisp	Adresse disponible	7 ns	10 ns
Trd0	Read à 0	8 ns	10 ns
Trd1	Read à 1	7 ns	10 ns
Tpres	Préselection	30 ns	
Tmaint	Temps de maintien	5 ns	

Questions

1. Quelle est la fréquence maximale à laquelle ce processeur peut fonctionner ?
2. Calculer en octets par seconde le débit possible sur le bus de ce microprocesseur à la fréquence maximale du microprocesseur.
3. Sachant que le nombre moyen d'instructions exécutées par période de l'horloge H est de 0,22 calculer la puissance moyenne de ce processeur.

2.3.1. Temps imposé à la mémoire par le microprocesseur

Des mémoires de 50 ns de temps d'accès sont reliées directement au bus de ce microprocesseur. Le processeur imposant à la mémoire le rythme des accès, au-delà d'une certaine fréquence de fonctionnement, la mémoire ne répondra pas suffisamment vite.

Questions

1. A quel instant les données en provenance de la mémoire doivent-elles être présentes sur le bus de données du processeur pour que la lecture s'effectue correctement ?
2. A quel instant la mémoire "voit-elle" démarrer le cycle de lecture ?
3. Calculer le temps d'accès imposé par le processeur fonctionnant à sa fréquence maximale ?
4. Quel temps d'accès la mémoire peut-elle tenir ?
5. A quelle fréquence maximale peut-on faire tourner le microprocesseur pour que les mémoires puissent tenir le temps d'accès imposé par le processeur ?

2.3.2. Overhead du système d'exploitation, puissance utile

Le système d'exploitation utilise pour son propre compte une partie du temps CPU, ceci réduit d'autant la puissance disponible pour les applications. Le terme "overhead" désigne le pourcentage du temps ou de puissance consommée par le système d'exploitation. Cette charge est du reste variable au cours du temps selon le type de situation auquel il a à faire face. On parle donc d'overhead moyen ou de pointe.

Questions

1. Quelle est la puissance utile minimale dont disposent les applications, si le système d'exploitation consomme en pointe 10% de la puissance CPU.

2.3.3. Durée des entrées - sorties

Le port série est le dispositif de communication asynchrone utilisé dans les modems. Il est chargé d'émettre sous forme série les données d'un programme et de paralléliser les données reçues sous forme série. Le processeur est ainsi libéré de cette tâche. L'horloge n'est pas émise sur un signal séparé. Le rythme de transmission doit être convenu entre l'émetteur et le récepteur et généré de chaque côté.

1. On parle de liaison série asynchrone. Pourquoi utilise-t-on ce terme ? A quelles occasions y a-t-il synchronisation ?
2. Que signifie 1,5 bit d'arrêt ? Peut-on couper un bit en 2 ?
3. A quoi correspond la notion de bit d'arrêt ?
4. Quels sont les paramètres relatifs à la parité côté émetteur ? côté récepteur ?
5. Qu'est-ce qu'une parité paire ? une parité impaire ?
6. On considère une liaison à 9600 bps, 8 bits de données utiles, 2 bits d'arrêt, parité paire.
7. Quel est le temps minimum qui s'écoule entre le début d'émission d'un caractère et le début d'émission du suivant ?
8. On utilise un sous-programme qui prend en charge l'émission d'un tampon de mémoire de 20 octets. Calculer le nombre d'instructions que cela représente pour un processeur 50 Mips ?

3. Le logiciel : modèle de programmation

Les principaux concepts liés à l'assembleur seront présentés de la façon la plus générale possible; cependant, les exemples donnés feront appel au jeu d'instructions de la famille 86 d'Intel.

3.1. Organisation de la mémoire

Bien que le 8086 ait un bus de données 16 bits, il lui est quand même possible d'adresser des octets. Ainsi, avec ses 20 bits d'adresses, il lui est possible d'adresser 1 Mégaoctet sous forme d'octets ou sous forme de mots de 16 bits. La terminologie employée est la suivante :

- octet : 8 bits,
- mot : 16 bits,
- double mot : 32 bits.

Deux octets situés à des adresses consécutives constituent un mot. Quatre octets (ou deux mots) consécutifs constituent un double mot. Les types d'accès à la mémoire générés par le microprocesseur sont soit des accès par octet, soit des accès par mot.

Supposons que la mémoire contienne les octets `5F, A3, 3C, 24` et `32` stockés à partir de l'adresse n

- @n : 5F
- @n+1 : A3
- @n+2 : 3C
- @n+3 : 24
- @n+4 : 32

Intel utilise le stockage dit "little-endian" alors que Motorola utilise le stockage "big-endian"[3]. Comme ce cours est illustré avec des processeurs Intel, voyons en quoi consiste le stockage "little-endian".

Un accès mot à l'adresse n lira en fait les octets consécutifs `5F` et `A3`. L'ordre "little-endian" considère que le poids faible 5F correspond à l'adresse la plus faible n. On peut alors dire que le mot stocké à l'adresse n est `A35F`. De la même façon, on aura les mots suivants :

- @n : A35F
- @n+2 : 243C

Un accès mot à l'adresse n+1 donnerait `3CA3`, mais 3C et A3 correspondent aux mêmes cases mémoires que celles utilisées dans les mots `A35F` et `243C`. Si le 8086 est capable de générer 2^{20} adresses différentes, cela signifie qu'il est capable d'adresser 2^{20} octets soit 1 Mo.

Un accès par double mot à l'adresse n lira les octets `5F`, `A3`, `3C` et `24`. Le stockage little-endian considère que le poids le plus faible est 5F et que le poids le plus fort est 24. On peut alors dire que le double mot stocké à l'adresse n est `243CA35F`. Un accès double mot à l'adresse suivante fera apparaître 3 octets commun à ce mot :

- @n : `243CA35F`
- @n+1 : `32`**`243CA3`**

La granularité dans l'inversion des poids est l'octet car les adresses sont en général des adresses d'octets.

Cette différence dans l'ordre de stockage pourrait poser des problèmes d'interopérabilité pour des machines reliées en réseau. En fait, c'est le réseau qui impose un ordre commun. La couche "présentation" de chaque machine est chargée d'adapter les informations transmises à l'ordre du réseau et celles reçues à l'ordre de la machine. TCP/IP et les autres protocoles d'Internet sont généralement big-endian.

---

[3] Cette allusion aux Voyages de Gulliver insiste sur l'aspect purement arbitraire de ce choix. Une traduction de cette fiction emploie les termes "petit-boutiens" et "gros-boutiens". Ces deux peuples se livraient une guerre sans merci, car ils n'étaient pas d'accord sur le côté par lequel ouvir les œufs qu'ils consommaient.

3.2. Le modèle de programmation

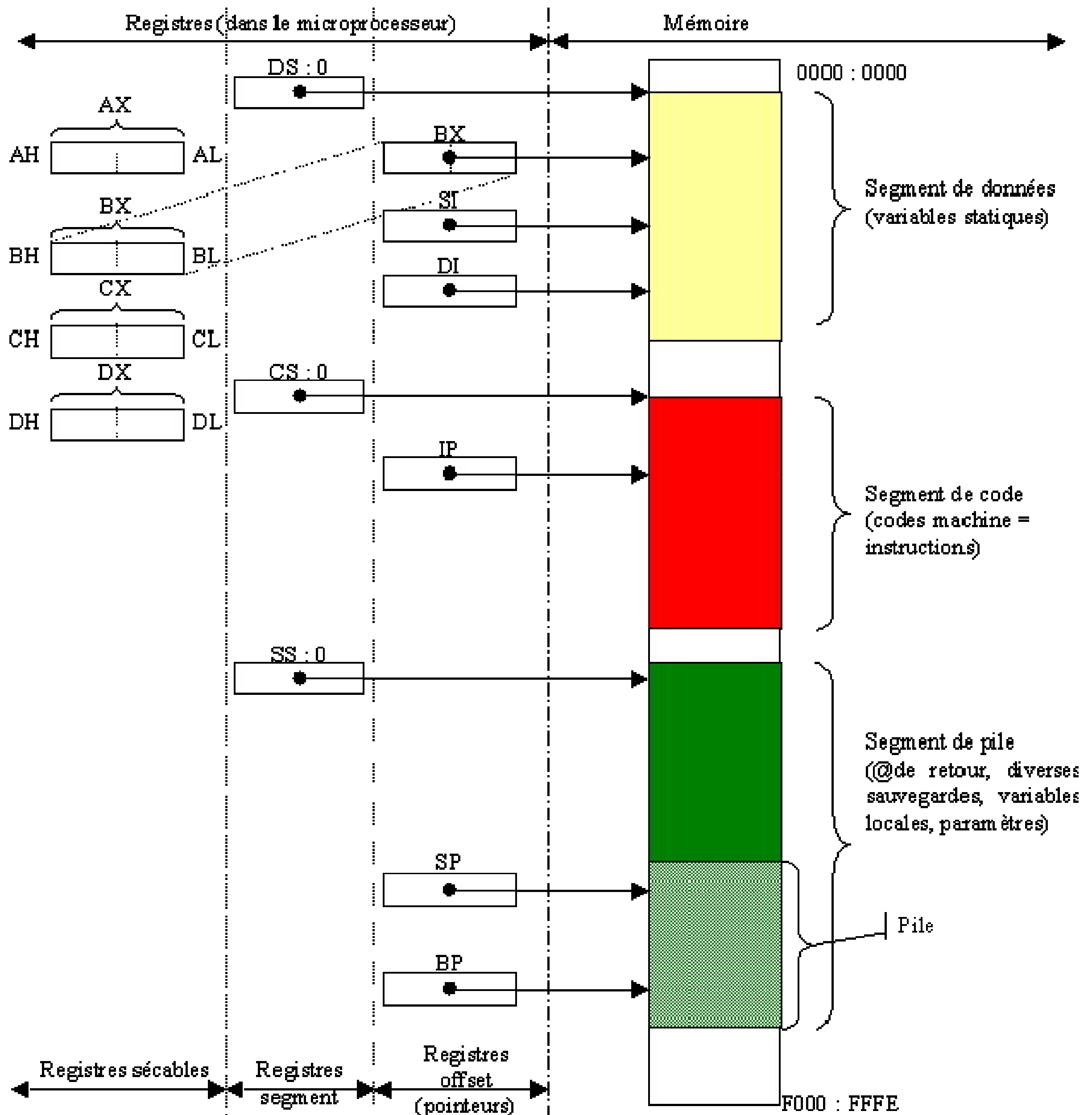
Certains **registres** du microprocesseur sont **accessibles au programmeur** par l'emploi des instructions appropriées. Du point de vue du programmeur le microprocesseur représente des **instructions** mais également un certain nombre de registres et de **mémoire**. Tout cela constitue le **modèle de programmation** du microprocesseur.

On peut distinguer deux types de registres : ceux qui servent pour mémoriser les opérandes des opérations et ceux qui sont là pour des raisons plus techniques comme le contrôle de l'exécution du programme ou la gestion des zones de mémoire.

Pour accéder aux registres, il n'est pas besoin d'adresse, il suffit de citer le nom du registre dans l'instruction. Ceci est le résultat du codage spécifique de l'instruction en une valeur numérique unique : le code machine. En fait, le registre à utiliser est codé sur un certain nombre de bits du mot constituant l'instruction. Ceci est également possible à cause de la **situation privilégiée des registres** au cœur du microprocesseur, en relation directe avec les autres éléments constituants.

Les registres sont caractérisés par leur **nom** bien sûr, mais également par leur **taille** et leur **rôle**. Les rôles principaux dévolus aux registres sont la **manipulation des données** et l'**adressage**, c'est-à-dire l'accès aux données en mémoire. Enfin, on trouve des registres plus particuliers ou qui en plus de leur utilisation traditionnelle ont un **comportement particulier** avec certaines instructions.

3.2.1. Modèle de programmation du 8086



Pour la famille 86 des processeurs INTEL, le modèle de programmation comprend les registres suivants :

AX, BX, CX, DX sont quatre registres de 16 bits également accessibles par demi-registres de 8 bits : AH, AL, BH, BL, CH, CL, DH, DL. Les suffixes H et L signifient "High" et "Low". Ces registres permettent de manipuler des données. De plus, BX peut aussi servir de registre d'adresse.

Les registres d'adresses sont employés lors des opérations d'adressage, c'est-à-dire l'accès à la mémoire externe. Tous les registres comportant le mot "pointer" sont des registres d'adresses :

- On trouve bien sûr le pointeur d'instructions IP (Instruction Pointer) encore appelé PC (Program Counter)
- Le registre SP (Stack Pointer) autorise la manipulation de la pile des appels de sous-programmes.
- Le registre BP (Base Pointer) permet d'accéder aux éléments de la pile autrement que par son sommet.

Il en va de même pour les registres comportant le mot index :

- Les registres SI (Source Index) et DI (destination Index) sont des registres autorisant l'adressage indirect.

Les registres BX, SI, DI et BP sont des registres d'adresses utilisés pour adresser les variables manipulées par les programmes. Les autres registres d'adresses comme SP et IP ont un rôle plus technique qui sera explicité.

Les registres de segment sont des registres ayant un rôle typiquement technique : ils permettent de distinguer des zones de mémoire selon les usages qu'on en fait. On distingue les segments de code (CS = Code Segment contenant les instructions), les segments de données (DS = Data Segment), les segments de pile (SS = Stack Segment). Les registres de segments CS, DS et SS repèrent à un instant donné, la position respective des segments de code, de données et de pile. Un registre segment supplémentaire (ES = Extra Segment) repère au besoin un deuxième segment de données.

On trouve également le mot d'état (FLAGS) qui est un registre regroupant comme son nom l'indique différents **indicateurs** (drapeaux) sur l'état du microprocesseur à la suite de l'exécution d'une instruction. L'exemple le plus évident est l'indicateur de retenue (CARRY), utile notamment pour l'addition, qui occupe un bit du registre FLAGS. Dans la description du fonctionnement de chaque instruction, on indique comment sont affectés les divers indicateurs.

Les indicateurs arithmétiques sont les suivants :

- CF (Carry Flag) : retenue pour l'addition ou pour la soustraction,
- AF (hAlf carry Flag) : retenue pour le quartet de poids faible,
- ZF (Zero Flag) : indique un résultat nul (quand il est positionné à 1 !!!),
- SF (Sign Flag) : indique un résultat négatif (recopie du bit de poids fort du résultat),
- PF (Parity Flag) : indique que le nombre de bits 1 dans l'octet de poids faible du résultat est pair,
- OF (Overflow Flag) : indique un dépassement de capacité de l'opération.

3.3. Segmentation du 8086

Nous présentons ici le modèle de segmentation de la famille Intel 8086. La segmentation du 386, plus évoluée est présentée plus loin dans les aspects relatifs aux systèmes d'exploitation.

3.3.1.1. Définition

La segmentation est une façon que le microprocesseur a d'adresser et de voir la mémoire. Le microprocesseur 8086 comporte 20 bits d'adresse, sa capacité d'adressage est donc d'un Mégaoctet. Cependant, les registres ne comportant que 16 bits pour des raisons historiques, la manipulation d'adresses 20 bits n'est donc normalement pas possible.

L'analyse des valeurs d'adresses permet d'établir que les **accès** aux instructions sont pendant un quantum de temps, **localisés** pour la plupart d'entre eux dans une zone d'adressage limité. Plus les accès sont lointains, plus ils sont rares. Les concepteurs du 8086 ont donc fait le pari que l'on pouvait travailler implicitement avec des adresses 16 bits la plupart du temps, et que la manipulation (explicite) d'adresses 20 bits étant plus rares pouvaient faire l'objet d'un mécanisme plus complexe.

C'est ce mécanisme qui porte le nom de **segmentation**. Il fait appel à des **registres de segment** contenant une composante de l'information d'adresse qui change peu (*composante continue*) et aux **registres d'adresses offsets** qui contiennent en quelque sorte la *composante variable* de l'information d'adresse. Les registres BX, SI, DI, SP, BP, IP sont des registres d'adresses offsets.

Comme le microprocesseur utilise **plusieurs flots d'adressage** (un pour le programme, un pour la pile, et un ou deux pour les données), **plusieurs registres segments** sont nécessaires. En effet, le 8086 comporte effectivement un registre segment pour le code (CS : Code Segment), un registre segment pour la pile (SS : Stack Segment) et deux registres de segment pour les données (DS : Data Segment et ES : Extra Segment).

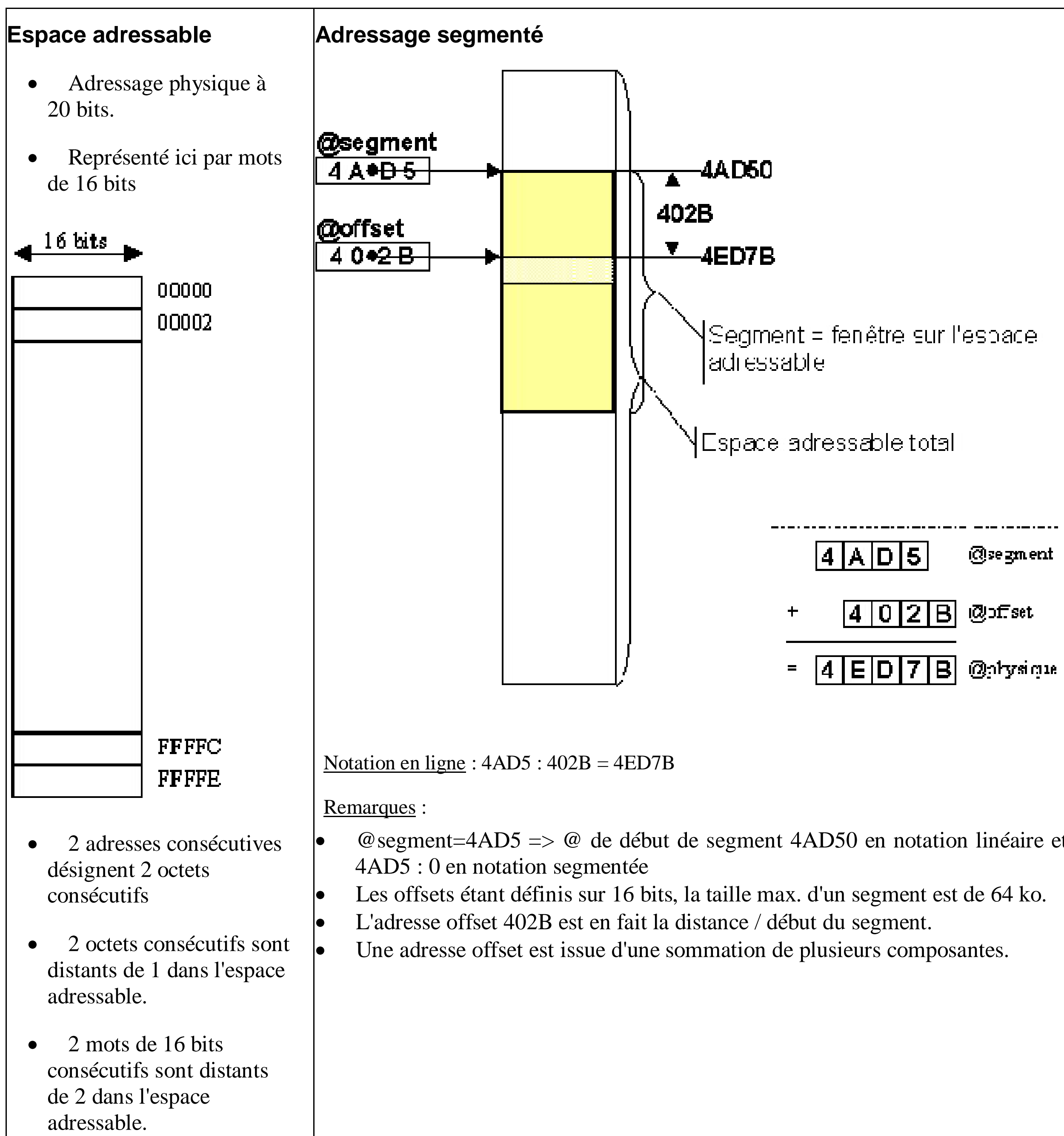
Certaines instructions manipulent seulement la partie offset, d'autres manipulent la partie offset et la partie segment. Il y a deux types d'adresses : les adresses "near" ou adresses offset, codées sur 16 bits et les adresses "far" ou adresses segmentées, codées sur 32 bits, c'est-à-dire comportant une partie segment et une partie offset. Il existe donc des pointeurs near occupant 16 bits en mémoire et des pointeurs far occupant 32 bits en mémoire. Les sous-programmes (les équivalents assembleur des procédures / fonctions des langages évolués) étant repérés par des adresses peuvent être des sous-programmes near ou des sous-programmes far.

3.3.1.2. Calcul de l'adresse

L'adresse physique, celle qui circule sur le bus d'adresses résulte d'un calcul mettant en œuvre une partie segment codée sur 16 bits provenant généralement d'un registre segment et une partie offset également codée sur 16 bits.

ADR_{20bits} = partie $segment_{16bits}$ * 16 + partie $offset_{16bits}$

On peut remarquer que multiplier par 16 décimal consiste à décaler de 4 bits vers la gauche la partie segment soit d'ajouter un 0 à droite en hexadécimal, le résultat comporte donc 20 bits soit 5 digits hexadécimaux. En rajoutant la partie offset, on peut atteindre 21 bits mais de tels calculs d'adresses ne sont pas souhaitables et le résultat est tronqué à 20 bits. Un segment peut comporter **64 ko** au maximum : la composante variable est exprimée sur 16 bits. Le début du segment correspond à un offset 0 et la fin du segment correspond à l'offset FFFF exprimé en hexadécimal. Un segment ne peut donc commencer qu'à une adresse multiple de 16. Une telle adresse, exprimée en hexadécimal se termine par un 0. Une telle adresse est appelée frontière de paragraphe. Une zone de 16 octets s'appelle en effet un **paragraphe**. Un segment comporte donc au moins **16 octets**. Le paragraphe est l'unité de granularité du segment et les segments commencent sur des frontières de paragraphes.



Espace adressable

- Adressage physique à 20 bits.
- Représenté ici par mots de 16 bits
- 2 adresses consécutives désignent 2 octets consécutifs
- 2 octets consécutifs sont distants de 1 dans l'espace adressable.
- 2 mots de 16 bits consécutifs sont distants de 2 dans l'espace adressable.

Adressage segmenté

Notation en ligne : 4AD5 : 402B = 4ED7B

Remarques :

- @segment=4AD5 => @ de début de segment 4AD50 en notation linéaire et 4AD5 : 0 en notation segmentée
- Les offsets étant définis sur 16 bits, la taille max. d'un segment est de 64 ko.
- L'adresse offset 402B est en fait la distance / début du segment.
- Une adresse offset est issue d'une sommation de plusieurs composantes.

3.3.2. Inconvénients

En introduisant le mécanisme de segmentation, les concepteurs du 8086 ont malheureusement **compliqué** sa programmation. La compréhensibilité du jeu d'instructions s'en trouve également réduite.

De plus, les segments ont une taille maximale fixe de 64 ko. Or, la propriété de localité d'exécution des programmes n'est pas applicable aux données. Prenons le cas de l'accès aléatoire à un tableau de données. Si le **tableau** fait moins de 64 ko, tout va bien, il suffit d'attribuer un registre segment pour le tableau et les accès seront tous dans le segment. Si le tableau fait **plus de 64 ko**, comme l'accès est aléatoire, les instructions doivent remettre en cause a priori le contenu du **registre segment** et donc le **recalculer** à chaque accès.

Dans la plupart des langages évolués, certaines restrictions du 8086 sont réellement contraignantes :

- Un tableau ne peut excéder 64 ko.
- Une variable (même dynamique) ne peut excéder 64 ko.
- La pile est limitée à 64 ko.

La limitation de la pile sera particulièrement sensible pour l'utilisation de la récursivité qui fait un usage important et systématique de la pile.

Avec l'architecture segmentée du 386, toutes ces contraintes disparaissent car les segments ont une taille maximale de 4 Go.

3.3.3. Avantages

Parmi les avantages qu'offre la segmentation, on citera la **compacité du code** obtenu à condition que le compilateur ne génère pas du code qui recalcule systématiquement le contenu des registres de segment.

Remarque préalable :

L'emplacement exact où sera logé un programme en mémoire vive pour y être exécuté, n'est pas figé. Une partie du système d'exploitation est chargée de trouver de la place. L'emplacement s'exprime en termes d'adresses et on comprend alors que ces adresses ne sont connues qu'au moment du **chargement** en mémoire, juste avant l'exécution. La qualité qu'a un programme de pouvoir s'exécuter à un endroit de la mémoire qui n'est pas connu avant ce moment s'appelle la **relogeabilité**.

Un des avantages de la segmentation est de **faciliter le relogement** des programmes. Le relogement d'un exécutable concerne le code, la pile et les données. Les adresses offsets sont déterminées lors de la production du programme exécutable. L'adresse segment est déterminée lors du chargement en mémoire vive.

Dans le cas idéal de programme où on n'a jamais à remettre en cause le contenu des registres de segment, le programme **chargeur** n'a qu'à définir le contenu des registres de segment une fois pour toutes.

En dehors du cas idéal, les programmes comporteront généralement plusieurs segments avec des instructions faisant des références entre segments. Ceci oblige à employer des instructions de chargement des registres de segment avec des adresses complètes.

3.4. Représentation à la Turbo-Debugger

La mémoire peut être représentée de différentes façons par les outils de mise au point (débogueurs). On peut la représenter en tant que données ou en tant qu'instructions. Dans tous les cas, la colonne de gauche précise à quelle adresse est situé le premier mot représenté sur la ligne. Dans le cas du 8086, l'adresse est représentée sous sa forme segmentée. La partie segment à gauche est séparée de la partie offset à droite par un double point. Il se peut que la partie segment soit celle contenue dans un registre. Dans ce cas, le nom du registre est affiché à la place de la valeur numérique.

3.4.1. Représentation de la mémoire en tant que données

Dans la représentation en tant que données, on a le choix entre une représentation par octets et une représentation par mots de 16 bits. Dans la représentation par octets, les données sont affichées en hexadécimal et en ascii lorsqu'un caractère affichable existe. Il s'agit de 2 représentations des mêmes données.

Adresses	Octets en hexa	Octets en ascii
ds:0000	73 61 6C 75 74 00 20 20	salut
Adresses	Mots de 16 bits en hexa	
53DC:0000	6173 756C 0074 2020	

Dans la représentation par mots, deux octets consécutifs sont groupés et remis dans l'ordre naturel qui est opposé à l'ordre de stockage en mémoire qui est l'ordre little-endian pour les processeurs Intel.

3.4.2. Représentation de la mémoire en tant qu'instructions

Les outils de mise au point représentent les instructions de deux façons : sous la forme numérique des codes machine d'abord, sous la forme symbolique des codes assembleur ensuite. Il s'agit de 2 représentations des mêmes instructions.

Adresses	Codes machine	Codes assembleur

cs:0100
cs:0103
cs:0106
	B80100
	BB0200
	03C3
		mov ax,0001
		mov bx,0002
		add ax,bx

3.4.3. Assembleur en ligne de Turbo-C++

Les assembleurs existent en tant qu'outils indépendants de traduction d'une syntaxe "langage d'assemblage" vers des fichiers objets puis vers un fichier exécutable en utilisant un éditeur de liens traditionnel. Les fichiers sources assembleur comportent, non seulement des instructions en langage d'assemblage, mais aussi des déclarations de données et de constantes, avec une syntaxe propre.

Une autre approche de l'assembleur consiste à inclure du code assembleur dans un programme écrit en langage évolué. Les compilateurs C/C++ de Borland permettent d'inclure du code assembleur dans du code C/C++. On parle d'assembleur en ligne ou inline. Le langage C est un langage à structure de bloc. Un bloc en langage C est délimité par des accolades. Borland introduit un nouveau type d'instructions et de blocs préfixé par le mot-clé asm (assembler). Ainsi, il est possible d'entrer une instruction assembleur dans un code C en le préfixant par asm :

```
asm mov ax,1
```

De la même façon, on peut définir un bloc asm :

```
asm {
      mov ax,1
      mov bx,2
}
```

Pour la définition des constantes et des variables, on a recours au langage C et on peut les manipuler en langage d'assemblage :

```
#define NULL 0
int maVar;
...
asm {
      mov ax,NULL
      mov maVar,ax
      ...
}
```

L'avantage est que les données sont typées, contrairement à ce qui se passe avec les syntaxes assembleur. De la même façon, on utilise les étiquettes du langage C qu'on référence par les instructions de sauts et de sauts conditionnels du

langage d'assemblage. L'inconvénient de cette technique est une certaine lourdeur introduite par la nécessité de fermer le bloc asm pour définir une étiquette en C et de rouvrir ensuite le bloc asm comme dans l'exemple suivant :

```
asm {
      jmp etiqDu_C
      ...
}
etiqDu_C:
asm {
      ...
}
```

Les variables locales et les paramètres peuvent aussi être référencés par les instructions assembleur. Dans ce cas, l'assembleur traduit l'instruction par un adressage basé sur BP comme on doit le faire pour ce type de variable. Ceci sera étudié dans le chapitre consacré aux paramètres et variables locales :

```
void proc1( int param1 )
{
      int varLoc1;
asm {
      mov   ax,param1;  // traduit par un adressage basé sur BP
      mov   varLoc1,ax; // idem
}
...
}
```

3.5. Association par défaut des registres de segment

Les registres de segment sont prévus pour travailler de concert avec les registres d'adresses offset. Il existe même des **associations par défaut** pour ces collaborations entre registres classiques et registres de segment.

Certaines **associations** sont **naturelles** : le registre IP (pointeur d'instructions) est ainsi associé au registre du segment de code CS. De même, le registre du segment de pile SS est associé aux registres SP et BP prévus pour adresser la pile.

Pour les données, le mode d'adressage, déterminera le registre de segment concerné. En fait, la **règle générale** spécifie que c'est DS qui est utilisé avec les registres BX, SI et DI.

Avec l'adressage basé et indexé, le segment associé par défaut est celui normalement associé avec le **registre de base**, c'est-à-dire soit BX soit BP. Le registre segment SI ou DI n'intervient plus dans ce cas dans le choix du registre de segment associé. BP étant un registre pour adresser la pile, l'adressage basé sur BP avec ou sans indexation utilisera le registre du segment de pile SS. L'adressage basé sur BX avec ou sans indexation utilisera le registre du segment de données DS.

3.6. Méthodologie

Afin de ne pas perdre les bonnes habitudes de programmation structurée, vous êtes invités à respecter la méthode suivante :

1. vous écrivez une ligne de "programme" en pseudo-langage,
2. vous la mettez en commentaire,
3. vous écrivez **au-dessous** de cette ligne les instructions assembleurs qui font ce que dit le commentaire, rien de plus, rien de moins.

3.6.1. Du bon usage des commentaires

Les commentaires isolés sur une ligne sont des commentaires de spécification : ils disent ce qu'est censé faire le code assembleur qui le suit. Ils doivent permettre la compréhension globale de ce qui est réalisé par le code assembleur. Pour cela, le lecteur n'est censé lire que ce type de commentaires en faisant abstraction des commentaires de fin de ligne.

Les commentaires de fin de ligne sont des commentaires d'implantation. Ils doivent normalement concerner des particularités ou des difficultés relatives au codage en assembleur. Ils pallient la pauvreté sémantique du langage d'assemblage.

4. Représentation des données

4.1. Les modes d'adressage

Une des activités importante que permette un microprocesseur est l'accès aux données. La *façon d'accéder aux données* est une notion importante pour l'implantation des algorithmes. On emploie le terme de **mode d'adressage**. On emploie un mode d'adressage plutôt qu'un autre en fonction des informations connues au moment de l'assemblage. Certaines informations ne seront connues qu'au moment de l'exécution.

4.2. L'adressage immédiat

Si l'opérande lui-même est connu lors de la compilation, alors on peut employer l'adressage immédiat. Celui-ci consiste à spécifier l'opérande directement dans le code machine donc dans le segment de code.

```
#define INCR 1
int iFill;
...
//étiquette opérateur   opérandes   commentaires
asm {
            MOV         AL,5;       // 5 : adressage immédiat
            ADD         AX,INCR     // INCR : adressage immédiat
            MOV         BX,offset iFill //offset iFill adressage immédiat
            ...
```

Dans un langage évolué, l'adressage immédiat correspond à l'utilisation de constantes. Ici l'opérande est la constante 5, qu'on charge dans le registre AL. De même, cette constante peut être symbolique. Dans le deuxième exemple, l'opérande est la constante INCR équivalente à la valeur 1. On l'utilise dans l'instruction d'addition dans le registre AX. Enfin, dans le troisième exemple, l'opérande n'est pas une variable mais l'adresse d'une variable. L'adresse offset des variables réservées par le compilateur peut être connue grâce au mot-clé offset suivi du nom de la variable.

4.3. L'adressage direct

Si l'opérande lui-même n'est pas connu lors de la compilation, on ne peut pas utiliser l'adressage immédiat; l'opérande doit alors être stocké dans une variable. Si l'adresse de l'opérande est connue lors de la compilation, alors on peut utiliser l'adressage direct. Celui-ci consiste à spécifier l'adresse de l'opérande dans les instructions du programme.

```
#define INCR 1
int iFill;
...
//étiquette opérateur   opérandes   commentaires
asm {
            MOV         AX,[iFill]
            ...
```

Notons que, contrairement à l'exemple précédent, c'est ici la valeur et non l'adresse de la variable iFill qui est l'opérande de l'instruction. C'est cette valeur qu'on charge dans AX.

L'adressage direct est normalement reconnaissable aux crochets entourant le nom ou l'adresse de la variable. Cependant, les crochets, bien que conseillés, ne sont pas obligatoires. Ainsi, dans l'exemple suivant, la syntaxe des deux instructions mov est la même : les crochets de l'adressage direct autour de iFill sont absents. Le compilateur/assembleur ayant compilé les déclarations de INCR et iFill, il est capable de voir s'il s'agit d'une variable (globale) ou d'une constante, quand il rencontre une instruction manipulant ces identificateurs. La manipulation de la valeur d'une constante fait appel à l'adressage immédiat. La manipulation du contenu d'une variable globale, même en l'absence de crochets, fait appel à l'adressage direct.

```
#define INCR 1
int iFill;
...
//étiquette opérateur   opérandes   commentaires
asm {
            mov         ax,INCR     // adressage immédiat
            mov         ax,iFill    // adressage direct
            ...
```

4.4. L'adressage indirect

Si l'adresse de l'opérande n'est pas connue lors de la compilation, on ne peut pas utiliser l'adressage direct ou plus précisément l'adressage indirect par registre. Il faut alors indiquer au microprocesseur un registre qui contiendra l'adresse de l'instruction au moment de l'exécution. Une ou plusieurs instructions préliminaires récupèrent dans le registre d'indirection (BX, BP, SI ou DI), l'adresse de l'opérande peu importe comment.

Puis, on met en œuvre l'adressage indirect. L'adresse de l'opérande à manipuler n'étant pas connue lors de la compilation, on indique au processeur que cette adresse est située dans tel ou tel registre d'adresse.

Dans un langage évolué, ce type d'adressage correspond à l'utilisation des variables dont l'adresse n'est pas connue lors de la compilation. C'est le cas des variables allouées dynamiquement, par une instruction malloc par exemple. On peut aussi utiliser ce mode d'adressage pour agir sur une diversité de variables : c'est ce qu'on fait dans les procédures et fonctions comportant des paramètres où l'adresse n'est connue qu'au moment du passage des arguments.

```
// déclarations
char *p;
...
// p = malloc(26);
      ...
// p[0] = 'A';
asm {
      mov         bx,[p]            // [p] : adressage direct
      mov         byte ptr[bx],'A'; // [bx] : adressage indirect
      ...
```

L'adressage direct change de nom selon le registre qu'on emploie. Si on utilise un registre de base comme BX ou BP, on parlera d'adressage basé, ou plus précisément d'adressage basé sur BX ou basé sur BP. Si on utilise un registre d'index comme SI ou DI, on parlera d'adressage indexé ou plus précisément d'adressage indexé par SI ou indexé par DI. Quel que soit le nom utilisé, le mécanisme est rigoureusement le même.

D'autre part, rappelons que ces registres d'adresses contiennent des adresses offsets et qu'ils sont implicitement associés à un registre de segment, donc à un segment. Par défaut, [BX], [SI] et [DI] sont utilisés pour accéder au segment de données repéré par DS, tandis que [BP] est utilisé pour accéder au segment de pile repéré par SS.

L'adressage indirect possède aussi plusieurs variantes :

- l'adressage indirect avec déplacement,
- l'adressage basé et indexé,
- l'adressage basé et indexé avec déplacement.

4.4.1. L'adressage indirect avec déplacement

L'adressage indirect avec déplacement accède à un opérande dont l'adresse est connue relativement à une autre adresse contenue dans un registre de base ou d'index. La distance entre l'adresse contenue dans le registre et l'adresse de l'opérande sera appelée déplacement (pour ne pas surcharger le mot offset) Le déplacement est connu lors de la compilation et codé dans l'instruction vient s'ajouter à l'adresse présente dans le registre.

```
// déclarations
int iaTab[26];

// instructions

// iaTab[2] = 0;
asm {
      mov   si,offset iaTab   // offset iaTab : adressage immédiat
      mov    [si+4],0         // [si+4] : adr. indexé avec déplacement
      ...
```

Dans cet exemple, l'adresse de iaTab[2] est située à 4 octets de distance du début du tableau iaTab car chaque case mémorise un entier, codé sur 2 octets.

On parlera également d'adressage basé avec déplacement avec les registres BX et BP et d'adressage indexé avec déplacement avec SI et DI.

4.4.2. L'adressage basé et indexé

L'adressage basé et indexé est un adressage indirect qui met en œuvre deux registres. C'est la somme de ces deux registres qui donne l'adresse de l'opérande : [BX+SI] par exemple. A cause de la structure interne du processeur, il est impossible d'utiliser conjointement BX et BP. De même, il est impossible d'utiliser conjointement SI et DI. Ainsi, les notations ~~[BX+BP]~~ et ~~[SI+DI]~~ sont illégales. Pour que les programmeurs s'y retrouvent dans les combinaisons de registres légales ou illégales, Intel a introduit une distinction entre "registres de base" et "registres d'index". Comme on l'a vu sous la rubrique sur l'adressage indirect, cette distinction est de pure forme : l'adressage indexé et l'adressage basé sont en fait deux noms différents pour un même mécanisme. Les seules combinaisons légales de registres font appel à un registre de base et un registre d'index : [BX+SI], [BX+DI], [BP+SI] et [BP+DI]. Les deux premières combinaisons adressent le segment de données et les deux suivantes adressent le segment de pile.

```
// déclarations
int iaTab[26];
int i;

// instructions

// iaTab[i] = 0;
asm {
      mov   bx,offset iaTab   // offset iaTab : adressage immédiat
      mov   si,[i]            // [i] : adressage direct
      add   si,si             // si = si+si = si*2
      mov    [bx+si],0        // [bx+si] : adr. basé+indexé
      ...
```

Ce code n'a qu'un but pédagogique. En fait, l'adresse offset de iaTab est ici connue au moment de la compilation. On pouvait donc se passer de l'adressage basé et indexé et se contenter d'un adressage indexé avec déplacement en remplaçant alors [bx+si] par [si+iaTab] (pas besoin de mettre offset devant iaTab).

Dans un langage évolué, ce mode d'adressage correspond à l'accès à la case d'un tableau par un indice qui n'est pas connu lors de la traduction. Un premier registre contient l'adresse de début du tableau et un second registre sert au calcul de l'adresse de la case relativement au début du tableau. L'adresse de début du tableau peut aussi ne pas être connu lors de la compilation et c'est là que ce mode d'adressage prend tout son intérêt.

4.4.3. L'adressage basé et indexé avec déplacement

Cette forme d'adressage indirect est la plus sophistiquée mais le principe est toujours le même. L'adresse offset résulte de la somme de 3 composantes : une fournie par un registre de base, une par un registre d'index et une fournie par un déplacement connu lors de la compilation.

```
      // déclarations
      struct {
            int a, b;
      } saTab[10];
      int i;

      // saTab[i].b = 0;
      asm {
            mov   bx,offset saTab   // offset saTab : adressage immédiat
            mov   si,[i]            // [i] : adressage direct
            add   si,si             // si = si+si = si*2
            add   si,si             // idem, => si = 4*i
            mov    [bx+si+2],0      // [bx+si+4] : adr. basé+indexé+déplacement
            ...
```

Dans un langage évolué, ce mode d'adressage correspondrait à l'accès à un champ d'une case d'un tableau d'enregistrements (struct du C). Le déplacement donnant, quant à lui, l'adresse du champ relativement au début de l'enregistrement. Comme précédemment, ce mode d'adressage trouve tout son intérêt lorsque l'adresse de début du tableau n'est pas connue lors de la compilation sinon on pourrait écrire "[si+saTab+2]" à la place de "[bx+si+2]". En effet, si l'adresse de saTab est connue lors de la compilation la valeur saTab+2 est calculable par le compilateur et on se retrouve avec un adressage indexé avec déplacement.

Dans le 8086, l'unité d'adressage comporte un sommateur spécialisé dans le calcul des adresses physiques. Ce sommateur fait la somme de 4 composantes : une partie segment, une partie "adresse de base" issue d'un registre de base, une partie "index" issue d'un registre d'index, une partie déplacement issue du code machine lui-même.

La partie offset résulte de la somme de 3 composantes que sont l'adresse de base, l'index et le déplacement. On peut voir l'adressage indirect, comme une version dégradée de l'adressage basé+indexé+déplacement où il l'adresse de base et l'offset valent 0. De la même façon, les autres variantes de l'adressage indirect, peuvent être vues comme une version dégradée du mode d'adressage le plus complet qu'est l'adressage basé+indexé+déplacement.

4.4.4. Usages des modes d'adressage

On peut résumer les usages des modes d'adressage de la façon suivante :

Mode d'adressage	l'instruction spécifie	l'opérande est
immédiat	l'opérande lui-même	une constante
direct	l'adresse de l'opérande	une variable globale, une case d'un tableau (adresse ET indice CLDC)
Indirect par registre	le registre contenant l'adresse de l'opérande	une variable manipulée via un pointeur, allouée dynamiquement ou paramètre d'une fonction ou procédure
Indirect + déplacement	un registre et un déplacement dont la somme fournit l'adresse de l'opérande	case d'un tableau (adresse OU indice ILDC) ou champ d'une structure (adresse ILDC)
basé+indexé	deux registres dont la somme fournit l'adresse de l'opérande	une case d'un tableau (adresse ET indice ILDC)
basé+indexé+déplacement	2 registres et un déplacement dont la somme fournit l'adresse de l'opérande	un champ d'un tableau de structures (adresse ET indice ILDC)

CLDC : connu lors de la compilation, ILDC : inconnu lors de la compilation

L'adressage basé sur BP et ses variantes accèdent au segment de pile. Les autres modes d'adressage accèdent au segment de données. L'adressage immédiat n'accède à aucun segment, en tout cas pas de manière programmée. Comme pour toute instruction, le processeur lit le segment de code pour acquérir le code machine. L'instruction spécifiant l'opérande lui-même, on peut dire qu'avec l'adressage immédiat, l'opérande est stocké dans le segment de code.

Enfin, lorsqu'il n'y a pas de crochets autour d'un registre, il s'agit de l'accès à un registre et non d'un mode d'adressage et il n'y a pas d'accès programmé à la mémoire.

4.4.5. Segments mis en œuvre

Ce tableau synthétise à gauche les segments mis en œuvre, à droite la syntaxe Turbo-C/C++.

<table>
<tr><th>Accès aux segments</th><th>Modes d'adressage</th></tr>
<tr><td>
Segment de code
• Le microprocesseur accède spontanément au segment de code (à l'adresse offset contenue dans le registre IP) pour trouver les codes machine. Dans l'adressage immédiat, l'opérande est partie intégrante du code machine.
Segment de données
• adressage direct : [offset],
• adressage indirect : [BX], [SI], [DI], [BX+SI], [BX+DI], [BX+/-dépl], [SI+/-dépl], [DI+/-dépl], [BX+SI+/-dépl], [BX+DI+/-dépl]
Segment de pile
• adressage indirect : [BP], [BP+/-dépl], [BP+SI+/-dépl], [BP+DI+/-dépl]
</td><td>
• adressage immédiat : le code machine spécifie l'opérande,
• adressage direct : le code machine spécifie l'adresse offset de l'opérande,
• adressage direct : le code machine spécifie les termes de la somme fournissant l'adresse offset de l'opérande.
Exemples
<code>#define MAX 10</code>
<code>int var;</code>
<u>// adressage immédiat</u>
<code> mov ax,10</code> ou
<code> mov ax,MAX</code>
<code> mov si,offset var</code>
<u>// adressage direct</u>
<code> mov var,ax ou</code>
<code> mov [var],ax</code>
<u>// adressage indirect</u>
<code> mov [si-2],ax</code>
</td></tr>
</table>

4.4.6. Exemples de modes d'adressage en Turbo C

Avec l'assembleur in-line de Turbo-C/C++, les syntaxes sont les suivantes :

Adressage immédiat	**Adressage direct**	**Adressage indirect**
`#define MA_CONST 4` `char tab[10];`	`char str[26];` `int iFill;`	`Char *p;`

`// ax = MA_CONST;`
`mov ax,MA_CONST`
`// si = @tab;`
`mov si,offset tab`
`offset` : mot-clé de l'assembleur signifiant "adresse offset de".
	`// ax = iFill;`
	`mov ax,[iFill]`
	`// al=str[0];`
	`mov al,[str]`
	La notation `mov al,str` est aussi légale (crochets optionnels) en adressage direct car `str` est une variable.
		`// p=(char *)malloc(26);`
		`mov ax,26`
		`push ax`
		`call _malloc`
		`pop cx`
		`// p[0]='A';`
		`mov bx,ax`
		`mov byte ptr [bx],'A'`

Adressage indirect+déplacement	**Adressage basé+indexé**	**Adressage basé+indexé+déplacement**

`int iaTab[26];`	`int iaTab[26];`
	`int i;`
		`struct {`
		`int a, b;`
		`} saTab[10];`

`// iaTab[10]='1';`
`mov si,offset iaTab // imm`
`mov [si+10],'1'`
`ou :`
`mov si,10 // imm`
`mov [si+iaTab],'1'`
	`// iaTab[i]=0;`
	`mov bx,offset iaTab`
	`mov si,i`
	`add si,si // si=si*2`
	`mov [bx+si],0`
		`// saTab[i].b=0;`
		`mov bx,offset saTab`
		`mov si,i`
		`add si,si`
		`add si,si`
		`mov [bx+si+2],0`

La notation `byte ptr` force une taille d'opérande de 8 bits (16 bits pour `word ptr`) Lorsque la taille des opérandes n'est pas connue, elle est supposée de 16 bits.

4.5. L'accès aux variables

Dans un processeur CISC (Complex Instruction Set Processor) comme le 8086, la plupart des instructions peuvent utiliser des opérandes situés dans la mémoire. Dans un processeur RISC (Reduced Instruction Set Computer), seule l'instruction MOV peut y accéder, les autres instructions manipulant des opérandes systématiquement situés dans un registre.

Les processeurs CISC ont quand même des limitations dues à leur architecture interne. Aussi, certaines combinaisons d'opérandes sont-elles impossibles. Par exemple, le 8086 ne peut pas combiner comme opérandes gauche et droit deux modes d'adressage accédant à la mémoire.

Illégales	**Légales**	**Légende**

`mov` **`imm`**`,xxx`
`mov` **`op8b,op16b`**
`mov` **`op16b,op8b`**
`mov mem,mem`
	`mov reg,imm`
	`mov mem,imm`
	`mov reg,reg`
	`mov mem,reg`
	`mov reg,mem`
	`mov op8b,op8b`
	`mov op16b,op16b`
		`imm`=adressage immédiat,
		`mem`=adressage direct ou indirect,
		`op8b`=opérande sur 8 bits
		`op16b`=opérande sur 16 bits
		`reg`=registre
		`xxx`=opérande indifférent
		`mov` ou toute autre instruction à 2 opérandes.

4.5.1. L'instruction ADD

Dans l'instruction ADD [SI],AX la somme des deux opérandes est stockée dans l'opérande de gauche. Ce dernier a donc un rôle double : opérande source pour la somme et opérande destination pour le résultat. Comme cet opérande est spécifié en adressage indirect, il est situé en mémoire. Cela signifie que cette instruction effectue d'abord une lecture mémoire, puis calcule la somme des opérandes, puis effectue une écriture mémoire. Pour ce type d'instruction, on parle de cycle de lecture-modification-écriture.

4.5.2. L'instruction LEA

LEA signifie Load Effective Address. L'adresse effective est l'adresse offset qui résulte d'un adressage direct ou indirect. L'instruction LEA utilise comme opérande gauche un registre d'adresse offset, tel que BX, BP, SI ou DI. L'opérande droit est un mode d'adressage direct ou indirect. Elle charge l'adresse effective dans le registre d'adresse offset spécifié comme opérande gauche.

Contrairement, aux autres instructions mettant en œuvre un mode d'adressage direct ou indirect, l'instruction LEA n'accède pas à la mémoire. Comme on l'a vu dans le récapitulatif consacré aux modes d'adressage, le 8086 comporte une unité d'adressage comprenant un sommateur spécialisé dans le calcul des adresses physiques. L'instruction LEA récupère dans un registre, la partie offset de l'adresse qui est la somme d'une adresse de base, d'un index et d'un déplacement.

4.5.3. Les instructions LDS et LES

Ce sont des instructions qui chargent dans un couple de registres un pointeur far stocké en mémoire. Rappelons qu'un pointeur far occupe 32 bits en mémoire : 16 bits pour la partie segment et 16 bits pour la partie offset. LDS (Load Data Segment) charge la partie segment du pointeur far dans le registre DS, alors que LES la charge dans le registre ES. Ces instructions admettent deux opérandes. L'opérande gauche est le registre d'adresse offset (BX, SI, DI ou BP) recevant la partie offset du pointeur far. L'opérande droit est un pointeur far recherché dans le segment de données, celui désigné par DS juste avant d'exécuter l'instruction LDS. LDS modifiant DS, cette instruction modifie la position du segment de données. LES modifiant ES, la position du segment désigné par ES est modifiée mais pas celle du segment désigné par DS.

4.5.4. Transferts

On en a pris conscience lors de l'étude des modes d'adressage, les mouvements de données représentent une activité importante du microprocesseur. Certaines instructions sont spécialisées dans ce type de travail.

Lors d'un transfert de donnée, on identifie une source (ou origine) et une destination. La source ou la destination peut être soit un registre, soit de la mémoire. Il peut y avoir des transferts entre registre et registre, entre registre et mémoire. Les seuls transferts entre mémoire et mémoire sont de la mémoire de programme (adressage immédiat) vers la mémoire de donnée.

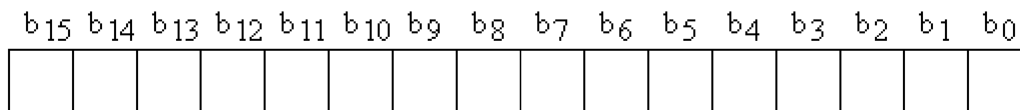
- MOV: mouvement de données,
- LEA: chargement de l'adresse d'accès dans un registre d'adresses.

4.5.5. Arithmétique

Il y a 2 types de nombres entiers : ceux pouvant prendre une valeur négative qu'on appelle les entiers signés et ceux qui sont toujours positifs ou nuls, les entiers non signés. La convention de représentation dite "en complément à 2" sert à représenter les nombres signés. L'avantage de cette représentation est qu'elle permet d'utiliser le même additionneur / soustracteur que pour les nombres non signés.

4.5.5.1. Nombres signés

Soit un nombre codé sur 16 bits :



On peut le considérer comme un nombre signé ou non signé. Pour les nombres signés, le bit de poids le plus fort (b15 pour un entier 16 bits, b31 pour un entier 32 bits) est considéré comme ayant un signe négatif. Tant que ce bit est à 0, cela n'a pas d'incidence sur le nombre qui reste positif ou nul. En revanche, si le bit de poids le plus fort est à 1, le nombre devient négatif puisqu'on ajoute une composante négative au nombre valant -2^{15}. Ainsi, un nombre composé uniquement de 0 à l'exception de b15 vaudra -2^{15} soit -32768 en décimal. Le nombre positif le plus élevé sera composé uniquement de 1 à l'exception de b15 et vaudra $2^{15}-1$ soit 32767. Le nombre -1 sera composé uniquement de 1 sans exception. Il sera la somme de 32767 et -32768. L'entier 0 sera composé uniquement de 0 sans exception.

NOMBRES NON SIGNES codés sur 16 bits	NOMBRES SIGNES codés sur 16 bits en complément à 2
$0 .. 2^{16} - 1$ 0 .. 65535	$-2^{15} .. 2^{15} - 1$ -32768 .. +32767
$N = \sum_{i=0}^{15} 2^i \times b_i$	$N = -2^{15} \times b_{15} + \sum_{i=0}^{14} 2^i \times b_i$

4.5.5.2. Instructions

Les quatre opérations + - * et / sont possibles sur les entiers. Pour les nombres en virgule flottante, on a besoin d'un coprocesseur. On peut également faire appel à une bibliothèque de sous-programmes appropriés. Il faut souligner ici l'importance des indicateurs (FLAGS) dans les opérations arithmétiques.

- ADD, ADC : addition et addition tenant compte de la retenue (CF = Carry Flag); ces deux instructions positionnent la retenue,
- SUB, SBB : soustraction et soustraction avec retenue (CF = Borrow Flag),
- importance de la notion de signe,
- DEC, INC : décrémentation (ôter 1) et incrémentation (ajouter 1),
- NEG prendre l'opposé (complément à 2).

La notion de signe a beaucoup d'importance pour la multiplication et la division. Ce n'est pas le même multiplicateur / diviseur qui est utilisé dans le cas de nombres signés ou non signés.

- MUL : multiplication non signée,
- IMUL : multiplication signée,
- DIV : division non signée,
- IDIV : division signée.

4.5.6. Logiques

Le 8086 comporte également les instructions de la logique booléenne. Chaque bit d'un mot mémoire ou d'un registre représente un élément booléen. Les opérations existantes sont le **et**, le **ou**, le **non**, et le **ou exclusif**. Les indicateurs sont également concernés par ces instructions.

Avec des techniques de masque, on peut isoler des bits dans un mot afin de ne retenir que les informations pertinentes.

- AND : le **et logique**
- OR : le ou logique
- NOT : la complémentation (le non logique)
- XOR : le ou exclusif

4.5.7. Décalages et rotations

Les instructions de décalage et de rotation permettent de cadrer les bits comme on le souhaite dans un mot. On peut également faire des multiplications et des divisions par 2.

4.5.7.1. Décalages

Les décalages peuvent s'effectuer vers la droite ou vers la gauche, soit un bit à la fois, soit du nombre de bits spécifié dans le registre CL.

- SHL : le décalage à gauche,
- SHR : le décalage à droite,
- SAL : le décalage arithmétique à gauche (équivalent à SHL),
- SAR : le décalage arithmétique à droite (différent du SHR : il y a conservation du signe).

Certaines opérations de décalage peuvent être considérées comme des opérations arithmétiques. Ces deux opérations conservent le bit de signe.

4.5.7.2. Rotations

Une rotation est un décalage avec rebouclage. L'indicateur de retenue participe ou non à la rotation mais est positionné dans tous les cas.

- ROL : rotation à gauche sans participation de la retenue,
- ROR : rotation à droite sans participation de la retenue,
- RCL : rotation à gauche avec participation de la retenue,
- RCR : rotation à droite avec participation de la retenue.

4.6. Plate-forme Turbo-C 8086 (taille des types de base)

Taille des variables selon leur type. Un pointeur est near ou far selon les options de compilation employées.

- char (entre -128 et +127), unsigned char (entre 0 et 255) : 1 octet
- int (entre -32768 et +32767), unsigned int (entre 0 et 65535), pointeur near : 2 octets
- long, unsigned long, float, pointeur far : 4 octets
- double : 8 octets

4.7. Schémas de mémoire

Les constantes et les types n'occupent pas de place en mémoire. Comme on l'a vu dans l'adressage immédiat, les constantes sont codées directement dans le code machine des instructions qui les utilisent.

On représente une **variable** par un rectangle, la valeur est indiquée à l'intérieur du rectangle. Le nom de la variable est indiqué à côté de la variable. var1	On peut aussi représenter l'adresse de la variable, à côté de la variable. 2A0 ou 4A2B:02A0 s'il s'agit d'une @ far.

4.8. Sujet du TP2

Voir http://wapiti.enic.fr/commun/ens/peda/modules/A23/Tp2/Tp2.html

4.8.1. Objectifs

Savoir mettre en œuvre les modes d'adressage. Seule la 1ère partie du TP2 est réalisable à l'issue de la séquence 4. Le reste de ce travail sera abordé après la séquence 5.

4.8.2. Mise en route

- Créez vous un répertoire de travail différent de celui où le logiciel Turbo-C est installé.
- Ne travaillez pas directement sur disquette car cela allonge beaucoup les temps d'accès.
- Pour imprimer le résultat de votre travail, n'utilisez pas la commande Print de Turbo-C. Imprimez à partir de Windows.
- Pour sauvegarder votre travail sur une disquette, n'utilisez pas la commande "Fichier/enregistrer sous…" (ou "File / save as…). Utilisez plutôt l'explorateur de fichiers de Windows.
- Téléchargez le fichier Adr.c dans votre répertoire de travail,
- Lancez le Borland-C ou Turbo-C, depuis une boîte DOS. Vous devez vous situer dans un répertoire où vous avez les droits d'écriture, typiquement votre répertoire de travail.

4.8.3. Le fichier Adr.c

```
#define constante 2

typedef struct {
        int champ1;
        char champ2, champ3;
} rec_t;

int variable1, variable2, variable3, *p1, i;
rec_t enreg;
unsigned char tab_byte[9];
rec_t tab_enr[9];

void proc() {
asm {
// AX = 1; (adressage immédiat)
// AX = constante; (adressage immédiat)
// AX = variable1; (adressage direct)
// AX = variable2; (adressage basé sur BX)
```

Travail à réaliser

Codez en langage d'assemblage les instructions spécifiées par les lignes de commentaires de la procédure `proc()`. Vous aurez soin de coder **en dessous** de chaque ligne de commentaire les instructions machine correspondantes.

Pour vérifier le travail, exécutez en pas à pas le programme en visualisant la fenêtre des registres (Windows / Registres). Observez l'effet de chaque instruction en guettant les modifications de registres. Le programme principal initialise les variables de telle sorte que, lors de l'exécution de la procédure `proc()`, le contenu de `AX`, `AH` ou `AL` s'incrémente 1, 2, 3, ... 10

Les valeurs intermédiaires des autres registres que vous utilisez verront également leur valeur se modifier. Interrogez-vous sur la nature de l'information visualisée dans ces registres.

L'utilisation de Turbo-Debugger n'est pas indispensable pour ce TP. On peut donc utiliser l'option Debugger / On.

5. Structuration des données

5.1. Les pointeurs

Les pointeurs sont présents dans les langages évolués. Ce sont des variables qui contiennent l'adresse d'une autre variable. En langage C, l'opérateur "&", préfixant une variable fournit son adresse. Grâce à cet opérateur, on peut initialiser un pointeur avec l'adresse d'une autre variable ou d'une partie d'une variable structurée. On dit alors que le pointeur pointe sur la variable ou que la variable est pointée par le pointeur. D'autre part, on peut initialiser un pointeur avec une adresse fournie comme résultat d'une fonction d'allocation mémoire telle que malloc. Dans ce cas, l'objet pointé n'a pas de nom et n'est accessible que via le pointeur.

En langage C, l'opérateur "*" préfixant un pointeur désigne l'objet pointé par le pointeur. Les pointeurs sont typés et pointent sur un certain type d'objets.

On peut initialiser un pointeur avec la constante NULL équivalente à la valeur 0. Quand un pointeur contient cette valeur particulière, cela ne signifie pas que l'objet pointé se situe à l'adresse 0. Par convention, cela veut dire qu'il n'y a pas d'objet pointé. On ne peut pas appliquer l'opérateur "*" à un pointeur nul. De manière générale, avant d'utiliser un pointeur, il faut s'assurer qu'il contient une valeur non nulle. De plus, pour que ce test ait un sens, avant de l'utiliser un pointeur, il faut toujours l'initialiser avec une valeur nulle ou avec l'adresse d'un objet légalement alloué.

5.1.1. Vocabulaire

Il ne faut pas confondre un pointeur avec une adresse. Un pointeur est une variable tandis qu'une adresse est une valeur. Il convient de distinguer au moins 3 notions différentes :

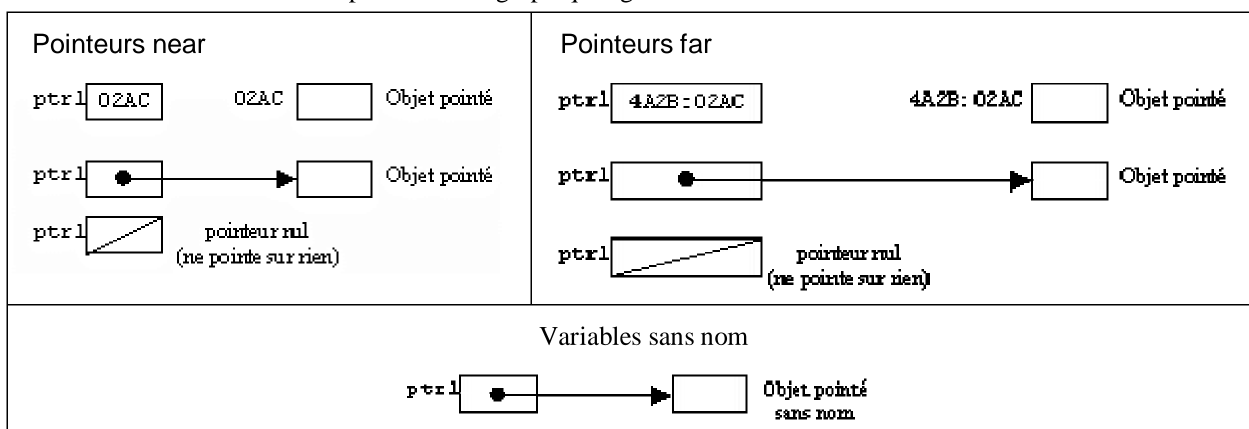
- le pointeur,
- l'adresse qu'il contient,
- l'objet pointé.

Pour parler de l'objet pointé, il ne faut pas dire "le contenu d'un pointeur" car le contenu d'un pointeur est l'adresse de l'objet pointé, pas l'objet pointé lui-même.

Pour parler de l'adresse stockée dans le pointeur, il ne faut pas dire "l'adresse du pointeur" car le pointeur est une variable et possède à ce titre une adresse comme n'importe quelle variable.

Avec l'architecture segmentée du 8086, il faut distinguer les pointeurs near et les pointeurs far. On peut choisir d'utiliser des pointeurs near ou des pointeurs far grâce à des options fournies au compilateur. Une extension au langage C, spécifique au 8086 permet aussi de le faire de façon plus spécifique grâce aux mots-clés near et far, mais le code ainsi obtenu n'est pas portable.

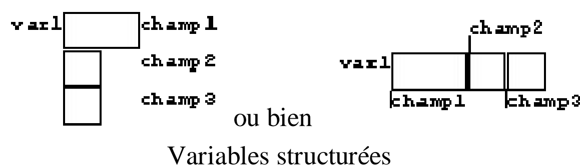
Le schéma suivant fournit les représentations graphiques généralement utilisées.



En langage d'assemblage, pour accéder à l'objet pointé, il faut d'abord copier son adresse dans un registre d'adresse offset au moyen d'un MOV. S'il s'agit d'un pointeur far, il faut charger un couple de registres (registre segment, registre d'adresse offset) au moyen d'un LDS ou d'un LES. On peut ensuite utiliser l'adressage indirect.

5.1.2. Pointeurs sur enregistrements

Un pointeur peut aussi pointer sur un enregistrement (traduction du mot "record" utilisé dans la terminologie du langage Pascal) ou sur une structure selon la terminologie du langage C.



Variables structurées

Dans ce langage, l'opérateur "->" est applicable uniquement aux pointeurs de structures. Il permet d'accéder à un champ de l'objet pointé. La notation p->champ est équivalente à (*p).champ, c'est donc une combinaison des opérateurs "*" et "."

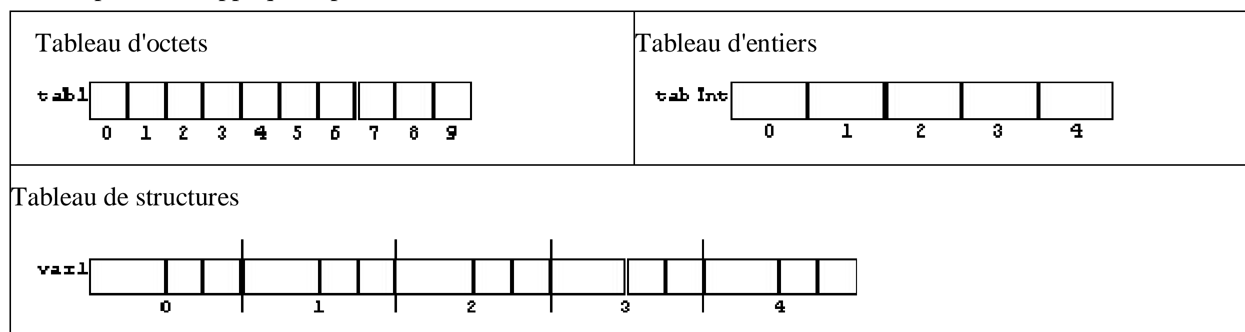
5.1.3. Pointeurs et tableaux

En langage C, pointeurs et tableaux sont des concepts différents bien qu'ayant des points communs. En fait, les opérateurs "*" et "[i]" sont applicables aussi bien aux tableaux qu'aux pointeurs. C'est au programmeur de savoir si cela a un sens ou pas d'utiliser tel ou tel opérateur. Le compilateur ne contrôle rien si ce n'est le type de l'objet pointé.

En fait, l'opérateur "[i]" est applicable à une adresse. **Le nom d'un tableau désigne l'adresse de début de ce tableau**. Si on postfixe le nom d'un tableau par "[i]", on accèdera à la case d'indice i de ce tableau. Les cases d'un tableau sont des zones mémoires successives et contiguës.

Si on postfixe le nom d'un pointeur par "[i]", l'opérateur s'appliquera en fait à l'adresse contenue dans le pointeur. Si cette adresse est celle d'un tableau d'objets pointés et non celle d'un objet pointé unique, on accèdera à la case d'indice i du tableau pointé par le pointeur.

D'autre part, si on applique l'opérateur "*" à un tableau, on accèdera à la case d'indice 0.



5.2. Les enregistrements

On a les déclarations suivantes :

```
// Constantes
#define constante 2

// Types
typedef struct {
        int champ1;
        char champ2, champ3;
} rec_t;

// Variables
int variable1, variable2, i;
rec_t  enreg, *pEnreg;
char tab_byte[9], tab_byte1[9], *pTabByte;
int tab_word[10], *pTabWord;
rec_t tab_enr[9];
```

On considère la représentation suivante. La surface des champs est représentative de l'espace qu'ils occupent en mémoire. Un type `char` est supposé occuper **un octet** en mémoire. Comme ceci dépend du système et du compilateur utilisés, les compilateurs C fournissent l'opérateur `sizeof` qui donne la taille en octets **d'une variable ou d'un type**. Sur la plate-forme utilisée pour les travaux pratique, `sizeof( char )` a la valeur 1.

- Trouvez parmi les déclarations précédentes le nom du type qui correspond à ce schéma.
- En C, quel est l'opérateur d'accès à un champ d'un enregistrement ?
- Quelle place un `int` occupe-t-il en mémoire ?
- Quelle place une variable de ce type occupe-t-elle en mémoire ?
- Parmi les déclarations précédentes, trouvez une variable de ce type.

Les champs 1, 2 et 3 sont stockés en mémoire dans l'ordre de la déclaration. Le champ1 est situé au début de l'enregistrement.

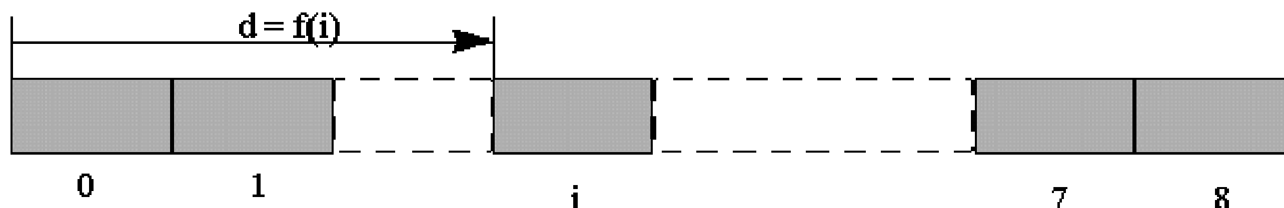
- Calculer la "distance" en octets entre le début de l'enregistrement et le champ2.
- Même question pour le champ3.
- Annotez le dessin ci-dessus avec les "distances" en octet, relatives au début d'enregistrement.

Accès via un pointeur

- Quel opérateur doit-on appliquer à un pointeur d'enregistrement pour accéder à l'un des champs ?
- Recherchez une notation équivalente mettant en œuvre deux opérateurs ?
- Pourquoi les parenthèses sont-elles obligatoires ?
- Quel est l'intérêt de la notation avec un seul opérateur ?
- Que se passe-t-il si on applique l'opérateur -> à un pointeur non initialisé ?

5.3. Tableau d'octets

En langage C, la première "case" d'un tableau est la case d'indice 0. Un tableau de 9 octets est donc indicé de 0 à 8.



- Parmi les déclarations précédentes, trouvez un tableau d'octets.
- En C, comment peut-on obtenir l'adresse d'un tableau ?
- En utilisant l'opérateur &, comment peut-on aussi obtenir cette adresse ?

Certains langages autorisent l'affectation d'un tableau dans un autre tableau. **Ce n'est pas le cas du langage C**.

- L'affectation `tab_byte = tab_byte1` a-t-elle un sens ?
- L'affectation du type `pTabByte = tab_byte` a-t-elle un sens ?
- Pour accéder à la case d'indice `i` du tableau d'octets, qu'a-t-on besoin de connaître ?
- Calculez la distance d en fonction de `i` (`i` étant une variable d'indice).
- L'adresse de début du tableau est-elle connue lors de la compilation ?
- Citez deux instructions permettant de charger l'adresse du tableau dans BX.
- Quel mode d'adressage ces deux instructions mettent-elles en oeuvre ?
- Que fait l'instruction `MOV BX, tab_byte` ?
- Comment accéder à la case d'indice `i` d'un tableau de caractères ?

5.4. Tableaux d'entiers

On suppose que les entiers occupent deux octets en mémoire.

- Calculez la distance d en fonction de `i` (`i` étant une variable d'indice).
- Comment accéder à la case d'indice `i` d'un tableau d'entiers ?

5.5. Tableau d'enregistrements

Les "cases" d'un tableau peuvent aussi être des objets plus complexes que des caractères ou des entiers, par exemple des enregistrements.

- Parmi les déclarations précédentes, trouvez un tableau d'enregistrements.
- Quelle est la taille occupée par ce tableau ?
- Dessinez une représentation graphique d'un tableau d'enregistrements.
- Calculez la distance d en fonction de i.
- Trouvez une formule de calcul de d en fonction de i convenant pour tout type de tableau.
- Dans cette formule, est-ce la valeur ou l'adresse de i qui intervient.
- Comment accéder au champ 2 de la case d'indice i ?

5.6. Tableaux à 2 dimensions

On considère les déclarations suivantes :

```
#define LIGS 10
#define COLS 5
char tab[LIGS][COLS];
```

- Dessinez une représentation graphique d'un tableau à deux dimensions.
- Quelle est la taille d'une "case" ?
- Quelle est la taille d'une ligne ?
- Que désigne l'expression `tab[i][j]` ?
- Cette fois-ci, d désigne la distance entre le début du tableau et la case `tab[i][j]` (`i` et `j` étant des variables). Proposez une formule de calcul de d en fonction de `i` et `j`.

5.7. Tableau de pointeurs

On considère les déclarations suivantes :

```
#define LIGS 10
#define COLS 5
char *tab[LIGS];
```

- Quelle est la taille d'une "case" ?

On considère le code d'initialisation suivant :

```
int i;
for (i=0; i<LIGS; i++)
        tab[i] = malloc( COLS*sizeof(char) );
```

La fonction `malloc` est chargée de réserver auprès du système d'exploitation une zone de mémoire ayant la taille spécifiée en argument (ici `COLS*sizeof( char )` c'est-à-dire 5 octets).

- Selon vous, que renvoie la fonction malloc ?
- Dessinez une représentation graphique d'un tel tableau après allocation.
- Que désigne l'expression `tab[i][j]` ?
- Que remarque-t-on à propos de la syntaxe ?

5.8. Fin du TP2

Implantez en langage d'assemblage les instructions indiquées par les commentaires.

```
// p1 = &variable3; (via SI)
// AX = *p1; (via BX)
// AH = enreg.champ2; (adressage basé sur BX avec déplacement)
// AX = i; (adressage indexé par SI)
// AL = enreg.champ3; (adressage indexé par SI avec déplacement)
// AH = tab_byte[ i ]; (adressage basé sur BX et indexé par SI)
// AL = tab_enr[ 4 ].champ3; (adressage basé, indexé avec déplacement)
// AH = tab_enr[ i ].champ3; (adressage basé, indexé avec déplacement)
}
}

int main() {

        variable1 = 3;
        variable2 = 4;
        variable3 = 5
        enreg.champ2 = 6;
        i = 7;
        enreg.champ3 = 8;
        tab_byte[ i ] = 9;
        tab_enr[ 4 ].champ3 = 10;
        tab_enr[ i ].champ3 = 11;
        proc();

return 0;
}
```

6. Structuration du code

6.1. Les instructions de saut

Les microprocesseurs ne sont pas seulement des machines capables d'exécuter des instructions les unes après les autres. Ils sont aussi capables de prendre des décisions sur la suite à donner au programme selon la situation qu'ils rencontrent. Autrement dit, en analysant les données qu'ils manipulent, ils décident d'emprunter des chemins différents tracés par le programmeur. Cela s'appelle le contrôle de l'exécution. Les langages évolués possèdent des structures de contrôle telles que le **si**... **alors**..., le **si**... **alors**... **sinon**..., le **tant** que..., le **jusque**..., le **pour**... et le **cas**...

L'assembleur n'étant pas un langage structuré, le contrôle de l'exécution du programme s'effectue grâce à des instructions d'appel de sous-programme et des instructions de saut. On peut avec les instructions de saut (conditionnels et inconditionnels) implanter les structures de contrôle des langages évolués. Une instruction analyse dans un premier temps les données et positionne le **mot d'état** du processeur encore appelé **Flags** ou **registre des indicateurs**. dans un second temps, une instruction de **saut conditionnel** exploite la valeur de ces indicateurs pour choisir entre deux chemins d'exécution : soit l'exécution se poursuit en séquence, c'est-à-dire avec l'instruction suivante dans l'ordre de la mémoire, soit l'exécution se poursuit à une adresse du segment de code spécifiée par l'instruction de saut conditionnel. Dans un assembleur, cette adresse est généralement repérée par une **étiquette**, c'est-à-dire un nom symbolique plus facile à mémoriser qu'un nombre. Le qualificatif "conditionnel" provient de ce que l'*exécution* du saut est *conditionnée par* la valeur de l'*indicateur*.

Il existe différents moyens de positionner les indicateurs en vue d'effectuer un saut conditionnel. La plupart des instructions arithmétiques et logiques les positionnent.

- AND : et logique,
- OR : ou logique,
- XOR : ou exclusif,
- NOT : négation,
- TEST : test de bits. C'est un "et" logique dont le résultat n'est pas stocké et qui ne sert qu'à positionner les indicateurs.

Certains indicateurs peuvent être positionnés par programme. Il est en effet parfois nécessaire de s'assurer de l'état de ces indicateurs avant de commencer un traitement qui les exploite. C'est notamment le cas de l'indicateur de retenue "carry flag" ou CF :

- CLC : remettre CF à 0,
- CMC : complémenter CF,
- STC : mettre CF à 1.

L'instruction CMP compare la valeur de deux opérandes dans la relation d'ordre des entiers. Il s'agit en fait d'une soustraction dont le résultat n'est pas stocké dans un registre et qui ne sert qu'à positionner les indicateurs. Elle est particulièrement intéressante car les mnémoniques des sauts conditionnels ont été choisis en fonction d'elle.

```
                CMP   AL,1
                JG    la_bas      // Jump to la_bas if al Greater than 1
                <instruction en séquence>
                ...
    }
    la_bas:     asm {
                ...
```

Dans la syntaxe de l'assembleur in-line de Turbo C, les étiquettes sont définies en dehors des blocs asm. Ce sont des étiquettes au sens du langage C (qui dispose de l'instruction goto etiq) qu'on peut utiliser dans des instructions de saut.

6.1.1. Les sauts conditionnels

Les sauts conditionnels ont lieu dans le segment de code et sont relatifs à la valeur courante du pointeur d'instructions. Un déplacement signé, codé sur 8 bits dans l'instruction, est rajouté à la valeur courante du registre IP. Cela signifie qu'il existe une **distance maximale de saut** en amont ou en aval dans le segment de code. Les étiquettes de saut doivent se trouver au maximum 126 octets avant ou 127 octets après l'instruction. Cette distance est de toute manière contrôlée par l'assembleur.

Les sauts du tableau suivant sont utilisables soit pour comparer deux nombres signés ou non signés et éventuellement leur égalité. Ils trouvent leur signification pour le couple d'instructions :

```
CMP   opg,opd
JXX   etiq
```

"JXX" est un générique pour les mnémoniques du tableau suivant. Par exemple, pour "JGE", ces instructions signifient ensemble "**Jump to etiq if opg Greater or Equal to opd**", c'est-à-dire "**sauter en etiq si opg >=opd**"

JE	Jump if Equal	JNE	Jump if Not Equal
JGE	Gump if Greater or Equal	JG	Jump if Greater
JLE	Jump if Lower or Equal	JL	Jump if Lower
JAE	Jump if Above or Equal	JA	Jump if Above
JBE	Jump if Below or Equal	JB	Jump if Below

Liste non exhaustive d'instructions de sauts conditionnels.

6.1.2. Sauts inconditionnels intra-segment

Les sauts inconditionnels sont des sauts qui ont toujours lieu. Par exemple à la fin de la branche "alors" d'un "si ... alors ... sinon ... ", il faut sauter au-dessus de la branche "sinon" pour éviter le comportement anormal qui consisterait à enchaîner la branche "alors" et la branche "sinon". De la même façon, le corps d'une boucle "tant que" se termine par un saut inconditionnel vers la portion de code qui évalue la condition de la boucle.

Comme un saut inconditionnel a toujours lieu, on peut penser que le code qui le suit ne sera jamais exécuté. Cela peut arriver si le programmeur n'y prend pas garde : cela s'appelle du code mort. Pour que le code qui suit un saut ne soit pas mort, il faut qu'il soit accessible donc repéré par une étiquette référencée par un autre saut.

```
            ...
            JMP   suite
}
sinon:      asm {
            <code pas mort>
```

Si l'adresse à laquelle on saute est située dans les 64 ko du segment de code courant, le saut peut être intra-segment, c'est-à-dire que l'adresse spécifiée dans le code machine est une adresse offset. Le code machine est donc plus compact. Au moment de l'exécution cette adresse offset est chargée dans le registre IP de telle sorte que l'instruction suivante sera recherchée à l'adresse de saut.

Si l'adresse de saut est située en dehors du segment de code courant, il faut impérativement utiliser un saut inter-segment.

6.1.3. Sauts inconditionnels inter-segment

Le code machine d'un saut inter-segment comporte une adresse segmentée codée sur 32 bits. Le code machine est donc plus long. Au moment de l'exécution, le saut inter-segment charge l'adresse de saut dans le couple CS:IP, ce qui change la position courante du segment de code.

Le tableau suivant présente les instructions de saut et leur équivalent avec des notations inspirées du langage C.

jmp etiq	goto etiq; *ou* **cs**:**ip**=etiq
cmp g,d **jge** etiq (voir sauts conditionnels)	if (g>=d) goto etiq

6.2. Les structures de contrôle

6.2.1. Le "si … alors …"

Voici un exemple en langage C de "si … alors …" :

```
int a, min;
...
if (a < min) {
      min = a;
}
...
```

L'instruction `min=a` est effectuée si `a` est inférieur à `min`. En langage d'assemblage, on utilise un saut conditionnel qui enjambe cette instruction si la condition inverse est vraie, c'est-à-dire si `a` est supérieure ou égal à `min`. Il faut une étiquette pour repérer les instructions situées après le "if".

6.2.2. Le "si … alors … sinon …"

Dans ce "si … alors … sinon …" l'évaluation de la condition `a<b` correspond à un `CMP` suivi d'un saut conditionnel. Si `a>=b`, le saut conditionnel fait sauter vers la branche `min=b`, repérée par une étiquette. Dans le cas contraire (`a<b`), le saut conditionnel n'a pas lieu, l'exécution continue en séquence et l'affectation `min = a` a lieu.

A la fin de la branche `min=a`, un saut inconditionnel saute au-dessus de la branche `min=b`, pour éviter d'enchaîner `min=a` et `min=b`. Une étiquette repère `<suite>` point de regroupement des deux branches.

```
int a, b, min;
...
if (a < b) {
      min = a;

}
else {
      min = b;
}
<suite>...
```

6.2.3. La structure de "cas"

La structure de cas envisage plusieurs cas pour la valeur de l'expression `etat`. Si elle vaut `0`, la branche `etat=2` est exécutée. Si elle vaut `1`, la branche `etat=3` est exécutée. Si elle vaut `2`, la branche `etat=0` est exécutée. Sinon, la branche `etat=1` est exécutée.

La valeur de l'expression `etat` est d'abord mise dans un registre. Pour chaque cas envisagé, on a une instruction CMP et un saut conditionnel. Le registre est comparé avec la valeur `0`. En cas d'inégalité, le saut conditionnel a lieu vers la comparaison avec la valeur suivante : `1` puis `2` éventuellement. Si aucune égalité n'est trouvée, le dernier saut conditionnel fait sauter vers la branche par défaut `etat=1`, repérée par une étiquette. Si une égalité est trouvée, le saut conditionnel, n'a pas lieu et la branche correspondante a lieu par exemple `etat=2` pour égalité avec `0`. L'instruction `break` correspond à un saut inconditionnel vers `<suite>` repéré par une étiquette.

```
switch (etat) {
case 0 :
      etat = 2;
      break;
case 1 :
      etat = 3;
      break;
case 2 :
      etat = 0;
      break;
default :
      etat = 1;
}
<suite>...
```

6.2.4. La boucle "tant que …"

Cette boucle "tant que..." initialise un tableau de `10` entiers `tab` avec la valeur `0`. La condition `i<10` correspond à un `CMP` suivi d'un saut conditionnel. Si `i>10`, le saut conditionnel fait sauter vers la `<suite>` de la boucle, repérée par une étiquette. Sinon (`i<10`) le saut conditionnel n'a pas lieu, l'exécution continue en séquence et le corps de boucle est exécuté : `tab[i]=0; i++;` un saut inconditionnel le termine et fait sauter vers l'évaluation de la condition (`CMP` + saut conditionnel) repérée par une étiquette.

```
int i;
int tab[10];

i = 0;
while (i < 10) {
      tab[i]=0;
      i++;
}
<suite>...
```

6.2.5. La boucle "pour …"

La boucle "pour" est une boucle "tant que" où l'initialisation (`i=0`) et l'itération (`i++`) sont intégrées. Son implantation est identique à la boucle tant que précédente.

```
int i;
int tab[10];

for (i=0; i<10; i++) {
      tab[i]=0;
}
```

6.2.6. La boucle "faire … tant que …"

La boucle "faire … tant que …" est une boucle qui s'exécute au moins une fois car la condition de sortie de boucle est située à la fin de la boucle. Le début du corps de boucle est repéré par une étiquette. Le corps de boucle se termine par l'itération. Puis, la condition est évaluée : `i<10` correspond à un `CMP` suivi d'un saut conditionnel. Si `i<10`, le saut conditionnel fait sauter vers le début du corps de boucle. Sinon, l'exécution se poursuit en séquence.

```
int i;
int tab[10];

i=0;
do {
      tab[i]=0;
      i++;
} while (i<10);
```

6.3. Sujet du TP3

6.3.1. Mise en route

- Lisez les recommandations générales,
- Téléchargez le fichier aff.c qui illustre l'affichage d'un caractère à l'écran au moyen de l'INT 21 du DOS.
- Compilez le et exécutez-le en pas à pas sous Borland C (visualisez en même temps la fenêtre des registres : menu Window / Registers)

Pour **afficher un caractère à l'écran**, on utilise la fonction 2 de l'interruption 21 hexa fournie par le DOS. Il faut mettre 2 dans `AH`, le code ascii dans `DL` et exécuter `INT 0x21`.

- Téléchargez le fichier HEX.C dans votre répertoire de travail,
- Lancez Turbo-C.

6.3.2. Le fichier Hex.c

Méthode : codez **sous** chaque ligne de commentaire les quelques instructions assembleur correspondantes.

```
#include <stdio.h>

void aff_hex( unsigned char car ) {
asm {
// -- affichage poids forts
// si <= 9
//   ajouter code ascii de '0'
// sinon -- si > 9
//   retirer 10 et ajouter le code ascii de 'A'
// fsi
// afficher
// -- afficher poids faibles
// si <= 9
//   ajouter code ascii de '0'
// sinon -- si > 9
//   retirer 10 et ajouter le code ascii de 'A'
// fsi
// afficher
}
}

int i, lignes;
```

```
int main() {
  clrscr();
  lignes = 0;
  for (i=0; i<=255; i++) {
        aff_hex( i );
        lignes++;
        if (lignes == 25) {
          getch();
          lignes=0;
        }
        printf( "\n" );;
  }

  return 0;
}
```

6.3.3. Indications

En général, ce sujet trouble un peu les étudiants. En effet, la plupart du temps, on se contente d'utiliser printf pour afficher un nombre. Cela paraît tellement naturel, qu'on ne se rend pas compte qu'un traitement est nécessaire pour passer du nombre lui-même tel qu'il est stocké dans la variable à une représentation affichable à l'écran, dans une base quelconque. Cela me fait penser au tableau de Magritte *"Ceci n'est pas une pipe"* qui représente bien sûr une pipe mais n'en est pas une. La représentation d'un nombre à l'écran est une chaîne de caractères, mais ce n'est pas le nombre lui-même.

Pour afficher un nombre en hexadécimal, il faut afficher deux caractères. Le premier représente le quartet de poids fort et le second, le quartet de poids faible. Le code ascii du 0 est 48 décimal (ou 30 hexa). Le code ascii du A majuscule est 65 décimal (ou 41 hexa).

Pour caler le quartet de poids fort en poids faible

4 décalages à droite (instructions SHR AL,1). Dans l'exemple 4 décalages sur 5A donnent 05.

Pour isoler le quartet de poids faible

Faire un et "bit à bit" avec la constante hexa 0F. Par exemple un "et" entre 5A et 0F donne 0A.

Pour obtenir le code ascii

Il faut envisager deux cas :

Si le quartet est un chiffre compris entre 0 et 9 bornes incluses, il suffit d'ajouter 30 (hexa)

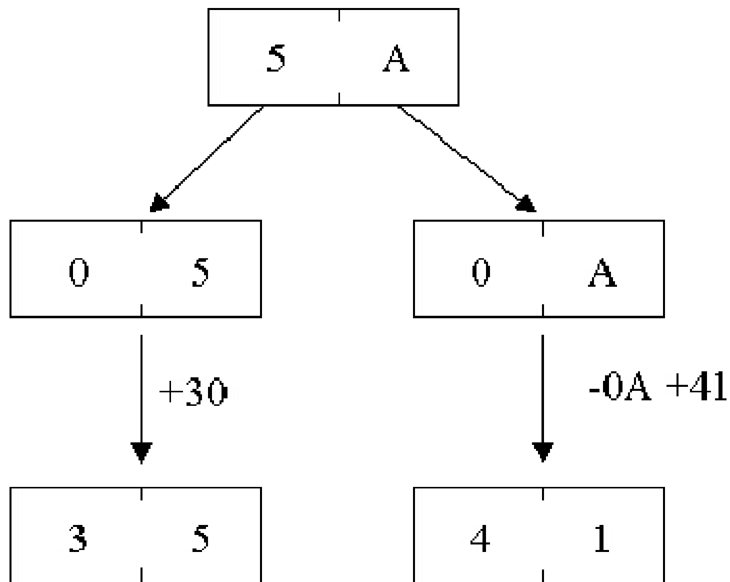
caractère	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
code ascii	30	31	32	33	34	35	36	37	38	39

Si le quartet est un chiffre compris entre A et F bornes incluses, il suffit d'ajouter (41 - 0A) hexa, 41 hexa étant le code ascii du caractère 'A'.

caractère	'A'	'B'	'C'	'D'	'E'	'F'
code ascii	41	42	43	44	45	46

On peut résumer l'algorithme par le schéma suivant :

Exemple avec car = 5A



7. Structuration du code : les sous-programmes

7.1. Fonctionnement d'une pile

La pile est une zone mémoire indispensable au fonctionnement du microprocesseur. Elle permet l'implantation de la notion de sous-programme qui correspond aux procédures et fonctions des langages évolués tels que le C.

D'un point de vue conceptuel, une pile est une **mémoire** gérée selon le principe "dernier entré - premier sorti". On parle en anglais de fonctionnement LIFO (**Last In - First Out**). Le **sommet** désigne le dernier entré qui est également le premier à sortir.

On peut implanter une pile dans la mémoire d'un ordinateur grâce au registre **pointeur de pile** (Stack Pointer). Une zone étant attribuée en mémoire à cet effet, le pointeur de pile est un registre dédié qui **repère** en permanence **le sommet** de la pile dans cette zone mémoire.

Pour le 8086, la pile est implantée à la fin du segment de pile. Le sommet de pile est le dernier élément empilé. Le sommet de pile et tous les éléments situés dessous (à des adresses supérieures) sont considérés comme faisant partie de la pile. Les éléments situés au-dessus bien que présents dans le segment de pile, ne sont pas présent dans la pile.

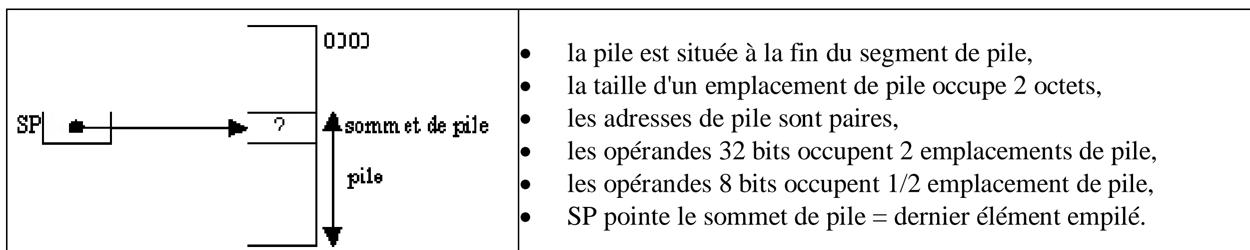
L'entrée d'un élément dans la pile porte le nom d'**empilement** et est représentée au niveau du microprocesseur par l'instruction "PUSH". Lors de l'exécution de l'instruction "PUSH", le pointeur de pile se trouve automatiquement mis à jour. La sortie d'un élément dans la pile (et sa récupération dans un registre) porte le nom de **dépilement** et est servie par l'instruction "POP". Lors de l'exécution de l'instruction "POP", le pointeur de pile se trouve automatiquement mis à jour.

push oper16b **push** Flags	*(--**sp**)=oper16b; *(--**sp**)=Flags;
pop oper16b	oper16b=*(**sp**++)

On peut empiler un registre ou un mot contenu en mémoire.

7.2. Représentation de la pile

On la représente verticalement :



- la pile est située à la fin du segment de pile,
- la taille d'un emplacement de pile occupe 2 octets,
- les adresses de pile sont paires,
- les opérandes 32 bits occupent 2 emplacements de pile,
- les opérandes 8 bits occupent 1/2 emplacement de pile,
- SP pointe le sommet de pile = dernier élément empilé.

En revanche, dans Turbo-Debugger, la pile est représentée à l'envers : quand on empile, le pointeur de pile se déplace vers le bas.

7.3. L'appel de sous-programme

Les appels de sous-programmes utilisent la pile. Dans les langages évolués, un sous-programme est appelé procédure ou fonction.

Une définition de sous-programme est une séquence d'instructions qu'on peut exécuter aux endroits où on en a besoin sans avoir à la répéter. L'**adresse du sous-programme** est en fait l'adresse de la 1ère instruction. Comme pour les étiquettes de saut, les assembleurs et compilateurs définissent un nom à l'adresse du sous-programme.

A chaque endroit où on en a besoin, une instruction du microprocesseur (CALL) fait référence au sous-programme par son adresse. Lors de l'exécution d'un CALL, un appel de sous-programme se produit, ce qui utilise la pile :

- Le processeur empile l'adresse de l'instruction qu'il faudra exécuter après l'exécution du sous-programme : l'**adresse de retour**. Une fois que le code machine du CALL a été lu, l'adresse de retour est celle normalement située dans le registre pointeur d'instructions (IP ou CS:IP).
- L'opérande de l'instruction CALL fait référence à l'adresse du sous-programme à exécuter. Cette adresse est chargée dans le registre **pointeur d'instructions**, comme lors d'un saut.
- L'exécution du sous-programme peut alors commencer.

Les possibilités sont les mêmes que pour le saut inconditionnel. Les appels peuvent être intra-segment (CALL near) ou inter-segment (CALL far). L'appel intra-segment manipule des adresses offsets pour l'adresse de sous-programme et l'adresse de retour. L'adresse de retour occupe alors un seul emplacement sur la pile soit deux octets. Inversement, l'appel inter-segment met en œuvre une adresse de sous-programme segmentée et une adresse de retour segmentée. L'adresse de retour occupe alors deux emplacements de pile soit quatre octets. Comme les sauts inter-segments, les appels inter-segments modifient le registre CS et donc la position du segment de code.

Certains assembleurs utilisent un mnémonique différent CALLF pour le CALL far.

call sprog	sprog(); ou *(--**sp**)=**ip**;
callf sprogf	sprogf(); ou *(--**sp**)=**cs**; *(--**sp**)=**ip**;

7.4. Retour de sous-programme

L'exécution du sous-programme se poursuit jusqu'à rencontrer une instruction de **retour de sous-programme** (RET)

- L'adresse de retour est **dépilée** et récupérée **dans** le registre **pointeur d'instructions** (IP ou CS:IP)
- l'exécution se poursuit donc avec l'instruction située en mémoire à la suite de l'instruction CALL.

Il existe deux types d'instructions RET :

- un retour qui restitue dans IP l'adresse de retour depuis le sommet de pile, (RET near ou intra-segment) ; cette instruction dépile un emplacement de pile soit deux octets,
- un retour qui restitue dans CS:IP l'adresse de retour depuis le sommet de complète (RET far ou inter-segment) ; cette instruction dépile deux emplacements de pile soit quatre octets. Certains assembleurs utilisent un mnémonique différent RETF pour le RET far.

Lorsqu'un sous-programme utilise un RET far, il doit être appelé au moyen d'un CALL far. Ce type de contrôle est réalisé par les assembleurs et les compilateurs.

ret	(**ip**)=*(**sp++**);
retf	(**ip**)=*(**sp++**); (**cs**)=*(**sp++**);

7.5. Imbrication des sous-programmes

Les **appels** de sous-programmes peuvent être **imbriqués**, c'est-à-dire qu'un sous-programme peut, dans sa définition, faire appel lui-même à un sous-programme. Un sous-programme peut s'appeler lui-même, moyennant certaines précautions de conception. Ce type d'appel est dit récursif. Toutes les adresses de retour sont empilées puis dépilées selon la **stratégie LIFO**.

7.6. Fonctions

Dans le cadre des langages évolués, une fonction est un sous-programme qui calcule et retourne un résultat au sous-programme appelant. Les sous-programmes appelant et appelé se conforment à une même convention de stockage pour le résultat. C'est en général un registre du processeur qui est utilisé. Lorsque la taille du résultat est supérieure à celle des opérandes couramment manipulés par le processeur, c'est un couple de registres qui est utilisé.

8. Passage des arguments

8.1. Le passage des arguments en langage C

Nous allons présenter ici le mécanisme que le langage C utilise pour passer les arguments aux sous-programmes. Rappelons-nous que le programme appelant fournit les paramètres effectifs (ou arguments) et qu'ils sont connus dans le programme appelé en tant que paramètres formels (ou tout simplement paramètres).

Ce programme montre l'appel à une fonction `rechercher` qui recherche un entier dans un tableau d'entiers et retourne l'indice de l'entier dans le tableau ou -1 si l'entier n'est pas trouvé. Le programme principal joue le rôle de l'appelant : il fournit les arguments et affiche le résultat de la fonction.

Les paramètres sont l'entier recherché `e` et un pointeur sur le tableau d'entiers `tab`.

```
#include <stdio.h>
#define N 10

int tabG[N] = { 10, 3, 7, 5, 4, 2, 7, 4, 5, 15 };

int rechercher( int e, int tab[N] ) {
        int i, fin;

        fin = 0;
        i = 0;
        while (i<N && !fin)
                if (e == tab[i])
                        fin = !fin;
                else
                        i++;

        return i;
}

main() {
        printf( "%d", rechercher( 5, tabG ) );

        return 0;
}
```

8.2. Passage des arguments par la pile

C'est généralement la pile qui est utilisée pour passer les arguments aux sous-programmes.

8.2.1. Convention utilisée par le langage C

Contrairement au langage Pascal, le langage C empile en premier le dernier paramètre de la liste des paramètres formels. Par conséquent, le premier de la liste est le dernier empilé et se trouve au sommet de la pile.

8.2.2. Passage par valeur ou par adresse

Contrairement à d'autres langages, le passage des arguments aux sous-programmes s'effectue toujours **par valeur** en langage C. Cependant, par analogie avec d'autres langages, on distinguera *passage par valeur* et *passage par adresse*, ce qui n'est pas contradictoire puisqu'une adresse est une valeur un peu particulière.

8.2.3. Passage de tableaux en arguments

Le passage d'arguments par valeur a une **sémantique de duplication**. Lorsque l'objet a une certaine taille, cela peut être pénalisant. C'est pourquoi le C **n'autorise pas** le passage des tableaux en arguments ; c'est en fait **leur adresse** qui est confiée au programme appelé.

Le programme appelé manipule alors le tableau via un pointeur et la syntaxe utilisée avec ce pointeur ressemble fort à celle utilisée pour un tableau. Si cela est commode, c'est également source de confusion.

Rappelez-vous : un paramètre d'une fonction, même s'il ressemble à une déclaration de tableau, est en fait une déclaration de pointeur.

Le panneau de données de Turbo-Debugger permet de voir le tableau tabG à l'adresse `00AA`.

```
ds:00A0 3A 79 00 00 3A 79 00 00 :y :y
ds:00A8 FF 9F 0A 00 03 00 07 00 ƒ
ds:00B0 05 00 04 00 02 00 07 00
ds:00B8 04 00 05 00 0F 00 25 64     ¤ %d
ds:00C0 00 00 00 00 CA 14 E4 67 -¶õg
ds:00C8 F5 02 E4 67 F5 02 E4 67 § õg§ õg
```

Attention, la représentation par octets d'entiers 16 bits met en évidence l'ordre de stockage little-endian.

8.3. Point de vue de l'appelant

```
#RECHLIN#22: printf( "%d", rechercher( 5, tabg ) );
 cs:0058 B8AA00 mov ax,00AA        // adresse de tabG
 cs:005B 50 push ax
 cs:005C B80500 mov ax,0005        // valeur entière 5
 cs:005F 50 push ax
 cs:0060 0E push cs                // nécessaire ici car call near
                                   // mais ret far
 cs:0061 E8A9FF call _rechercher   // exécution de la fonction
 cs:0064 59 pop cx                 // nettoyage pile
 cs:0065 59 pop cx                 // cx inexploité
                                   // le résultat est dans ax
```

8.4. Point de vue de l'appelé

On se place d'un point de vue du programme appelé. On suppose que l'adresse de `tabG` et la valeur à rechercher `5` ont été empilées, que l'instruction `call _rechercher` vient de s'exécuter et qu'on est sur le point d'exécuter la 1ère instruction de la fonction rechercher.

8.4.1. Prologue de la fonction

```
_rechercher: int rechercher( int e, int tab[N] ) {
 cs:000D 55 push bp
 cs:000E 8BEC mov bp,sp
 cs:0010 83EC04 sub sp,0004
```

L'instruction push bp sauvegarde bp qui pointe sur le contexte de l'appelant. L'instruction mov bp,sp fait pointer bp au même endroit que sp, c'est-à-dire sur la sauvegarde de bp. L'instruction sub sp,4 réserve (sans initialiser) 4 octets pour les variables locales i et fin. La valeur 4 correspond à la taille occupée par l'ensemble des variables locales du sous-programme. Le contexte du sous-programme est maintenant établi.

8.4.2. Accès aux variables locales

Nous allons voir comment il faut accéder aux variables locales. Les variables locales n'existent que pendant l'exécution de la procédure. Elles sont allouées sur la pile au moyen de l'instruction `sub sp,4` où `4` est la taille en octets qu'occupe l'ensemble des variables locales. Le pointeur de pile `sp` subit le même mouvement que lors d'un empilement; en revanche, le registre `bp` n'est pas modifié.

Les variables locales sont accessibles en utilisant l'adressage basé sur `bp` avec un déplacement (négatif) différent pour chaque variable locale.

```
#RECHLIN#10: fin = 0;
 cs:0013 C746FC0000 mov word ptr [bp-04],0000
#RECHLIN#11: i = 0;
 cs:0018 C746FE0000 mov word ptr [bp-02],0000
```

8.4.3. Accès aux paramètres

Les paramètres sont accessibles en utilisant l'adressage basé sur `bp` avec un déplacement positif. Remarquons cependant qu'on n'utilise pas la pile en tant que pile, en empilant ou en dépilant.

Le compilateur se base uniquement sur la déclaration des paramètres formels pour déterminer les déplacements et la taille de la mémoire à réserver pour chaque paramètre. Il doit aussi savoir s'il s'agit d'un sous-programme near ou d'un sous-programme far. On accède aux paramètres de **la même façon** quelle que soit la valeur des paramètres effectifs et quel que soit le point d'appel de la procédure. C'est cette indépendance vis-à-vis du contexte du programme appelant qui vaut aux paramètres le qualificatif de "**formels**".

```
#RECHLIN#13: if (e == tab[i])
 cs:001F 8B46FE mov ax,[bp-02]      // [bp-2] : i
 cs:0022 D1E0 shl ax,1
 cs:0024 8B5E08 mov bx,[bp+08]      // [bp+8] : tab
 cs:0027 03D8 add bx,ax
 cs:0029 8B07 mov ax,[bx]
 cs:002B 3B4606 cmp ax,[bp+06]      // [bp+6] : e
 cs:002E 750D jne #RECHLIN#16 (003D)
...
```

8.4.4. Retour du résultat de la fonction

Par convention, le résultat de la fonction est mis dans le registre ax avant de revenir au sous-programme appelant.

```
#RECHLIN#18: return i;
 cs:004C 8B46FE mov ax,[bp-02]
 cs:004F EB00 jmp #RECHLIN#19 (0051)    // saut vers fin de fonction
```

8.4.5. Epilogue de la fonction

L'épilogue a une fonction inverse de celle du prologue. L'instruction `mov sp,bp` supprime de la pile l'espace réservé pour les variables locales. L'instruction `pop bp` refait pointer `bp` sur le contexte de l'appelant. L'instruction `retf` fait poursuivre l'exécution à l'adresse de retour.

```
#RECHLIN#19: }
 cs:0051 8BE5 mov sp,bp
 cs:0053 5D pop bp
 cs:0054 CB retf
```

8.5. Exploitation du résultat de la fonction

Une fois le prologue passé, on se retrouve dans le **contexte** de l'appelé. Les deux `pop cx` déjà examinés nettoient la pile des paramètres qui avaient été empilés avant l'exécution du `call`. La fonction `printf` peut ensuite afficher le résultat de la fonction qui, comme nous l'avons vu, est dans le registre `ax`.

```
#RECHLIN#22: printf( "%d", rechercher( 5, tabg ) );
```

```
...
 cs:0066 50 push ax            // le résultat de la recherche est dans AX
 cs:0067 B8BE00 mov ax,00BE
 cs:006A 50 push ax
 cs:006B 9A8A0FE467 call far _printf
 cs:0070 59 pop cx
 cs:0071 59 pop cx
```

8.6. Récursivité

Une procédure récursive est une procédure qui s'appelle elle-même (avec un contrôle programmé de la profondeur d'appel). Elle joue donc à la fois le rôle de l'appelant et de l'appelé. Un contexte est empilé pour chaque niveau de récursivité. On comprend ainsi qu'il existe un exemplaire des paramètres et des variables locales pour chaque niveau de récursivité. Les solutions récursives saturent la pile d'autant plus vite que le la taille occupée par les variables locales et les paramètres est importante. On comprend mieux ainsi le caractère contraignant de la limitation à 64 ko des segments du 8086.

8.7. Paramètres en nombre variable

Ceci autorise le passage de **paramètres en nombre variable** comme pour la procédure `printf` par exemple :

```
printf( const char *format, ... );
```

Le premier paramètre est une chaîne de format. C'est l'adresse de début de cette chaîne qui se retrouve au sommet de la pile. Les autres paramètres s'ils existent sont dans la pile sous la chaîne de format. Si cette chaîne de format est obligatoire, le prototype de `printf` précise par les points de suspension que les paramètres suivants sont optionnels et en nombre variable. Leur type n'est d'ailleurs pas indiqué.

La procédure `printf` ne connaît rien du programme appelant qui l'utilise. Tout ce qu'elle sait, c'est qu'au sommet de la pile se trouve l'adresse de la chaîne de format. Elle ne connaît ni le nombre ni le type des paramètres suivants. En fait c'est en analysant la chaîne de format que la procédure `printf` peut connaître le nombre et le type des paramètres qui suivent dans la pile. Si la chaîne de format ne se trouvait pas au sommet de la pile, rien ne serait possible. De façon générale, il faut que les paramètres obligatoires soient au sommet de la pile et que les paramètres optionnels soient dessous dans la pile. De plus, les paramètres obligatoires doivent fournir par leur valeur un moyen de connaître le nombre de paramètres exact au moment de l'exécution.

9. Gestion des entrées-sorties

9.1. Introduction

Les événements liés aux entrées-sorties posent des problèmes spécifiques au microprocesseur car ils interviennent de manière imprévisible (asynchrone) par rapport au déroulement de l'exécution du programme. Le fait qu'une donnée devienne disponible dans un registre d'un contrôleur d'entrée-sortie est un événement. La fin d'une sous-traitance confiée à un contrôleur d'entrée-sortie est aussi un événement. La détection d'une circonstance particulière par un contrôleur d'entrée-sortie est aussi un événement.

Les fabricants de microprocesseurs ont mis en place deux techniques permettant de prendre en compte cet asynchronisme : les interruptions et le DMA.

9.2. Le polling

La technique de "polling" est la plus basique, et c'est elle qui pose problème si le microprocesseur a d'autres occupations que les entrées-sorties. Le problème est particulièrement crucial pour des événements se produisant de manière sporadique mais nécessitant une réaction rapide par le processeur sous peine de perte de données. Dans un système temps-réel critique comme un ABS d'automobile, une réaction trop lente peut même entraîner la perte de vies humaines.

Dans le mode de fonctionnement par polling, le microprocesseur doit lire un registre d'entrée-sortie pour savoir si un événement s'est produit. Si l'événement est critique et se produit rarement, le microprocesseur doit dépenser l'essentiel de sa puissance à guetter l'événement. Ce type de fonctionnement n'est envisageable que si le processeur n'a pas d'autres traitements à réaliser.

9.3. Le mécanisme d'interruption

Ce mécanisme autorise un dispositif d'entrées-sorties à se signaler auprès du processeur grâce à un signal spécifique propagé jusqu'au processeur. C'est donc le dispositif qui a l'initiative du déclenchement de ce signal. Pour le processeur, cela produit l'effet d'une sonnerie de téléphone : il a la possibilité de répondre ou de ne pas répondre à cette sollicitation, ce qui, dans notre analogie, correspond à décrocher ou ne pas décrocher. Si le processeur répond à la demande d'interruption, il le fait en laissant tomber provisoirement le travail qu'il était entrain d'exécuter, en allant exécuter un programme spécifique de traitement de la demande d'interruption, puis, la demande étant traitée, en reprenant le travail laissé en suspens.

Le mécanisme d'interruption modifie le séquencement des instructions tel qu'il a été présenté précédemment. On obtient la variante suivante :

- lecture de l'instruction,
- décodage de l'instruction,
- exécution de l'instruction identifiée,
- examen de la présence éventuelle d'une demande d'interruption.
- D'éventuels signaux d'**acquittement d'interruption** signalent au dispositif d'entrées-sorties que la demande d'interruption va être prise en compte.

9.3.1. La logique d'interruption du processeur

Deux cas peuvent alors se présenter :

- si aucune demande d'interruption n'est présente, le cycle recommence avec l'instruction suivante dans la mémoire.
- si une interruption est présente et que son traitement est autorisé à ce moment là, alors un cycle de fonctionnement spécial se déroule pour détourner l'exécution des instructions vers un programme spécial de traitement de l'interruption. Une fois traitée, une instruction permet de reprendre le fil de l'exécution là où on l'avait laissé.

Cela suppose, au niveau du processeur, l'existence de broches supplémentaires :

- Une ou plusieurs entrées dites de **demande d'interruption** permettent à des événements externes (issus de dispositifs d'entrées-sorties) de se signaler auprès du microprocesseur dans le but d'interrompre, selon certaines modalités, le déroulement de l'exécution actuelle du programme.

Une **logique interne** du microprocesseur autorise la **gestion** des signaux liés aux interruptions. Les *signaux* de demande d'interruption sont éventuellement *mémorisés* dans des bascules qui sont examinées à la fin de chaque cycle de fonctionnement du microprocesseur. La logique détermine si l'interruption peut être traitée tout de suite (*interruption autorisée*) ou si son traitement doit être différé (interruption *interdite*). Les interruptions peuvent être autorisées ou interdites par programme, soit globalement soit individuellement.

9.3.2. Traitement d'une interruption

Le **cycle d'acquittement d'interruption** qui détourne l'exécution du programme normal au profit du programme de traitement de l'interruption prend en compte les actions suivantes :

- *sauvegarde du contexte* de l'exécution du programme normal pour pouvoir le restituer plus tard,
- mémorisation du mode "traitement d'interruption",
- *détermination* de l'adresse de début du programme de *traitement* de l'interruption en fonction de sa cause,
- aiguillage de l'*exécution* vers cette adresse.

La **sauvegarde du contexte** peut être plus ou moins complète selon le modèle de microprocesseur. En cas de sauvegarde incomplète, le programmeur a l'initiative de compléter cette sauvegarde par des instructions appropriées.

De même, pendant l'exécution du programme de traitement d'une interruption, le traitement d'une **autre interruption** peut être autorisé ou non. Selon le modèle de microprocesseur, l'interdiction sera soit automatique, soit laissée à l'appréciation du programmeur.

Le sous-programme de traitement de la demande d'interruption consiste généralement à lire le registre d'état du dispositif d'entrées-sorties qui en est à l'origine. L'analyse du registre d'état détermine la suite des traitements à réaliser.

A la fin du traitement de l'interruption, une instruction permet un **retour au programme interrompu** en restituant le contexte qui avait été sauvegardé par le cycle d'acquittement d'interruption.

9.4. Le contrôleur d'interruptions

Un **contrôleur d'interruption** prend en compte d'une *multiplicité* de causes d'interruption. Un système de *priorité* entre signaux d'interruption permet de sélectionner une cause d'interruption plutôt qu'une autre en cas de simultanéité. Pour poursuivre l'analogie avec la sonnerie de téléphone, on peut dire qu'un contrôleur d'interruption joue le rôle de la secrétaire qui filtre les appels selon les directives qui lui ont été données.

Les contrôleurs d'interruption comportent des registres permettant d'interdire certaines causes d'interruption et d'en autoriser d'autres. On parle de *masquage* de certaines interruptions. Pendant le cycle d'acquittement, le contrôleur d'interruptions envoie sur le bus de données un **numéro de vecteur d'interruption** dépendant de la cause identifiée.

9.5. Vectorisation des interruptions

Grâce à ce numéro de vecteur, le microprocesseur peut déterminer l'adresse du sous-programme de traitement de l'interruption avec l'aide d'une table d'adresses située en mémoire. Le numéro de vecteur, multiplié par 4 donne l'adresse du vecteur d'interruption dans la Table des Vecteurs d'Interruption (TVI). Chaque vecteur est une adresse segmentée (sur 4 octets) du sous-programme de traitement d'une interruption.

- n° vecteur --(* 4)--> adresse du vecteur,
- adresse du vecteur --(accès mémoire)--> vecteur,
- vecteur = adresse du sous-programme de traitement de l'interruption.

Le déroulement est ensuite le suivant :

- le contenu du registre FLAGS est empilé,
- l'indicateur IF est positionné à 0 (interruptions interdites),
- l'adresse de retour segmentée est empilée,

- le couple de registres CS:IP est chargé avec l'adresse du sous-programme de traitement d'interruption.

Dans un ordinateur, la TVI est gérée en mémoire vive par le système d'exploitation.

9.6. Les interruptions logicielles

Il s'agit d'interruptions déclenchées par programme, sans l'intervention de signaux d'interruption. L'instant du déclenchement du processus d'interruption est donc connu contrairement aux interruptions d'origine électronique. On distingue les exceptions et les interruptions logicielles.

Les **exceptions** sont déclenchées par certaines instructions dans des situations exceptionnelles. L'exemple le plus évident est la division par 0. On peut également citer la tentative d'exécution d'un code machine absent du jeu d'instructions ou l'accès à une case mémoire qui n'existe pas dans un système à mémoire virtuelle. Pour les exceptions, le numéro de vecteur est déterminé implicitement par la nature de la situation exceptionnelle.

Les **interruptions logicielles** sont généralement utilisées pour accéder aux services offerts par le système d'exploitation. L'interruption logicielle n'est pas le fait d'une situation exceptionnelle. L'instruction INT appelle un sous-programme de traitement d'interruption. Le numéro du vecteur d'interruption est précisé en opérande. Elles ne sont pas *masquables*. L'instruction d'interruption logicielle est parfois appelée TRAP (piège).

9.7. Mécanisme d'accès direct à la mémoire

Certains contrôleurs se présentant sous forme de circuits intégrés sont spécialisés dans l'accès direct à la mémoire (DMA pour **D**irect **M**emory **A**ccess). Ils assurent des **transferts rapides** de blocs entiers entre la **mémoire** et les dispositifs d'**entrées - sorties**. Un contrôleur de DMA est capable de prendre le contrôle des bus d'adresses, données et des signaux de contrôle (notamment ceux concernant l'accès à la mémoire). Par rapport à un transfert effectué par le processeur, une première cause d'amélioration des performances est **l'absence de cycles fetch** ; le contrôleur de DMA étant spécialisé, il n'a pas d'instruction à lire, contrairement à un processeur.

La seconde cause d'amélioration des performances tient au caractère direct des accès. L'**accès est direct** en ce sens que les données *ne transitent pas par le processeur* mais par le contrôleur de DMA, voire dans certaines architectures directement du périphérique à la mémoire ou de la mémoire au périphérique. Les transferts de blocs de mémoire à mémoire sont possibles et passent obligatoirement par un registre intermédiaire. Le processeur programme le contrôleur de DMA, puis lance la sous-traitance et n'intervient plus ensuite.

Enfin, des signaux de **synchronisation** des transferts à l'instigation du périphérique (DMA REQUEST ou DATA REQUEST = demande de transfert, DMA ACKNOWLEDGE ou DATA ACNOWLEDGE = transférez) résolvent **deux types de problème** selon le sens de transfert :

- Pour un transfert de *périphérique à mémoire*, la synchronisation effectuée par le périphérique résout la question de **l'asynchronisme** évoquée pour les interruptions : l'instant de disponibilité de la donnée du périphérique n'est connu que de lui, c'est donc à lui de synchroniser les transferts.
- Pour un transfert de *mémoire à périphérique*, la synchronisation à l'instigation du périphérique résout l'éventuel problème de **l'engorgement** du périphérique : le périphérique ne demande le transfert du mot suivant que lorsqu'il est prêt à l'accepter. Cette synchronisation opère un *contrôle de flux*.

La mise en relation d'un boîtier d'entrées - sorties avec la mémoire au moyen des signaux de synchronisation des transferts forme un **canal de DMA**. Ces signaux de requête / acquittement étant a priori dédiés à un dispositif, il existe un jeu de ces signaux par canal. Ceci explique également l'absence d'adressage proprement dit du dispositif. Un contrôleur de DMA est généralement capable de piloter plusieurs canaux de DMA.

Le contrôle des bus est soit entièrement dédié au DMA, soit attribué tantôt au microprocesseur, tantôt au contrôleur de DMA. Pendant la sous-traitance DMA, le processeur est soit inactif (**processeur arrêté**), soit en fonctionnement normal (**multiplexage**), soit ralenti (**vol de cycle**).

Le processeur programme le contrôleur de DMA en termes d'adresse du bloc à transférer, de taille du bloc, de mode, de largeur du bus de données, de sens de transfert et de numéro de canal. Le processeur lance ensuite la sous-traitance. Le **mode processeur arrêté** conduit aux transferts les plus rapides puisque le DMA dispose des bus à temps plein. Le processeur se met en sommeil (instruction HALT juste après avoir lancé le transfert). Un signal d'interruption de fin de sous-traitance est généré à l'attention du processeur. Ce signal "réveille" le processeur qui reprend le fil de l'exécution à

l'instruction qui suit le HALT. Ce mode présente l'inconvénient de rendre impossible le traitement d'événements plus prioritaires pendant une sous-traitance DMA.

Dans le **mode multiplexage**, le contrôleur de DMA met à profit les cycles de bus inutilisés par le processeur : il n'utilise les bus que lorsque le processeur n'en a pas besoin, par exemple, lorsqu'il réalise une opération purement interne. L'avantage est que le processeur n'est pas du tout ralenti. L'inconvénient est que ce mode ne peut pas être utilisé pour des transferts présentant un caractère d'urgence ou de priorité. De plus, l'évolution des processeurs tend à éliminer les cycles de bus inutilisés par le processeur.

Dans le **mode par "vol de cycle",** le processeur continue de fonctionner mais ménage des cycles disponibles pour le contrôleur de DMA. Ce dernier demande au processeur de libérer les bus grâce au signal HOLD (demande de bus). Le processeur répond à l'aide d'un signal HLDA (HOLD ACKNOWLEDGE : bus accordé) : c'est le seul overhead introduit par ce mode. Cette demande peut concerner un ou plusieurs cycles mémoire : le signal HLDA est maintenu tant que le signal HOLD est maintenu. Les cycles accordés sont mis à profit pour effectuer le transfert d'un ou plusieurs mots selon que les instants séparant le transfert de deux mots consécutifs sont éloignés (transfert sporadique pour un périphérique "caractères" lent) ou au contraire rapprochés (transfert par blocs de mots à vitesse maximale pour un périphérique "bloc"). Finalement, c'est ce mode le plus convaincant, car :

- le processeur continue à fonctionner,
- le processeur ne cède l'usage des bus qu'aux instants où le périphérique est prêt ; il n'est ralenti que du temps nécessaire au transfert ; on a donc le meilleur compromis entre le rendement du processeur et les performances absolues du contrôleur de DMA,
- les événements (= interruptions) plus prioritaires continuent d'être pris en compte.

10. Les logiciels d'exploitation

10.1. Rôle d'un système d'exploitation

Les processeurs 16 bits de la génération du 68000 et du 80286 ainsi que les générations suivantes, de même que les processeurs 32 bits intègrent dans le silicium des dispositifs (MMU=Memory Management Unit) en relation directe avec les préoccupations d'un système d'exploitation. Les concepteurs de ces processeurs ont pris en compte ces préoccupations afin de faciliter l'implantation de mécanismes de base des systèmes d'exploitation et d'en augmenter l'efficacité.

On peut classifier ces mécanismes en plusieurs grands thèmes :

- la mémoire,
- le parallélisme,
- la protection.

Un système d'exploitation prend en charge des opérations pour le compte des programmes d'application s'exécutant sur l'ordinateur. Le terme programme d'application s'oppose justement à système d'exploitation par le fait que n'ayant pas les mêmes préoccupations, ils ont des prérogatives différentes.

Le système d'exploitation a pour but d'**exploiter le matériel** et de **fournir des services** aux programmes d'application. Les programmes d'application s'appuient sur le système d'exploitation pour fournir des services aux utilisateurs du programme.

La fourniture de ces services passe par la notion d'**appel-système** qui sont des appels de sous-programmes au sens des langages évolués, c'est-à-dire avec des paramètres. Ces sous-programmes sont documentés à l'attention des développeurs qui les utilisent.

En plus des services d'exploitation du matériel, le système d'exploitation a comme ambition de **garantir** une certaine qualité de service. Le système d'exploitation ne souhaite donc pas que les programmes d'application manipulent directement le matériel. Ceci pourrait en effet perturber la gestion qu'il opère et l'empêcherait donc de garantir cette qualité de service.

Prenons comme comparaison la gestion d'un magasin de pièces; un magasinier en assure la gestion. Il tient à jour les stocks pour chaque référence, connaît la localisation des pièces, assure les approvisionnements, et distribue les pièces aux utilisateurs selon certaines conditions, sur présentation d'un bon par exemple. L'interface avec les utilisateurs du magasin est constituée d'un guichet qui est un point de passage obligé.

Imaginons maintenant que le magasin de pièces serait accessible directement aux utilisateurs. Ne connaissant pas leur emplacement, ils chercheraient longtemps après les pièces (manque d'efficacité). Indisciplinés par nature, ils se serviraient sans remplir les fiches de stock et n'assureraient pas l'approvisionnement des stocks (absence de qualité de service). La gestion mise en œuvre par le magasinier s'écroule.

Un ordinateur est comparable à une usine utilisant un magasin de pièces. Le **noyau** du système d'exploitation est un programme possédant certains **privilèges** par rapport aux programmes d'application. Il est comparable au magasinier. Il gère efficacement le matériel selon des procédures bien pensées qui assurent une certaine qualité de service. Il tient à jour pour cela un certain nombre de **tables** de gestion comparables aux fiches de stock. Les programmes d'application s'identifient aux utilisateurs du magasin. L'utilisation du noyau s'effectue au moyen d'**appels-noyau** par le mécanisme d'interruption logicielle (INT ou TRAP) qui s'apparente au guichet. Les **appels-systèmes** sont des appels de sous-programmes ayant une utilisation et un rôle bien définis; ils sont comparables aux procédures suivies par les utilisateurs du magasin de pièces (fourniture de bons, etc. ...).

10.2. La gestion de la mémoire

La gestion de la mémoire procède de plusieurs préoccupations :

- la confidentialité et l'intégrité de la mémoire,
- la taille et la disponibilité de la mémoire,
- la vitesse d'accès à la mémoire

10.2.1. Allocation de la mémoire

Lorsqu'un **programme** doit s'exécuter, il faut lui allouer de la place en mémoire. De même, les programmes peuvent utiliser des **données dynamiques** dont la taille, essentiellement variable, n'est connue qu'au fur et à mesure de l'exécution.

Une partie du système d'exploitation, l'**allocateur**, s'occupe de tenir à jour les stocks de mémoire disponible et de répondre aux demandes de mémoire émanant des programmes d'application ou d'autres parties du système d'exploitation.

La mémoire est constituée de blocs de tailles diverses qui sont soit libres, soit occupés par un autre programme. L'allocateur de mémoire effectue généralement un **chaînage** de ces blocs de mémoire. Lors d'une demande d'allocation de mémoire, l'allocateur parcourt la liste des blocs à la recherche d'un bloc libre de taille suffisante dans le but de l'allouer. Le bloc, rarement de taille égale à la demande, est alors **fractionné**, et la quantité exacte de mémoire est attribuée au demandeur, le reste étant disponible pour d'autres allocations.

Différentes stratégies existent pour le choix du bloc de mémoire :

- La stratégie "first fit" consiste à allouer le premier bloc trouvé de taille suffisante.
- La stratégie "best fit" consiste à allouer le bloc dont la taille est la plus proche par excès de la demande à satisfaire.
- La stratégie "worst fit" consiste à allouer le bloc de plus grande taille de façon à maximiser la taille du bloc restant après fractionnement.

Des variantes de ces stratégies existent de façon à en améliorer les performances. La création de **listes séparées** pour les blocs libres et les blocs occupés améliore le temps d'allocation en augmentant légèrement le temps de libération.

Le **tri de la liste des blocs libres** par ordre croissant améliore le temps d'allocation de l'algorithme "best fit" mais augmente le temps de libération.

Des simulations ont montré que la stratégie "first fit" donne curieusement les meilleurs résultats. La stratégie "best fit" présente l'inconvénient de créer par fractionnement une multitude de blocs trop petits pour être alloués ce qui conduit au morcellement de la mémoire.

Une stratégie de gestion de la mémoire par **dichotomie** fragmente la mémoire uniquement en blocs dont la taille est une puissance de 2. Lors de la libération, le fusionnement de blocs 2 voisins de taille égale permet de retrouver un bloc de taille double qui est donc également une puissance de 2. Ce système évite le morcellement de la mémoire mais conduit à un **mauvais taux d'utilisation** de la mémoire : en effet, une demande de taille immédiatement supérieure à une puissance de 2 se verra allouer un bloc de taille presque double de celle dont on a besoin.

10.2.2. Chargement par parties

La taille de la mémoire d'un micro-ordinateur est toujours limitée. Plusieurs raisons font que l'on a toujours besoin de plus de mémoire :

- les programmes sont de plus en plus gourmands en mémoire,
- plusieurs programmes peuvent être simultanément chargés en mémoire.

La limite est liée au coût et à la vitesse d'accès :

- plus une mémoire est importante plus cela coûte cher (à vitesse constante),
- plus une mémoire est importante, plus elle est lente (à coût constant).

La **technologie** tend à augmenter la capacité et la rapidité des mémoires.

Jusqu'ici, nous avons envisagé uniquement la gestion de la mémoire centrale (RAM). Lorsqu'elle ne suffit plus aux besoins des programmes, d'autres techniques font appel à la **mémoire secondaire** (disque) au prix d'une certaine dégradation des performances. Le disque a en effet une taille supérieure, mais c'est une mémoire plus lente.

Lorsque les programmes ont une taille supérieure à la mémoire effectivement disponible, il est possible de les charger par parties au cours de l'exécution. C'est le chargement de "partiels" ou d'overlays. Le chargement des partiels fait l'objet d'un code exécutable inséré soit "manuellement" par le programmeur, soit automatiquement par l'éditeur de liens.

Aujourd'hui cette technique est abandonnée au profit d'un mécanisme transparent pour le programmeur d'applications : la mémoire virtuelle paginée, prise en compte par le système d'exploitation. Elle est rendue possible grâce aux unités de gestion de mémoire.

10.2.3. Les unités de gestion mémoire

Ce sont des dispositifs matériels qui associés au système d'exploitation rendent possibles la protection et la mémoire virtuelle. Une unité de gestion mémoire (MMU = Memory Management Unit) :

- restreint l'accès aux segments de mémoire selon le droit attribué au processus en cours d'exécution
- transcrit les adresses logiques (ou virtuelles) générées par les programmes en adresses physiques destinées à la mémoire.

10.2.4. La mémoire virtuelle

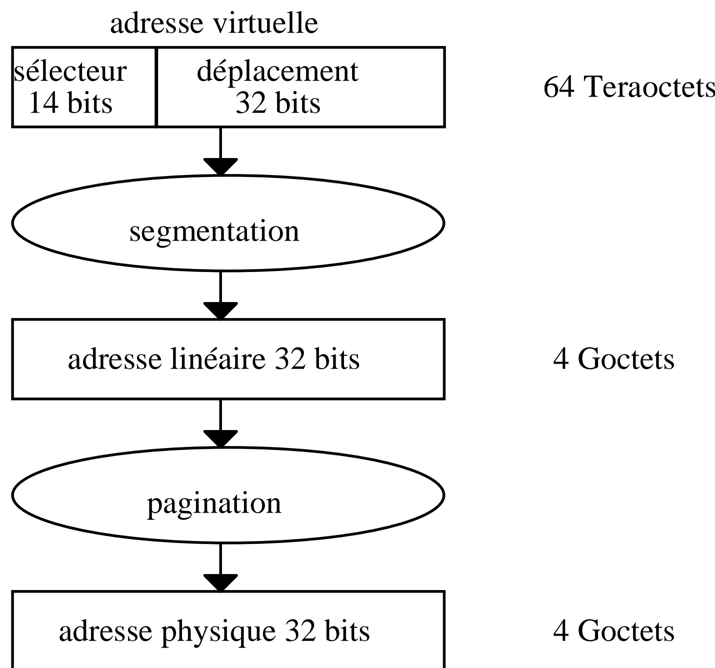
Il s'agit d'un concept où l'on considère la mémoire d'un point de vue **logique** : on dispose alors toujours de la mémoire dont on a besoin sans qu'on ait à se soucier de la mémoire **physique** réellement disponible. Des mécanismes sous-jacents du système d'exploitation prennent en charge la mise à disposition de la mémoire centrale physiquement disponible. Comme ces mécanismes font appel au disque, on accepte que les performances soient dégradées dans des proportions acceptables.

En pratique, la **limite** est celle de *l'espace disque* disponible. Une autre limite est la *capacité d'adressage* du processeur. C'est donc une conception de la mémoire où les programmes avec leurs données et leur pile peuvent être d'une taille supérieure à celle de la mémoire effectivement disponible.

10.2.4.1. La mémoire virtuelle du 386

On présente ici la segmentation et pagination des MMU intégrés au processeur Intel 386 et ses successeurs Pentium. Les logiciels s'exécutent en générant des adresses virtuelles composées d'une partie sélecteur et d'une partie déplacement. Les registres segments rebaptisés registres sélecteurs ont toujours 16 bits dont 14 sélectionnent le descripteur de segment.

Un programme adressant un mot fournit un déplacement, adresse ayant un sens à l'intérieur d'un segment d'une taille maximale de 4 giga octets. L'association du sélecteur et du déplacement permet d'adresser 16384 segments de 4 Gigaoctets soit 64 Teraoctets.



Les segments sont des zones mémoire de longueur variable. Les adresses générées par les programmes n'ont de sens que dans les segments affectés au programme par le système d'exploitation. Les segments peuvent être découpés en des blocs de mémoire de taille fixe appelés pages. La pagination, optionnelle, fait correspondre des pages physiques à des pages logiques au fur et à mesure des besoins.

Si la pagination n'est pas mise en œuvre, la segmentation fournit directement l'adresse physique. La mémoire virtuelle fait une image disque de chaque page dans une zone spéciale du disque dur. Un segment est constitué d'un ensemble de pages logiques contiguës correspondant à des pages physiques qui ne le sont pas nécessairement. Le morcellement de la mémoire n'est donc plus un problème.

10.2.5. La segmentation

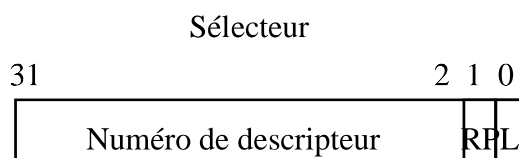
Dans un système de mémoire virtuelle segmentée, on dispose de multiples espaces d'adressage, chacun d'eux est appelé segment. Chaque espace d'adressage étant sensé être suffisant pour un type de besoin. Le découpage en segments est un découpage logique donc **géré par le programmeur** d'une part, et les **tailles des segments** ne sont pas toutes les mêmes d'autre part.

Dans le système de segmentation du processeur Intel 8086, l'espace linéaire d'adressage d'un Mégaoctet peut être partagé en **espaces d'adressage** pouvant varier entre 16 octets (1 paragraphe) et 64 ko : les segments. A l'intérieur de chaque segment, les adresses peuvent varier entre 0 et 65535. Cette multiplicité des espaces d'adressage facilite le relogement des programmes en mémoire.

Dans le cas où plusieurs processus exécuteraient le même programme, le fait de séparer le segment de code des segments de données, permet d'attribuer un segment de données à chaque processus et de partager le segment de code.

10.2.5.1. La segmentation du 386

Les registres de segment du 386 ne contiennent plus des numéros de paragraphe mais des sélecteurs de segment. Seuls 14 bits constituent le sélecteur de segment, les 2 bits restant codent un niveau de privilège.



Le sélecteur permet de sélectionné un **descripteur de segment**. Les descripteurs de segment sont regroupés dans des **tables de descripteurs de segment**. Un descripteur de segment contient les informations suivantes.

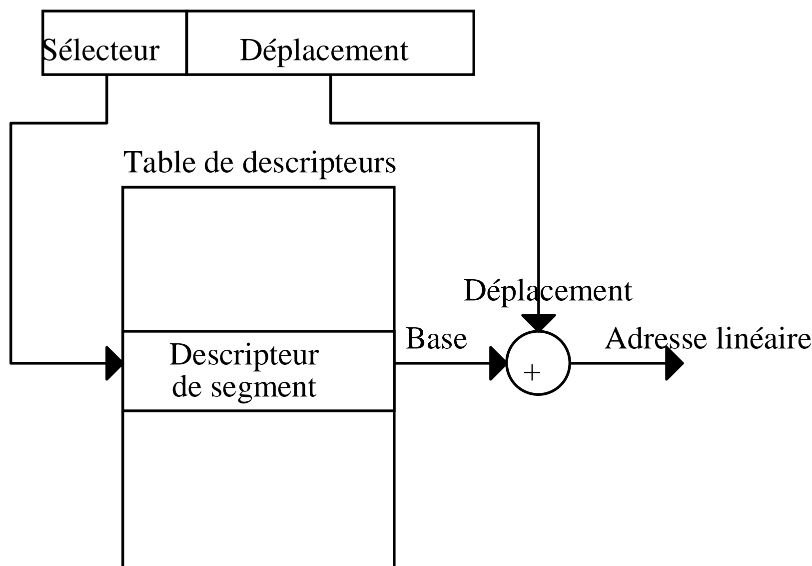
base 31-24	attributs (16 bits)	base 23-16
base 15-0	limite 15-0	

Une information de base sur 32 bits désigne l'adresse linéaire de base du segment. Une information limite de segment sur 16 bits autorise le contrôle des accès au-delà du segment. Des attributs également stockés dans le descripteur de segment caractérisent le segment notamment quant au privilège requis pour y accéder.

Lorsqu'une valeur de sélecteur est chargée dans un registre de segment, deux cas peuvent se produire. Soit le descripteur désigné par la valeur de sélecteur existe dans une table de descripteurs, il est alors chargé dans un registre caché du processeur appelé registre descripteur. Soit la valeur de sélecteur ne désigne aucun descripteur, un bit de présence parmi les attributs du descripteur passe à 0 dans le registre descripteur indiquant que le contenu n'est pas valide.

10.2.5.2. Translation de segment

Lors d'une instruction d'accès à la mémoire, l'adresse effective résultant du mode d'adressage fournit la partie déplacement de l'adresse virtuelle. L'adresse de base stockée dans le descripteur est additionnée à la partie déplacement pour fournir l'adresse linéaire.



Le 386 gère plusieurs types de tables de descripteurs. Les LDT (Local Descriptor Table) sont des tables de descripteurs locaux à une **tâche** ou à un petit nombre de tâches. Les GDT (Global Descriptor Table) sont des tables de descripteurs communs à toutes les tâches. Les instructions LLDT et LGDT permettent de charger l'adresse de début de ces tables dans les registres LDTR et GDTR.

Les tables de descripteurs sont stockées en mémoire, mais le registre descripteur de segment joue le rôle de cache lors des translations de segment. Pour chaque registre de segment, CS, SS, DS, ES, FS et GS il existe un registre descripteur de segment correspondant dans le processeur.

Si la pagination n'est pas mise en œuvre l'adresse linéaire sur 32 bits correspond à l'adresse physique. Dans le cas contraire, une translation de page se produit.

10.2.6. La pagination

Dans un système à mémoire virtuelle paginée, on dispose d'un espace d'adressage unique très important et donc supérieur à la mémoire physiquement disponible. La mémoire est découpée en **pages** qui sont toutes de longueur

égale. La MMU établit une **correspondance** au moyen de tables programmées par le noyau du système d'exploitation entre :

- les adresses logiques utilisées dans les programmes d'une part,
- les adresses physiques c'est-à-dire correspondant à de la mémoire physiquement présente d'autre part.

Ainsi, avec un adressage logique de 4 Gigaoctets, et une mémoire physique de 8 Mégaoctets, un programme ayant besoin de 16 Mégaoctets peut s'exécuter. Supposons que les 8 Mégaoctets physiquement disponibles lui soient alloués; lorsqu'il veut utiliser une page qui ne correspond pas à de la mémoire physiquement disponible, le système de correspondance géré par la MMU est mis en défaut.

C'est ce que l'on appelle un **défaut de page** : ceci correspond à une exception générée par la MMU sur le processeur et à l'attention du superviseur. Le traitement de l'exception "défaut de page" consiste à éliminer le défaut de page :

- Un algorithme de remplacement des pages choisit une page physique à mettre en correspondance avec la page en défaut. Elle est sauvegardée sur le disque dur si elle a été modifiée depuis son chargement.
- elle est mise en correspondance avec la page logique ayant causé le défaut de page,
- elle est chargée avec son contenu présent sur disque dur,
- une fois le défaut de page disparu, l'instruction l'ayant causé est **réexécutée** à son début; l'exception "défaut de pages" l'avait en effet interrompue.

10.2.7. Pagination du 386

Le 386 met en œuvre à la fois la segmentation (d'abord) et la pagination (ensuite). La pagination est optionnelle mais si elle est mise en œuvre, l'adresse linéaire est ensuite translatée en adresse physique. Le mécanisme de translation de page considère que l'adresse linéaire est composée des trois champs suivants.

Adresse linéaire

REP	PAGE	DEPL

La pagination, quand elle est mise en œuvre, scinde l'adresse linéaire en 3 champs. Elle utilise 2 types de tables que le système d'exploitation gère dans une zone non paginée.

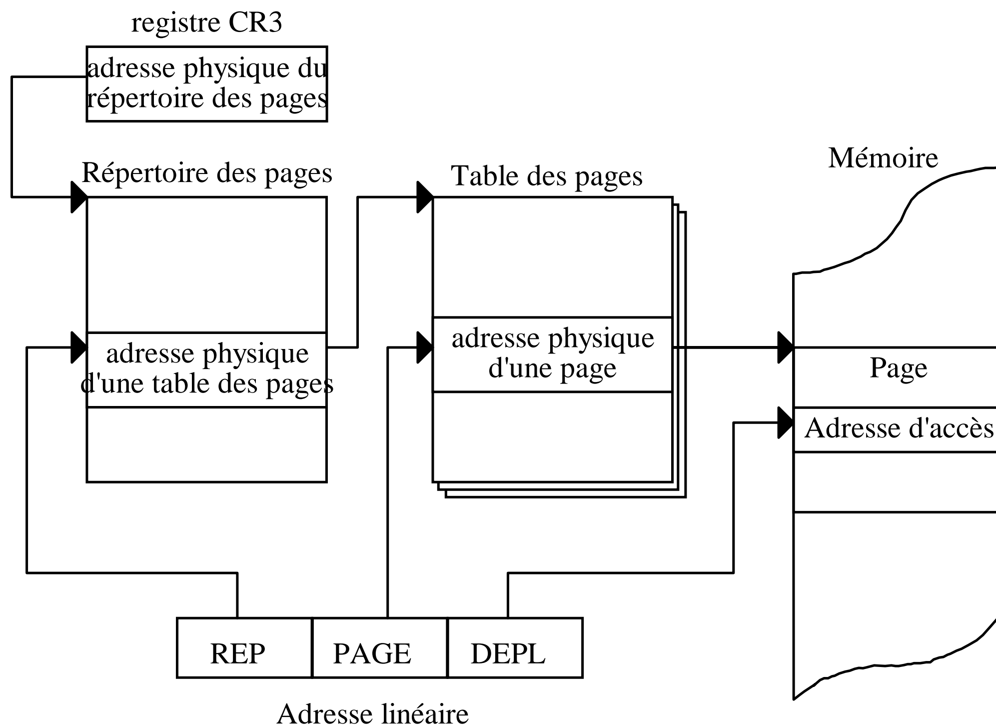
10.2.7.1. Les tables de pages

Le champ PAGE fournit un numéro de page dans une table des pages. Les tables de pages ont une longueur de 4 koctets. Chaque entrée occupant 32 bits, chaque table décrit un maximum de 1024 pages. On comprend donc que le champ PAGE occupe 10 bits. Les entrées de la table des pages comportent 12 bits d'attributs et 20 bits d'adresse de page.

Parmi les attributs, les **pages utilisables** pour la translation de page sont marquées du bit P (Present) dans l'entrée correspondant à la page.

10.2.7.2. Le répertoire des tables de pages

Le champ REP fournit un numéro d'entrée dans le répertoire des pages. Il a une taille de 4 koctets et ses entrées sont des pointeurs de 32 bits. Un maximum de 1024 tables des pages est donc possible. Les entrées du répertoire des pages désignent les adresses de début des tables des pages.



10.2.7.3. Le champ DEPL

Le champ DEPL quant à lui fournit un déplacement dans la page. Les pages ayant une longueur de 4 koctets, le champ DEPL est codé sur 12 bits.

10.2.8. Marquage des pages

Parmi les attributs de page, les bits A (Accessed) et D (Dirty) sont marqués lors des accès à la page permettant l'implantation des algorithmes de remplacement de page. Accessed et Dirty correspondent respectivement aux bits R et M classiquement utilisés dans la littérature relative systèmes d'exploitation.

Le bit A est positionné lors d'un accès en lecture ou en écriture. Il est positionné d'une part dans l'entrée correspondante de la table des pages, d'autre part dans l'entrée du répertoire des pages correspondant à la table des pages concernée.

Le bit D est positionné uniquement lors des écritures. Il est positionné uniquement dans l'entrée correspondante de la table des pages.

10.2.9. Remplacement des pages

Le système d'exploitation intervient pour reprogrammer les descripteurs de la MMU et pour effectuer les transferts disque - mémoire nécessaires. Il doit prendre une décision stratégique : quelle page choisir comme page de remplacement ? Tant que la mémoire physique n'est pas entièrement utilisée, il n'y a pas de problème. Une fois qu'elle est entièrement attribuée, il faut choisir une page dont on suppose que le programme n'a plus besoin pour l'instant. L'idéal serait de pouvoir prévoir l'avenir est de choisir, parmi les pages utilisées, celle qui sera référencée dans l'avenir le plus lointain. Cet algorithme n'est pas réaliste.

Différents choix sont possibles :

- l'algorithme NRU (Not Recently Used) choisit une page non récemment utilisée,
- l'algorithme FIFO choisit la page *chargée* en mémoire depuis le temps le plus long,
- l'algorithme LRU (Least Recently Used) choisit la page la moins récemment utilisée,
- l'algorithme NFU (Not Least Frequently Used) choisit une page qui n'est pas fréquemment utilisée

L'**algorithme NRU** choisit une page non récemment utilisée. Les MMUs permettent de savoir si une page a été utilisée récemment grâce à deux bits R (pour Referenced) et M (pour Modified). Ces deux bits, initialement à 0 pour toutes les pages. Le bit R est remis à zéro à intervalles réguliers. Lorsqu'une page est lue, seul le bit R est mis à 1; lorsqu'elle est écrite, les deux bits R et M sont mis à 1. On peut grâce à cela classer les pages en quatre catégories correspondant aux combinaisons de ces deux bits. On choisira une page au hasard dans une des catégories dans l'ordre de préférence suivant :

- les pages avec R=0 et M=0,
- les pages avec R=0 et M=1,
- les pages avec R=1 et M=0,
- les pages avec R=1 et M=1.

On choisira les catégories suivantes si les premières catégories sont vides. Les pages avec R=0 et M=1 peuvent exister car R est remis à zéro périodiquement. Ce sont des pages qui ont été modifiées depuis leur chargement mais qui n'ont pas été modifiées ou lues récemment, c'est-à-dire depuis la dernière remise à 0 de R.

Le bit M n'est pas remis à zéro périodiquement car il est utile lors du remplacement de page pour savoir si la page doit être sauvegardée sur le disque. Il n'est remis à zéro que lors du chargement de la page en mémoire centrale depuis le disque dur. Cet algorithme est simple à mettre en œuvre et fonctionne bien en pratique; il est donc très utilisé.

L'algorithme FIFO pourrait s'appeler LRL (pour Least Recently Loaded) car ici le critère n'est pas l'utilisation mais le chargement. Une file d'attente gérée en FIFO (First In - First Out) mémorise l'ordre dans lequel les pages sont chargées depuis le disque. La première page à avoir été chargée parmi celles encore présentes en mémoire est la première à être remplacée.

Une variante de **l'algorithme FIFO** consiste à tester les bits R et M de la page qui doit sortir de la FIFO. Si R=M=0 la page sort effectivement (elle sert au remplacement), sinon ce sont les pages suivantes dans la FIFO qui subissent cet examen. Si aucune page de la FIFO n'a R=0 et M=0 alors on applique la même méthode pour les catégories suivantes.

Dans une autre variante de l'algorithme FIFO, l'**algorithme de la seconde chance**, on ne teste que le bit R de la page qui doit sortir de la FIFO. Si R=0, la page sort effectivement, sinon le bit R est remis à zéro et la page est mise à la queue de la FIFO comme si elle venait d'être chargée en mémoire. On lui accorde une chance d'échapper à l'évacuation sur disque dur et de ne pas servir au remplacement.

L'algorithme LRU choisit la page la moins récemment utilisée comme page de remplacement. Ceci est très contraignant car il faut pouvoir classer les pages dans cet ordre. Cet ordre peut être remis en cause à chaque accès à la mémoire. Il faudrait disposer d'un support matériel pour tenir à jour l'ordre des pages de façon satisfaisante.

L'algorithme NFU possède une implémentation logicielle, il nécessite un compteur par page. A intervalles réguliers, les pages avec R=1 voient leur compteur s'incrémenter puis les bits R sont remis à zéro. On arrive donc bien à savoir avec cet algorithme quelles sont les pages qui sont souvent référencées. En cas de défaut de page, la page qui a le compteur le plus faible est utilisée pour le remplacement.

Le problème avec l'algorithme NFU est que les compteurs ne sont jamais remis à zéro. Un compteur peut donc avoir une valeur élevée et cependant n'avoir pas été fréquemment utilisé récemment. Il existe une variante appelée **algorithme du vieillissement** où à la fin de chaque intervalle, les compteurs sont décalés d'un bit vers la droite, le bit R entrant sur la gauche. Une page, qui est fréquemment référencée, a une valeur de compteur plus élevée qu'une qui l'est moins; cependant si une page a été référencée souvent par le passé plus lointain et qu'elle ne l'est plus dans un passé plus récent sa valeur diminue jusqu'à atteindre zéro. L'algorithme du vieillissement se comporte sensiblement comme l'algorithme LRU.

10.2.9.1. La mémoire cache

Les données, au sens large du terme, peuvent être stockées dans des mémoires d'accès de plus en plus rapide jusqu'au moment de leur exécution. La bande magnétique est la mémoire la plus lente. Le disque est un dispositif plus rapide (temps d'accès en millisecondes) et permet l'accès direct. La mémoire centrale l'est encore plus. Le haut de la hiérarchie des temps d'accès est constitué par la mémoire cache externe ou interne au microprocesseur.

Le principe de la mémoire virtuelle commencé avec la mémoire centrale et le disque peut être appliqué entre une mémoire centrale normale et une mémoire centrale très rapide mais aussi plus petite appelée mémoire cache. Celle-ci est divisée en blocs de taille inférieure à la taille d'une page. Grâce à un **dispositif matériel** (un contrôleur de mémoire cache interne ou externe au microprocesseur), les blocs les plus utilisés sont présents en mémoire centrale mais sont de plus **dupliqués** dans le cache. Les adresses physiques normalement destinées à la mémoire centrale normale sont, du point de vue du contrôleur de mémoire cache, des adresses logiques, les adresses physiques étant alors celles de la mémoire cache. Le **défaut de cache** correspond à la notion de défaut de page.

Le mécanisme de cache est le plus souvent **complètement** pris en compte par le **matériel** et n'est pas connu du système d'exploitation.

10.3. Le parallélisme d'exécution

Il faut d'abord distinguer le parallélisme vrai où chaque programme (processus) dispose d'un processeur et le pseudo-parallélisme où les processus se partagent un même processeur. Nous allons parler ici de pseudo-parallélisme et nous utiliserons le terme parallélisme sans préciser davantage.

Tout se passe **comme si chaque processus disposait d'un processeur**. Avec un processeur Intel 386, chaque processus d'application a l'illusion de disposer d'un processeur Intel 8086 à quelques instructions supplémentaires près. On accepte cependant des performances moindres par rapport à autant de processeurs réels : en fait, le temps et donc la **puissance est donc partagé** entre les différents processus qui s'exécutent en pseudo-parallélisme. La multiplicité des processus va autoriser la multiplicité des utilisateurs simultanés d'une même machine. On parle alors de système d'exploitation multi-tâches et multi-utilisateurs.

Lors des opérations d'**entrées-sorties**, les temps mis en œuvre par les dispositifs périphériques sont dans des ordres de grandeur très supérieurs à ceux utilisés par les instructions. Il y donc beaucoup de **temps perdu** par le processeur à attendre que le périphérique soit prêt ou qu'il envoie l'information demandée. De la même façon, lorsqu'on achète un article, les délais de livraison sont tels qu'on a largement le temps de s'atteler à un autre travail que celui qui avait nécessité l'achat.

Ceci est l'idée à la base du pseudo-parallélisme. Dès qu'un processus est en attente d'un événement extérieur, il doit donner la possibilité de s'exécuter à un **autre processus** qui lui n'attend que le processeur pour poursuivre son travail.

Par exemple, lorsqu'un **défaut de page** se produit, on fait appel au disque pour lire le contenu de la page en mémoire. Le temps d'accès à l'information s'exprime en millisecondes alors que le temps d'exécution des instructions s'exprime en en nanosecondes. En attendant que l'information soit disponible, le processus qui a subi le défaut de page se voit retirer la ressource processeur qui est attribuée à un autre processus prêt à s'exécuter.

C'est le système d'exploitation qui attribue le processeur aux processus d'application; il peut également en disposer pour ses propres besoins. La partie du système d'exploitation qui alloue le processeur s'appelle **l'ordonnanceur** (scheduler). Un ordonnancement peut avoir lieu à l'occasion d'une entrée - sortie, d'un appel système, du traitement d'une interruption ou d'une exception.

Dans certains ordonnanceurs, une interruption liée au temps écoulé peut provoquer l'ordonnancement de telle sorte qu'en l'absence d'autres occasions, un minimum de rotation du processeur soit assuré. Le processus actif se voit subtiliser le processeur sans avoir fait appel au système d'exploitation, sans qu'une cause externe d'interruption se soit manifestée et même sans qu'une exception se soit produite. Cette possibilité s'appelle la **préemption** ou la réquisition. Un ordonnanceur capable de préemption est dit préemptif. En fait, il y a bien une interruption mais issue de l'horloge. La cause est donc temporelle. Les processus du noyau lui-même peuvent être préemptés par l'ordonnanceur.

10.3.1. Algorithmes d'attribution du processeur

Il existe différents algorithmes d'attribution du processeur qui se fondent sur les critères suivants :

- équité dans l'attribution du processeur,
- taux d'utilisation du processeur,
- réactivité aux sollicitations,
- efficacité dans les travaux.

On ne tient pas compte de ces critères de la même façon selon la **vocation** de l'ordinateur et le **type des travaux** à réaliser. En effet, on ne demande pas la même chose à un ordinateur de bureautique ou au développement et à un ordinateur de process traitant les alarmes d'une centrale nucléaire. De même, certains travaux soumis à un ordinateur n'exigent pas d'intervention humaine (on parle de **traitements par lots**), d'autres travaux sont en relation directe avec l'utilisateur (ce sont des **programmes interactifs**).

Sur le thème de **l'équité**, tous les processus ne seront pas dotés de la même façon. Certains utilisateurs travaillant sur des travaux plus **urgents** se verront allouer plus de puissance de traitement. Les programmes nécessitant une certaine **réactivité** auront éventuellement besoin de plus de puissance également. Cependant, tout processus devra pouvoir disposer même faiblement du processeur. L'ordonnanceur ne devra mettre en **famine** oublier aucun processus.

Le processeur doit être utilisé le plus possible **pour le compte des utilisateurs** et le moins possible **pour le compte du système** d'exploitation. Le temps passé à ordonnancer, à remplacer des pages ou à attendre les périphériques doit être minimisé. Ceci sera crucial dans un ordinateur central ayant un coût d'exploitation élevé. Cela en aura moins dans un ordinateur personnel. Cela en aura moins encore dans un ordinateur de process si c'est là le prix à payer pour gagner en réactivité.

Un ordinateur de process possède un système d'exploitation **temps-réel**. On emploie également le terme de système réactif. Dans un tel système, le critère essentiel est le temps de réponse à certains événements. Pour être digne de ce nom, un système temps-réel doit pouvoir garantir certains temps de réponse, en termes de **borne supérieure de temps** dans le pire cas. Certains travaux exigent peu de puissance mais une réactivité importante. On acceptera à la rigueur un mauvais taux d'utilisation du processeur qui "passera une partie de son temps à se tenir prêt".

L'efficacité dans les travaux signifie réaliser, **en moyenne**, un travail le plus important possible, en le moins de temps possible.

Chaque processus est un programme qui dispose de segments de pile, de données et de code. Lorsqu'un processus dispose du processeur, il occupe notamment ses registres. L'ensemble des registres et la mémoire attribuée au processus constitue son **contexte**.

Lorsque le processus est interrompu par l'ordonnanceur et que le processeur doit être attribué à un autre processus, le contexte du processus doit être sauvegardé pour pouvoir être restitué plus tard. Le contexte du processus qui obtient le processeur doit être restitué; notamment, les registres du processeur doivent être rechargés dans l'état antérieur. Il y a donc **changement de contexte** (context switching = task switching = process switching).

Les **processeurs** actuels **facilitent** le travail du système d'exploitation en assurant un changement de contexte le plus rapide possible. Cependant, ils ne préjugent en rien de la stratégie à employer pour la réquisition éventuelle du processeur auprès du processus en cours d'exécution. Ils **n'imposent pas d'algorithme** dans le choix du processus qui obtient la ressource processeur.

10.3.2. Ordonnancement tourniquet

Dans l'**ordonnancement circulaire** (tourniquet, "round-robin"), un quantum de temps est alloué à chaque processus, pendant lequel il peut s'exécuter. Lorsque le quantum de temps est écoulé, le processus suivant obtient le processeur. Si un processus se trouve en attente de ressource avant la fin de son quantum de temps, il rend le processeur à l'ordonnanceur pour qu'un autre processus puisse en bénéficier. L'ordre dans lequel les processus bénéficient du processeur est circulaire. En fait, l'ordonnanceur utilise une file d'attente circulaire : les processus servis sont remis à la fin de la file d'attente.

Le quantum de temps doit être suffisamment important en regard du temps nécessaire au choix du processus et au changement de contexte. Le rapport du temps passé à ordonnancer sur le du temps passé à exécuter les autres processus systèmes ou les processus utilisateurs est appelé **overhead**.

10.3.3. Ordonnancement temps-réel

Dans l'**ordonnancement avec priorité**, les processus sont dotés d'une priorité. Parmi les processus prêts à s'exécuter, les processus les plus prioritaires s'exécutent jusqu'à ce qu'ils soient en manque d'une ressource ou jusqu'à ce qu'ils rendent le processeur d'eux-mêmes. Ceci peut éventuellement conduire à une situation de famine pour les autres processus. A l'intérieur d'un même niveau de priorité les processus sont ordonnancés circulairement.

Eventuellement, les **priorités peuvent changer** au cours de l'exécution. Elle diminue périodiquement si le processus n'a pas d'autre raison de rendre le processeur. Elle augmente, si en moyenne le processus n'utilise pas tout son quantum de temps.

Dans un système temps-réel, c'est le choix des priorités, la **conception** même des processus et d'autres mécanismes pris en compte par le programmeur qui **évitent la famine** aux processus de faible priorité.

10.4. La notion de protection

Un système d'exploitation protégé restreint l'accès à la mémoire, non seulement en écriture, mais aussi en lecture. Le but recherché est d'une part la sûreté de fonctionnement, d'autre part la lutte contre des actions malveillantes consistant à détruire espionner des données sensibles.

Certains processeurs utilisés dans des systèmes protégés font appel à deux types d'instructions : d'une part les **instructions normales**, destinées aussi bien à l'écriture des programmes des utilisateurs qu'à celle du noyau du système d'exploitation, d'autre part les **instructions privilégiées** dont l'emploi est réservé à l'écriture du noyau.

Le processeur peut donc se trouver dans deux modes : d'une part, le **mode utilisateur** où seules les instructions normales sont exécutables, et d'autre part, le **mode superviseur** où l'ensemble du jeu d'instructions est exécutable. Les instructions supplémentaires, privilégiées, concernent notamment les entrées - sorties.

Une **interruption** d'origine **électronique**, directement liée au matériel, fait passer le processeur en mode superviseur s'il existe. De même, **l'interruption logicielle** est une instruction normale qui fait passer le processeur du mode utilisateur au mode superviseur. On comprend qu'elle soit l'interface d'accès au noyau du système d'exploitation. Le retour d'interruption fait revenir au mode utilisateur.

Quand le processeur en mode utilisateur tente d'exécuter une instruction privilégiée, son exécution est refusée et une exception "**violation de privilège**" est générée. Elle donne lieu comme son nom l'indique à un traitement exceptionnel.

Les microprocesseurs Intel 8086 et 8088 ne possèdent qu'un mode de fonctionnement où toutes les instructions sont utilisables par tout type de programme. Ces processeurs ne peuvent pas être utilisés dans des systèmes protégés.

A partir du 80286, on trouve un système hiérarchique à quatre niveaux de privilège, qui offre non seulement un accès contrôlé aux instructions, mais également à des tables de gestion utilisées par le processeur pour faciliter l'implantation du noyau des systèmes d'exploitation tout en améliorant les performances.

En effet, le mécanisme d'instruction privilégiée est **insuffisant** en lui-même pour permettre l'écriture de noyaux inviolables. En effet, le traitement d'une interruption ou d'une exception s'effectue en mode superviseur. Donc, si l'on n'interdit pas aux utilisateurs d'implanter en mémoire leurs propres programmes de traitement d'interruption, sans contrôle du noyau, ce dernier n'est pas inviolable.

Aussi, les concepteurs des générations successives de processeurs 16 et 32 bits ont intégré dans le silicium les dispositifs allant dans le sens de l'inviolabilité du noyau. Ces mécanismes concernent notamment la **protection d'accès à la mémoire** et rejoignent en cela d'autres préoccupations de gestion de la mémoire, du domaine des systèmes d'exploitation.

10.4.1. La protection du 386

Un mécanisme de protection est disponible sur le 386. Il opère principalement au niveau du segment encore que quelques contrôles soient effectués au niveau de la page. Les bits d'attributs stockés dans les descripteurs de segments et dans les entrées des tables de pages sont utiles à la mise en œuvre de la protection.

10.4.1.1. Les niveaux de privilège

Le processeur offre quatre niveaux de privilège numérotés de 0 à 3, le plus privilégié étant le niveau 0. Les niveaux de privilèges sont stockés dans les descripteurs de segment. En effet, parmi les attributs du segment, on trouve 2 bits constituant le champ DPL (Descriptor Privilege Level).

Les deux bits de poids faible des sélecteurs codent également un niveau de privilège et sont notés RPL (Requestor's Privilege Level).

Le niveau de privilège du processeur est appelé CPL (Current Privilege Level). Il est normalement défini par le champ DPL du descripteur du segment de code courant. Le processeur change de niveau selon qu'il est entrain d'exécuter une application de l'utilisateur ou qu'il se trouve dans telle ou telle partie du système d'exploitation.

Dans certaines situations, il se peut que le CPL soit différent du DPL du segment de code courant. En effet, il est besoin parfois qu'une procédure s'exécute avec le niveau de privilège de la procédure appelante. Pour cela, il faut que la procédure appelée se situe dans un segment **conforme**. Un segment est dit conforme si un des bits d'attribut (bit C de conformité) est positionné dans le descripteur du segment. Le bit de conformité n'existe que pour les segments de code. Les segments conformes permettent d'offrir des procédures exécutables sous plusieurs niveaux de privilèges.

10.4.1.2. Protection au niveau du segment

10.4.1.2.1. Contrôle de la limite des segments

Ces vérifications ont lieu avant que l'accès à la mémoire ne se produise. La partie déplacement de l'adresse virtuelle est notamment confrontée avec le champ limite du descripteur de segment. Tout accès au-delà du segment est ainsi empêché et une exception "défaut général de protection" se déclenche. Il en va de même pour une tentative d'exécution d'une instruction située au-delà de la limite du segment de code envisagé.

10.4.1.2.2. Contrôle du type de descripteur

Parmi les attributs d'un descripteur, 5 bits déterminent son type. Ainsi, on distingue les segments de code et les segments de données.

Il n'est pas possible, sous peine de déclencher une exception, de charger dans CS un sélecteur qui ne désigne pas un descripteur de segment de code ; de même il n'est pas possible de charger dans DS un sélecteur qui désigne un descripteur de segment de code non accessible en lecture.

Il n'est possible à aucune instruction d'écrire dans un segment de code ou dans un segment de données protégé en écriture.

10.4.1.2.3. Contrôle d'accès aux données

Le contrôle est effectué au moment où on charge une valeur de sélecteur dans un registre de segment. Pour qu'une instruction puisse accéder à un segment de données, c'est-à-dire en utilisant les registres de segment DS, ES, FS GS ou SS, il faut que son privilège soit suffisant par rapport au privilège du segment auquel elle veut accéder.

Pour cela, il faut confronter le privilège du processeur (CPL) précédemment défini, le privilège du sélecteur du segment de code courant (RPL) et le privilège du segment de données (DPL). Numériquement, Le DPL doit être supérieur ou égal à CPL et RPL, ce qui signifie que le privilège du processeur et du sélecteur doivent être plus important que (ou égal à) le privilège du segment de données.

Ainsi, un segment de données muni d'un DPL de niveau 2, sera accessible à des instructions d'un segment dont le RPL est compris entre 0 et 2, à condition que le CPL soit lui aussi compris entre 0 et 2.

10.4.1.2.4. Protection des segments de code

Les instructions de saut inconditionnel, les instructions d'appel et de retour de sous-programme, les appels et les retours d'interruption font l'objet d'un contrôle fondé sur les niveaux de privilège. Les niveaux de privilège sont représentatifs de la confiance qu'on accorde à une procédure. Les procédures du noyau sont les plus privilégiées et aussi les plus dignes de confiance. Les procédures utilisateur sont les moins privilégiées, les moins dignes de confiance. On comprend dès lors qu'une procédure ne puisse pas demander l'exécution d'une procédure moins digne de confiance. **La procédure appelée doit être au moins aussi privilégiée (sinon plus privilégiée) que la procédure appelante**. Pour un segment non conforme, la procédure appelée doit être de même niveau de privilège que la procédure appelante. Pour un segment conforme, la procédure appelée peut éventuellement être plus privilégiée que la procédure appelante.

10.4.1.2.5. Protection par guichet

Un changement de privilège du processeur (changement de la valeur du CPL) ne peut s'effectuer qu'à travers un guichet. Un **guichet est un descripteur** particulier qui constitue une indirection supplémentaire pour les instructions CALL ou JMP qui les utilisent. Le champ type du descripteur précise de quel type de descripteur il s'agit.

Le guichet contrôle les changements de privilège lors des appels de procédure (guichet CALL), lors des commutations de tâches (guichet de tâche), lors du traitement des interruptions (guichet d'interruption). Un guichet CALL précise le point d'entrée de la procédure apportant ainsi la garantie que le contrôle de l'exécution ne sera pas transféré n'importe où.

Un guichet introduit une sélection supplémentaire dans les tâches autorisées à utiliser une procédure, une tâche ou une interruption. En effet, un guichet étant une structure de données, les règles concernant les segments de données s'appliquent quant à son utilisation. Seules les procédures suffisamment privilégiées pourront utiliser le guichet. De plus, la règle de protection des segments de code continue de s'appliquer.

10.4.1.3. Protection au niveau de la page

La protection au niveau de la page ne considère que deux niveaux de privilège : le niveau **utilisateur** (qui correspond au niveau 3) et le niveau **superviseur** (qui correspond aux niveaux 0, 1 et 2).

Des droits d'accès sont affectés pour chaque page qui peut être soit accessible en lecture et en écriture, soit accessible en lecture seulement. Une exception "défaut de page" est générée :

- lors d'une tentative d'accès à un segment dont le bit P n'est pas positionné,
- lors d'une tentative d'accès à un segment avec un privilège insuffisant,
- lors d'une tentative d'écriture dans un segment protégé en écriture.

10.4.2. Protection des entrées-sorties

Le registre EFLAGS (Extended Flags) du 386 définit notamment un champ IOPL codé sur 2 bits qui spécifie un niveau de privilège pour l'accès aux entrées-sorties. Le champ IOPL est associé à une tâche puisque les contextes de tâches comportent un champ IOPL qui est chargé dans le champ IOPL du registre EFLAGS lors d'une commutation de tâches. Pour qu'une procédure puisse utiliser une instruction d'entrées-sorties, il faut que le CPL soit numériquement inférieur ou égal à IOPL ce qui signifie que le privilège du processeur (CPL) doit être au moins aussi important que le privilège d'entrées-sorties de la tâche (IOPL).

Dans le cas où le privilège du processeur serait insuffisant, une carte des permissions des entrées-sorties fournit un accès sélectif aux 65536 ports d'entrées-sorties possibles. A raison d'un bit par port, la carte autorise ou interdit l'accès à tel ou tel port. La carte des permissions des entrées-sorties est stockée dans le segment de tâche.

11. Système d'exploitation : approfondissement

11.1. Implantation du pseudo-parallélisme

Les petits systèmes à microprocesseur ne disposent pas forcément d'un système d'exploitation ou d'un "noyau". Il n'y a donc qu'un seul "fil" d'exécution (thread). Seul le traitement des interruptions peut prétendre contredire cette affirmation. Le thread dans lequel s'exécutent les interruptions a cependant un statut tout à fait particulier : déclenché par un périphérique. La nature des traitements est particulière : en relation avec les entrées-sorties. Un tel thread n'existe également que de façon limitée dans le temps. Un des principaux intérêts d'un système d'exploitation réside dans le support qu'il offre pour l'exécution apparemment simultanée de plusieurs tâches. Plusieurs threads semblent exister de façon simultanée sans que l'un soit favorisé par rapport à l'autre et sans qu'une nature de traitement soit imposée. Parfois, le système d'exploitation existe mais n'offre pas le support pour le pseudo-parallélisme. On s'intéresse dans ce travail dirigé à la conception d'un support à la multi-exécution.

11.1.1. Hypothèses

Dans un premier temps, on va supposer que ces sont les processus eux-mêmes qui sont suffisamment "raisonnables" pour "rendre" le processeur afin que les autres processus puissent travailler, un tel type de multi-tâche est dit non préemptif : aucune entité n'a l'autorité pour reprendre le processeur (préempter) à un processus qui le conserverait de façon abusive.

11.1.2. Commutation de contexte

Le code A empile l'ensemble des registres du processeur. Seul le pointeur de pile SP et SS n'est pas empilé.

```
        pushf
        callf   ici
        ...
ici:    push    ax
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    es
        push    ds
```

Le code B restitue la pile et le processeur dans l'état de départ.

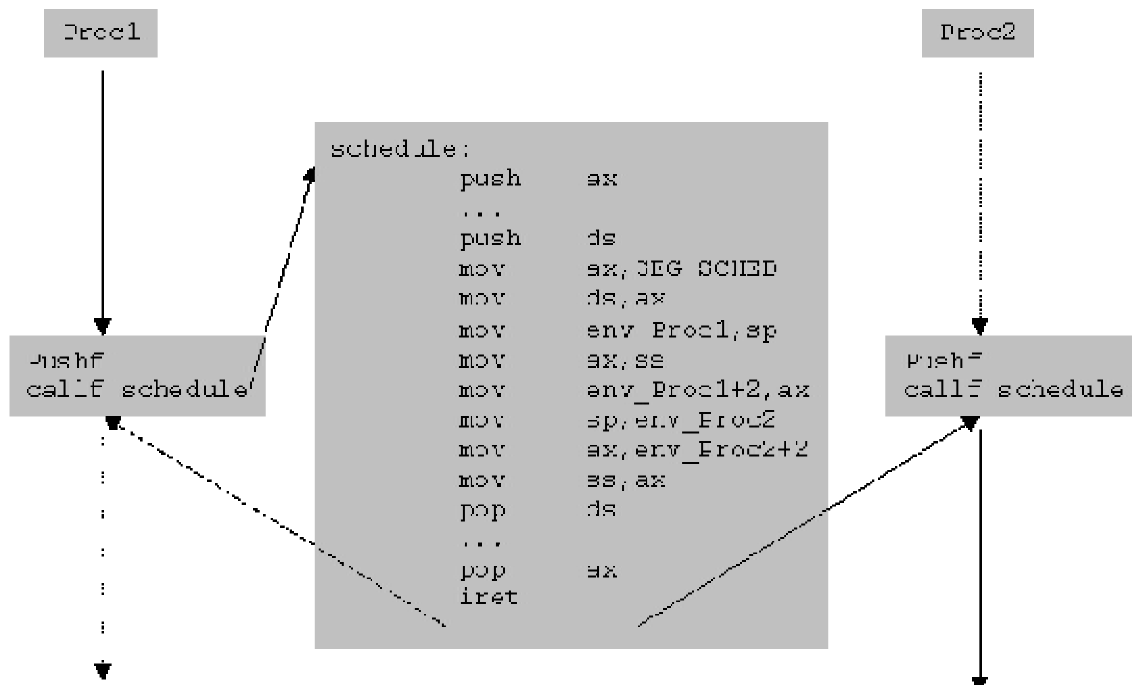
```
        pop     ds
        pop     es
        pop     bp
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        iret
```

Pour partager le processeur entre 2 processus P1 et P2, on exécute le code C entre les codes A et B. P1 et P2 possèdent chacun un segment de code, données et pile et les pointeurs env_Proc sont alloués dans un segment distinct. Le code A sauvegarde le contexte d'exécution c'est-à-dire la valeur actuelle des registres dans la pile de P1. Le pointeur de pile complet est constitué de SS et SP. Le segment distinct repéré par la valeur SEG_SCHED.

On sauvegarde SS et SP dans le pointeur far env_Proc1 et on les recharge à partir d'env_Proc2 censé pointer sur le contexte dans la pile de P2. Le code B restitue le contexte de P2 dans les registres du processeur.

```
        <code A>
mov       ax,SEG_SCHED   ┐
mov       ds,ax          │
mov       env_Proc1,sp   │
mov       ax,ss          ├── code C
mov       env_Proc1+2,ax │
mov       sp,env_Proc2   │
mov       ax,env_Proc2+2 │
mov       ss,ax          ┘
        <code B>
```

Quand un processus n'a plus besoin du processeur, il appelle une procédure far après avoir empilé les indicateurs. On reconnaît ici le code A. L'étiquette ici de la page précédente est renommée schedule qu'on peut traduire par ordonnancer. On y ajoute le code C. Puis le code B et la procédure schedule est complète. Après avoir sauvegardé le contexte de l'appelant et son pointeur de pile dans env_Proc1, son rôle est de commuter le pointeur de pile vers un contexte pointé par env_Proc2. On suppose que c'est un Processus Proc1 qui, temporairement, n'a plus besoin du processeur et souhaite en céder l'usage au processus Proc2. Au départ env_Proc1 sur des données qui n'ont pas d'importance pour l'explication. On montre donc une pile vide. Env_Proc2 pointe sur une pile contenant le contexte de Proc2 supposé préalablement empilé. Proc1 appelle schedule. Schedule sauvegarde le contexte de Proc1, positionne le segment de données de façon à voir ses variables env_Proc1 et 2, sauvegarde le pointeur de pile dans env_Proc1, charge le pointeur de pile avec env_Proc2 et dépile dans les registres le contexte pointé par env_Proc2. La restitution du contexte de Proc2 se termine par iret qui dépile dans CS:IP une adresse de retour en direction de Proc2.



11.1.3. Ordonnancement fondé sur l'équité

On cherche à ce que les processus aient la main chacun leur tour de façon circulaire. Ce type d'ordonnancement est parfois qualifié de tourniquet (round-robin) ou tour de table.

On propose ici une structure de données en langage C et un **code C'** qui généralise le mécanisme précédent à n processus et fournit le fonctionnement en tourniquet.

```
void far * tab_Proc[MAX_PROC];     // contient les CS:IP des processus
int currentProc = 0;
```

```
void schedule() asm {
// sauvegarder le contexte du process appelant
// flags, CS, IP déjà empilés
        push    ax
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    es
        push    ds
// chargee la valeur de ds du scheduler
        mov     ax,SEG tab_Proc
        mov     ds,ax
// tasab_Proc[currentProc] = SS:SP du process appelant
        mov     bx,OFFSET tab_Proc
        mov     si,currentProc
        mov     di,si
        add     di,di
        add     di,di
        mov     [bx+di],sp
        mov     ax,ss
        mov     [bx+di+2],ax
// currentProc = (currentProc+1) % MAX_PROC
        inc     si
        cmp     si,MAX_PROC
        jne     fsi0
        xor     si,si
fis0:   mov     currentProc,si
// SS:SP = tab_Proc[currentProc]
        add     si,si
        add     si,si
        mov     sp,[bx+si]
        mov     ax,[bx+si+2]
        mov     ss,ax
// restituer le contexte du process nouvellement élu
        pop     ds
        pop     es
        pop     bp
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        iret
}
```

On appelle **schedule** le "sous-programme" commençant par l'étiquette "ici:" et constitué des codes A, C' et B. Lorsqu'un processus souhaite céder le processeur, il suffit qu'il appelle le sous-programme `schedule` avec pour seule contrainte d'utiliser `pushf` puis `callf` plutôt qu'un simple `call`.

Dans ce cadre, l'instruction `int xx` appelle un sous-programme d'interruption via le numéro de vecteur xx (xx étant un nombre compris entre 00 et FF). Cette instruction présente l'avantage de sauvegarder les indicateurs et d'utiliser des adresses segmentées. Sur les processeurs possédant un système de protection ce type d'instruction fait également passer en mode d'exécution privilégié qui est le mode d'exécution des systèmes d'exploitation. Schedule ferait alors partie du système d'exploitation.

11.1.4. Temps d'attente

Lorsqu'un processus a besoin d'effectuer une entrée ou une sortie, cela peut prendre du temps. Une fois l'opération d'entrée-sortie lancée, il serait pénalisant que le processeur passe son temps à attendre la fin de l'opération. L'idée est de céder le processeur à un autre processus jusqu'à ce que l'opération d'entrée-sortie soit terminée.

Les opérations d'entrée-sortie étant généralement accessibles aux applications par l'intermédiaire de bibliothèques. Pour l'écriture de ces bibliothèques, il faut prévoir un appel à schedule une fois l'opération lancée. L'ensemble des services d'entrées-sorties offert par le système d'exploitation est donc susceptible d'appeler schedule.

On appelle processus **éligible** un processus qui n'a pas besoin d'attendre quoi que ce soit : il ne lui manque que le processeur pour être le processus **élu**, à savoir le processus qui s'exécute. Cette notion complique le sous-programme schedule. En effet, seuls les processus éligibles doivent participer à l'ordonnancement. Il ne s'agit plus simplement d'incrémenter `currentProc` modulo `MAX_PROC`. Il faut une structure de données qui ne stocke que les processus éligibles qui sont en nombre variable.

La fin d'une opération d'entrée-sortie est signalée au processeur par une interruption. Cela implique que le traitement de l'interruption marque comme éligible le processus qui était en attente de terminaison de l'opération d'entrée-sortie, puis fasse appel à schedule. Une telle interruption est un événement susceptible de provoquer un ordonnancement.

11.1.5. Ordonnancement préemptif

Dans les ordonnanceurs étudiés jusqu'à présent, les interruptions signalant la fin d'une opération d'entrée-sortie peuvent provoquer un ordonnancement c'est-à-dire le réexamen de l'attribution du processeur. De même, un processus "discipliné" qui considère ne plus avoir besoin du processeur pour l'instant peut provoquer l'ordonnancement.

Une interruption se produisant quoi qu'il advienne au bout d'un certain temps permettrait de reprendre le processeur (préempter) à un processus non discipliné. Ainsi, si un autre événement ne vient pas provoquer l'ordonnancement, il y aura au moins celui-là.

Dans un système à microprocesseur, un temporisateur matériel pouvant déclencher une interruption est indispensable à l'implantation d'un noyau multi-tâche préemptif.

Dans le cas d'un ordonnancement préemptif, un processus peut être préempté en tout point du code où l'interruption temporisateur est autorisée.

12. Evolution des performances

12.1. Introduction

Les processeurs ont évolué rapidement ces dernières années. Les progrès sont de plusieurs ordres :

- progrès de la technologie de gravure du silicium,
- progrès dans l'architecture des processeurs,

En revanche, les mémoires ont surtout progressé en capacité. Leur temps d'accès n'a que peu diminué, justement à cause de l'augmentation des capacités. Pour combler ce divorce, on a augmenté la largeur du bus de données et une hiérarchie de mémoires cache s'est interposée entre le processeur et la mémoire centrale.

12.2. Structure d'un ordinateur

Les ordinateurs sont parfois conçus autour d'un bus système obéissant à une spécification publique. Des cartes d'extension peuvent ainsi y être connectées. Certains bus système autorisent la présence de plusieurs dispositifs pouvant se rendre maître du bus tels qu'une carte d'extension possédant un processeur, comme les cartes graphiques. Un contrôleur de bus réalise l'interface entre le bus processeur et le bus système. Il s'interface aussi avec la mémoire accessible concurremment par le processeur ou par un maître du bus système. Il cumule aussi parfois le rôle de contrôle de la mémoire cache afin d'en gérer les échanges avec la mémoire.

12.3. Progrès technologiques

Selon la loi de Moore, le nombre de transistors intégrables sur une puce de silicium unique double tous les 18 mois. La finesse de gravure diminue. La taille des puces réalisables de manière fiable dans un process industriel augmente. Ceci a les conséquences suivantes :

- plus une puce est petite, plus sa fréquence de fonctionnement peut être élevée,
- plus une puce est petite, plus sa consommation est faible (à fréquence égale).

La fréquence de fonctionnement atteint le GHz en 2000. On prévoit que les transistors au silicium ne pourront pas dépasser une limite de fréquence de commutation de l'ordre de 20 GHz.

La consommation est proportionnelle au carré de la tension d'alimentation. La tendance est également à la réduction de la tension de fonctionnement qui est passée de 5V à 3,3V puis 2,8V. Mais on ne voit pas comment des transistors pourront commuter au-dessous de 1V.

12.4. Amélioration des performances

Les progrès dans l'architecture des processeurs reposent sur des techniques d'amélioration des performances telles que :

- le pipeline et super-pipeline,
- la mémoire cache,
- la prédiction de branchement,
- le super-scalaire,
- la granularité plus fine des instructions,
- le réordonnancement des instructions,
- le multi-scalaire.

Ces techniques reposent sur la suppression des temps d'attente et l'exécution en parallèle. Une évolution significative a été franchie avec les architectures RISC (Reduced Instruction Set Computer) faisant appel à certaines de ces techniques. Ces types de processeur ont joué un important rôle d'aiguillon vis-à-vis des

processeurs à architecture traditionnelle, rebaptisés a posteriori du nom d'architecture CISC (Complex Instruction Set Computer).

Les progrès architecturaux des processeurs doivent être rapprochés de ceux réalisés pour les compilateurs quant à l'optimisation du code machine qu'ils génèrent. Ceci est vrai pour les processeurs RISC et également pour les architectures VLIW (Very Long Instruction Word) qui font appel à des codes machine très longs.

12.4.1. Mémoire cache

Les performances des processeurs ont évolué beaucoup plus vite que celles des mémoires. Ces dernières ont surtout crû en capacité, mais guère en performances.

En fait, la technologie des mémoires vives est régie par l'équation technico-économique "coût-taille-temps d'accès" Une des solutions consiste à mettre en place des mémoires cache ou antémémoires. Une mémoire cache est une mémoire de taille plus faible mais d'un accès plus rapide que la mémoire principale. A cause de cette taille réduite, on n'y stocke que les informations les plus probables d'accès. Dans le domaine de l'approvisionnement des marchandises, on peut comparer la mémoire cache à un entrepôt qui ne contiendrait que les produits les plus demandés afin d'améliorer leur délai de livraison.

Dans le domaine des processeurs, la mémoire cache s'interpose entre le processeur et la mémoire centrale des ordinateurs. La mémoire cache contient une copie des informations de la mémoire centrale qui ont l'accès le plus probable. Les besoins de communication entre le processeur et la mémoire sont principalement de deux ordres : l'accès aux instructions et l'accès aux données. Les mémoires cache améliorent uniquement les accès en lecture. Heureusement ceux-ci sont majoritaires : les accès aux instructions notamment sont toujours des lectures. Dans certaines architectures, la mémoire cache des instructions est séparée de celle des données ; dans d'autres, le cache est commun ; d'autres encores panachent les deux techniques.

La notion de cache est relative : dans un système, on peut donc introduire plusieurs niveaux de mémoire cache entre le processeur et la mémoire centrale. S'installe alors une hiérarchie de temps d'accès croissants en s'éloignant du processeur. Dans le même temps, les tailles de mémoire augmentent. Les premiers niveaux de mémoire cache sont intégrés à la même puce de silicium que le processeur. Les niveaux suivants peuvent être dans le même boîtier mais sur une autre puce ou dans un autre boîtier. On peut considérer que la mémoire centrale est une mémoire cache vis-à-vis de la mémoire secondaire que sont les disques durs.

Le choix des informations à mettre dans la mémoire cache fait appel aux principes de localité spatiale et de localité temporelle. L'observation du comportement des programmes montre que si une donnée a fait l'objet d'un accès, dans un passé proche à l'échelle des temps d'exécution, il est probable qu'elle fera à nouveau l'objet d'un accès dans un futur proche. C'est la localité temporelle. De la même façon, si une donnée d'adresse n fait l'objet d'un accès, il est probable que les données voisines feront l'objet d'un accès. C'est la localité spatiale. Ainsi, les échanges entre la mémoire centrale et la mémoire cache s'effectuent par pages qui sont des blocs de données d'adresses voisines.

Lorsque la mémoire cache est pleine, il faut choisir les données qui pourront être éliminées au profit d'autres plus probables. Plusieurs stratégies peuvent être mises en œuvre dans ces algorithmes de remplacement. La plus utilisée (LRU pour Least Recently Used) consiste à éliminer les données ayant fait l'objet des accès les plus anciens.

Une mémoire cache contient une copie d'informations disséminées par blocs dans la mémoire centrale. L'adresse ou une information équivalent est stockée dans la mémoire cache en même temps que la donnée et des bits d'état. Lorsque le processeur cherche à accéder à une donnée, il génère l'adresse de cette donnée. Pour savoir si la donnée est dans le cache, les adresses stockées dans la mémoire cache sont comparées en parallèle avec l'adresse de la donnée recherchée. Si la donnée est dans le cache, on parle de "cache hit", l'accès est rapide. Si elle n'y est pas, on parle de "cache miss" et il faut accéder à la donnée en mémoire centrale et la dupliquer dans le cache. Le temps d'accès est celui de la mémoire centrale. Si le cache est plein, la nouvelle donnée vient écraser une donnée périmée.

12.4.1.1. Cas de l'écriture

Les accès en lecture sont largement majoritaires ce qui fait que la présence d'une mémoire cache améliore les performances. Cependant, lorsque le processeur écrit une donnée, elle doit être écrite dans le cache et dans la mémoire centrale afin de garder la cohérence. Deux méthodes existent :

- le "write through" consiste à écrire en même temps dans la mémoire cache et dans la mémoire centrale, le temps d'accès est celui de la mémoire centrale,
- le "write back" consiste à écrire dans le cache afin que l'accès soit rapide, et de mettre en cohérence la mémoire centrale par rapport au cache à l'occasion du prochain "cache miss"

12.4.2. La technique du pipeline

C'est une technique utilisée depuis longtemps en organisation du travail où on l'appelle "travail à la chaîne". Elle est appliquée ici à l'exécution des instructions d'un processeur. En effet, l'exécution d'une instruction fait appel à plusieurs étapes qu'on peut comparer à des postes de travail successifs d'une ligne de montage. En première approche, on peut considérer que l'exécution se déroule en quatre phases :

- lecture de l'instruction LI,
- décodage de l'instruction DI,
- exécution proprement dite EX,
- rangement du résultat RR,

Si le processeur possède quatre unités, chacune étant affectée à l'une de ces tâches, il est possible de faire fonctionner ces unités en parallèle. C'est ce qu'on cherche à faire dans un processeur avec pipeline.

12.4.3. Amorçage du pipeline

- étape 1 : l'unité de lecture lit l'instruction 1,
- étape 2 :
  - l'unité de décodage décode l'instruction 1,
  - l'unité de lecture lit l'instruction 2,
- étape 3 :
  - l'unité d'exécution exécute l'instruction 1,
  - l'unité de décodage décode l'instruction 2,
  - l'unité de lecture lit l'instruction 3,
- étape 4 :
  - l'unité de rangement range le résultat de l'instruction 1,
  - l'unité d'exécution exécute l'instruction 2,
  - l'unité de décodage décode l'instruction 3,
- l'unité de lecture lit l'instruction 4

12.5. L'architecture RISC

Les processeurs à architecture RISC possèdent les caractéristiques suivantes :

- un pipeline composé d'au moins 4 unités,
- un jeu réduit d'instructions conçu pour un compilateur optimisant,
- un séquenceur câblé,
- des instructions codées sur un mot mémoire, agissant de registre à registre,
- de nombreux registres.

12.5.1. Redéploiement du "budget" silicium

Les concepteurs de l'architecture RISC ont réexaminé l'architecture des processeurs, sans faire table rase du passé, mais en ayant un point de vue critique sur l'existant. Des statistiques sur l'utilisation des instructions réellement exécutées par les programmes montrent que 80% du temps est passé dans seulement 20% des instructions du jeu d'instructions des microprocesseurs traditionnels. L'idée est donc d'économiser le silicium pour des instructions inutiles et d'en dépenser plus pour les instructions stratégiques et l'amélioration des performances : sophistication du pipeline et augmentation du nombre de ses étages, mémoire cache. Pour les instructions d'usage moins fréquent, on se repose sur le compilateur pour qu'il les remplace par plusieurs instructions simples, ce qui n'est pas pénalisant dans un tel cas.

L'usage d'instructions complexes exécutées en plusieurs phases, impose l'implantation des séquenceurs sous forme microprogrammée. A chaque phase, une microinstruction active les parties du processeur qu'il faut mettre en œuvre. Avec des instructions plus simples et moins nombreuses, il est plus aisé de concevoir un séquenceur

câblé ce qui utilise moins de silicium. Dans un processeur RISC, lorsqu'on doit implanter une instruction comprenant plusieurs phases, il faut s'arranger pour que chaque phase corresponde à un étage de pipeline. Il est donc parfois nécessaire d'augmenter le nombre d'étages. L'essentiel est qu'à chaque front d'horloge, une instruction termine son exécution.

Dans les processeurs traditionnels, les instructions ayant un ou plusieurs opérandes ont le choix entre des opérandes situés dans le code de l'instruction (adressage immédiat), dans la mémoire (adressages direct et indirect) ou dans un registre (pas d'adressage). Pour économiser le silicium, les processeurs RISC limitent l'accès à la mémoire aux seules instructions de type MOV. Les autres doivent agir sur des opérandes situés dans des registres. Elles peuvent aussi utiliser des constantes spécifiées dans le code machine (adressage immédiat) mais avec des limitations sur les valeurs.

Ce principe du "registre à registre" permet aussi de coder la plupart des instructions sur un mot unique et de taille fixe (typiquement 32 bits) là où les processeurs traditionnels utilisent des codes machine de longueur variable. Ceci est intéressant pour la régularité du pipeline. Comme les opérandes doivent toujours être dans des registres, ceux-ci doivent être plus nombreux (souvent 32).

12.5.2. Rôle des compilateurs

Pour ce type d'architecture, les compilateurs ont un rôle déterminant. En modifiant l'ordre des instructions, il est possible d'éviter certaines irrégularités prévisibles dans le fonctionnement du pipeline. L'optimisation est aussi plus simple avec des instructions plus élémentaires. Il y a plus de possibilités d'optimisation lorsque les registres sont nombreux, notamment pour les variables locales et les paramètres. Le compilateur garde trace de l'usage des registres et les alloue de manière rationnelle. Les registres peuvent être utilisés pour passer des arguments aux sous-programmes ce qui est plus rapide que par la pile. Celle-ci n'est donc pas utilisée systématiquement mais pour sauvegarder/restituer les registres qui doivent être rendus non modifiés au programme appelant.

Un nombre de registres important devient un inconvénient dans les occasions où il faut les sauvegarder/restituer tous. Par exemple, lors de la commutation de contexte, il faut sauvegarder tous les registres utilisés par la tâche qui perd l'usage du processeur et restituer tous les registres utilisés par la tâche qui gagne l'usage du processeur. Ceci est un frein à l'augmentation du nombre des registres. Dans les noyaux temps-réel, le temps nécessaire à une commutation de contexte est primordial et des compromis sont souvent nécessaires.

12.6. Architectures, super-pipeline et super-scalaire

Un super-pipeline n'est qu'une évolution du pipeline vers un plus grand nombre d'étages. Un super-scalaire met en parallèle plusieurs pipelines afin de pouvoir exécuter plusieurs instructions par cycle d'horloge.

Une dépendance de ressources se produit quand 2 instructions doivent utiliser une même unité du processeur. Dans un super-scalaire, 2 instructions peuvent utiliser simultanément une unité similaire dans chacun des pipelines.

Une dépendance de données se produit quand une instruction a besoin du résultat d'une instruction précédente. Il faut alors attendre ce résultat ce qui réduit les performances.

Une autre solution consiste à propager le résultat vers l'instruction suivante sans attendre la phase "rangement résultat". C'est le "data forwarding"

Les instructions peuvent aussi être réordonnées par le processeur. Le compilateur peut aussi éviter de générer des instructions dépendantes l'une derrière l'autre.

Une dépendance de contrôle se produit lors d'un saut conditionnel. Si le saut a lieu, les instructions dans le pipeline sont normalement les instructions en séquence et non les instructions situées à l'étiquette de saut.

12.7. Processeurs CISC

Les processeurs CISC devant assurer la compatibilité avec le parc des logiciels existants doivent conserver leur important jeu d'instructions parfois complexes.

Les codes machine étant de longueur variable, un étage de "prefetch" est nécessaire. Un tampon est rempli à partir du cache instruction. Les frontières des instructions peuvent y être déterminées. Le ou les pipelines sont ensuite alimentés de façon à éviter les dépendances de données.

12.8. L'architecture VLIW

L'architecture VLIW (Very Long Instruction Word) fait appel comme son l'indique à des instructions à mot long. Ce type de programmation est très ancien puisqu'elle est même antérieure aux microprocesseurs. Le décodeur d'un processeur CISC identifie un code machine, lui associe un microprogramme que le séquenceur "exécute" ensuite. Avant l'avènement des microprocesseurs, il était possible de microprogrammer directement les ordinateurs de façon à commander l'activité de leurs différentes unités de façon à avoir un maximum de parallélisme. Dans une architecture VLIW, on cherche à utiliser au mieux les unités d'un microprocesseur pipeliné en se reposant sur un optimiseur de microprogramme agissant au moment de la compilation.

12.9. Applications

Le gain en puissance des processeurs autorise une croissance dans l'ambition des logiciels. Citons quelques exemples d'activité consommatrices de puissance CPU.

Les systèmes d'exploitation multi-utilisateurs et l'usage des réseaux exigent de la sécurisation. Il faut vérifier les droits sur les ressources de la part des utilisateurs.

Les réseaux à bas débit rendent nécessaires les algorithmes de compression des données. Les modems logiciels utilisent la puissance du processeur principal à la place d'un processeur dédié. Internet incite au cryptage de l'information.

Le "bytecode" des programmes écrits en langage Java sont des "codes machine" destinés à une machine virtuelle qui est émulée par le processeur réel. L'intérêt est que le code Java peut s'exécuter quel que soit le processeur réel et le système d'exploitation sous-jacent. Dans certains langages à objet - et Java fait partie -, une tâche récupère à l'arrière-plan la mémoire qui n'est plus utilisée. Le vérifieur de code Java détecte, lors du chargement, certaines séquences d'instructions apparentées à des virus.

En bureautique, certaines vérifications peuvent s'exécuter à l'arrière plan comme la vérification/correction orthographique et grammaticale. La reconnaissance vocale exige aussi de la puissance.

Certaines parties du calcul des images de synthèse utilisent la puissance du processeur principal.

13. Bibliographie

Le micro, Architecture matérielle et logicielle. Christian Schüller. Editions ellipses. ISBN 2-7298-0194-4

Architecture de l'ordinateur. A. Tannenbaum.