Architecture ARM/x86

Vincent Migliore

vincent.migliore@insa-toulouse.fr

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Objectifs, méthode et organisation

INSA TOULOUSE

Objectifs

- Connaître les principaux aspects des couches basses du logiciel (contraints pas le matériel) :
  - Les appels de fonctions ;
  - Le stockage des données ;
  - Les aspects de sécurité associés.
- Connaître les contraintes associées aux architectures standard (CISC et RISC).
- Être capable d'utiliser efficacement des outils de débogage standards.

Méthode

- Compilation d'un même code C vers ces 3 type de processeurs suivants :
  - ARM Cortex-M3
  - Intel x86 (architecture 32 bits) et x86-64 (architecture 64 bits).
- utilisation de la chaîne de compilation de GNU (make, gcc, binutils, ...).

Pré-requis

- Architecture d'une machine (harvard et harvard modifée).
- Langage d'assemblage.

Organisation

La séquence est composée de 2 CM et 3 TP, l'évaluation ce fait par rendu de TP.

Jeu d'instruction

C'est l'ensemble des instructions supportées par un processeur donné.

Architecture de jeu d'instruction

Appelé Instruction Set Architecture (ISA) en anglais, c'est la description fonctionnelle du processeur du point de vu du programmeur, c'est-à-dire tout ce qui est visible au niveau du langage machine.
Ce qui est **visible** au niveau de l'ISA :

- Les instructions supportées par le processeur;
- Les registres utilisables par les instructions et leur rôle ;
- L'organisation de la mémoire et des entrées/sorties ;
- Les éléments architecturaux (présence ou non de plusieurs coeurs, présence ou non de caches,$\cdots$).

(Exemple) de ce qui est **invisible** au niveau de l'ISA :

- Les éléments internes du pipeline processeur (nombre d'étages, exécution in-order ou out-of-order, prédiction de branchement,$\cdots$) ;
- Le nombre de cache et leur associativité ;
- Description interne du circuit du processeur (ALU, contrôleur mémoire, unité de contrôle,$\cdots$).

Architectures CISC/RISC

RISC

Reduced **I**nstruction **S**et **C**omputer

- Premier jeu d'instruction formalisé (Université de Stanford et Berkeley, 1981).
- code d'instructions de longueur fixe = 1 mot de mémoire programme, pour faciliter l'utilisation d'un pipe-line
- durée d'exécution d'une instruction = 1 cycle d'horloge registres banalisés
- éventuellement architecture load-store (2 instructions seulement ont accès à la mémoire)

CISC

Complex **I**nstruction **S**et **C**omputer

- Concept non formalisé : Acronyme donné aux familles non RISC.
- code d'instructions de longueur variable, partant de 1 byte (pas d'alignement mémoire).
- développement incrémental d'une famille de CPUs de plus en plus perfectionnés.
- registres spécialisés.
- ajout d'instructions facilité par le microcodage.
- durée d'exécution d'une instruction très variable.

Architectures CISC/RISC

RISC

Reduced **I**nstruction **S**et **C**omputer

- Premier jeu d'instruction formalisé (Université de Stanford et Berkeley, 1981).
- code d'instructions de longueur fixe = 1 mot de mémoire programme, pour faciliter l'utilisation d'un pipe-line
- durée d'exécution d'une instruction = 1 cycle d'horloge registres banalisés
- éventuellement architecture load-store (2 instructions seulement ont accès à la mémoire)

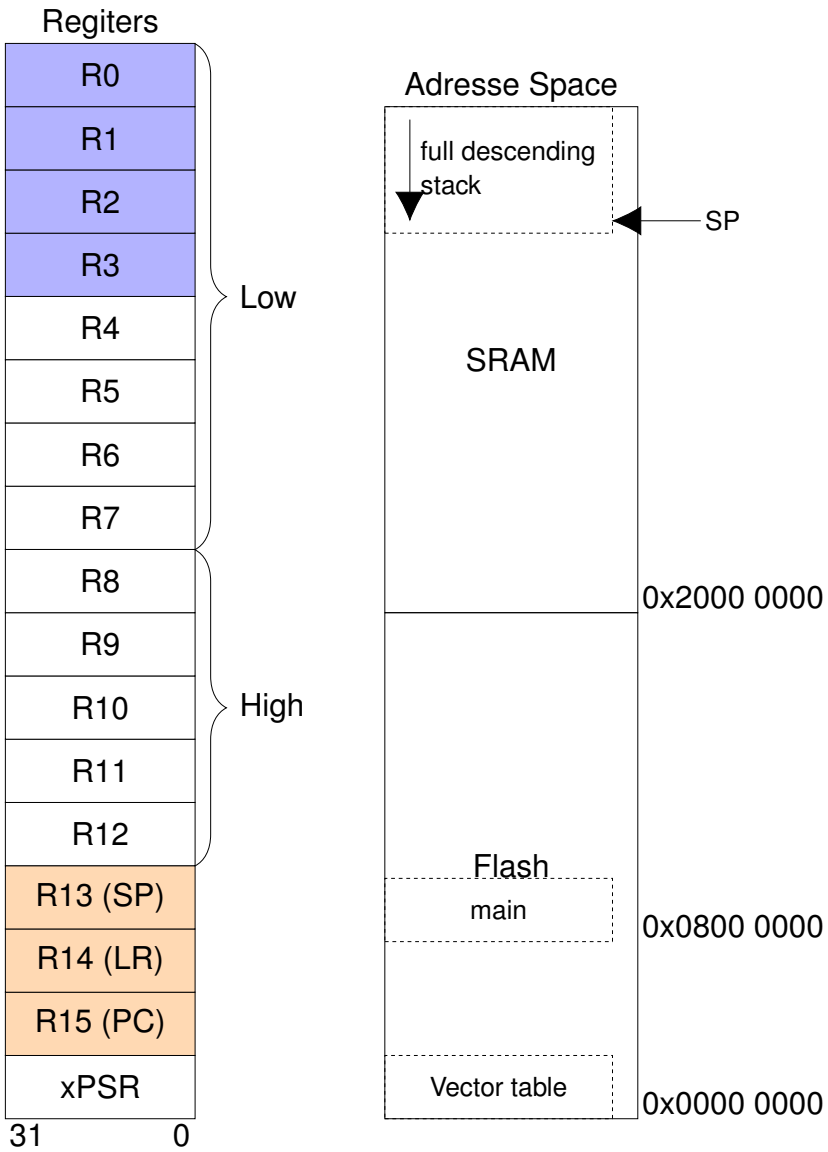
CISC

Complex **I**nstruction **S**et **C**omputer

- Concept non formalisé : Acronyme donné aux familles non RISC.
- code d'instructions de longueur variable, partant de 1 byte (pas d'alignement mémoire).
- développement incrémental d'une famille de CPUs de plus en plus perfectionnés.
- registres spécialisés.
- ajout d'instructions facilité par le microcodage.
- durée d'exécution d'une instruction très variable.

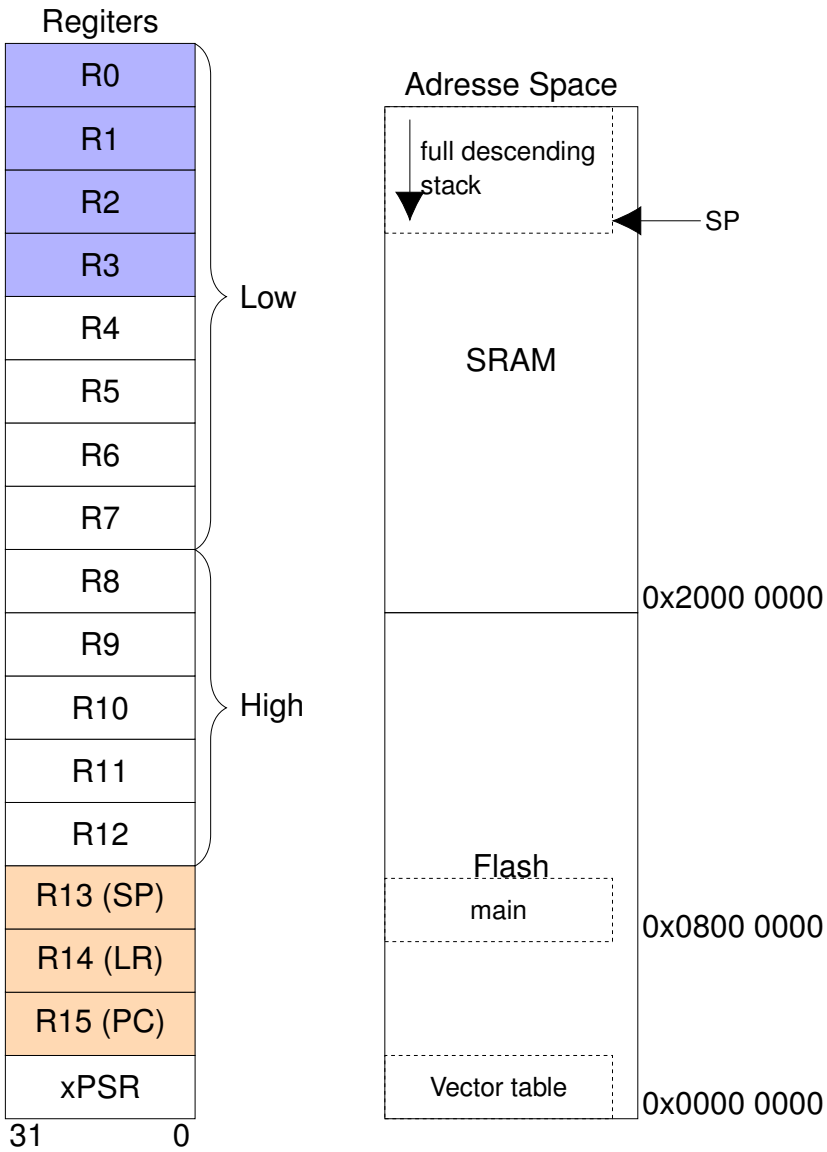
Aujourd'hui la barriere entre les deux jeux d'instruction s'est réduite :

- les processeurs CISC se sont dotés de pipelines efficaces ;
- le jeu d'instruction des RISC est devenu de plus en plus complexe, les règles ont été assouplies (Arm autorise des instructions sur 16 et 32 bits + spécialisations de registres) ;
- vers un dépassement des performances du RISC par rapport à CISC (Apple M1).



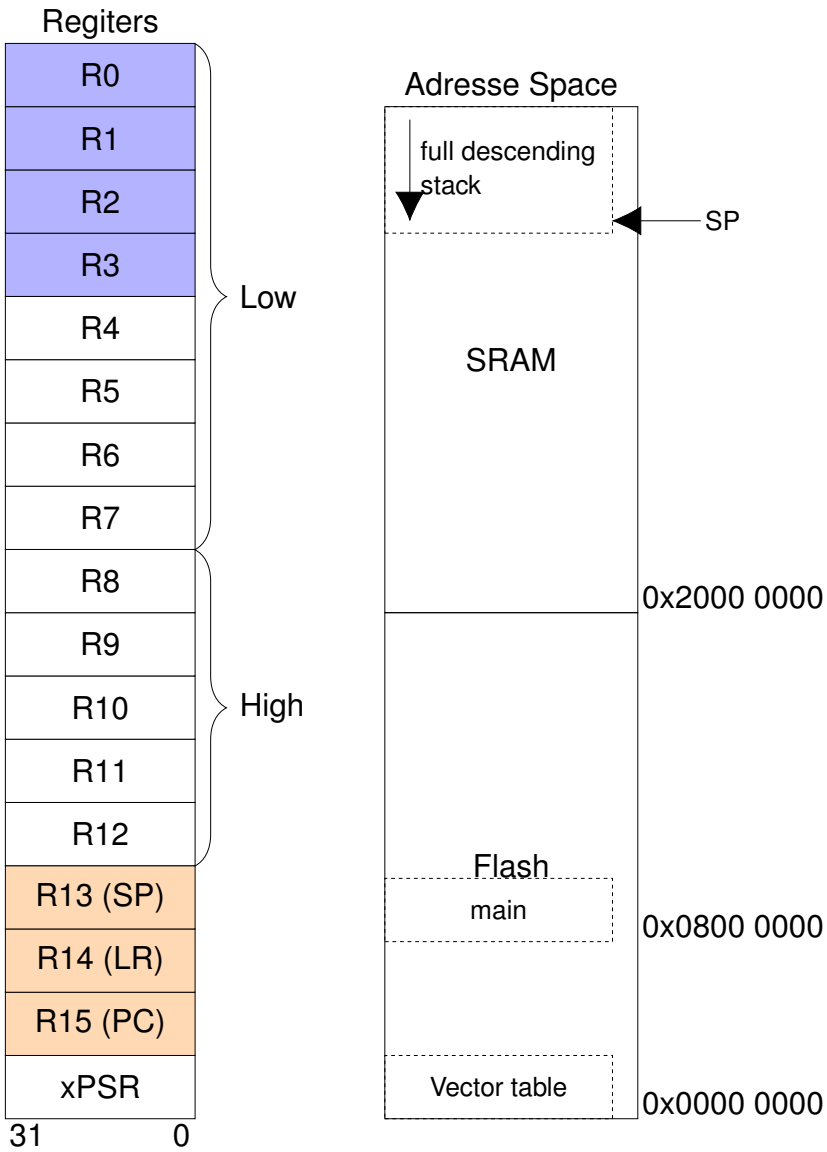
Characteristics

- RISC Architecture
- Little endian
- 16 bits and 32 bits instructions
- Harvard architecture model:
  - 0x0000 0000 to 0x1FFF FFFF: ROM address space (instructions + constants).
  - 0x2000 0000 to 0x3FFF FFFF: RAM address space (stack + variables).
- Full descending stack.
- Vector table is always at address 0.
- Main program is in general at address 0x0800 0000.



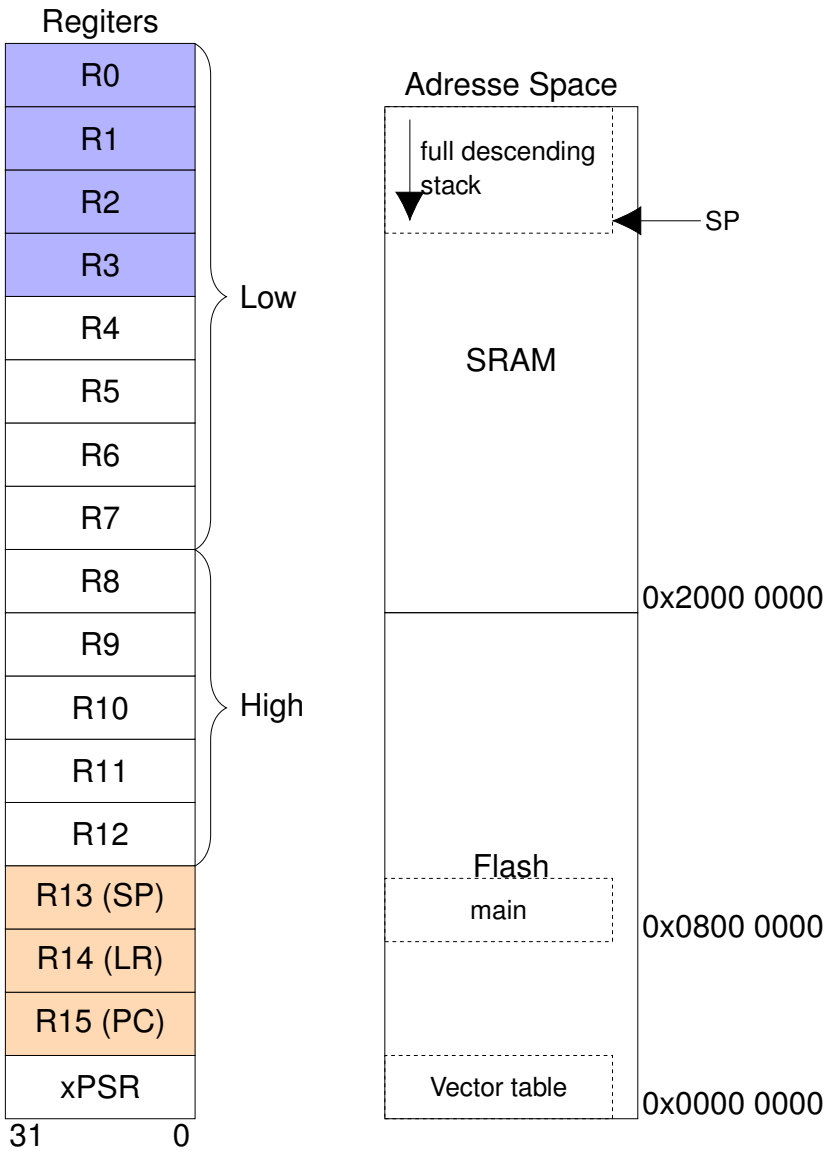
General purpose registers

- 13 general purpose registers (R0 to R12).
- 8 ”low” registers. Most 16 bits instructions can only access them.
- 5 ”high” registers Some 16 bits instructions can only access them.
- R0 to R3 hold function arguments.
- R0 and R1 hold return value.
- R4 to R11 must not be altered during a function call.



Special purpose registers [1]

- 4 special purpose registers (R13 to R15 + xPSR).
- **SP**: Stack Pointer. Pointer to the top address of the stack.
- **LR**: Link Register. Holds return address.
- **PC**: Program Counter. Holds next instruction address.



Special purpose registers [2]

xPSR: Processor Status Register. Holds the processor state:

- **N**: Negative condition code flag. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N = 0 if it is positive or zero.
- **Z**: Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.
- **C**:Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.
- **V**:Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.
- **Q**: Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result

Jusqu'à ARMv7

Jusqu'à ARMv7, seulement des architectures jusqu'à 32 bits étaient supportés, avec les jeux d'instructions suivants :

ARM	:	Jeu d'instructions de taille fixe (32 bits). C'est le jeu d'instructions historique de Arm.
Thumb	:	Jeu d'instructions de taille fixe (16 bits). Il est constitué d'un sous-ensemble d'instructions du ARM (réduction de la taille du code, mais réduction des performances)
Thumb-2	:	Jeu d'instructions de taille variable (16 et 32 bits). Il s'agit d'une extension du jeu d'instruction Thumb avec des instructions 32 bits (issus de ARM) afin d'obtenir un compromois temps/taille du code.

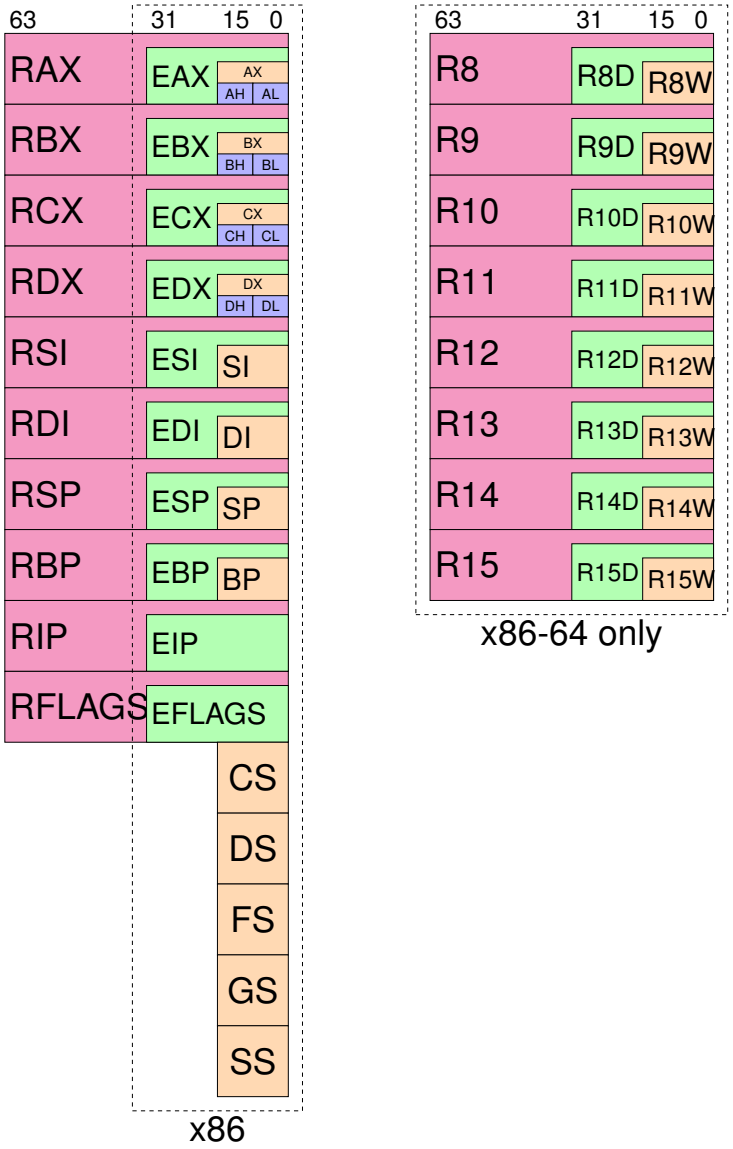
ARMv8 et + (à partir d'octobre 2011)

Arrivée des architectures 64 bits, avec refonte de la nomenclature : Processeur 32 bits $\rightarrow$ AArch32, Processeur 64 bits $\rightarrow$ AArch64. Chaque architecture ne supporte pas le même jeu d'instruction :

AArch32		
A32 ("ARM")	:	Jeu d'instructions de taille fixe (32 bits) avec support de ARM
T32 ("THUMB")	:	Jeu d'instructions de taille variable (16 et 32 bits) avec support du Thumb-2

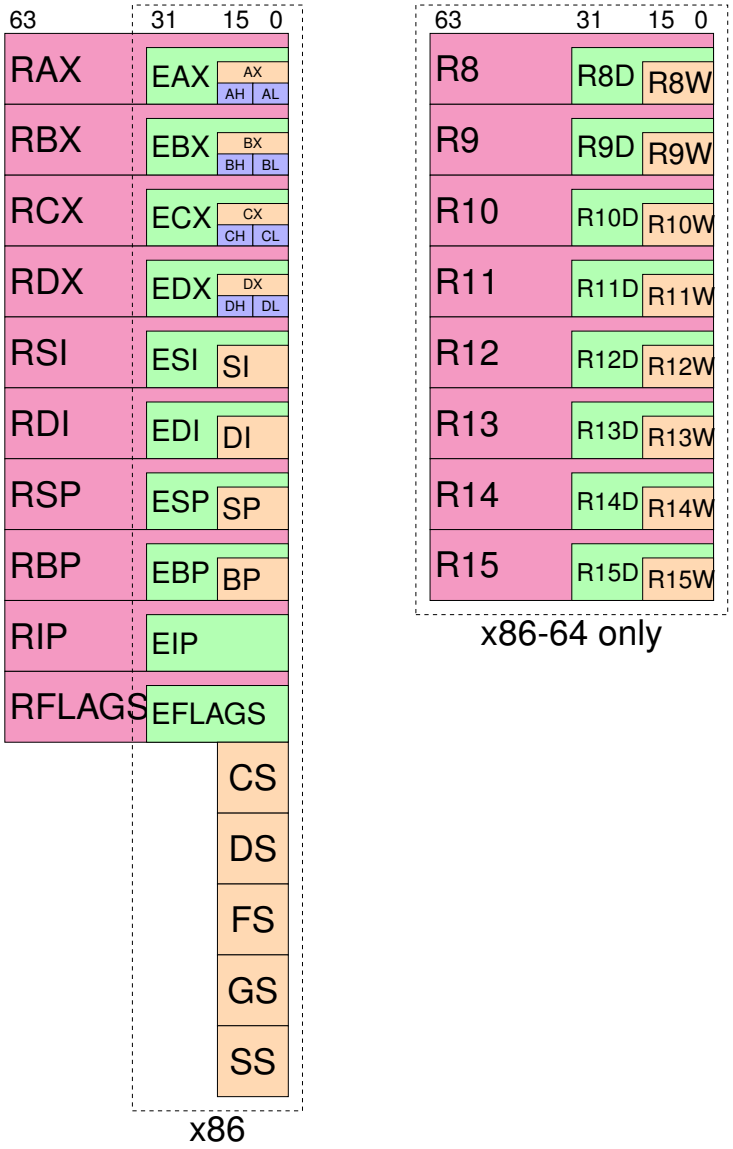
AArch64		
A64	:	Jeu d'instructions de taille fixe (32 bits)

Intel (x86 and x86-64)



Characteristics

- **x86** is a 32 bits CISC architecture.
- **x86-64** is a 64 bits CISC architecture.
- For majority of regiters, the prefix defines a fraction of the register:
  - **R** (RAX, RBX, ...): Full 64 bits register (x86-64 only);
  - **E** (EAX, EBX, ...): Full 32 bits register in X86, or lower 32 bits in x86-64;
  - **No prefix** (AX, BX, ...): Lower 16 bits register.
- The lower 16 bits can be further split using a suffix:
  - **X** (AX, BX, ...): Full 16 bits register;
  - **H** (AH, BH, ...): Higher 8 bits;
  - **L** (AL, BL, ...): Lower 8 bits.

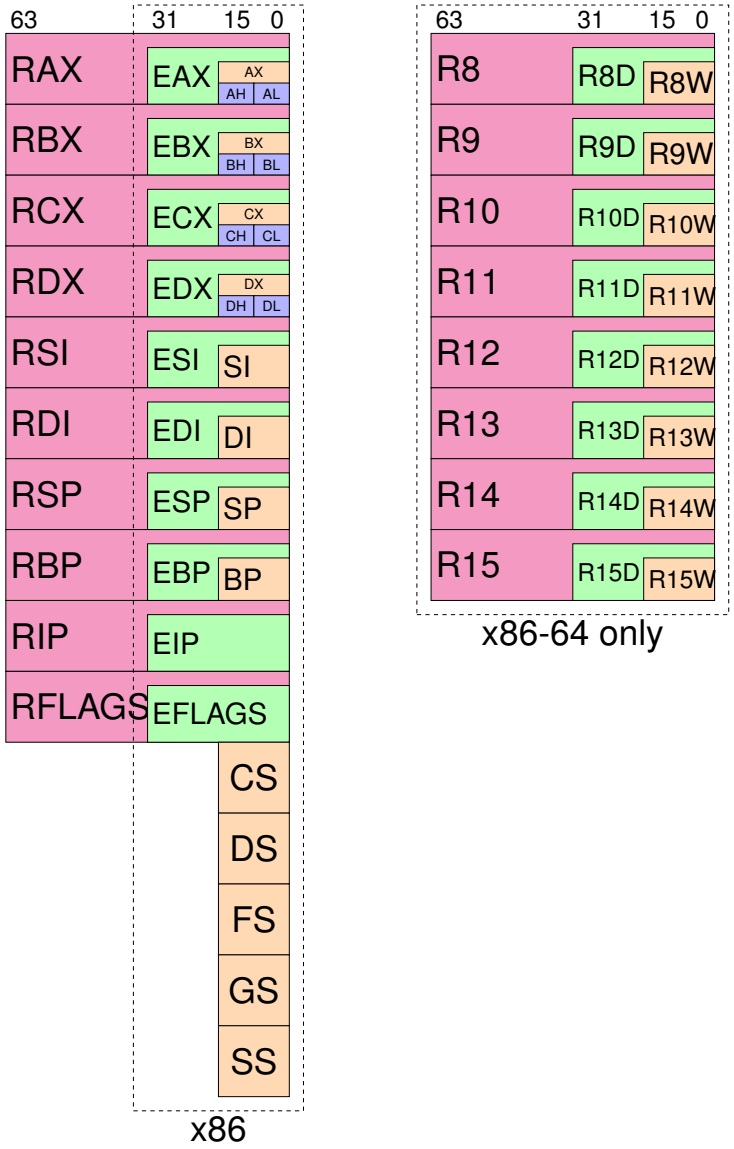


General purpose registers

4 general purpose registers which can have some special use:

- **EAX**: Accumulator Register, usually be used for I/O port access, arithmetic, interrupt calls.
- **EBX**: Base Register, usually used as a base pointer for memory access and some interrupt return values.
- **ECX**: Counter Register, usually used as a loop counter, shifts and some interrupt values.
- **EDX**: Data Register, usually used for I/O port access, arithmetic, some interrupt calls.

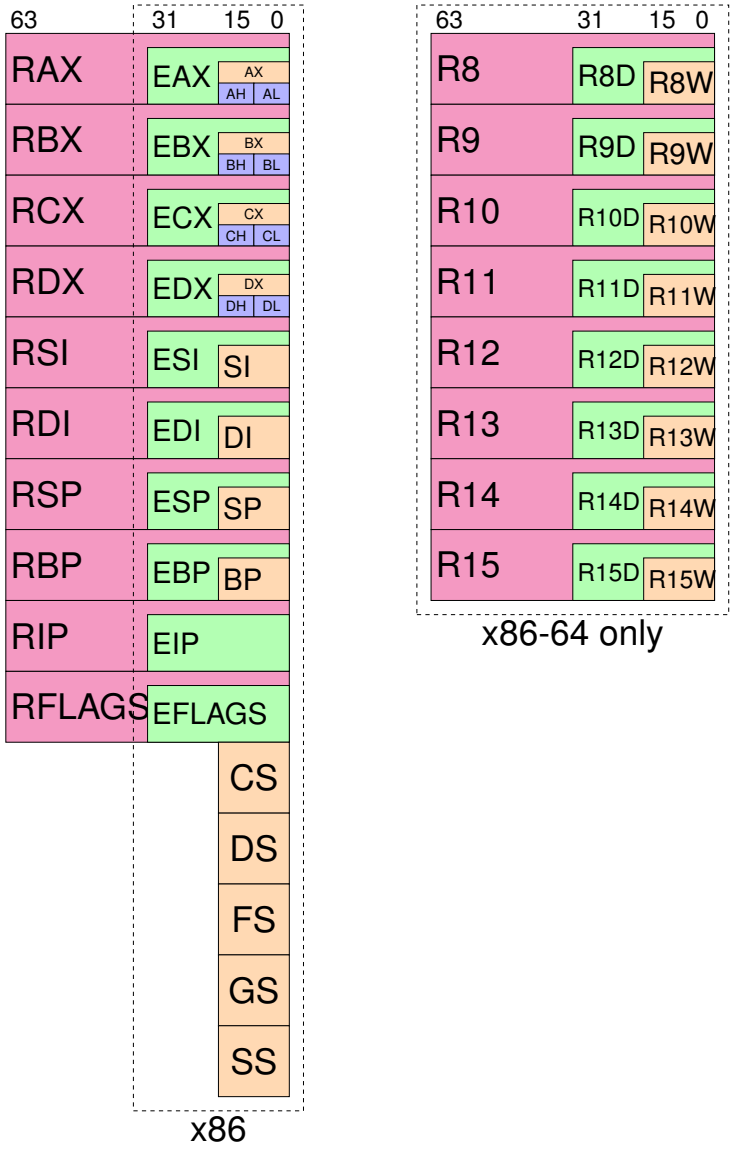
Intel (x86 and x86-64)



Segment registers

- No longer used in modern operating systems (with some exception).
- Used when memory was split into segments with different roles.
- Such registers where used to store information about address of segments
- Now, pagination is used instead.
- **CS**: Code Segment, holds the Code segment in which your program runs.
- **DS**: Data Segment, holds the Data segment that your program accesses
- **ES,FS,GS**: extra segment registers available for far pointer addressing like video memory and such. Used also for stack canaries.
- **SS**: Stack Segment, holds the Stack segment your program uses.

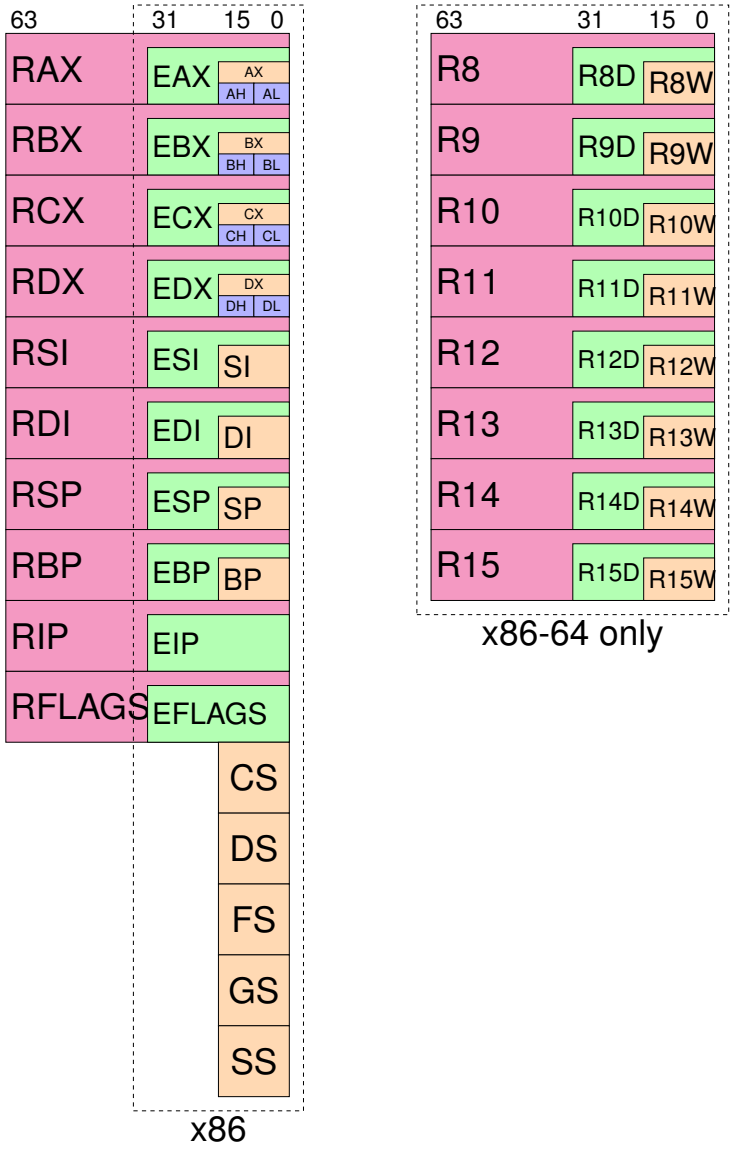
Intel (x86 and x86-64)



Index and pointer registers

- **EDI**: Destination Index, used for string, memory array copying and setting and for far pointer addressing with ES.
- **ESI**: Source Index, used for string and memory array copying.
- **EBP**: Base Pointer, holds the base address of the stack.
- **ESP**: Stack Pointer, holds the top address of the stack.
- **EIP**: Instruction Pointer, holds the offset of the next instruction.

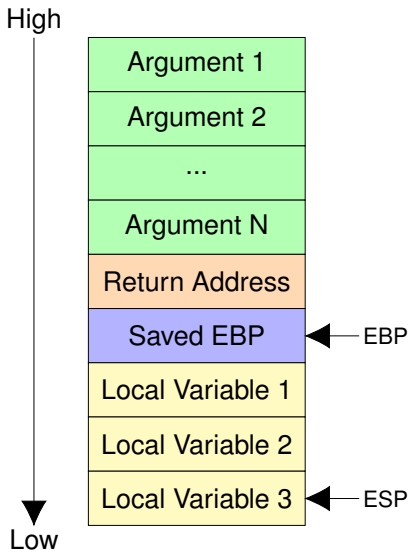
Intel (x86 and x86-64)



EFLAGS register

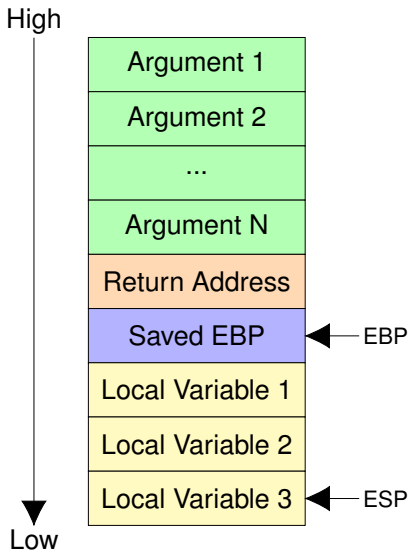
Used to hold the state of the processor:

- **CF**: Carry Flag.
- **PF**: Parity Flag.
- **AF**: Auxilliary Carry Flag.
- **ZF**: Zero flag.
- **SF**: Sign flag.
- **TF**: Trap flag.
- **IF**: Interrupt enable flag.
- **DF**: Direction flag.
- **OF**: Overflow flag.
- **IOPL**: I/O Priviledge level.
- **NT**: Nested task flag.
- **RF**: Resume flag.
- **VM**: Virtual 8086 mode flag.
- **AC**: Alignment check flag (486+).
- **VIF**: Virutal interrupt flag.
- **VIP**: Virtual interrupt pending flag.
- **ID**: ID flag.



Stack

- The Stack is full descending.
- Stack Pointer holds the current top address of the stack.
- Since Stack Pointer can change during routine execution, x86/x86-64 architectures introduce a stable pointer called Base Pointer.
- Note that Base Pointer is not necessary (Arm does not use such can of pointer), and is in general not used by compliers except for debug.



Stack Frame

- Each function holds a portion of the stack called Stack Frame.
- A stack frame can hold Local variables, return address, function arguments, ...
- for function call, arguments are stored in ascending order.
- Return value is stored in EAX.

Jeu d'instructions - Tableau comparatif

Opération	Arm Thumb2	x86 - Intel (Microsoft)	x86 - AT&T (Linux)
Copie registre	mov r0, r1 ←	mov eax, ebx ←	mov %ebx, %eax →
Adr immédiat	mov r0, #0x1C ←	mov eax, 0x1C ←	mov $0x1C, %eax →
Load indirect	ldr r0, [r2] ←	mov eax, [eci] ←	mov (%eci), %eax →
Store indirect	str r0, [r2] →	mov [ecx], eax ←	mov %eax,(%ecx) →
Load indirect + offset	ldr r0, [r2,0x18] ←	mov eax, [ecx+0x18] ←	mov 0x18(%ecx), %eax →
Store indirect + offset	str r0, [r2,0x1C] →	mov [ecx+0x1C], eax ←	mov %eax,0x1C(%ecx) →

Jeu d'instructions - Tableau comparatif

Opération	Arm Thumb2	x86 - Intel (Microsoft)	x86 - AT&T (Linux)
Load indexé	ldr r0, [r2,r3,LSL #2] ←	mov eax, [ecx+edx*4] ←	mov (%ecx,%edx,4), %eax →
Store indexé	str r0, [r2,r3,LSL #2] →	mov [ecx+edx*4], eax ←	mov %eax,(%ecx,%edx,4) →
Base + Index + offset		mov eax, [ecx+edx*8+0x24] ←	mov 0x24(%ecx,%edx,8), %eax →
Load effective address		lea eax, [ecx+edx*8+0x24] ←	lea 0x24(%ecx,%edx,8), %eax →

Jeu d'instructions - Tableau comparatif

Opération	Arm Thumb2	x86 - Intel (Microsoft)	x86 - AT&T (Linux)
Appel de sous-programme	bl Label	call label	call label
Retour de sous-programme	bx LR (feuille) pop {PC} (branche)	ret	ret
Retour de sous-programme avec incrément du pointeur de pile	 POP {PC} (branche)	ret 0x1C ret	ret 0x1C ret
Saut inconditionnel	b Label	jmp Label	jmp Label
Saut conditionnel (exemple : Z=1)	beq Label	je Label	je Label

Outils utilisés pour les TP

Chaîne de compilation GNU

La chaîne de compilation GNU est un ensemble d'outils libres permettant la compilation de fichier décrits en langage C/C++ et d'assemblage. Elle est composée de :

- **binutils :** suite d'outils permettant de manipuler les fichiers binaires. En particulier, contient un assembleur et un éditeur de lien.
- **gcc :** compilateur pour langage C/C++.
- **glibc :** bibliothèque permettant de gérer les appels systèmes ainsi que la gestion de fonctionnalités requises par certaines normes logicielles (POSIX, ISO C99, ...).
- **gdb :** debugger de fichiers binaires compilés avec GNU.
- **make :** interpréteur de scripts qui facilite l'automatisation des séquences de construction du projet (build).

Simulateur qemu

- Simulateur de nombreuses ISA (dont le cortex-M).
- Capable de s'interfacer avec gdb.