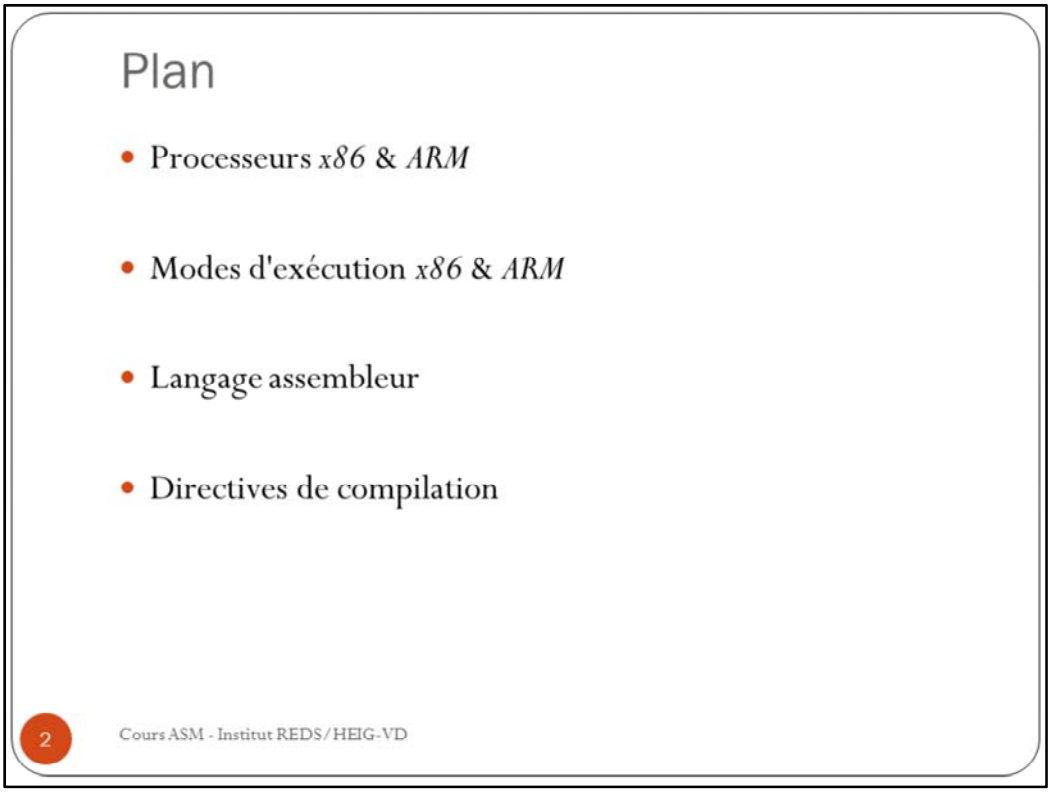
REDS
Reconfigurable & embedded Digital Systems

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Programmation assembleur (ASM)
Assembleurs x86 & ARM

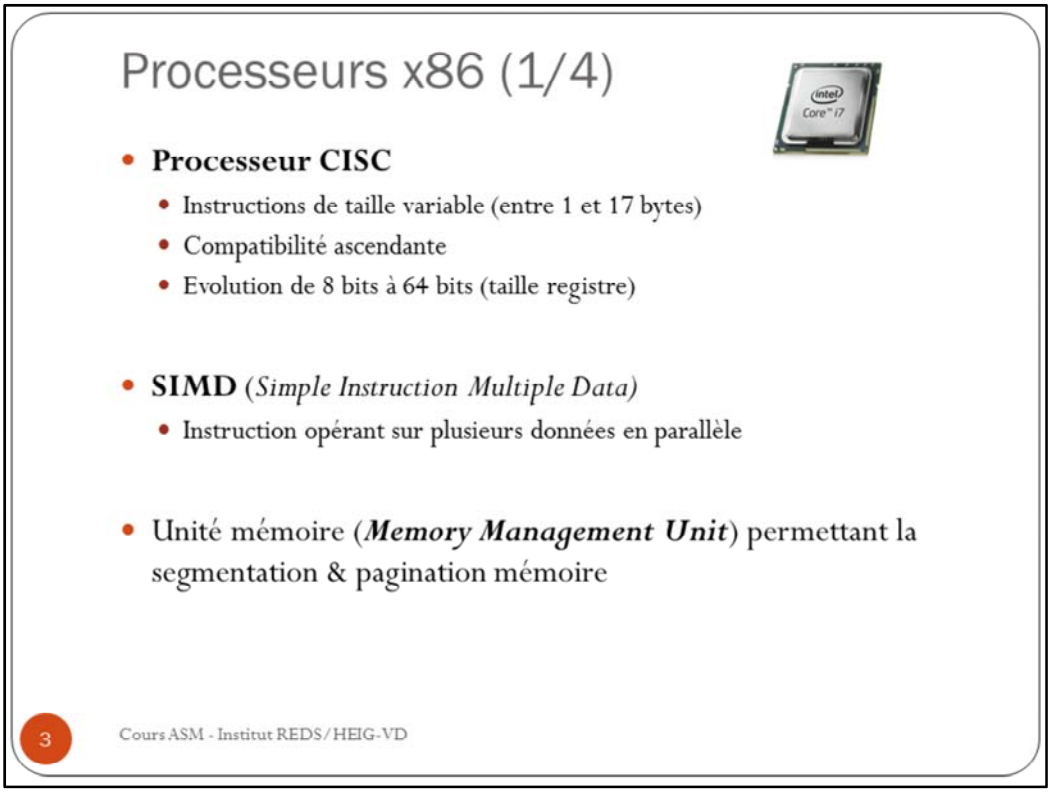
Prof. Daniel Rossier

Version 1.4 (2015-2016)



Le cours ASM propose un aperçu de deux langages d'assemblage: ***x86*** et ***ARM***, avec un accent particulier sur ce dernier. En effet, plus récent que l'assembleur *x86*, l'assembleur *ARM* joue un rôle prépondérant puisqu'il est largement répandu non seulement dans les systèmes embarqués (comme les smartphones ou tablettes PC), mais tend aussi à se répandre dans les *desktops* et serveurs.

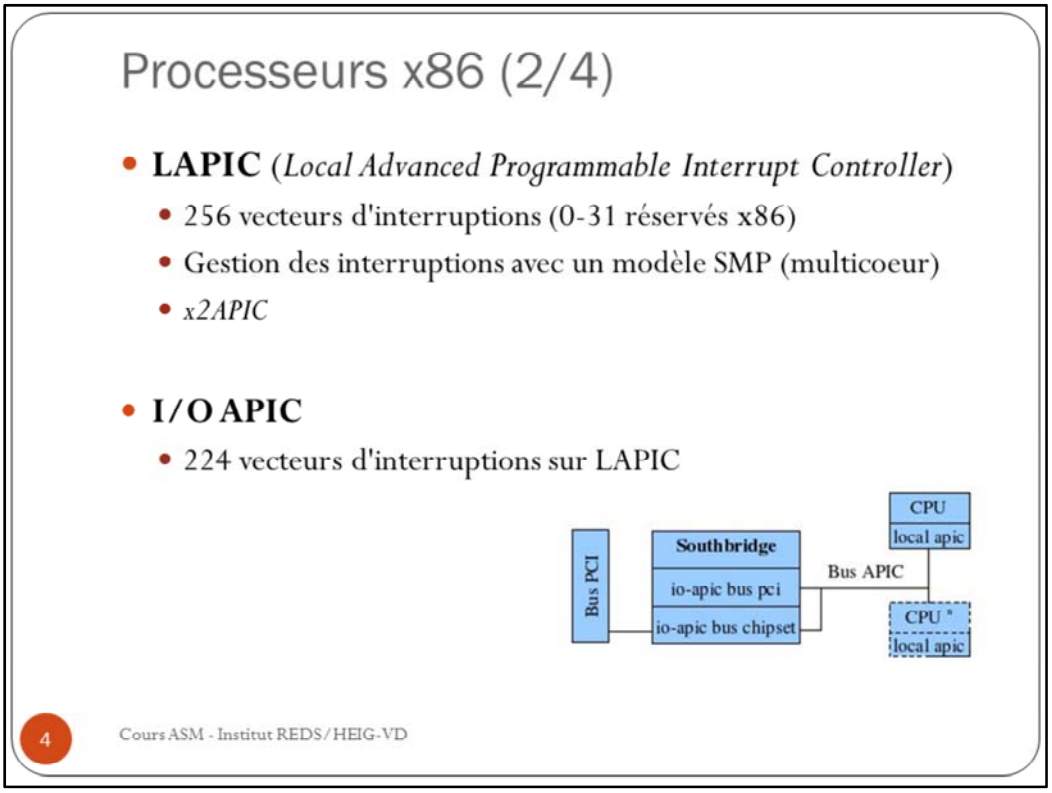
Ce chapitre introduit les spécificités des processeurs implémentant les jeux d'instructions compatibles avec ces deux langages d'assemblage, et donne également une brève introduction au langage assembleur.



Les familles des processeurs de type *x86* ont une architecture *CISC* avec des instructions de taille variable pouvant aller de 1 à 17 bytes. Elles sont bâties sur des microarchitectures de type *x86* avec un souci permanent de préserver une compatibilité ascendante: ainsi, les générations successives de processeurs admettent plusieurs modes de fonctionnement, qui diffèrent en particulier du point de vue de l'accès à la mémoire.
L'architecture fortement *pipelinée* des processeurs *x86* (pouvant aller jusqu'à plus de 30 étages de *pipeline*) permet à plusieurs instructions d'être exécutées en parallèle; ce type d'architecture est également nommée *superscalaire*. Bien entendu, aujourd'hui, tous les processeurs de type *x86* (32/64 bits) sont multicoeurs.

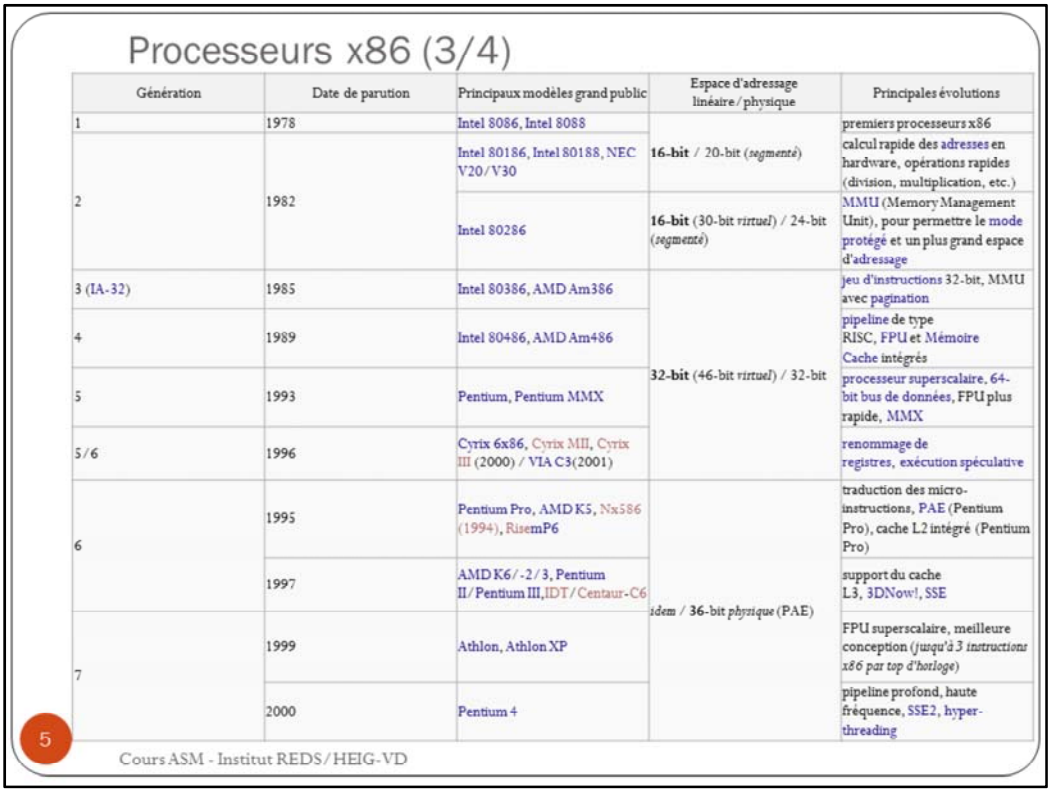
Une particularité intéressante au niveau des instructions est la technologie *SIMD (Simple Instruction Multiple Data)*. C'est une technique largement utilisé dans les processeurs & microcontrôleurs permettant d'appliquer des opérations (arithmétiques) sur un vecteur de données, et cela durant le même cycle d'exécution de l'instruction (donc en parallèle).

L'unité matérielle de gestion mémoire *(Memory Management Unit*) sur *x86* est très complexe: elle permet de faire de la segmentation (premières générations de processeur *x86*) et de la pagination. Pour assurer la compatibilité, la traduction d'une adresse virtuelle sur *x86* passe obligatoirement par une traduction de l'adresse virtuelle vers une adresse linéaire (via les segments), puis de l'adresse linéaire vers l'adresse physique.



Un processeur x86 peut gérer plusieurs sources d'interruptions: le contrôleur primaire d'interruption sur les dernières versions peut gérer jusqu'à 256 vecteurs; ces vecteurs sont utilisés pour des interruptions de type synchrones (interruption logicielle) ou de type asynchrone (interruption matérielle) ou **IRQ** *(Interrupt Request).* A cela s'ajoute un (voire plusieurs) contrôleur d'interruption externe comme le contrôleur "*I/O APIC*" qui occupe une ligne du LAPIC, et peut lui-même gérer 224 sources supplémentaires: il s'agit dans ce cas d'IRQs liées aux périphériques externes.

Dans une architecture multicoeur, chaque cœur dispose de son contrôleur *LAPIC* et un contrôleur maître sera responsable de *dispatcher* l'interruption sur un cœur disponible (cas général pour une architecture de type SMP – *Symmetric MultiProcessing)*, chaque cœur pouvant traiter indépendamment tout type d'interruption externe (la routine de service associée est implantée en mémoire et tous les cœurs ont accès à cette *même* mémoire).



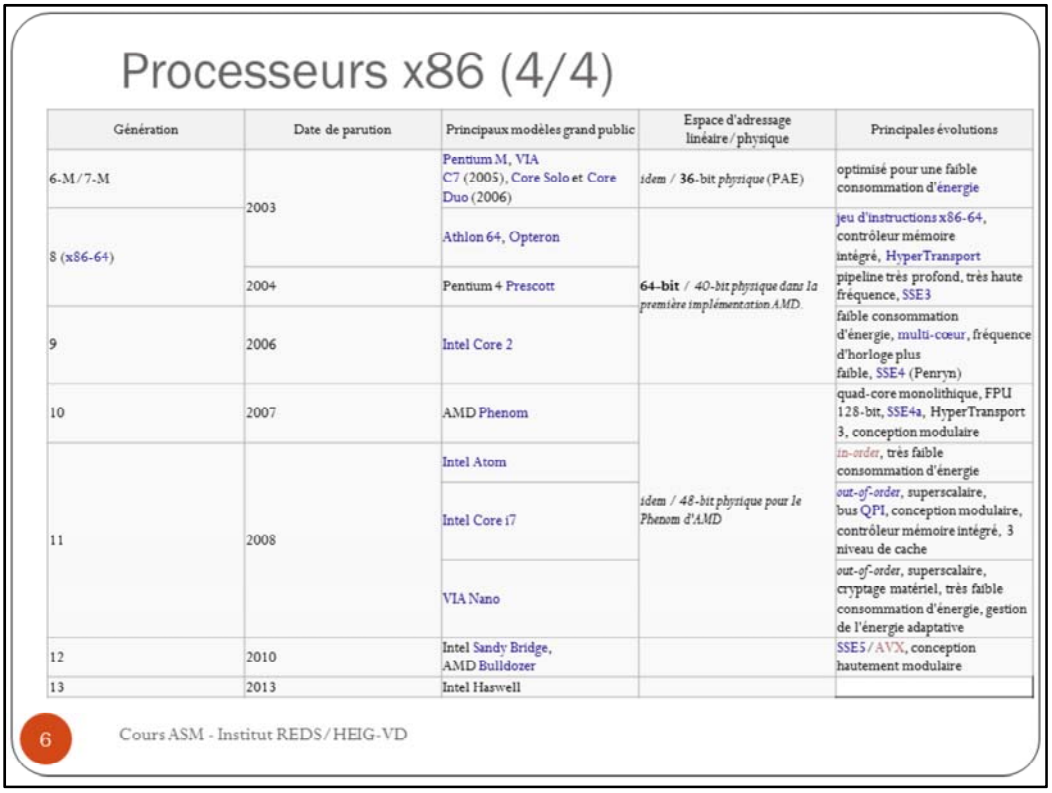
Processeurs x86 (3/4)

Génération	Date de parution	Principaux modèles grand public	Espace d'adressage linéaire / physique	Principales évolutions
1	1978	Intel 8086, Intel 8088	**16-bit** / 20-bit (*segmenté*)	premiers processeurs x86
2	1982	Intel 80186, Intel 80188, NEC V20 / V30		calcul rapide des adresses en hardware, opérations rapides (division, multiplication, etc.)
		Intel 80286	**16-bit** (30-bit *virtuel*) / 24-bit (*segmenté*)	MMU (Memory Management Unit), pour permettre le mode protégé et un plus grand espace d'adressage
3 (IA-32)	1985	Intel 80386, AMD Am386	**32-bit** (46-bit *virtuel*) / 32-bit	jeu d'instructions 32-bit, MMU avec pagination
4	1989	Intel 80486, AMD Am486		pipeline de type RISC, FPU et Mémoire Cache intégrés
5	1993	Pentium, Pentium MMX		processeur superscalaire, 64-bit bus de données, FPU plus rapide, MMX
5/6	1996	Cyrix 6x86, Cyrix MII, Cyrix III (2000) / VIA C3(2001)		renommage de registres, exécution spéculative
6	1995	Pentium Pro, AMD K5, Nx586 (1994), RisemP6	*idem* / **36**-bit *physique* (PAE)	traduction des micro-instructions, PAE (Pentium Pro), cache L2 intégré (Pentium Pro)
	1997	AMD K6 / -2 / 3, Pentium II / Pentium III, IDT / Centaur-C6		support du cache L3, 3DNow!, SSE
7	1999	Athlon, Athlon XP		FPU superscalaire, meilleure conception (*jusqu'à 3 instructions x86 par top d'horloge*)
	2000	Pentium 4		pipeline profond, haute fréquence, SSE2, hyper-threading

5

Cours ASM - Institut REDS / HEIG-VD

Le tableau ci-dessus résume l'évolution de la technologie *Intel* au fil des ans. On remarquera l'apparition de caractéristiques importantes comme la présence d'un coprocesseur *MMX* (instructions multimédia, principalement de type *SIMD*), le support des 64 bits, les extensions *PAE* (possibilité d'adresser plus de 4 Go de mémoire physique), les jeux d'instructions (*SSE, SSE2, L3, 3DNow!,* etc.) ainsi que la technologie *hyperthreading* permettant l'exécution parallèle de plusieurs *threads*.



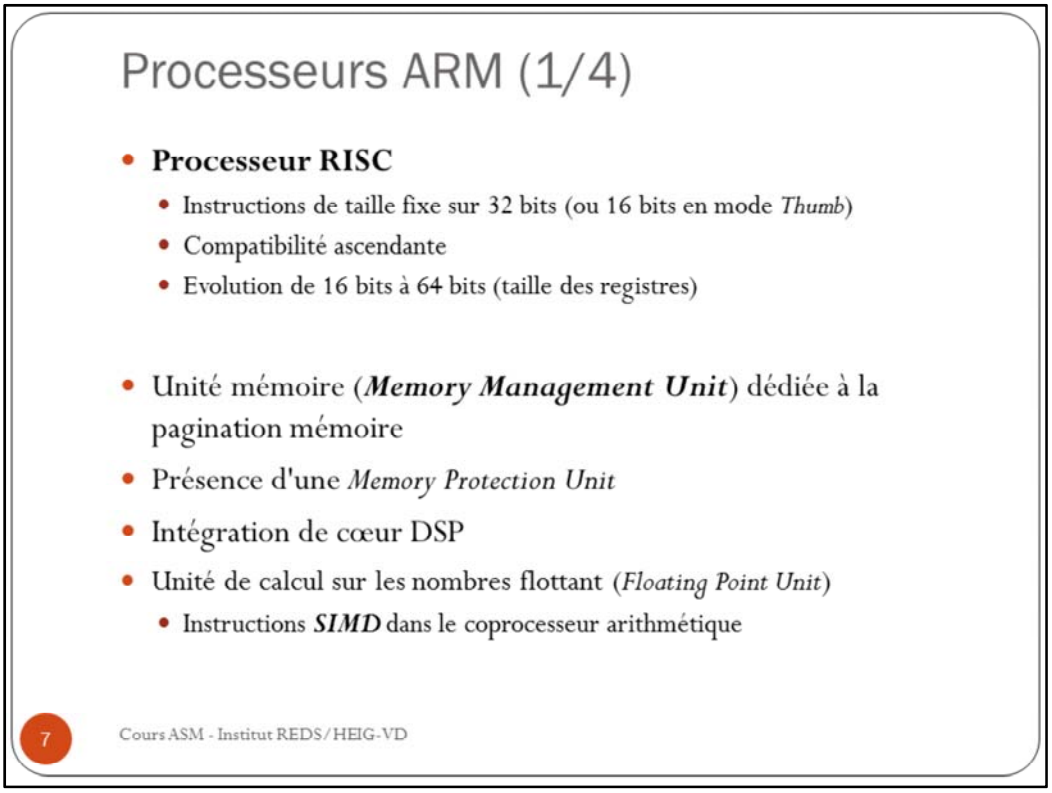
Processeurs x86 (4/4)

Génération	Date de parution	Principaux modèles grand public	Espace d'adressage linéaire / physique	Principales évolutions
6-M / 7-M	2003	Pentium M, VIA C7 (2005), Core Solo et Core Duo (2006)	*idem* / 36-bit *physique* (PAE)	optimisé pour une faible consommation d'énergie
8 (x86-64)		Athlon 64, Opteron	64-bit / *40-bit physique dans la première implémentation AMD.*	jeu d'instructions x86-64, contrôleur mémoire intégré, HyperTransport
	2004	Pentium 4 Prescott		pipeline très profond, très haute fréquence, SSE3
9	2006	Intel Core 2		faible consommation d'énergie, multi-cœur, fréquence d'horloge plus faible, SSE4 (Penryn)
10	2007	AMD Phenom	*idem / 48-bit physique pour le Phenom d'AMD*	quad-core monolithique, FPU 128-bit, SSE4a, HyperTransport 3, conception modulaire
11	2008	Intel Atom		*in-order*, très faible consommation d'énergie
		Intel Core i7		*out-of-order*, superscalaire, bus QPI, conception modulaire, contrôleur mémoire intégré, 3 niveau de cache
		VIA Nano		*out-of-order*, superscalaire, cryptage matériel, très faible consommation d'énergie, gestion de l'énergie adaptative
12	2010	Intel Sandy Bridge, AMD Bulldozer		SSE5 / AVX, conception hautement modulaire
13	2013	Intel Haswell		

6 Cours ASM - Institut REDS/HEIG-VD

Au début des années 2000, *Intel* a évolué fortement sur le plan *marketing* en renommant les familles de processeurs (*Athlon, Opteron, Intel Core 2, etc.)*, ainsi que les noms de leurs microarchitectures (non montrés dans ce tableau). Des exemples de microarchitectures sont *Pentium Pro, P6, Nehalem, Sandy Bridge, Haswell, Skylake, Larrabee, Itanium, etc.*).

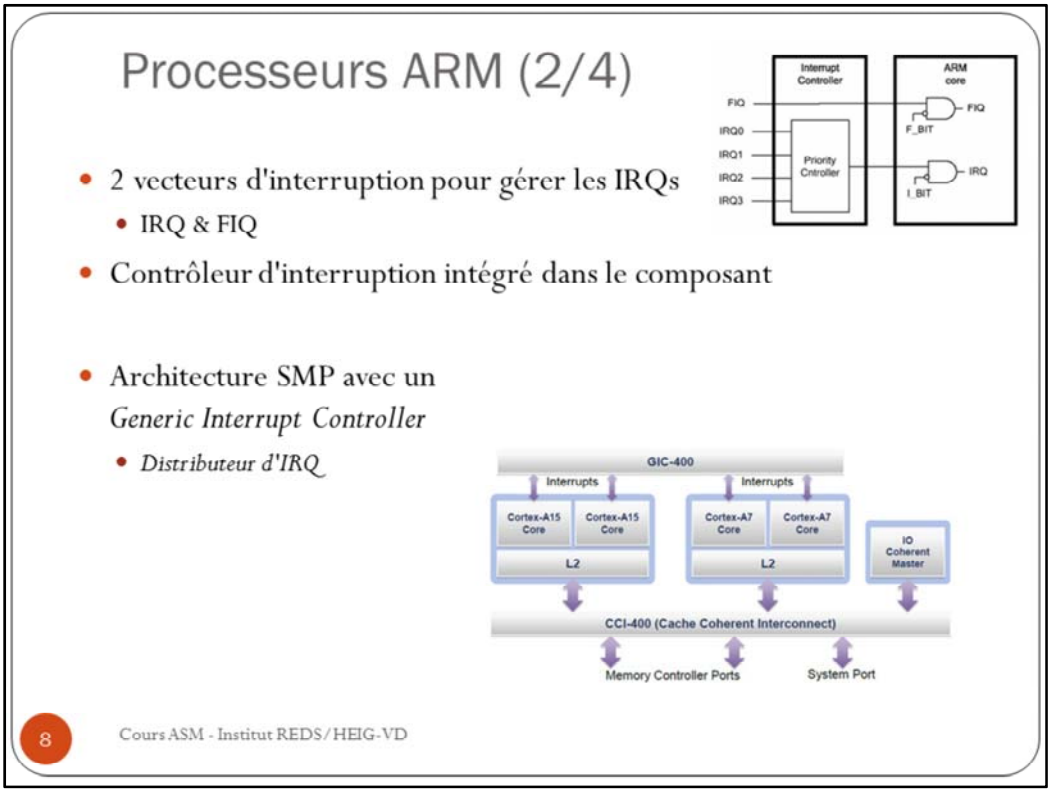
L'évolution de ces technologies s'est accompagné d'une concurrence féroce, notamment avec la présence de la firme ***AMD***, principal rival d'Intel. *AMD* a développé ses propres familles de microprocesseurs (réputées nettement moins cher *qu'Intel*), tout en implémentant des microarchitectures *x86*. Les autres concurrents sont le coréen *Samsung*, l'américain *Texas Instruments*, le japonais *Toshiba* et la société franco-italienne *STMicroelectronics*.



Les processeurs de type *ARM* sont principalement utilisés dans le contexte des systèmes embarqués au sens large. Le processeur, de type *RISC*, est beaucoup plus "simple" qu'un processeur *CISC*. En revanche, il existe une multitude de contrôleurs intégré au sein même du composant électronique *(chipset)*. Les instructions sont en principe de taille fixe (soit 16 ou 32 bits). La plupart des instructions s'exécutent en un seul cycle d'horloge.

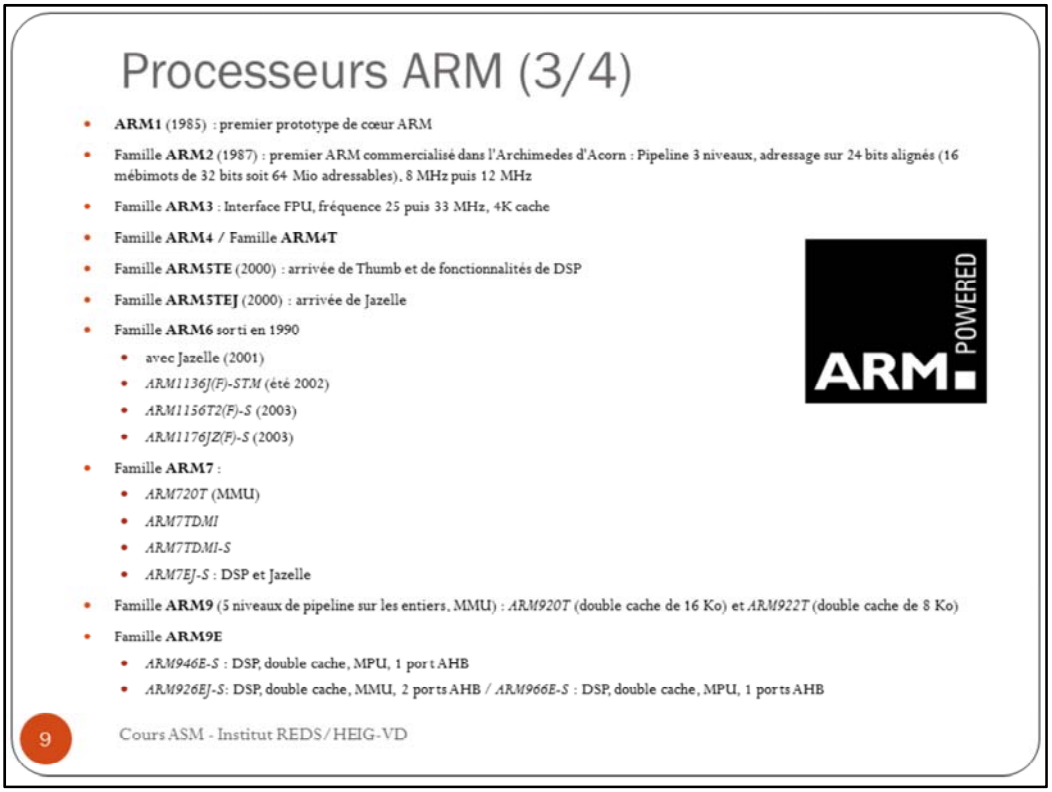
Il est fréquent aujourd'hui d'avoir plusieurs coprocesseurs arithmétiques offrant des instructions de type SIMD, ainsi qu'un DSP *(Digital Signal Processing)* afin d'offrir une puissance de calcul élevée tout en maintenant une consommation relativement basse.

Il n'y a pas de segmentation mémoire comme avec un processeur *x86*. Seule la pagination est supportée par la *MMU*. En revanche, il est possible de *mapper* des zones mémoires de tailles différentes.

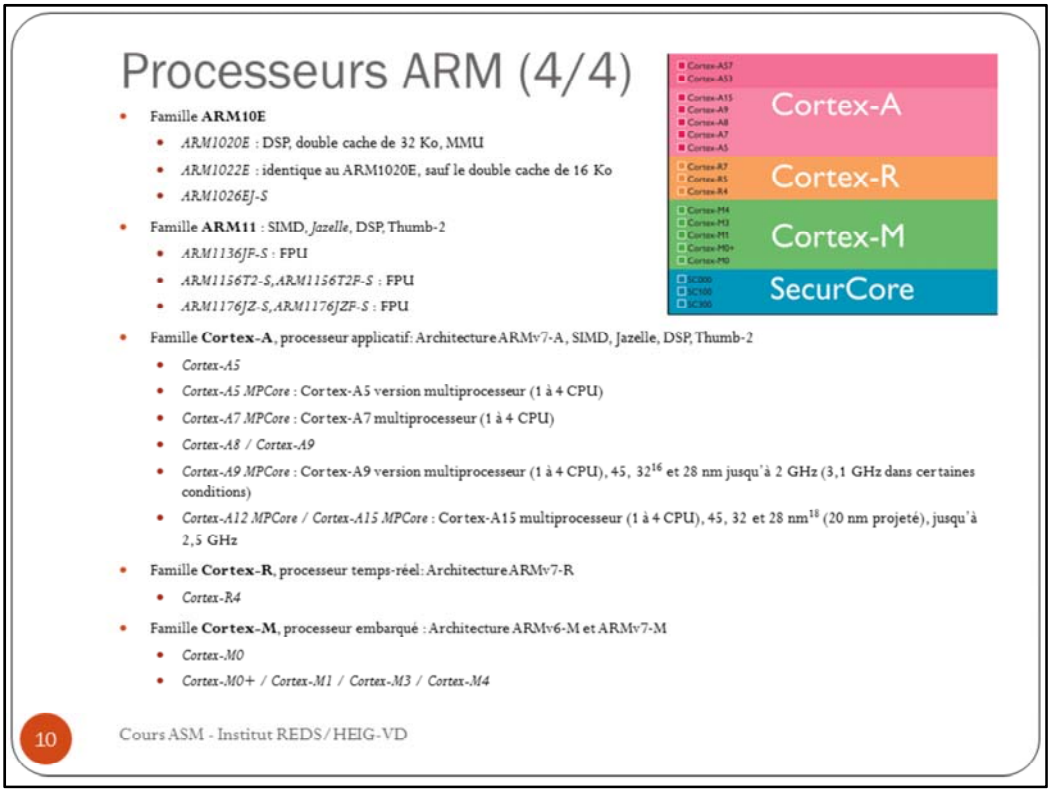


La gestion des interruptions sur un processeur *ARM* est assez différente que sur un processeur *x86*. Il n'y a en fait que deux lignes physiques qui sont reliées du contrôleur d'interruption vers le cœur de processeur, à savoir les lignes ***IRQ*** et ***FIQ*** (pour *Fast IRQ*, celle-ci étant par ailleurs peu utilisée). Il n'y a donc que deux vecteurs d'interruption associés aux interruptions matérielles (avec bien entendu d'autres vecteurs pour les interruptions logicielles/exceptions).

Au niveau de la gestion des IRQs, une architecture multicoeur basée sur *ARM* est relativement similaire à une architecture *x86*: un contrôleur primaire (appelé aussi *distributeur*) est responsable de *dispatcher* les interruptions aux contrôleurs locaux à chaque cœur.

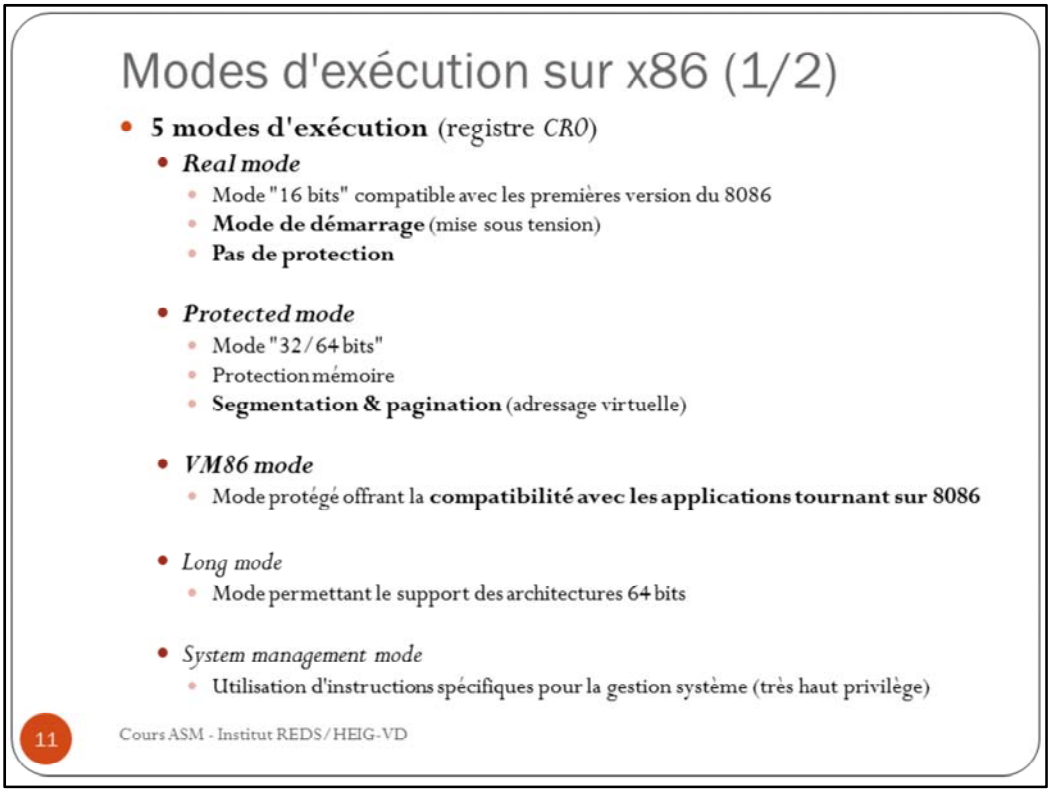


L'évolution des processeurs *ARM* est décrite ci-dessus. On remarque des caractéristiques propre à ce genre de composant, à savoir un mode *Thumb* pour des instructions 16 bits, un coprocesseur *Jazelle* normalement capable d'exécuter directement du *bytecode Java* (peu voire pas documenté, donc quasiment pas utilisé!), un *DSP*, un mécanisme matériel de protection mémoire (MPU) en plus de la MMU, etc.



Les familles *Cortex* constituent une des familles de processeur *ARM* les plus en vogue aujourd'hui. Elles se déclinent en fonction des types d'application (temps-réel, multimédia, sécurité, etc.).

Les dernières versions de *Cortex (Cortex-A15* notamment) sont multicoeurs (jusqu'à 4 cœurs) et supportent la virtualisation, du point de vue de l'architecture matérielle et du jeu d'instructions. Les performances de ces processeurs (> 2 Hz) se rapprochent de plus en plus des processeurs *x86* tout en consommant moins d'énergie et en ayant une architecture évolutive moins lourde qu'un *x86*. Ils constituent donc une concurrence sérieuse à *Intel*, pour les serveurs notamment.

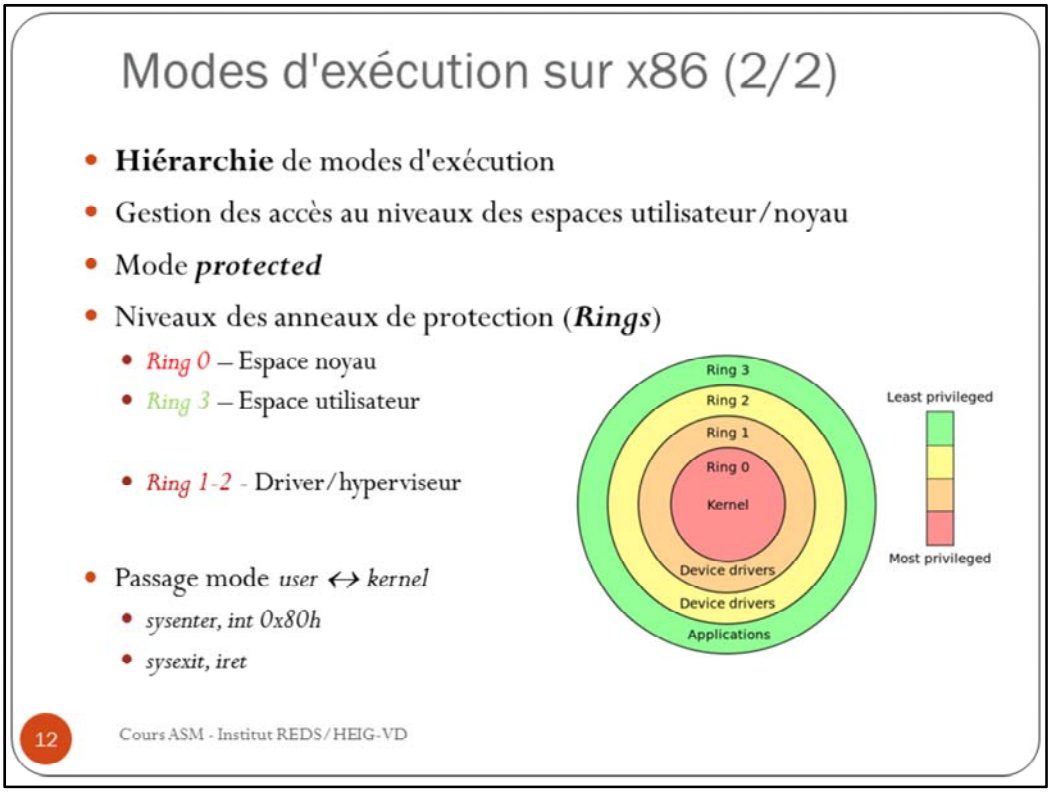


Les processeurs *x86* offrent 3 modes de base: ***real mode***, ***protected mode*** et ***vm86 mode*****.**

Le **mode réel** (premier mode) n'offre aucune protection et était utilisé dans les premières générations de processeur, alors que la mémoire ne pouvait dépasser 1 Mo, du fait que seule 20 lignes d'adresse étaient présentes (2^{20} = 1 Mo). Les registres de segment, comme nous le verrons plus tard, étant **implémentés sur 16 bits**, une adresse mémoire était composée d'un numéro de segment, puis d'un ***offset*** codé dans un autre registre de 16 bits. Le numéro de segment correspond alors aux bits de poids fort de l'adresse, et le déplacement ne pouvait être codé que sur 4 bits (on remarque la possibilité d'avoir un chevauchement (*overlap*) des segments).

Le **mode protégé** change considérablement la manière d'utiliser les registres de segment: ils ne contiennent plus les bits de poids forts, mais un index dans une table de segment qui donne une adresse physique contenant le début du segment. Le déplacement cette fois-ci peut être codé sur 16 bits si les registres sont de 16 bits, ou **32 bits** avec des registres 32 bits. Avec le mode protégé, il est possible d'utiliser la pagination. Aujourd'hui, seule la pagination est utilisée et les segments se réduisent au nombre d'un seul pour chaque type, avec un ***offset*** **de 2^{32}** (ce qui permet bien de gérer un espace linéaire de **4 Go**).

Nous nous étendrons pas davantage sur les autres modes.

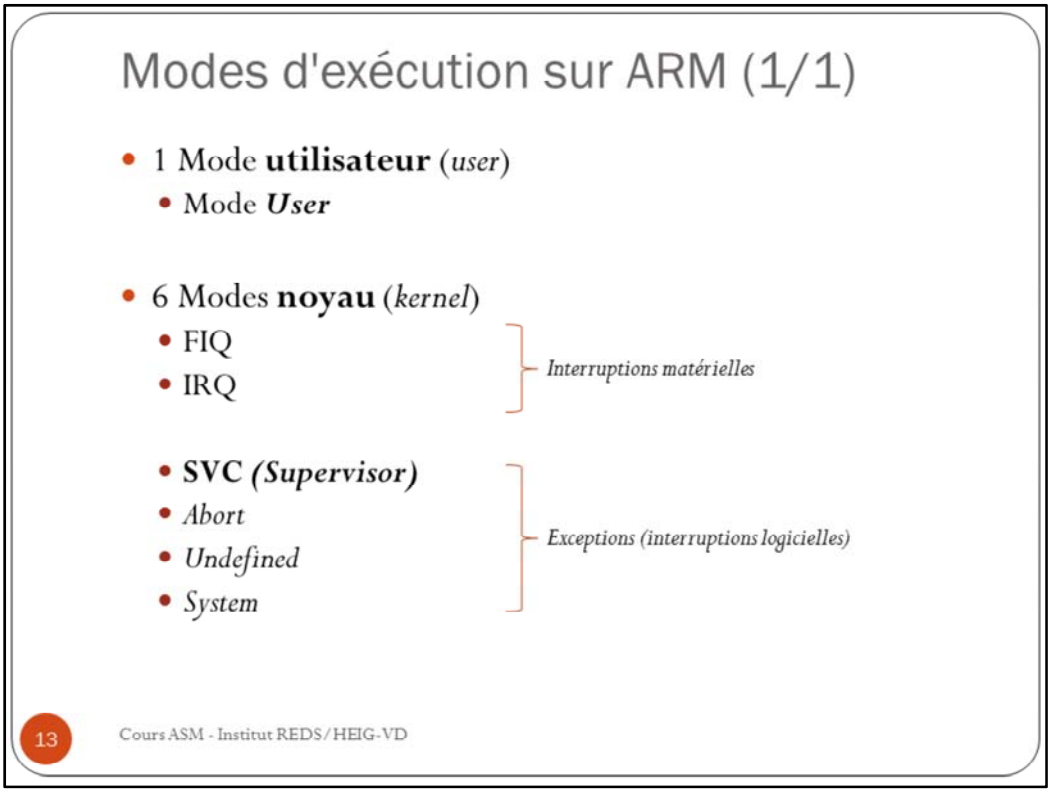


Lorsqu'un processeur *x86* démarre, il est dans le mode "*real mode*". Très rapidement, le code logiciel (en principe du *bootloader*, voire de l'OS, va implanter une table de segment avec un minimum d'information afin de passer en mode protégé, et de disposer ainsi d'un adressage ad-hoc. L'OS pourra par la suite configurer les tables de page afin d'activer la pagination.

Dans le mode protégé, il faut encore pouvoir distinguer entre le mode utilisateur (*user mode*) et le mode noyau *(kernel mode)*, afin que tout OS moderne puisse gérer correctement la sécurité entre l'espace utilisateur et l'espace noyau.

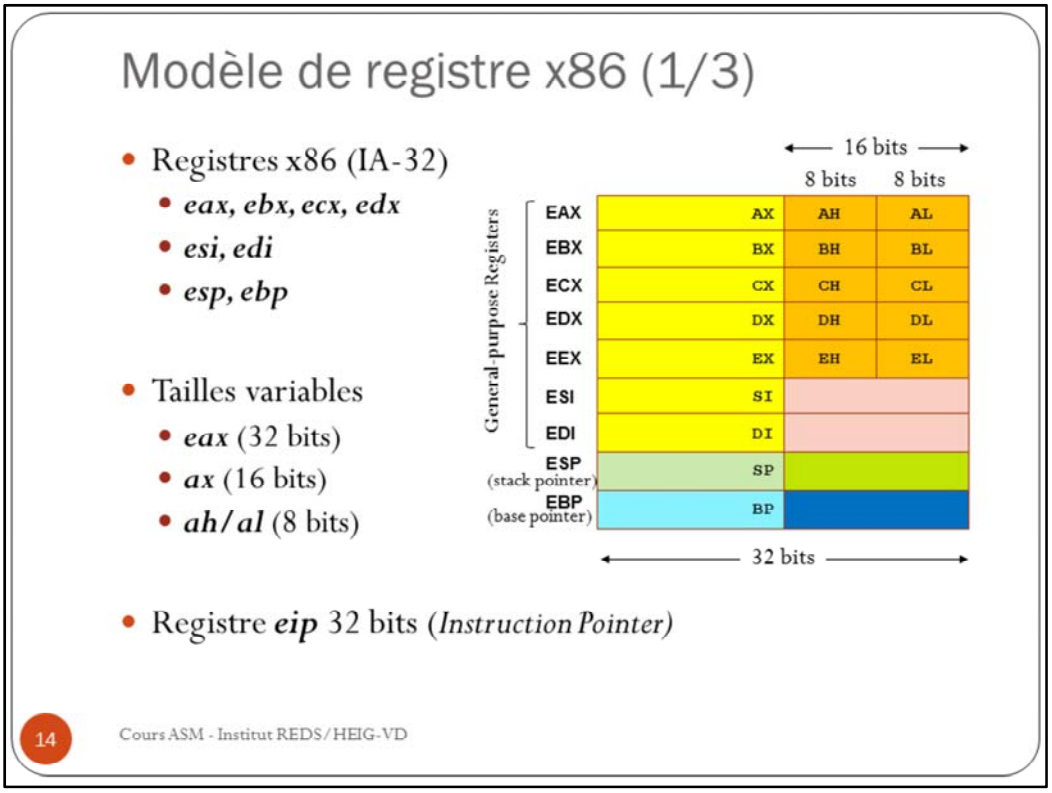
C'est la notion **d'anneau de protection** - ou ***ring*** - qui permettra de faire la différence. Généralement, dans un environnement non virtualisé, seul deux *rings* sont utilisés: le *ring 0* pour le code noyau et le *ring 3* pour l'espace utilisateur.

Le *bootloader* reste en principe dans le *ring 0*. C'est le système d'exploitation, lors du *bootstrap*, qui commutera en mode utilisateur lorsque toute l'initialisation au niveau noyau sera terminée.



Un processeur ARM dispose de **7 modes d'exécution**: un mode ***user*** et 6 modes ***privilégiés*** (ou noyau) qui seront utilisés dans des contextes différents. Les modes *noyau* sont associés à chaque interruption de type matérielle ou logicielle pouvant survenir durant l'exécution. De plus, un mode *system* est utilisé au démarrage du processeur qui permet la manipulation du registre d'état servant à changer de mode. En principe, le moniteur ou l'OS utilisera le mode *SVC.*

L'utilisation de ces modes sera détaillée plus loin dans ce document lors de la présentation du modèle de registre sur *ARM*.



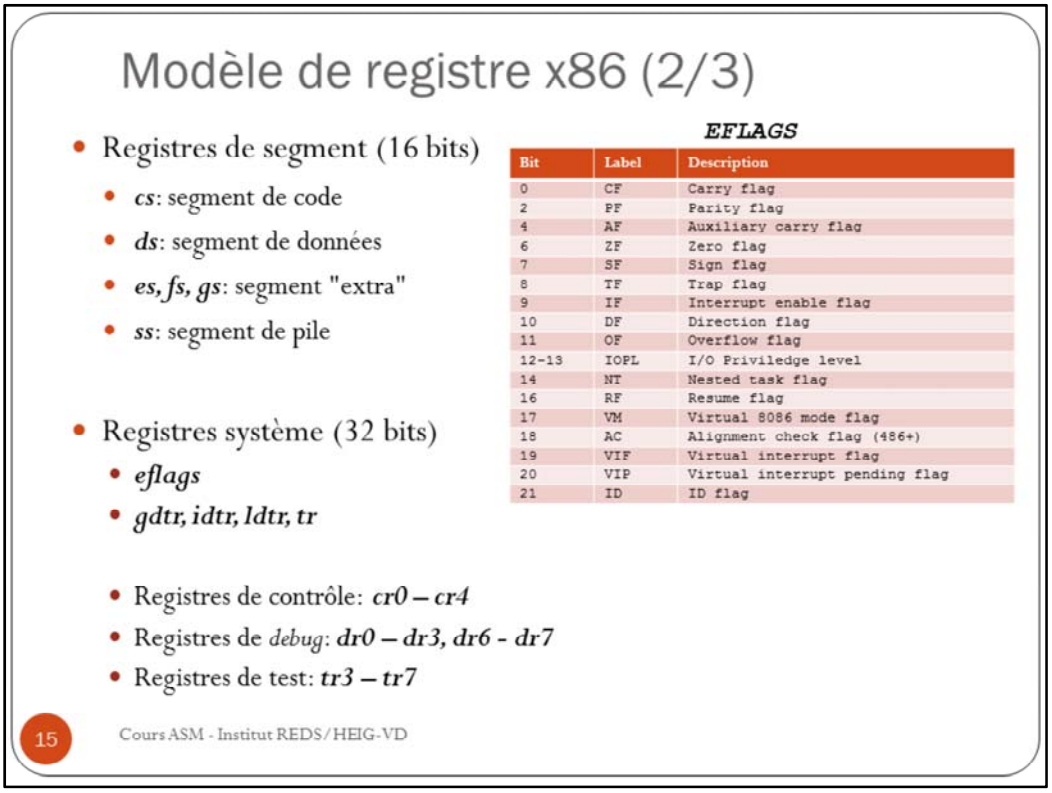
Dans le cours ASM, nous ne considèrerons que les architectures *x86* **32 bits** (et non 64 bits).

Le processeur *x86* dispose de 8 registres généraux dont 4 peuvent être facilement accessibles en 8, 16 ou 32 bits, selon le modèle de la figure ci-dessus.

Les registres *eax, ebx, ecx, edx* sont utilisés librement pour stocker n'importe quelle valeur et peuvent être utilisés comme opérandes d'une instruction.

Les registres ***esi*** et ***edi*** sont plutôt utilisés dans le contexte d'adressage par indexation (index d'une table, d'une zone mémoire, etc.), mais du point de vue du processeur, ils n'ont pas de signification particulière. C'est le cas aussi pour le registre ***bp***, qui est généralement utilisé pour préserver une référence dans la pile. En revanche, le registre ***sp*** est réservé pour stocker le pointeur de pile et peut être altéré de manière intrinsèque avec certaines instructions, comme nous le verrons plus tard. De plus les registres *bp* et *sp* sont associés au segment utilisé pour la pile (registre *SS*).

Le registre ***eip*** contient l'adresse de la prochaine instruction à exécuter (ce registre correspond au *Program Counter (PC)* d'autres processeurs/microcontrôleurs). Il ne peut être altéré directement, mais est mis à jour lors d'instructions de saut.



Modèle de registre x86 (2/3)

- Registres de segment (16 bits)
  - *cs*: segment de code
  - *ds*: segment de données
  - *es, fs, gs*: segment "extra"
  - *ss*: segment de pile
- Registres système (32 bits)
  - *eflags*
  - *gdtr, idtr, ldtr, tr*
  - Registres de contrôle: *cr0 – cr4*
  - Registres de *debug*: *dr0 – dr3, dr6 - dr7*
  - Registres de test: *tr3 – tr7*

EFLAGS

Bit	Label	Description
0	CF	Carry flag
2	PF	Parity flag
4	AF	Auxiliary carry flag
6	ZF	Zero flag
7	SF	Sign flag
8	TF	Trap flag
9	IF	Interrupt enable flag
10	DF	Direction flag
11	OF	Overflow flag
12-13	IOPL	I/O Priviledge level
14	NT	Nested task flag
16	RF	Resume flag
17	VM	Virtual 8086 mode flag
18	AC	Alignment check flag (486+)
19	VIF	Virtual interrupt flag
20	VIP	Virtual interrupt pending flag
21	ID	ID flag

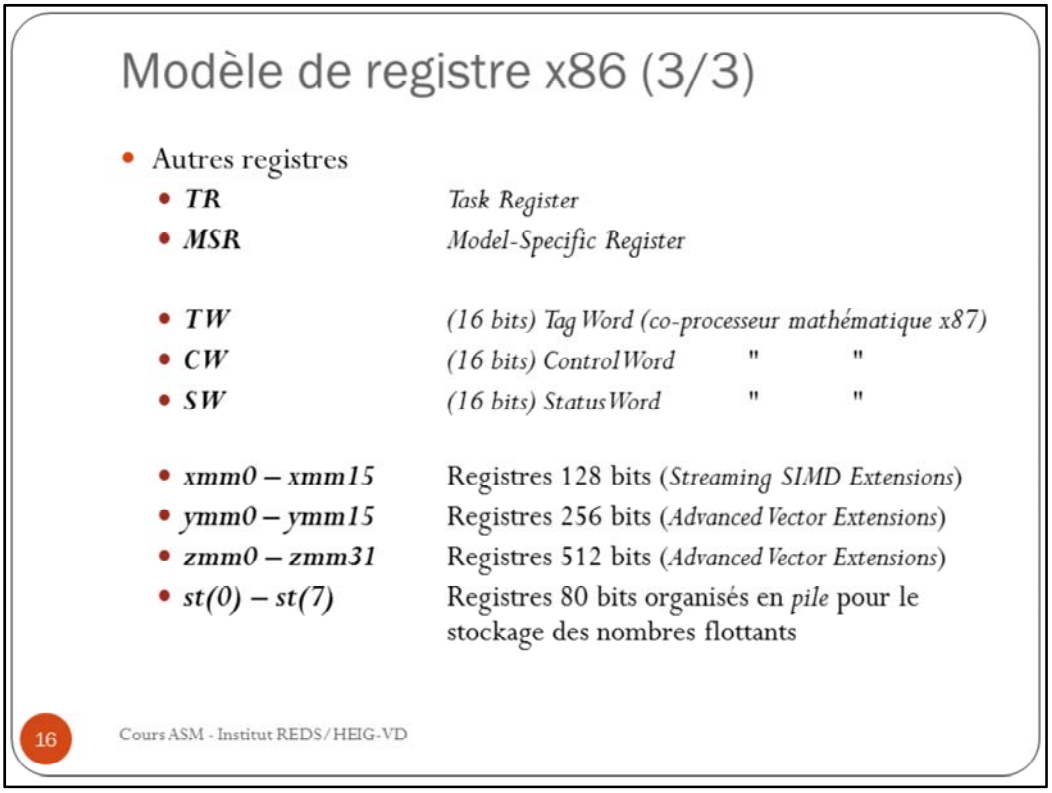
15 Cours ASM - Institut REDS/HEIG-VD

Sur *x86*, les registres de segment interviennent dans le mécanisme d'adressage comme il a déjà été évoqué brièvement sur la partie des modes d'exécution. Dans le mode réel, il constitue les bits de poids forts de l'adresse physique, dans le mode protégé, ils identifient le descripteur de segment intervenant dans la traduction de l'adresse virtuelle vers l'adresse physique. Dans les architectures modernes, ces registres peuvent référer à un et un seul segment correspondant à l'espace mémoire complet. L'*offset* dont la valeur se trouve dans les registres généraux lors de l'accès mémoire pourra ainsi aller de *0x0* à *0xffffffff* (2^{32} - 1).

Un registre très important est le registre ***eflags***. Celui-ci contient les *flags* mis à jour par les instructions arithmétiques/logiques, et contient également d'autres informations relative au mode d'exécution et aux interruptions.

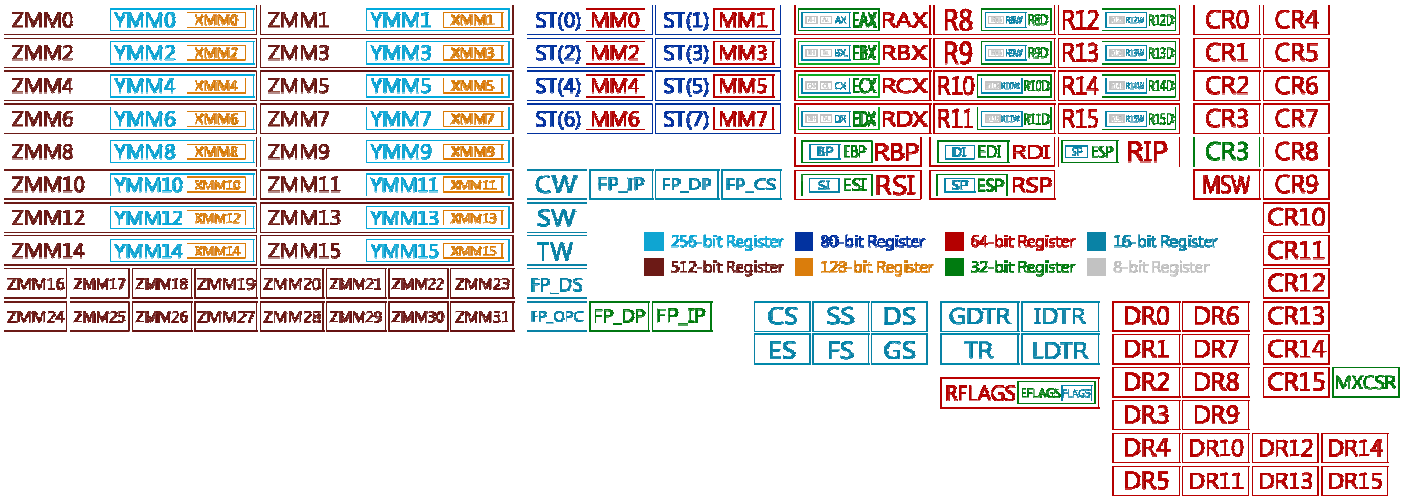
Le processeur *x86* dispose également d'autres registres systèmes dont les plus importants sont les suivants:

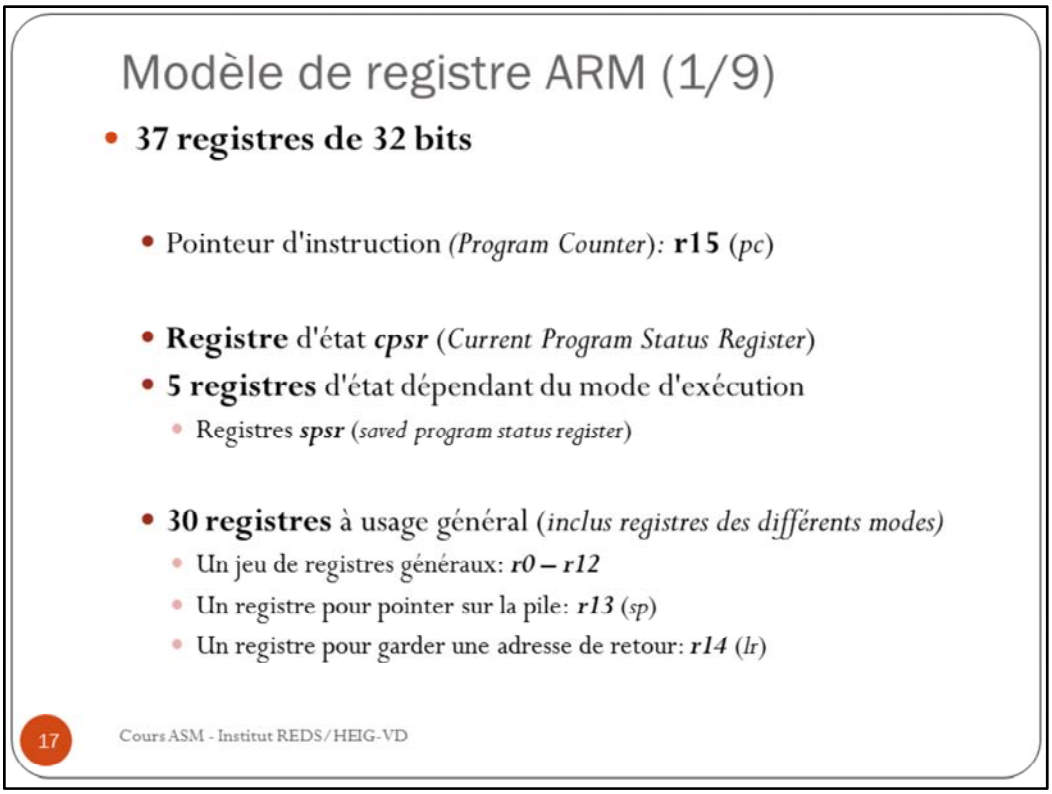
- ***gdtr, ldtr*** *(Global/Local Descriptor Table Register*): ces registres sont utilisés pour stocker l'adresse de la table (globale/locale) des segments
- ***idtr*** *(Interrupt Descriptor Table Register): c*e registre contient l'adresse de la table des vecteurs d'interruption
- ***cr0*** *(Control Register 0): c*e registre contient notamment les bits d'activation/désactivation de la pagination *(cr3* contient l'adresse de la table de pages).



D'autres registres système et principalement liés aux coprocesseurs mathématiques et multimédias sont présentés ci-dessous. Il est intéressant de remarquer la variabilité de la taille des registres, ainsi que la présence de nombreux registres dédiés au traitement des instructions *SIMD*.

Il est à noter également que pour les registres "traditionnels" du *x86*, la forme 64-bits existe avec le préfixe *r* (*rax, rip, etc.*).

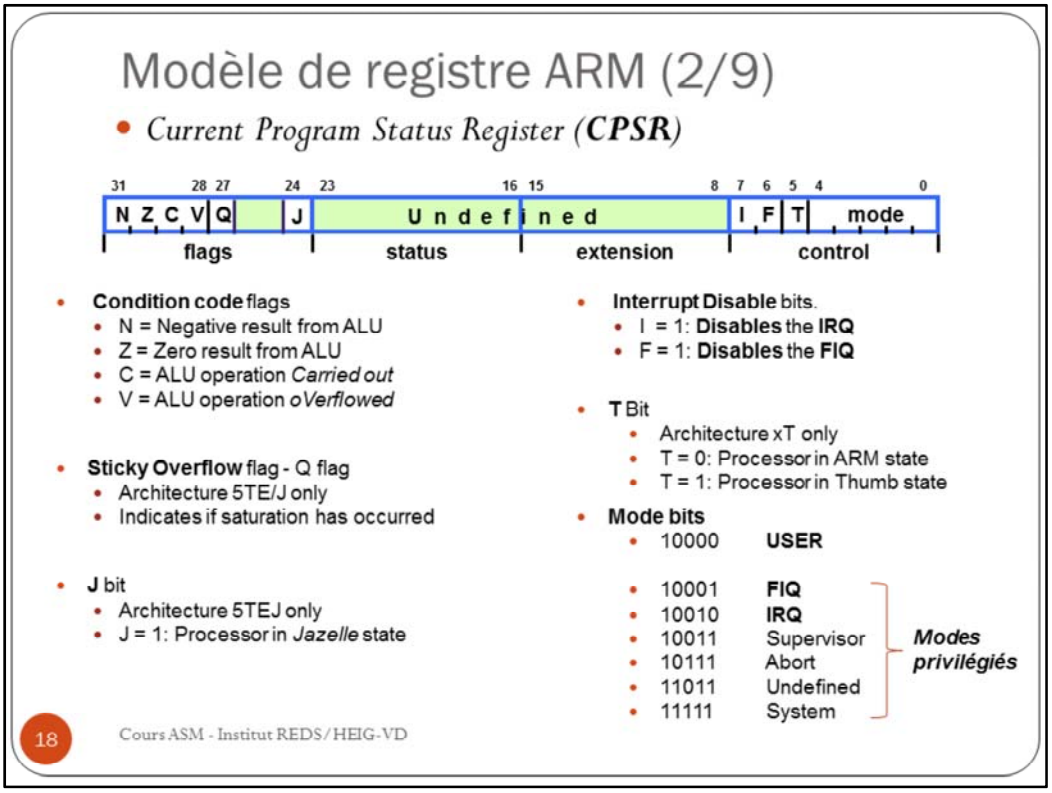




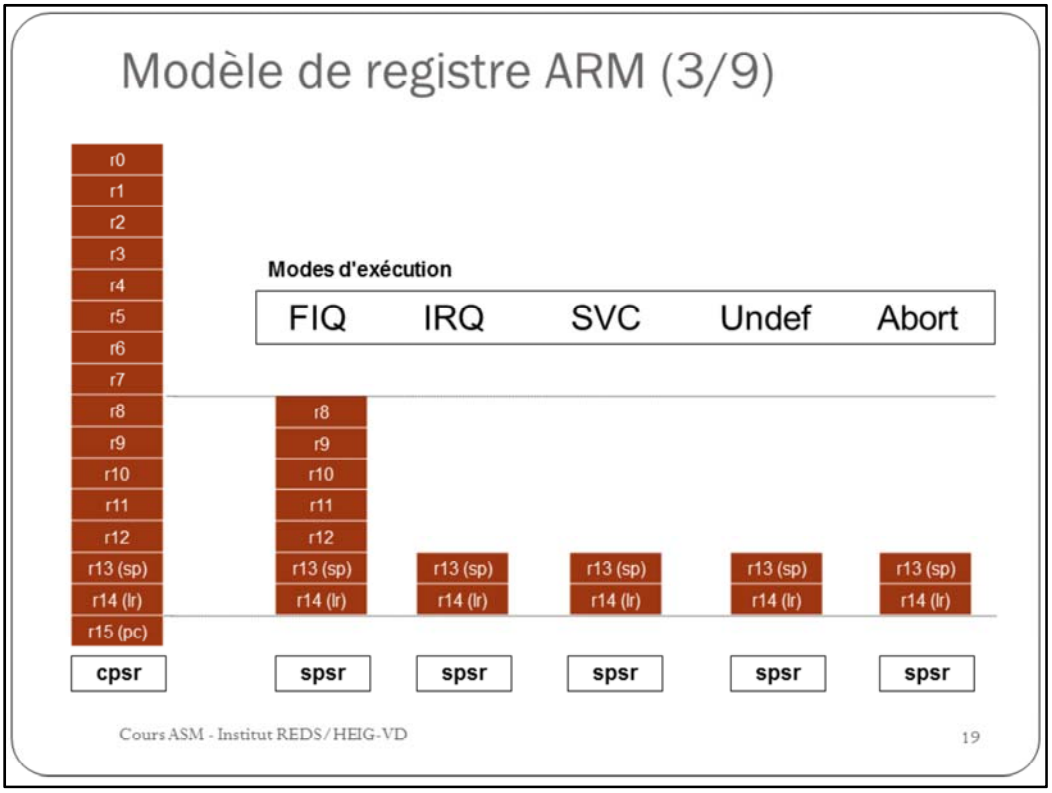
A l'inverse du *x86*, le processeur ARM dispose de beaucoup plus de registres généraux (sans signification particulière du point de vue du processeur) dont leur utilisation dépende directement du programmeur qui peut suivre des conventions ou non. Deux registres cependant ont une signification particulière: ***lr*** et ***pc***, en plus des registres d'état ***cpsr*** *et* ***spsr***.

Par convention, certains registres ARM prennent des significations particulières et possèdent dès lors des *alias* connus par le compilateur d'assemblage et les désassembleurs. Il s'agit des registres suivants:

r10: ***SL*** (*Stack Limit*)
r11: ***FP*** (*Frame Pointer*)
r12: ***IP*** (*Scratch Register*)
r13: ***SP*** (*Stack Pointer*)
r14: ***LR*** (*Link Register*)
r15: ***PC*** (*Program Counter*)



Le registre ***cpsr*** correspond au registre ***eflags*** du *x86*. Il contient l'état des bits liés à l'ALU et comporte des bits *système* indiquant le mode d'exécution du processeur, l'état d'activation des interruptions (*IRQ/FIQ*), et d'autres bits spécifiques au modèle de processeur et liés au jeu d'instructions; par exemple, le mode *Thumb* est déterminé par un bit de ce registre.

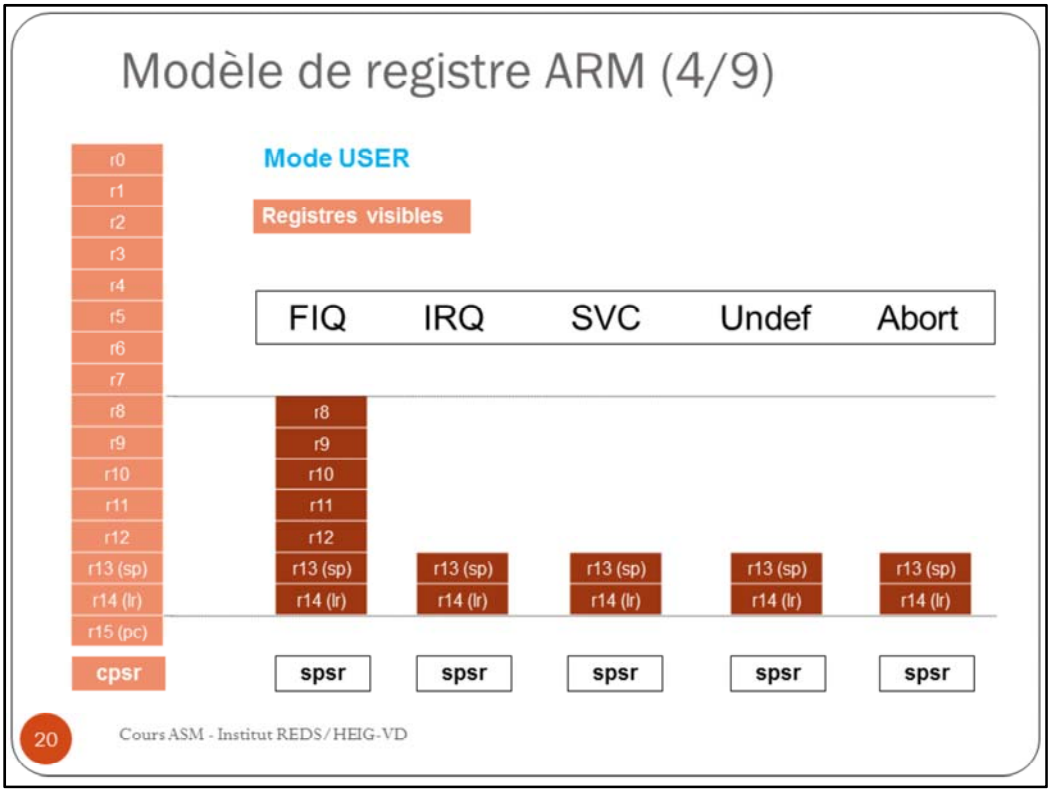


Certains registres *ARM* sont *répliqués* physiquement dans les différents modes d'exécution. Par exemple, les registres ***r13*** (***sp***) et ***r14*** (***lr***) sont *physiquement* différents d'un mode à l'autre. En revanche, il n'existe pas au niveau du langage des noms différents pour les différencier. Cela complique quelque peu la programmation, car lorsque l'on utilise ces registres, il est important de connaître le mode d'exécution du processeur.

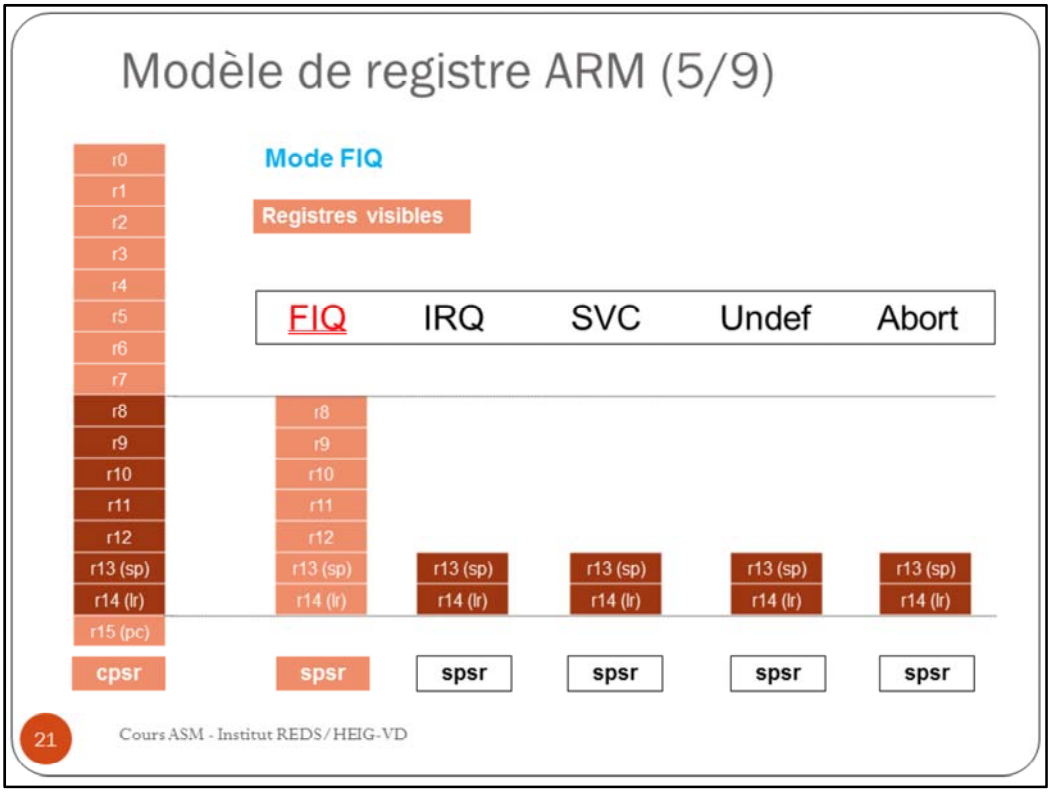
La réplication de registres dans les différents modes permet de *préserver* les valeurs de registre d'un mode à l'autre et diminue ainsi l'utilisation de la pile.

Lorsque l'on passe d'un mode à l'autre, le programmeur manipule donc des ***fenêtres de registres*** différentes, chaque fenêtre étant propre au mode d'exécution.

Les différentes fenêtres de registres sont examinées en détail dans les pages suivantes.

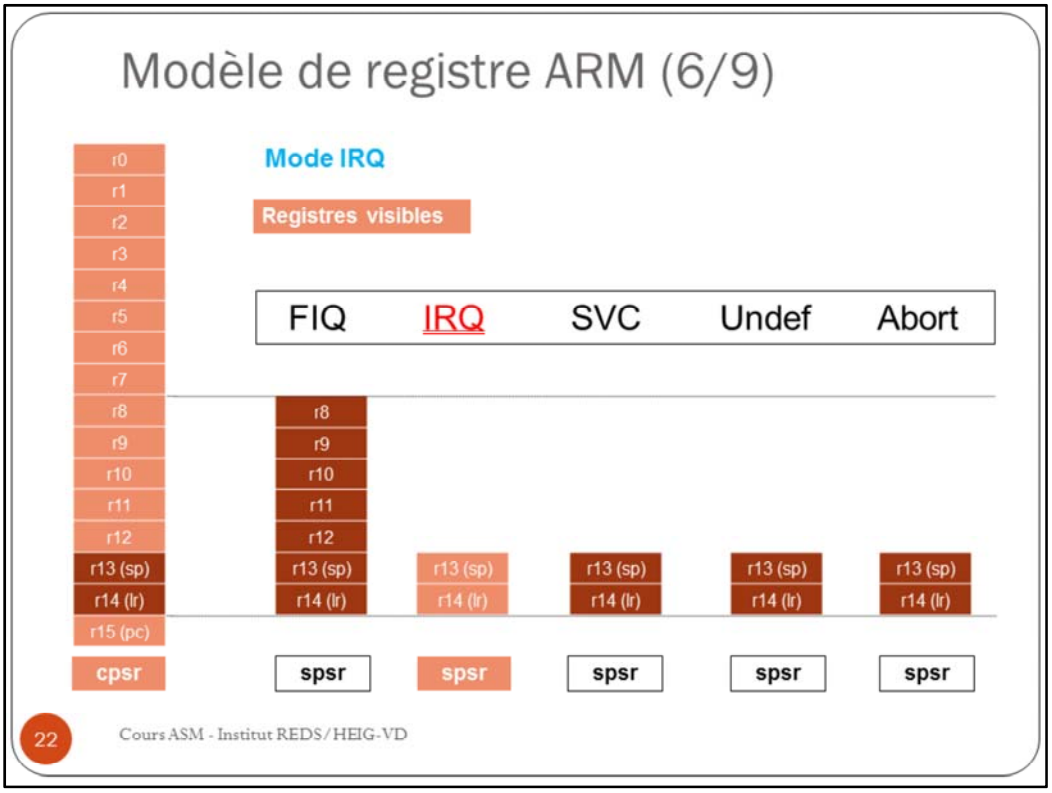


Le mode ***user*** correspond au mode utilisé par le système d'exploitation lorsque le code s'exécute dans l'espace utilisateur. Le registre d'état est utilisé lors d'opérations arithmétiques par exemple, mais il ne peut être accédé directement; les instructions manipulant le registre d'état (*cpsr*) sont des instructions privilégiées.



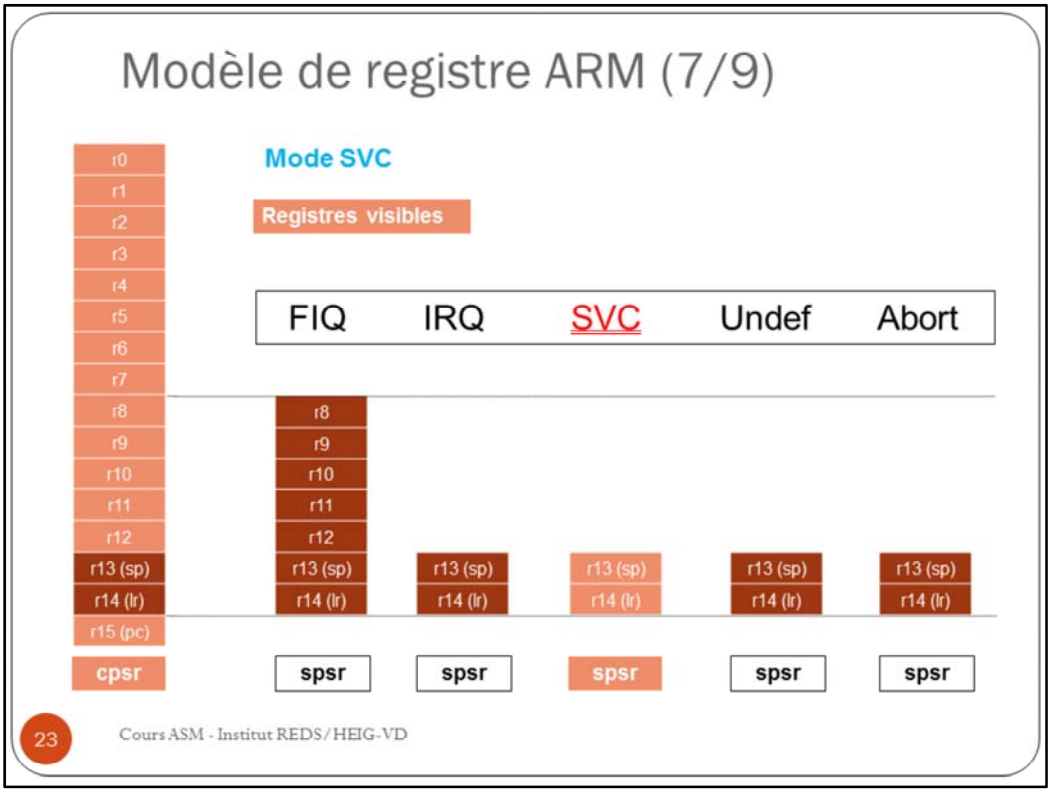
Le mode ***FIQ*** est utilisé lors d'une interruption matérielle de **haute priorité**. Le cœur ARM dispose en effet de deux lignes d'interruption matérielle, l'une étant réservée pour le type d'interruption FIQ (l'utilisation des deux lignes est sous la gestion du contrôleur d'interruption).

Comme le traitement doit être très rapide, ce mode dispose de registres ***r8-r14*** différents. Ces registres (notamment de *r8* à *r12*) peuvent donc être utilisés directement dans la routine de service liée à cette interruption, **sans avoir besoin** de préserver les valeurs courantes sur la pile. On évite ainsi des transferts mémoire.

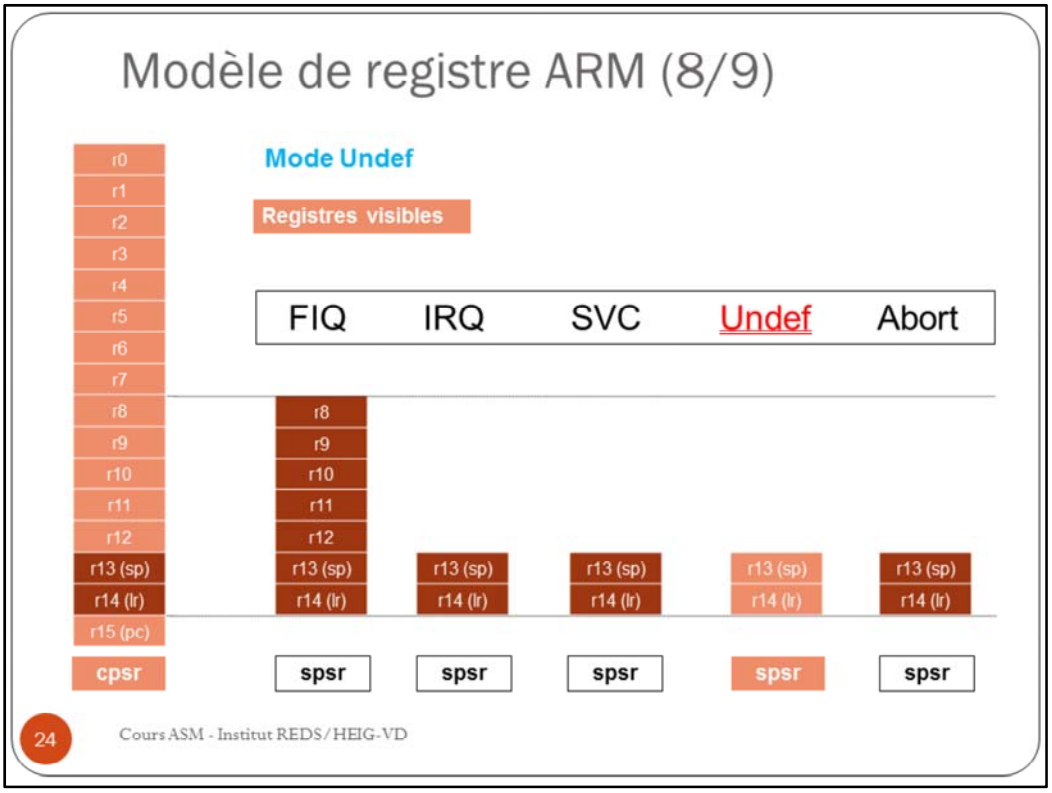


Alors que le mode *FIQ* peut gérer des interruptions hautement prioritaire, le mode ***IRQ*** est le mode le plus souvent utilisé pour gérer des interruptions matérielles; les temps de latence et de traitement sont suffisamment faibles et conviennent pour la plupart des applications. Dans ce mode, seuls les registres ***sp, lr*** et ***spsr*** sont répliqués.

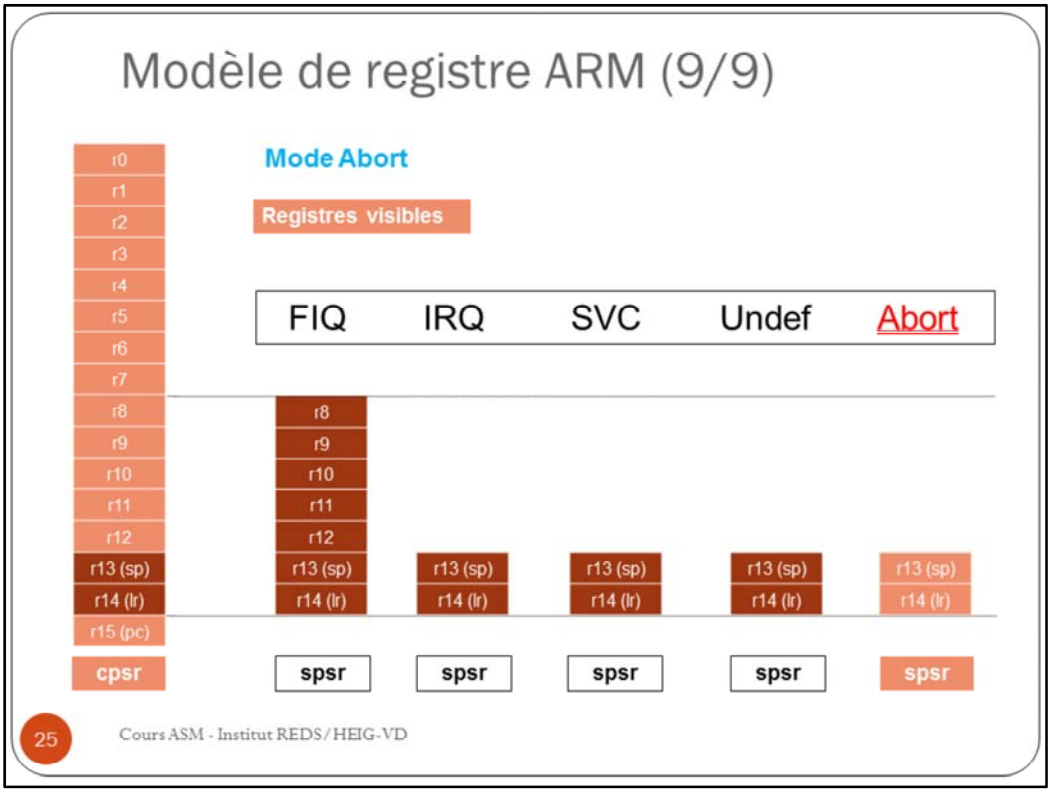
A l'instar du mode ***FIQ***, le mode ***IRQ*** est utilisé dans un contexte d'interruption matérielle et dispose de sa propre routine de service (un vecteur d'interruption *IRQ* dédié). Comme toute routine de service, celle-ci devra rester aussi petite que possible afin de ne pas *désactiver* les interruptions durant une trop longue période. C'est pourquoi, en principe, on cherchera à quitter ce mode aussi vite que possible afin d'éviter les problèmes de *réentrance*. C'est le mode *SVC* qui servira de "mode principal" pour l'exécution de code noyau. Le mode *SVC* est présenté sur la page suivante.



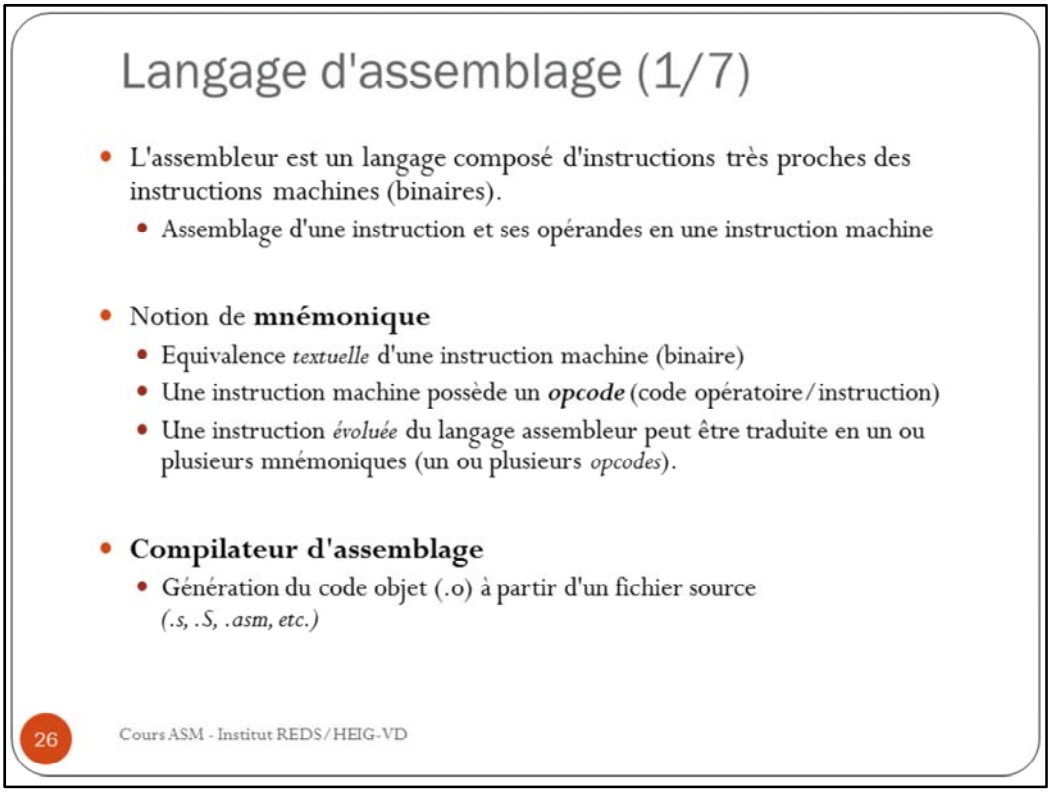
Le mode ***SVC*** est le mode "préféré" pour l'exécution de code en mode privilégié (mode *superviseur*). Le code peut manipuler toutes les instructions sensibles et dispose de tous les privilèges au niveau de l'espace d'adressage.



Le mode ***Undef*** est utilisé lors d'une interruption logicielle correspondant à un échec à l'exécution d'une instruction (instruction non valide).



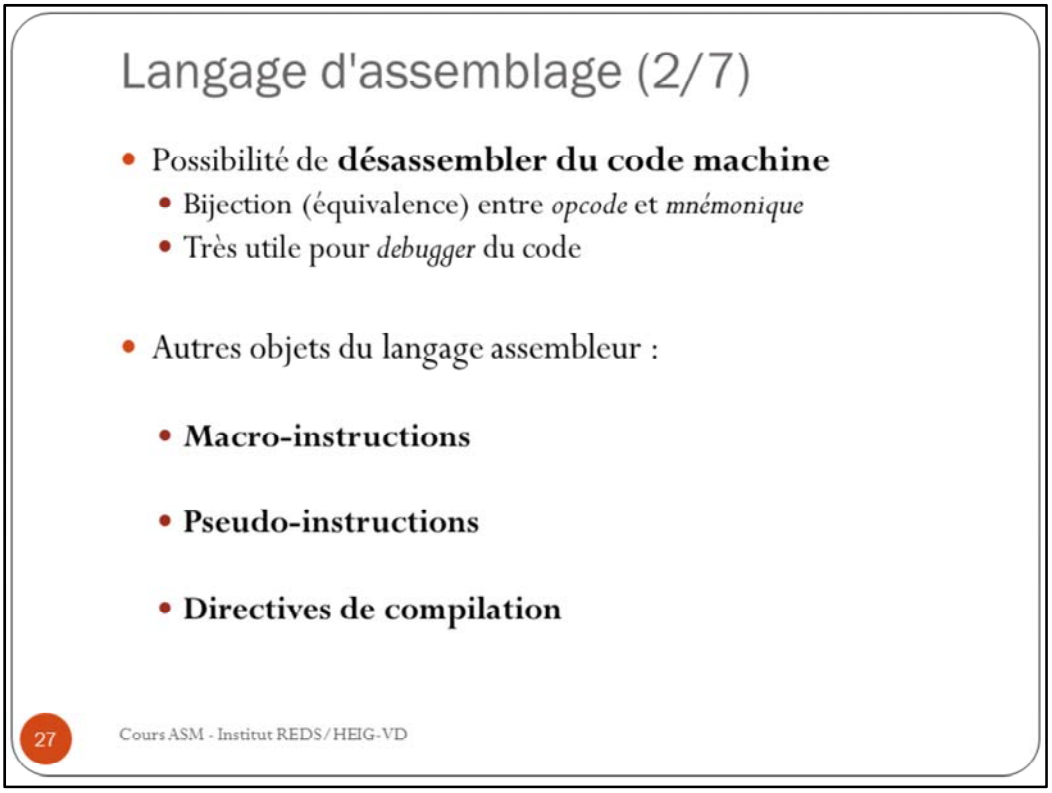
Le mode ***Abort*** est utilisé lorsqu'un accès mémoire est invalide. Typiquement, cela se produit lorsque l'accès à une adresse mémoire n'est pas permis, ou qu'il n'existe aucun *mappage* entre l'adresse virtuelle et l'adresse physique (*faute de page*). Ces aspects sont traités au cours de système d'exploitation dans le chapitre consacré à la gestion mémoire.



Le langage assembleur est composé d'instructions très proches (parfois équivalentes) aux instructions machines, c-à-d faisant partie du jeu d'instructions du processeur/microcontrôleur.

Chaque instruction assembleur devra être *convertie* en une instruction machine par un ***compilateur d'assemblage***. Comme tout autre langage, le compilateur génèrera ainsi un code objet (.o) pour chaque fichier assembleur compilé. La différence avec un langage de haut niveau réside bien dans le fait que les instructions utilisées dans le cadre de l'assembleur *dépendent* totalement du jeu d'instructions machines, donc du processeur.

Une instruction assembleur peut toutefois dériver sur différentes instructions machines, car elle peut être utilisée sous diverses formes, avec des *extensions* ou *paramètres* différents. Pour chaque *variété* d'une instruction, il peut donc y avoir différents *codes opératoires (opcodes)* correspondants. C'est pourquoi, on nuancera la notion d'instruction en utilisant la notion de ***mnémonique*** qui ce dernier représente une équivalence *textuelle* d'une instruction machine.

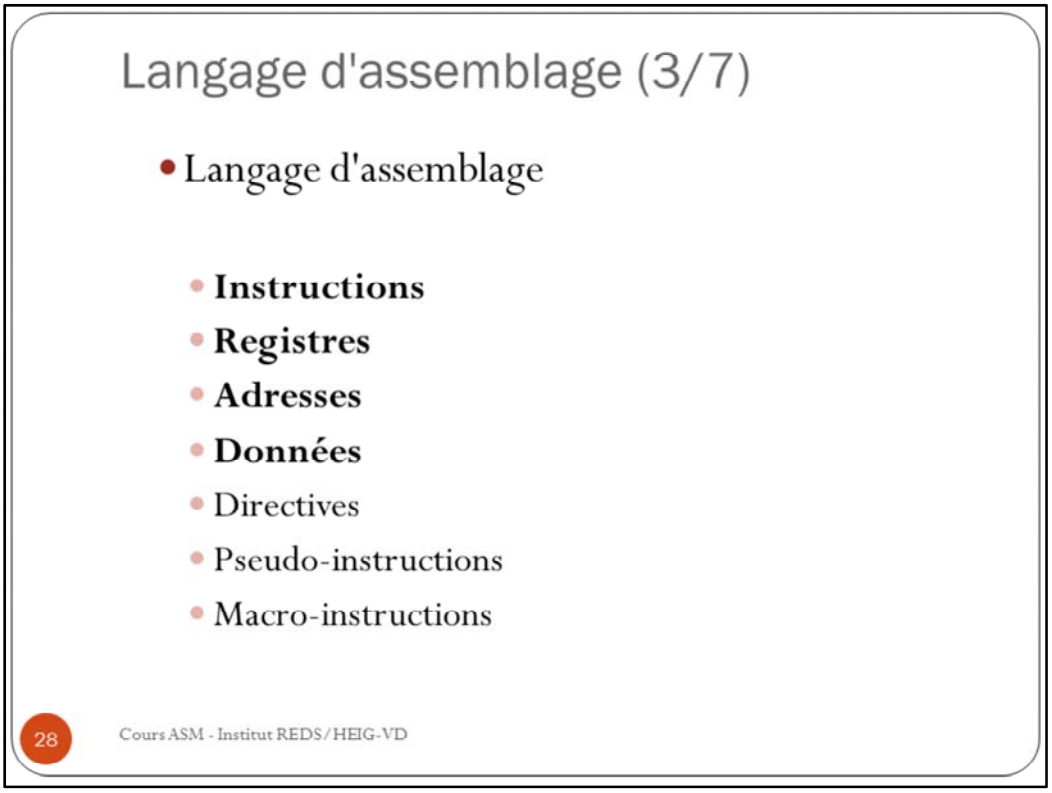


Le langage assembleur permet – en plus du jeu d'instructions de base – d'utiliser des *artifices* de langage plus évolués qui sont présentés ci-après.

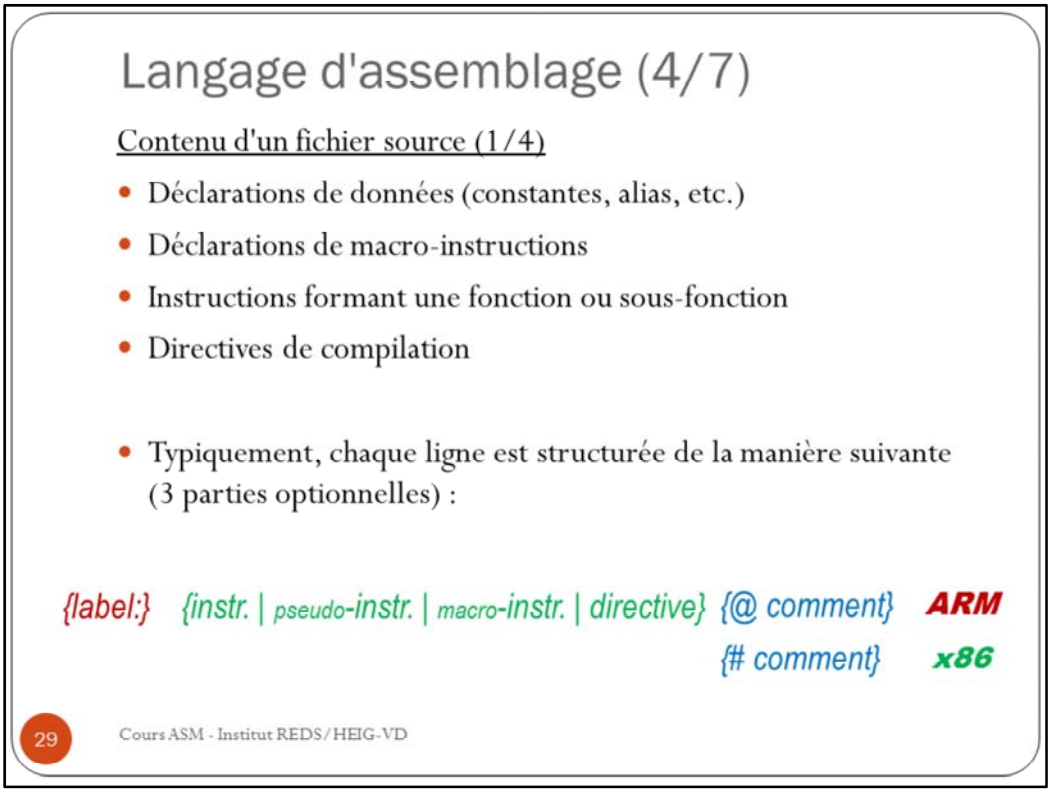
Les **macro-instructions** permettent de définir de nouvelles instructions basées sur des instructions de base. Les macro-instructions sont exclusivement des compositions *textuelles*, dans le sens qu'elles ne sont pas directement compilées en tant que telles, mais traitées par le préprocesseur qui applique systématiquement un remplacement de la macro-instruction par son équivalent *code* telle que défini par le programmeur. Cette notion est tout à fait semblable à celle utilisée avec la directive *#define* en langage C par exemple.

Les **pseudo-instructions** en revanche sont liées au type d'assembleur et sont traduites par le compilateur cette fois-ci en instructions de base. La traduction d'une pseudo en instructions natives est plus compliquée car elle fait intervenir souvent des mécanismes d'optimisation et d'adressage complexes.

Les **directives** de compilation ne sont pas directement liées au type d'assembleur mais au compilateur. Elles permettent d'informer le compilateur sur la manière de considérer le texte qui suit la directive, ou de déclarer des constantes, ou des étiquettes (*labels*), etc. Les directives ne correspondent donc pas à des instructions machines.



En résumé, un langage d'assemblage (ou *assembleur*) contient toujours des structures fondamentales, à savoir des instructions, des registres, des adresses et des données. En outre, le compilateur comprend des directives de compilation, des pseudo-instructions et des facilités pour structurer le code à un plus haut niveau d'abstraction telles que les macro-instructions.



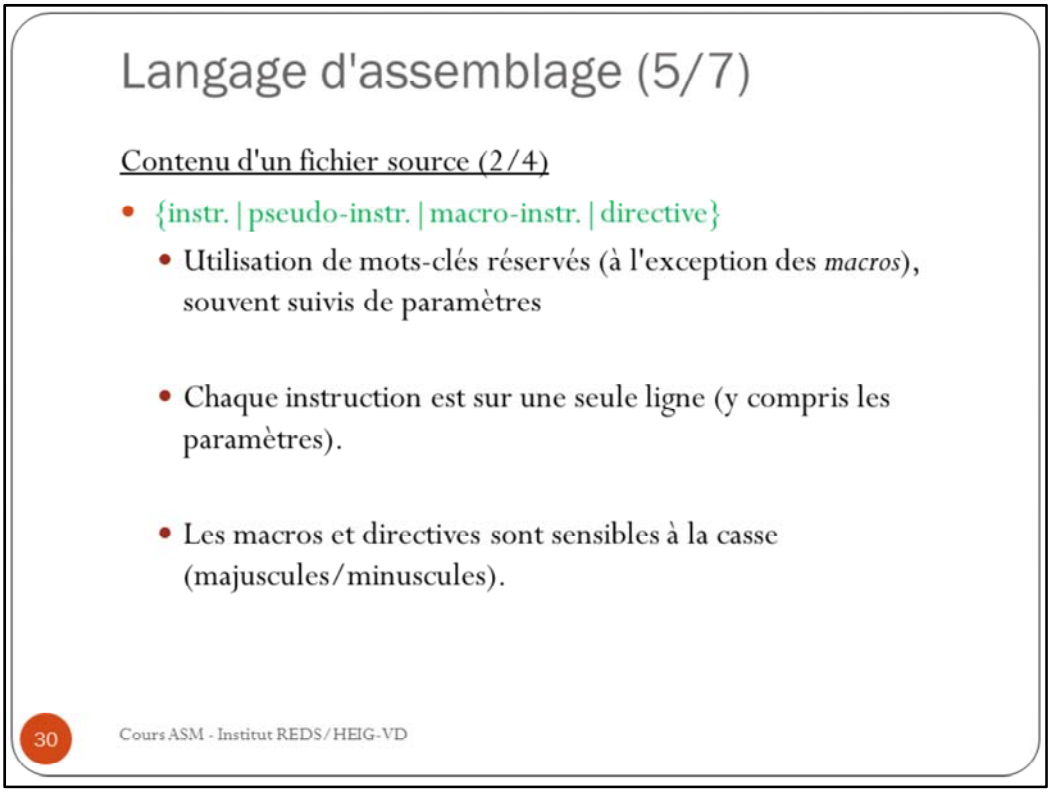
Un fichier source contenant un programme en langage assembleur nécessite une structure claire et lisible comme à peu près n'importe quel autre langage. On s'efforcera de commencer le fichier par la partie déclarative, notamment les constantes et redéfinitions de registre (alias). Puis, les directives de compilation peuvent être utilisés à des endroits très différents. Les directives commencent généralement par un point (*.text, .global, etc.)* ou un "#" (*#include <file>*) pour les directives traitées par le préprocesseur.

Un fichier assembleur peut inclure d'autres fichiers à l'instar d'un code C.

La découpe du programme doit bien faire apparaître les différentes fonctions du programme, même s'il n'existe pas de notion de fonction en tant que telle en langage assembleur.

Bien entendu, le code doit être suffisamment commenté. On adoptera les conventions ci-dessus pour les commentaires, à savoir le signe "@" pour *ARM* et "#" pour *x86*.

Ces considérations sont valables pour l'assembleur *x86* et *ARM*.

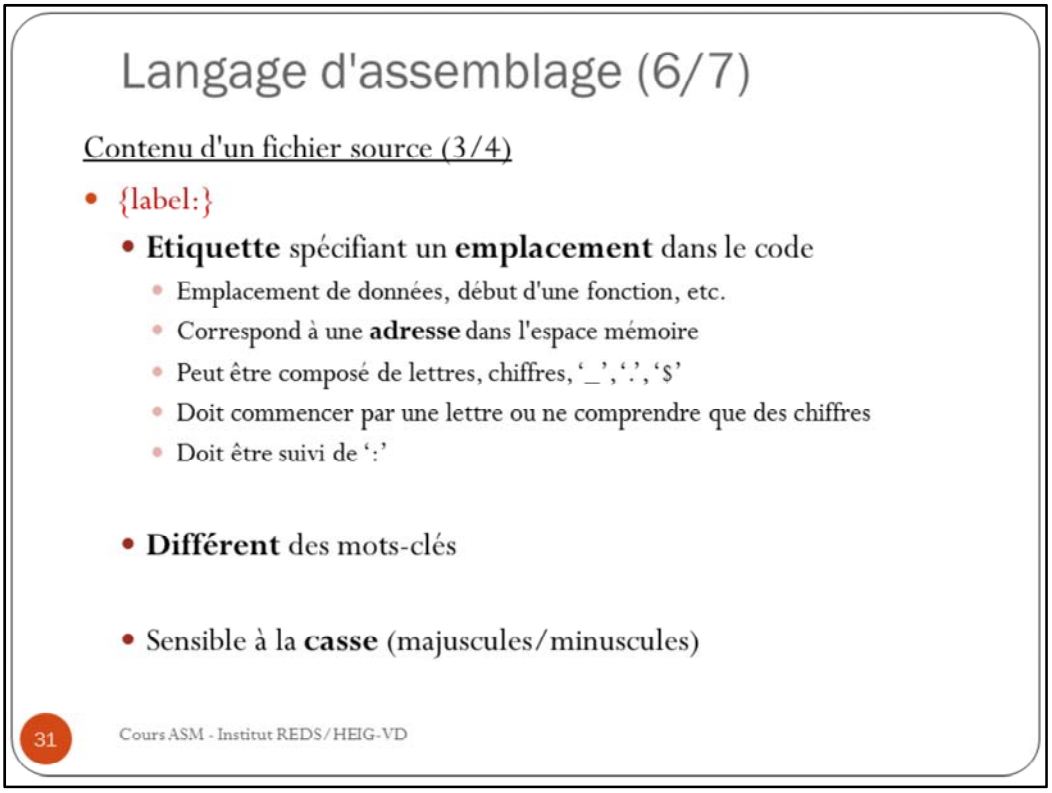


On veillera à ce que les instructions du programme soient correctement indentées; il faut éviter à tout prix de varier l'espacement entre les mnémoniques et les opérandes (paramètres).

On évitera également l'utilisation de plusieurs mnémoniques sur une même ligne; une ligne doit correspondre à une instruction (avec ses opérandes/paramètres).

De manière identique à une compilation d'un code C, la compilation d'un code assembleur passe par une phase de *pre-processing*. Le préprocesseur parcours le fichier texte et traite ses directives comme l'inclusion de fichier (*#include*) ou autres pragmas.

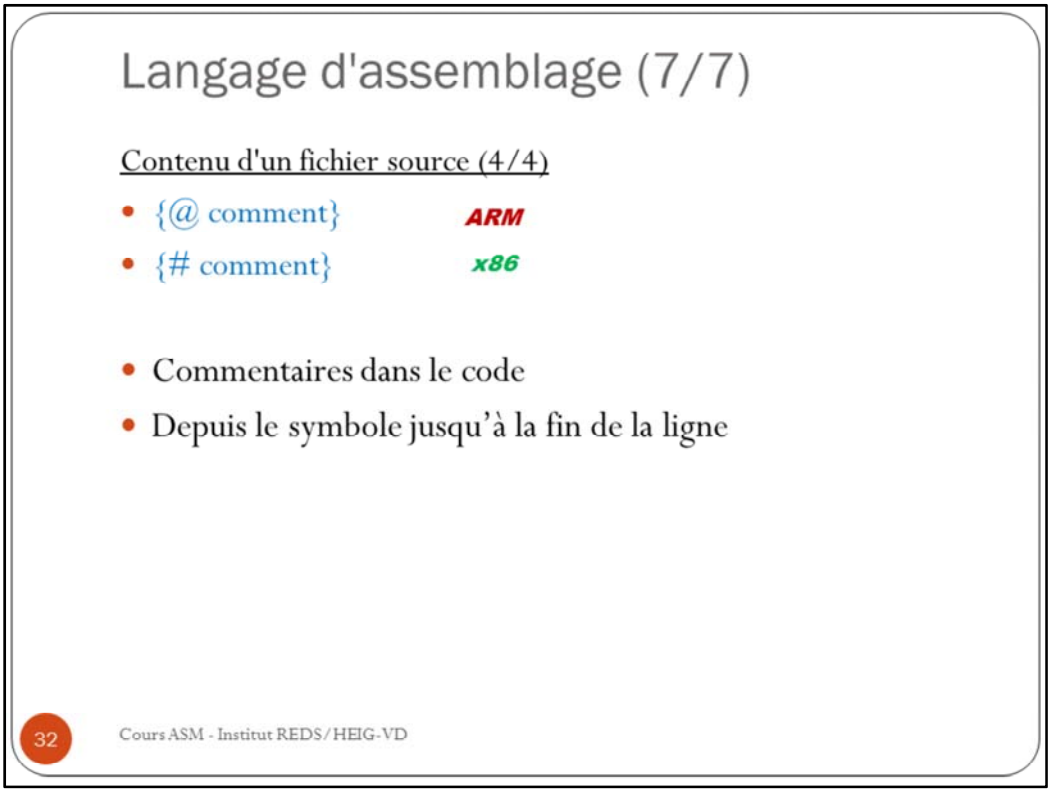
Avec *gcc*, on peut obtenir les productions intermédiaires (résultat après le *pre-processing*, traduction dans le langage assembleur natif, etc.) en utilisant l'option ***--save-temps*** à l'invocation de la compilation. On trouvera ainsi un fichier avec l'extension ***.s*** (à la place de *.S*) signifiant que cette version de code est le résultat intermédiaire du compilateur avant le passage au code objet.



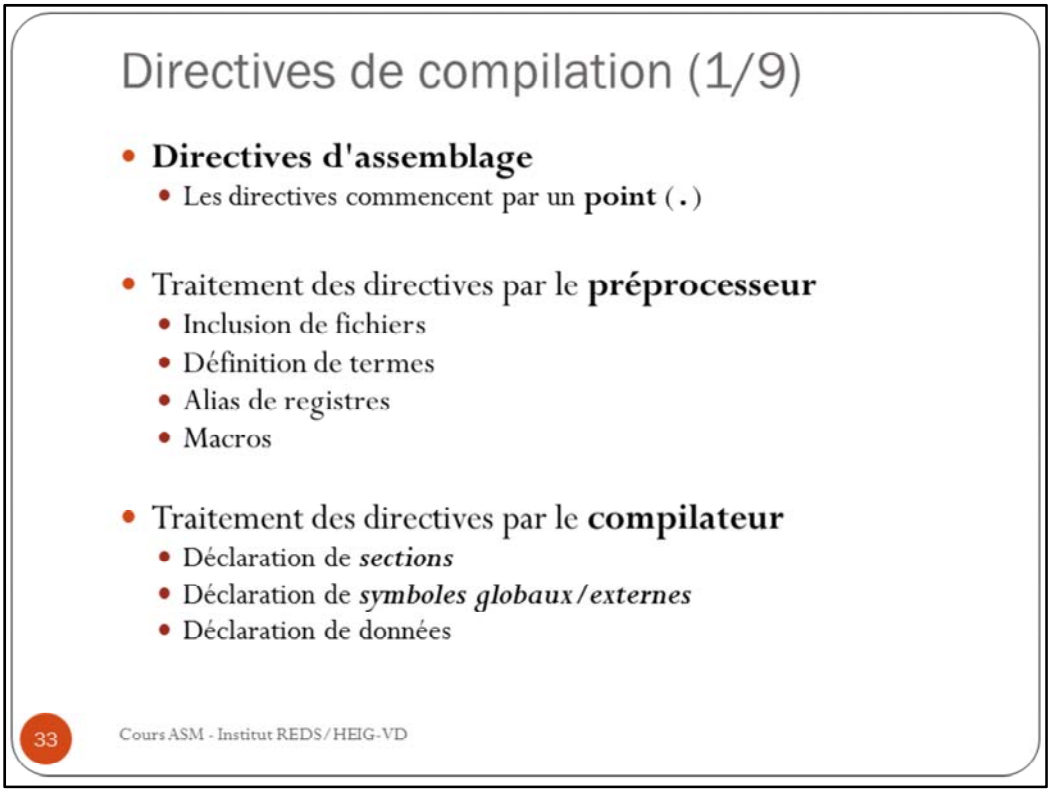
Un *label* dénote un emplacement précis dans le code. Il faut se représenter cette emplacement au sein de l'espace d'adressage dans lequel le code s'exécutera. Ainsi, grâce aux *labels*, il est possible de spécifier le début d'une fonction, ou encore l'emplacement d'une zone particulière dans laquelle des données peuvent être déclarées, ou encore l'emplacement d'un espace mémoire non initialisé.

Le *label* joue donc un rôle central dans la structure du programme (implémentation d'une boucle par exemple) et permet de rediriger l'exécution courante à des emplacements bien précis. Il peut être utilisé à l'intérieur d'une *macro*.

Un *label* peut être composé de chiffres et de lettres. Il est généralement unique. Si un *label* est utilisé dans une *macro* par exemple, et que celle-ci est instanciée plusieurs fois dans le même fichier, le *label* doit être local et ne peut être **que numérique**. L'utilisation du **suffixe *b*** pour ***backward*** ou ***f*** pour ***forward*** permettra au compilateur de sélectionner le *label* le plus proche en arrière ou en avant.



Dès que le symbole de commentaire est rencontré, il est valable jusqu'à la fin de la ligne, c-à-d jusqu'au prochain retour de ligne (*CR* ou *carriage return*). Un commentaire peut commencer en début de ligne, ou suivre tout type de code (instruction, *label*, etc.).

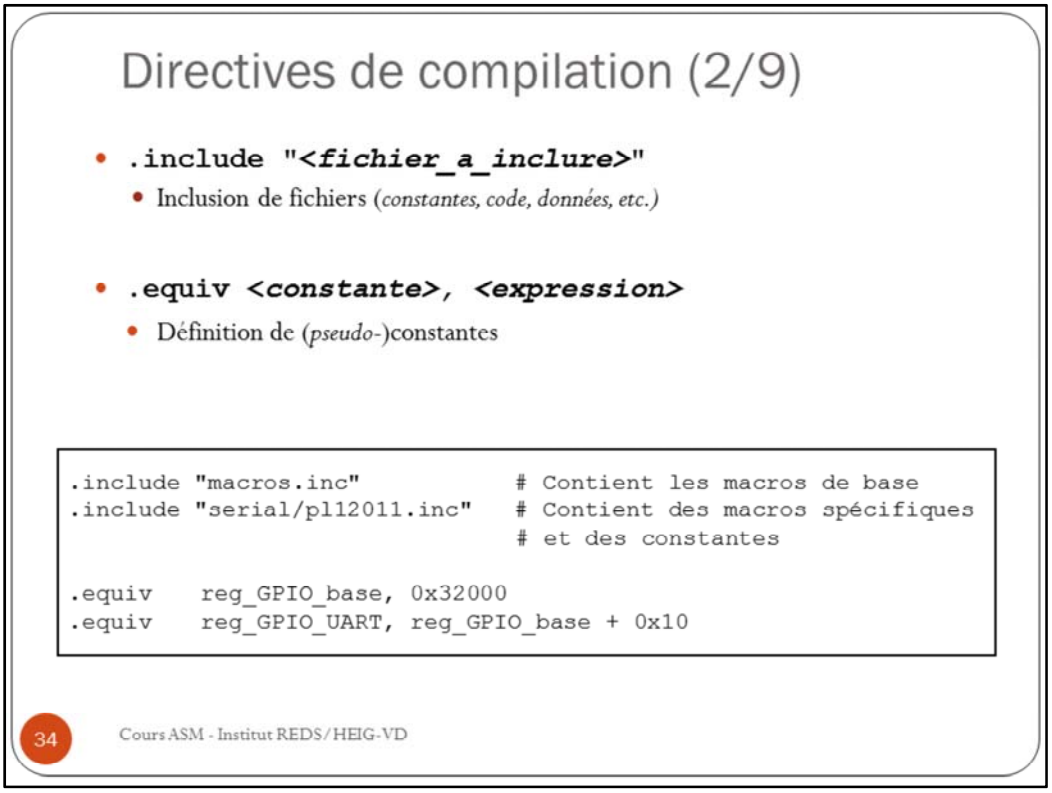


Les directives de compilation (ou directives d'assemblage) sont des éléments du langage d'assembleur qui, d'une part facilitent la programmation d'un code assembleur, et d'autre part fournissent au **préprocesseur** et au **compilateur** des informations utiles à la production d'un code objet et d'un exécutable.

A l'instar du langage C, le préprocesseur dispose de directives particulières comme *.include, .equiv, .macro*, etc. qui agissent directement sur le texte du programme. Ce prétraitement génère alors un nouveau code source intermédiaire qui sera repris directement par le compilateur.

Les directives d'assemblage liés au compilateur mettent souvent en jeu des calculs d'adresse parfois complexe, et permettent de gérer au mieux l'allocation (statique ou retardée) de zones de données présentent dans le code objet.

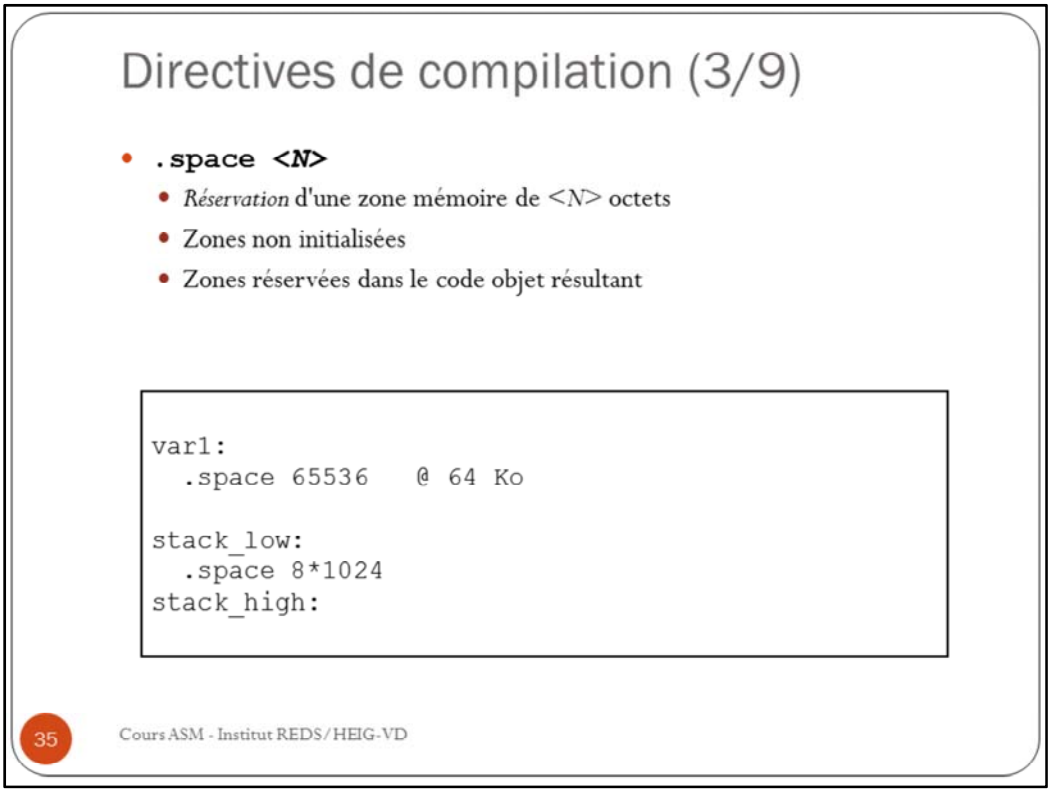
Dans l'univers des compilateurs de la famille *GNU*, la plupart des directives d'assemblage sont identiques entre les différents langages d'assemblage (*x86, ARM, PowerPC, etc.*).



Les deux directives ci-dessus sont liées au préprocesseur : elles auront pour effet une transformation du code source (en tant que texte) vers le code intermédiaire qui sera "présenté" au compilateur.

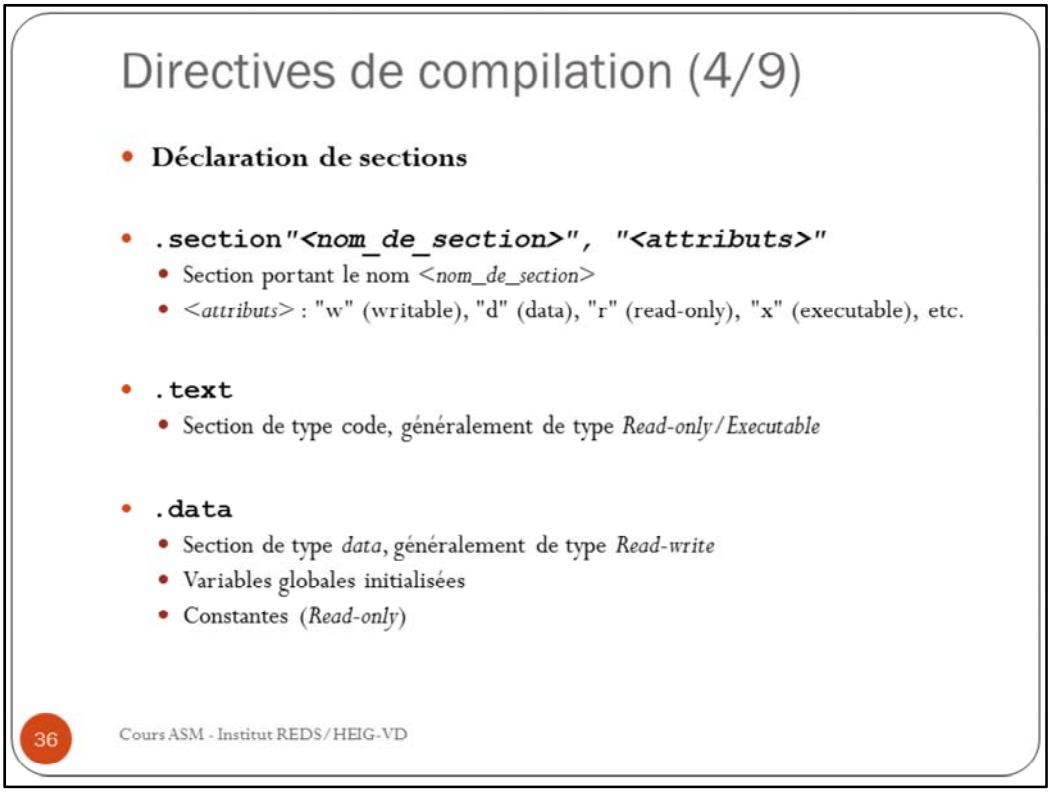
La directive ***.include*** permet d'inclure un fichier à l'intérieur du fichier en cours de compilation.

La directive ***.equiv*** permet la redéfinition de termes à l'intérieur du fichier; c'est le cas par exemple de constantes; chaque occurrence d'une constante sera systématiquement remplacée par sa valeur correspondante, à un niveau textuel. Dans ce sens, la notion de constante n'a pas de signification particulière du point de vue du compilateur.



La directive ***.space*** est très utile pour la réservation explicite de zone mémoire dans le code objet, autrement dit dans l'espace d'adressage du code compilé. Cette directive peut être utilisée pour réserver une zone mémoire dédiée à la **pile**, par exemple, ou tout autre type de zone mémoire dans lequel des données pourront prendre place.

Il est intéressant de se rendre compte que les adresses croissent en parcourant le fichier du haut vers le bas. C'est la raison pour laquelle la référence *stack_high* de l'exemple ci-dessus indique le **sommet** de la pile (et non le bas).



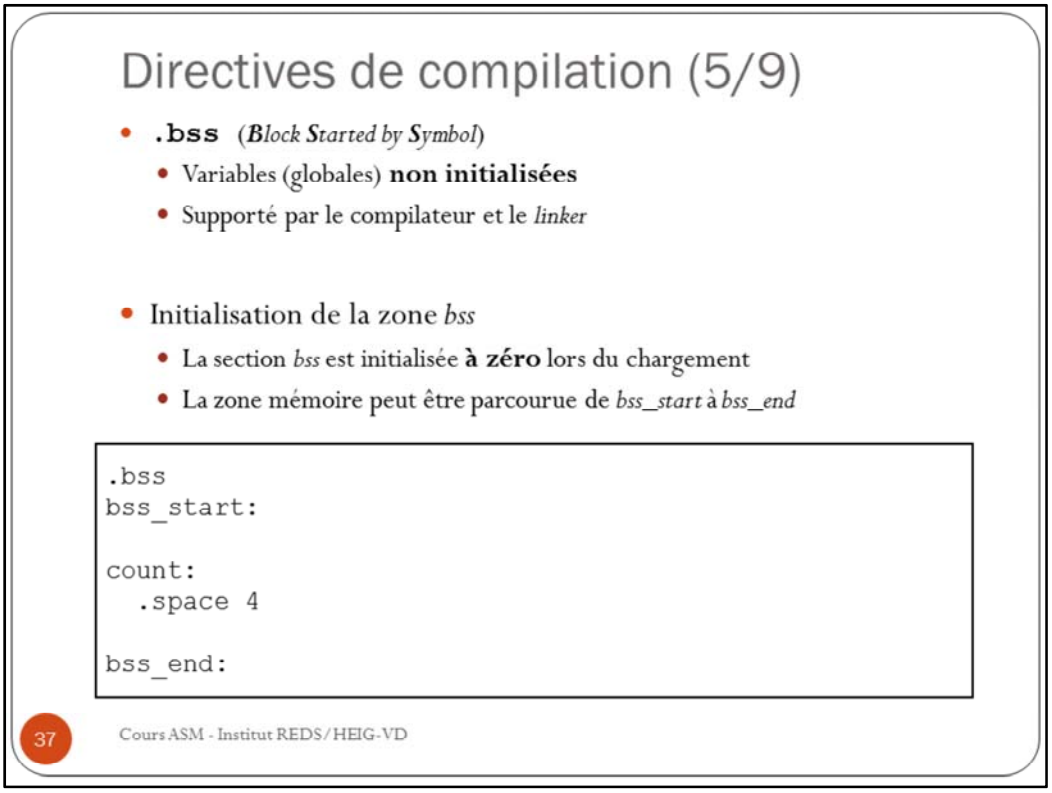
Les directives de section permettent de déclarer une portion de code ou de données qui suit directement la directive comme faisant partie d'une certaine section.

Dans un programme, différents types de section peuvent exister; ces types de section servent avant tout à organiser le code et les données d'une manière structurée et sécurisée : différents *attributs de protection* peuvent en effet être liés aux différentes zones mémoire correspondantes. Par ailleurs, l'utilisation de sections permettent au *linker* et au chargeur de reloger plus aisément le code et les données le cas échéant (les adresses sont alors relatives par rapport au début d'une section).

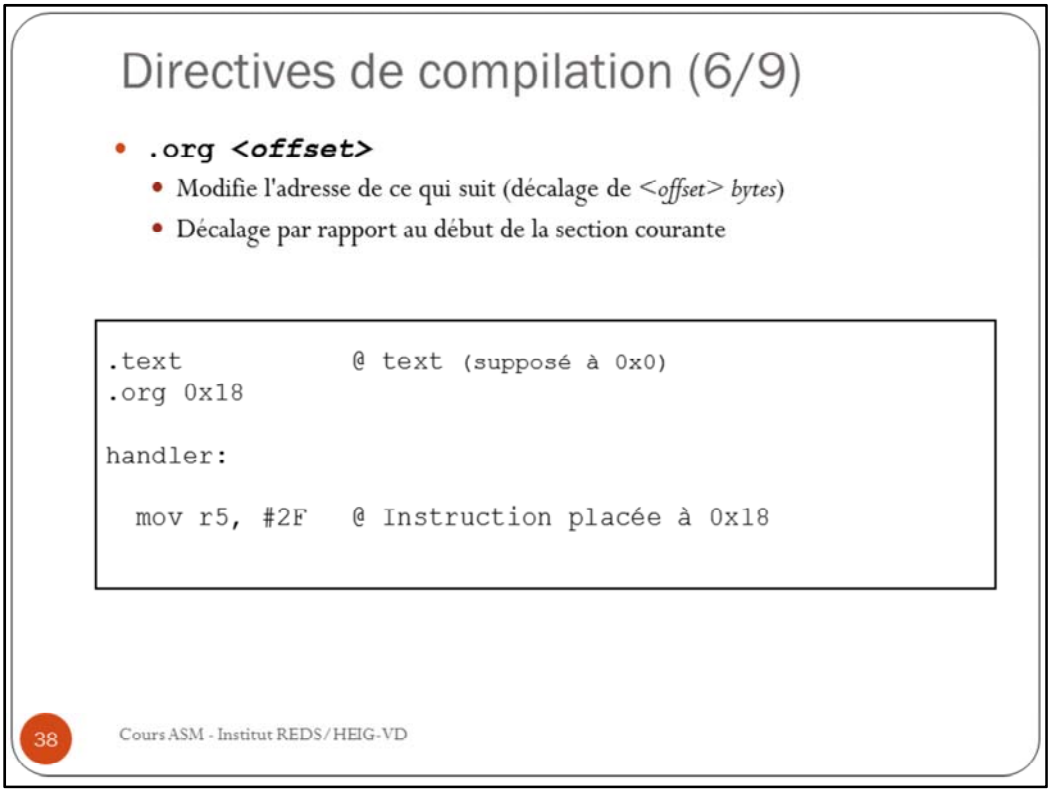
Sur un système d'exploitation, les sections habituelles sont de type **code** (section *text*), **données** (section *data* pour les variables globales initialisées et les constantes, ou section *bss* pour les variables globales non initialisées), **pile** (section *stack*), etc.

Il n'y a pas vraiment de limitations liées au nombre de section ou à leur type; il existe un certain nombre de sections de type prédéfini (comme mentionné ci-dessus), mais un programme peut en contenir beaucoup pour des usages très différents (*runtime, debugger,* sections définies par le développeur, etc.).

Notons que la notion de section apparaîtra dans le code objet (et l'exécutable); le *linker* se chargera de rassembler tous les codes objet d'un même type dans leurs emplacements respectifs. L'emplacement des sections diverses est souvent décrite dans un fichier spécial appelé ***script de linkage***.



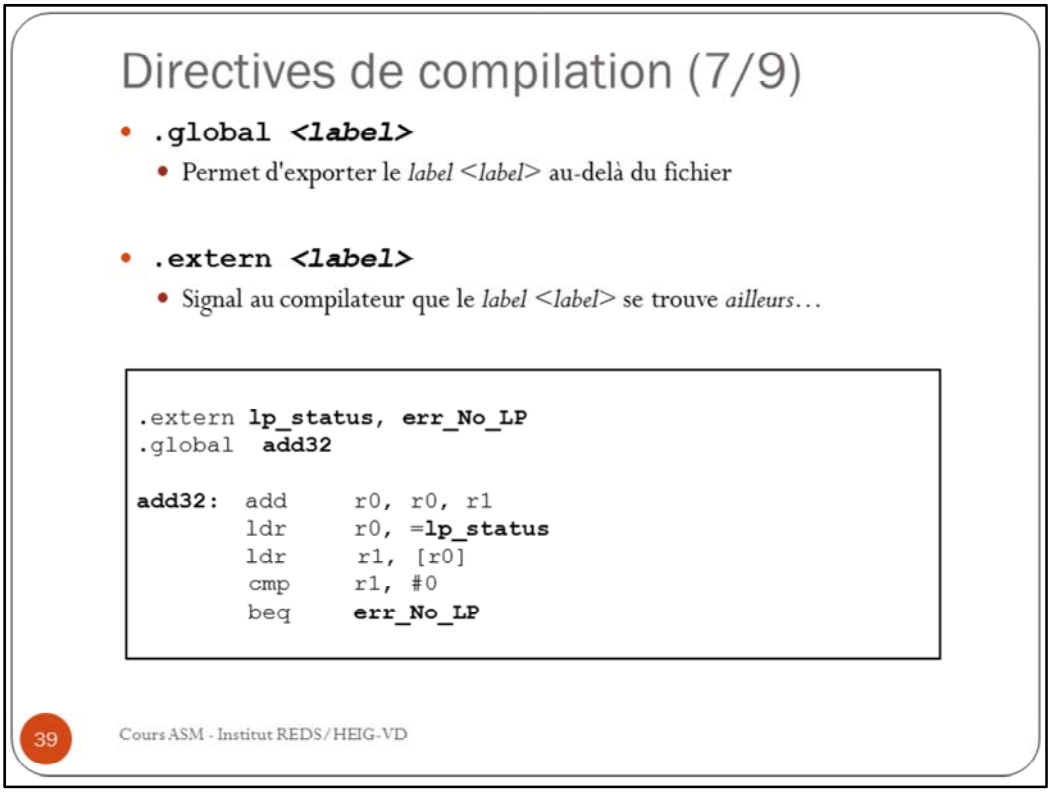
La section ***bss*** est généralement utilisée par le compilateur pour définir l'ensemble des variables – et donc des zones mémoire correspondantes – non initialisées. De ce fait, le code objet stocké dans le fichier est réduit car il n'est pas nécessaire de stocker un ensemble de *bytes* non initialisés. De plus, la section *BSS* pourra être initialisée à 0 lors du chargement, après avoir alloué la mémoire nécessaire.



La directive ***.org*** informe le compilateur que ce qui suit est à placer à un certain *offset* par rapport au début de la section. L'exemple ci-dessus montre que l'instruction suivant la directive *.org* sera placée à l'adresse 0x18, puisque dans cet exemple, la section courante (*.text*) est supposée à l'adresse 0x0.

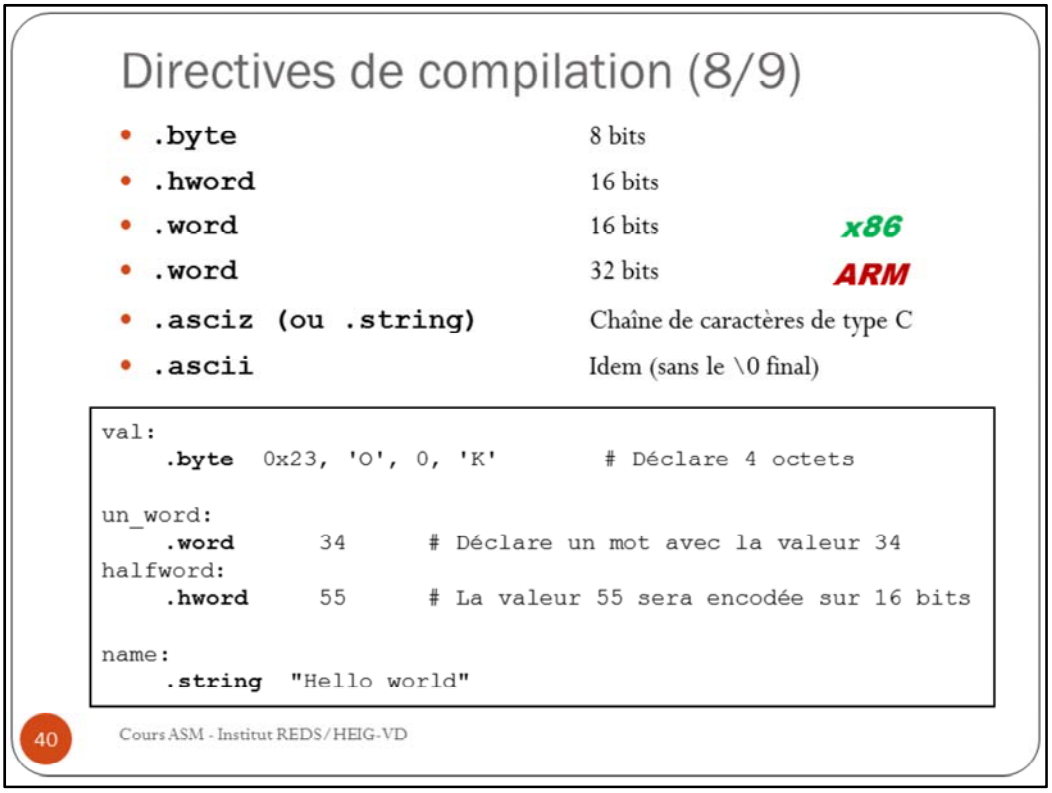
Cette directive peut être utilisée n'importe où dans le fichier, mais ne peut provoquer une discontinuité de l'espace d'adressage, i.e. il n'est pas possible de donner un *offset* qui aboutirait à une adresse inférieure à la position courante.

Le *linker* déterminera l'adresse des sections. Un réajustage d'adresse aura lieu durant l'édition des liens.



Les directives ***.global*** et ***.extern*** sont utilisées par le compilateur et le *linker*. Lorsque le compilateur traite une référence (*label*) suivant la directive *.global*, il donne la possibilité au *linker* de l'utiliser comme référence externe, c-à-d qu'un autre code objet pourra y référer.

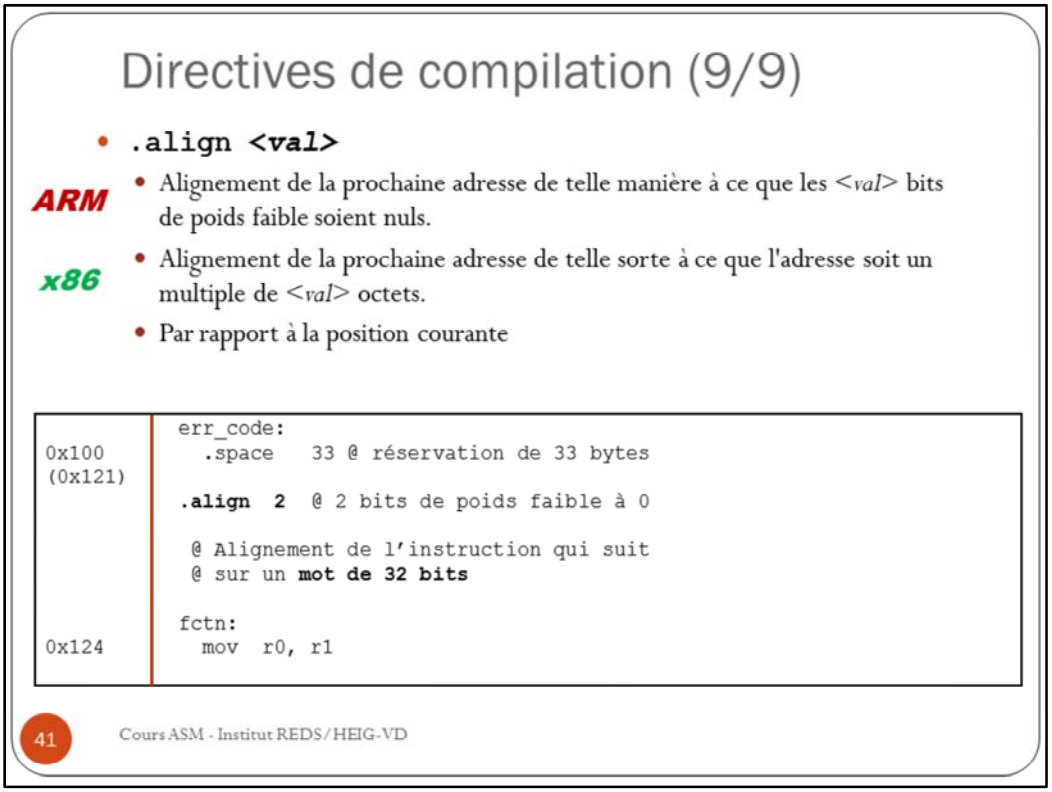
Respectivement, lorsqu'une instruction fait référence à un *label* qui a été déclaré *.extern*, le compilateur produira une référence symbolique correspondant au nom du *label*, référence qui sera par la suite ***résolue*** par le *linker*.



La famille de directives ci-dessus permettent de déclarer des données de tout type, y compris des chaînes de caractères.

Les directives principales sont bien entendu ***.byte***, ***.word*** et ***.string***

Il faut faire très attention au fait que les valeurs qui suivent ces directives correspondent directement aux valeurs qui seront implantées en mémoire, et **non à leur taille**.



La directive ***.align*** permet l'alignement de l'adresse qui suit sur un multiple d'un certain nombre de *bytes*. Par exemple, un alignement sur un *mot de 32 bits* conduira à ce que l'adresse soit un multiple de 4 octets, et en d'autres termes, cela signifie que les 2 derniers bits soient nuls (en effet, on le voit aisément en incrémentant l'adresse de 4 depuis la valeur 0).

Sur *ARM*, un tel alignement sur 4 octets est toujours requis au niveau des instructions; par exemple. le registre *pc (r15)* est toujours incrémenté de 4 octets et, partant de ce constat, les instructions de saut peuvent encoder une adresse sur un nombre de bits réduit comme nous le verrons plus tard.

Malheureusement, pour des raisons historiques, l'argument de la directive ***.align*** n'a **pas la même signification sur *x86* que sur *ARM*** *:* sur cette dernière, c'est le nombre de bits à 0 que doit comporter la prochaine adresse, alors que sur *x86*, il s'agit du nombre d'octets dont l'adresse doit être un multiple. L'effet est cependant le même.

Références

- ARM Infocenter. Site ARM principal : *http://infocenter.arm.com*
- Intel x86 32 & 64 bits.
  http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- Rajat Moona, *Assembly Language Programming in GNU/Linux for IA32 Architectures*, Eastern Economy Edition, New Dehli, 2009.
- Andrew N.Sloss, Dominic Symes, *ARM System Developer's Guide, Designing and Optimizing System Software*, Elsevier, 2004.