

Introduction au développement logiciel

Langage machine et langage d'assemblage : x86-64 et ARMv8-A

Georges-André Silber, Centre de recherche en informatique

Mines Paris — PSL, octobre 2026

Architectures externes et internes de processeurs

- **Spécification fonctionnelle** d'un processeur, du point de vue du programmeur en langage machine.
- Comprend :
 - le **jeu d'instructions** ;
 - les **registres** visibles par le programmeur ;
 - l'organisation de la **mémoire** et des **entrées-sorties** ;
 - les modalités d'un éventuel support **multiprocesseur**.
- Synonymes : *Instruction Set Architecture*, architecture de processeur, ou simplement *architecture* dans ce cours.

- L'**architecture interne**, ou **microarchitecture**, est l'implémentation pratique du comportement spécifié par l'architecture externe.
- Une architecture externe peut être implémentée par **plusieurs** microarchitectures : Intel et AMD proposent tous deux des processeurs x86; Apple, Fujitsu, Qualcomm et Arm elle-même conçoivent des cœurs ARMv8.
- Dans ce cours : **architectures externes** uniquement; la microarchitecture n'est abordée que par ses aspects comportementaux utiles à une programmation système efficace.
- Deux familles qui couvrent la majorité des machines actuelles : **x86-64** (Intel, AMD) et **ARMv8-A** en 64 bits (AArch64).

Ajouter un entier signé 32 bits à un emplacement mémoire contenant un entier signé 64 bits.

```
void f(long *a, int c)
{
    *a = *a + c;
}
```

```
fn f(a: &mut i64, c: i32) {
    let cst = c as i64;
    *a = *a + cst;
}
```

x86-64, avec a dans rdi et c dans esi :

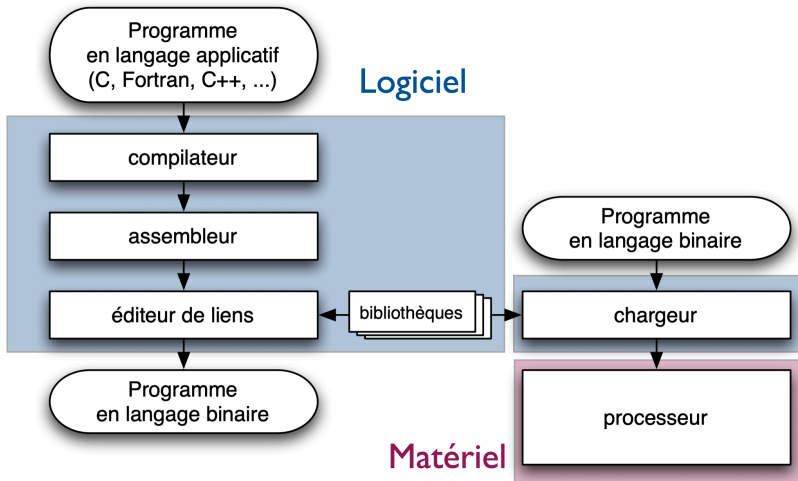
```
movsxd rsi, esi          # rsi <- SignExtend(esi)
add qword ptr [rdi], rsi  # memory[rdi] <- memory[rdi] + rsi
```

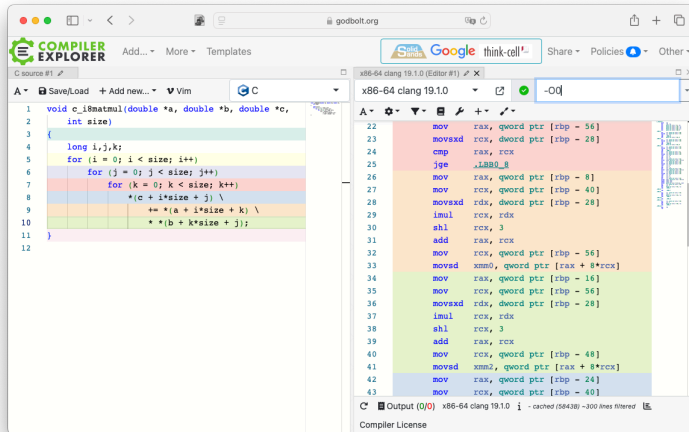
AArch64, avec a dans x0 et c dans w1 :

```
ldr      x2, [x0]          // x2 <- memory[x0]
add      x1, x2, w1, sxtw  // x1 <- x2 + SignExtend(w1)
str      x1, [x0]          // memory[x0] <- x1
```

x86 lit, additionne et écrit en mémoire en une instruction ; ARM, architecture *load/store*, en trois.

Chaîne de compilation





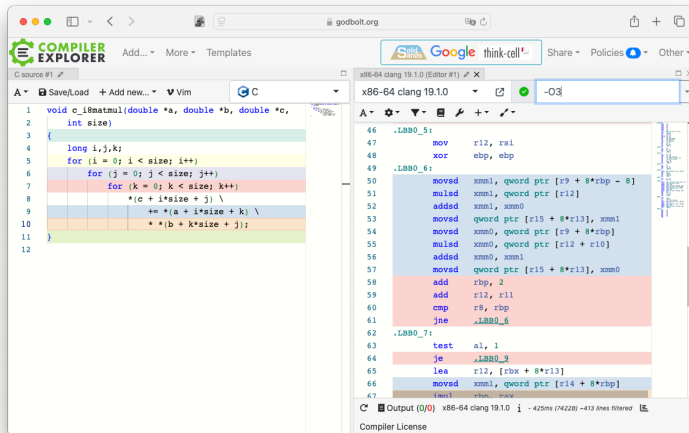
The screenshot shows the Compiler Explorer interface. On the left, the C source code is displayed in a file named 'C source #1'. The code is a function `void c_i8matmul(double *a, double *b, double *c, int size)` that performs a naive matrix multiplication using three nested loops. The code is highlighted with a color scheme. On the right, the assembly output for 'x86-64 clang 19.1.0' is shown. The assembly code is color-coded to match the C code, showing instructions like `mov rax, qword ptr [rbp - 56]`, `movsxd rcx, dword ptr [rbp - 28]`, `cmp rax, rcx`, `jge .LBB0_8`, `mov rax, qword ptr [rbp - 8]`, `mov rcx, qword ptr [rbp - 40]`, `movsxd rdx, dword ptr [rbp - 28]`, `imul rcx, rdx`, `shl rcx, 3`, `add rax, rcx`, `mov rcx, qword ptr [rbp - 56]`, `movsxd xmm0, qword ptr [rax + 8*rcx]`, `mov rax, qword ptr [rbp - 16]`, `mov rcx, qword ptr [rbp - 56]`, `movsxd rdx, dword ptr [rbp - 28]`, `imul rcx, rdx`, `shl rcx, 3`, `add rax, rcx`, `mov rcx, qword ptr [rbp - 48]`, `movsxd xmm2, qword ptr [rax + 8*rcx]`, `mov rax, qword ptr [rbp - 24]`, and `mov rcx, qword ptr [rbp - 40]`. The assembly is also color-coded. At the bottom, the output is shown as 'Output (0/0) x86-64 clang 19.1.0 i - cached (58438) ~300 lines filtered'. The Compiler License is also visible at the bottom.

```
1 void c_i8matmul(double *a, double *b, double *c,  
2 int size)  
3 {  
4     long i,j,k;  
5     for (i = 0; i < size; i++)  
6         for (j = 0; j < size; j++)  
7             for (k = 0; k < size; k++)  
8                 *(c + i*size + j) \  
9                 += *(a + i*size + k) \  
10                 * *(b + k*size + j);  
11 }  
12
```

```
22 mov     rax, qword ptr [rbp - 56]  
23 movsxd  rcx, dword ptr [rbp - 28]  
24 cmp     rax, rcx  
25 jge     .LBB0_8  
26 mov     rax, qword ptr [rbp - 8]  
27 mov     rcx, qword ptr [rbp - 40]  
28 movsxd  rdx, dword ptr [rbp - 28]  
29 imul    rcx, rdx  
30 shl     rcx, 3  
31 add     rax, rcx  
32 mov     rcx, qword ptr [rbp - 56]  
33 movsxd  xmm0, qword ptr [rax + 8*rcx]  
34 mov     rax, qword ptr [rbp - 16]  
35 mov     rcx, qword ptr [rbp - 56]  
36 movsxd  rdx, dword ptr [rbp - 28]  
37 imul    rcx, rdx  
38 shl     rcx, 3  
39 add     rax, rcx  
40 mov     rcx, qword ptr [rbp - 48]  
41 movsxd  xmm2, qword ptr [rax + 8*rcx]  
42 mov     rax, qword ptr [rbp - 24]  
43 mov     rcx, qword ptr [rbp - 40]
```

Output (0/0) x86-64 clang 19.1.0 i - cached (58438) ~300 lines filtered
Compiler License

Compilation avec optimisation -O3 (x86-64 "classique")

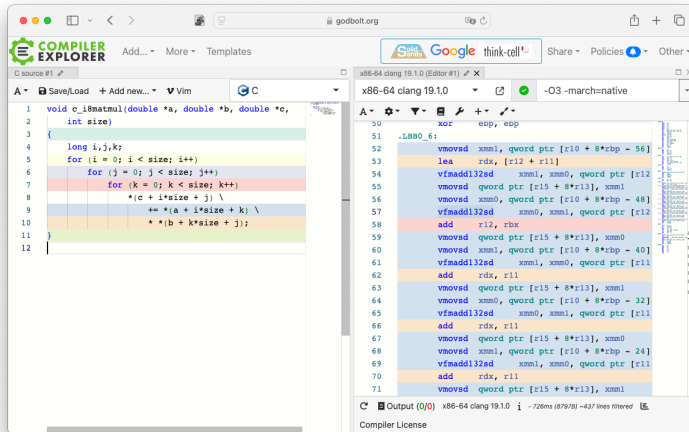


The screenshot shows the Compiler Explorer interface. On the left, the C source code is displayed with syntax highlighting and line numbers. The code is a function `c_i8matmul` that takes three double pointers and a size, and performs a matrix multiplication. On the right, the generated assembly code for x86-64 using clang 19.1.0 with the `-O3` optimization flag is shown. The assembly includes labels like `.LBB_5:`, `.LBB_6:`, and `.LBB_7:`, and instructions such as `mov r12, rsi`, `xor ebp, ebp`, `movsd xmm1, qword ptr [r9 + 8*rbp - 8]`, `mulsd xmm1, qword ptr [r12]`, `addsd xmm1, xmm0`, `movsd qword ptr [r15 + 8*r13], xmm1`, `movsd xmm0, qword ptr [r9 + 8*rbp]`, `mulsd xmm0, qword ptr [r12 + r10]`, `addsd xmm0, xmm1`, `movsd qword ptr [r15 + 8*r13], xmm0`, `add rbp, 2`, `add r12, r11`, `cmp r8, rbp`, `jne .LBB_6`, `test al, 1`, `je .LBB_9`, `lea r12, [rbx + 8*r13]`, and `movsd xmm1, qword ptr [r14 + 8*rbp]`. The bottom status bar indicates the output is for x86-64 clang 19.1.0, with a size of 425ms (74228) and 413 lines filtered.

```
1 void c_i8matmul(double *a, double *b, double *c,  
2 int size)  
3 {  
4     long i,j,k;  
5     for (i = 0; i < size; i++)  
6         for (j = 0; j < size; j++)  
7             for (k = 0; k < size; k++)  
8                 *(c + i*size + j) \  
9                 += *(a + i*size + k) \  
10                * *(b + k*size + j);  
11 }  
12
```

```
46 .LBB_5:  
47     mov     r12, rsi  
48     xor     ebp, ebp  
49 .LBB_6:  
50     movsd   xmm1, qword ptr [r9 + 8*rbp - 8]  
51     mulsd   xmm1, qword ptr [r12]  
52     addsd   xmm1, xmm0  
53     movsd   qword ptr [r15 + 8*r13], xmm1  
54     movsd   xmm0, qword ptr [r9 + 8*rbp]  
55     mulsd   xmm0, qword ptr [r12 + r10]  
56     addsd   xmm0, xmm1  
57     movsd   qword ptr [r15 + 8*r13], xmm0  
58     add     rbp, 2  
59     add     r12, r11  
60     cmp     r8, rbp  
61     jne     .LBB_6  
62 .LBB_7:  
63     test    al, 1  
64     je      .LBB_9  
65     lea     r12, [rbx + 8*r13]  
66     movsd   xmm1, qword ptr [r14 + 8*rbp]  
67     movsd   xmm1, qword ptr [r14 + 8*rbp]
```

Compilation avec optimisation -O3 (x86-64 spécifique)



The screenshot shows the Compiler Explorer interface. On the left, the C source code is displayed:

```
1 void c_i8matmul(double *a, double *b, double *c,  
2 int size)  
3 {  
4     long i,j,k;  
5     for (i = 0; i < size; i++)  
6         for (j = 0; j < size; j++)  
7             for (k = 0; k < size; k++)  
8                 *c + i*size + j \\  
9                 += *(a + i*size + k) \\  
10                * *(b + k*size + j);  
11 }  
12
```

On the right, the assembly output for x86-64 clang 19.1.0 with -O3 -march=native is shown:

```
30 xor     ebp, ebp  
51 .LBB0_6:  
52 vmovsd  xmm1, qword ptr [r10 + 8*rbp - 56]  
53 lea     rdx, [r12 + r11]  
54 vfmadd132sd  xmm1, xmm0, qword ptr [r12  
55 vmovsd  qword ptr [r15 + 8*r13], xmm1  
56 vmovsd  xmm0, qword ptr [r10 + 8*rbp - 48]  
57 vfmadd132sd  xmm0, xmm1, qword ptr [r12  
58 add     r12, rbx  
59 vmovsd  qword ptr [r15 + 8*r13], xmm0  
60 vmovsd  xmm1, qword ptr [r10 + 8*rbp - 40]  
61 vfmadd132sd  xmm1, xmm0, qword ptr [r11  
62 add     rdx, r11  
63 vmovsd  qword ptr [r15 + 8*r13], xmm1  
64 vmovsd  xmm0, qword ptr [r10 + 8*rbp - 32]  
65 vfmadd132sd  xmm0, xmm1, qword ptr [r11  
66 add     rdx, r11  
67 vmovsd  qword ptr [r15 + 8*r13], xmm0  
68 vmovsd  xmm1, qword ptr [r10 + 8*rbp - 24]  
69 vfmadd132sd  xmm1, xmm0, qword ptr [r11  
70 add     rdx, r11  
71 vmovsd  qword ptr [r15 + 8*r13], xmm1
```

The bottom status bar indicates: Output (0/0) x86-64 clang 19.1.0 i - 726ms (879/78) -437 lines filtered. Compiler License

**Ce qui est commun : x86-64 et
AArch64 côte à côte**

	x86-64	AArch64 (ARMv8-A)
Origine	Intel 8086 (1978); 64 bits : AMD (2003)	Acorn (1985); 64 bits : ARMv8 (2011)
Philosophie	CISC	RISC, <i>load/store</i>
Instructions	longueur variable, 1 à 15 octets	longueur fixe, 4 octets
Opérandes mémoire	dans presque toute instruction	<i>ldr/str</i> uniquement
Registres généraux	16 × 64 bits	31 × 64 bits + <i>sp, zr</i>
Drapeaux	<i>eflags</i> : CF ZF SF OF ...	PSTATE : N Z C V
SIMD	SSE/AVX <i>xmm/ymm/zmm</i> , 128–512 bits	NEON <i>v0–v31</i> 128 bits; SVE 128–2048
Privilèges	anneaux 0–3	niveaux d'exception EL0–EL3
Appel système	<i>syscall</i>	<i>svc #n</i>
Modèle mémoire	fortement ordonné (TSO)	faiblement ordonné (<i>dmb, ldar/stlr</i>)
Boutisme	<i>little-endian</i>	bi-boutiste; <i>little-endian</i> sous Linux, macOS
Syntaxes	Intel et AT&T	une seule (UAL)

- **Modèle de Von Neumann** : registres, mémoire adressée par octets, compteur ordinal, pile.
- **Mêmes types de données** : entiers de 8 à 64 bits, non signés ou en **complément à deux**; flottants **IEEE 754** simple et double précision; vecteurs SIMD.
- **Little-endian** en pratique : l'octet de poids faible à l'adresse la plus basse.
- **Trois classes d'instructions** : affectations (calcul, transfert), tests (positionnent des drapeaux), branchements (conditionnels ou non, appels et retours).
- **Deux niveaux de privilège** au moins : utilisateur et noyau, avec une instruction d'appel système pour passer de l'un à l'autre.
- **Une ABI** par couple OS et processeur : convention d'appel, représentation des types, cadre de pile.

Les figures Intel qui suivent décrivent donc aussi AArch64, au vocabulaire près.

- La mémoire est adressée par **octets**. Un entier de 32 bits occupe **quatre adresses consécutives** : lequel de ses octets va à la plus basse ?
- Deux conventions, pour $x = 0x12345678$ rangé à l'adresse $0x1000$:

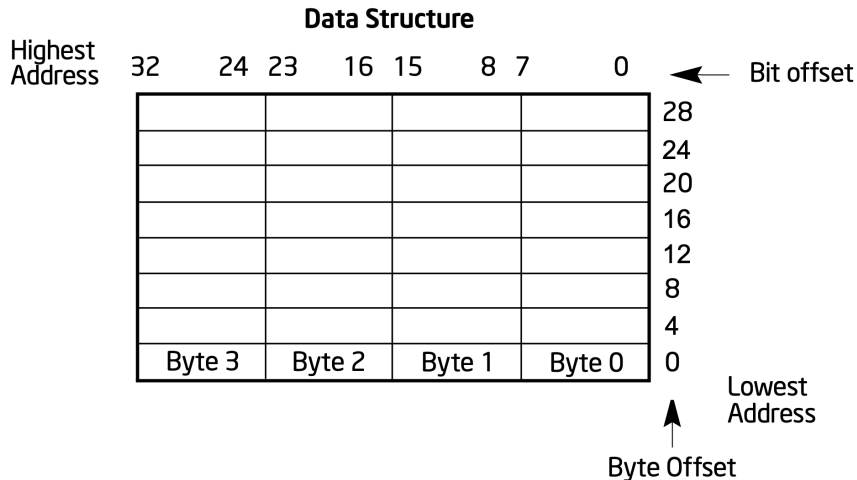
Adresse	0x1000	0x1001	0x1002	0x1003
little-endian , poids faible d'abord	78	56	34	12
big-endian , poids fort d'abord	12	34	56	78

- Les noms viennent de *Gulliver* : la guerre entre ceux qui cassent leur œuf par le petit bout et par le gros bout. Danny Cohen, *On Holy Wars and a Plea for Peace*, 1980 : le choix est arbitraire, l'important est de s'y tenir.
- x86** est *little-endian*. **ARM** sait faire les deux, et tous les systèmes courants le configurent en *little*. *Big-endian* : les anciens Motorola, SPARC et PowerPC, et surtout l'**ordre réseau** d'IP et TCP, d'où les fonctions `htonl` et `ntohl`.

- **Dans un vidage mémoire** : 0x12345678 se lit 78 56 34 12 sous xxd ou gdb.
- **En relisant une case avec un type plus court** : en *little-endian*, l'octet de poids faible d'un int est à la **même adresse** que l'int; a1 est le bas de eax pour la même raison. Ce test répond 1 sur x86 et ARM :

```
int x = 1;  
char *p = (char *) &x;    /* p[0]: 1 en little-endian, 0 en big-endian */
```

- **Dans les fichiers et sur le réseau** : chaque format fixe son ordre (PNG, Java, TCP en *big*; BMP, ELF x86, FAT en *little*). Écrire une structure C telle quelle sur disque n'est pas portable.
- **Dans le jeu d'instructions** : bswap et movbe sur x86-64, rev sur AArch64 retournent les octets d'un registre; c'est ce que compile htonl. Le compilateur connaît sa cible : __BYTE_ORDER__ en C.



Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.

Avec n bits on code 2^n valeurs. Pour des entiers naturels, de 0 à $2^n - 1$. Comment coder aussi les négatifs?

- **Signe et valeur absolue** : un bit de signe, $n - 1$ bits de valeur. Deux zéros (+0 et -0), et une addition qui doit regarder les signes : un circuit à part pour soustraire.
- **Complément à deux** : le bit de poids fort compte **négativement**. Sur 8 bits, les poids sont

-128	64	32	16	8	4	2	1
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

Autrement dit, $-x$ est représenté par $2^n - x$: c'est le « complément » de x à 2^n .

- **Plage sur n bits** : de -2^{n-1} à $2^{n-1} - 1$. Sur 8 bits, de -128 à 127; sur 32 bits, de -2 147 483 648 à 2 147 483 647. Un seul zéro, et un négatif de plus que de positifs.

Pour obtenir $-x$: **inverser tous les bits,**
puis ajouter 1.

5	0000 0101
bits inversés	1111 1010
+1	1111 1011 = -5

Vérification :

$-128 + 64 + 32 + 16 + 8 + 2 + 1 = -5$.

La même recette ramène de -5 à 5 :

1111 1011 inversé donne 0000 0100,
plus 1 donne 0000 0101.

Pourquoi ? Inverser donne $(2^n - 1) - x$;
ajouter 1 donne $2^n - x$.

Octet		Non signé	Signé
0000	0000	0	0
0000	0001	1	1
0111	1111	127	127
1000	0000	128	-128
1000	0001	129	-127
1111	1110	254	-2
1111	1111	255	-1

Les mêmes octets, deux lectures : le processeur ne sait pas lequel vous voulez dire, c'est l'instruction choisie qui le dit.

- **Un seul additionneur** pour les signés et les non signés : on additionne les bits et on jette la retenue sortante, qui vaut 2^n . $5 + (-5) = 0000\ 0101 + 1111\ 1011 = 1\ 0000\ 0000$, soit 0 sur 8 bits.
- **Soustraire**, c'est ajouter l'opposé : sub est add avec les bits inversés et une retenue entrante à 1.
- **Débordement** : $127 + 1 = 1000\ 0000 = -128$. Le résultat est juste modulo 2^8 , faux comme entier signé. Le processeur le signale par un drapeau (OF sur x86, V sur ARM), que C ignore : dépasser un `int` est un comportement indéfini.

- **Comparer** : `cmp` calcule une soustraction et positionne les drapeaux; c'est le **saut** qui choisit la lecture. Non signé : retenue (`jb/ja`, `lo/hi`); signé : signe et débordement (`j1/jg`, `lt/gt`). `1111 1111` est inférieur à `0000 0001` pour `j1`, supérieur pour `jb`.
- **Élargir** : de 32 à 64 bits, un non signé se complète par des zéros, un signé par des **copies de son bit de signe** (*sign extension*) : c'est `movsxd` et `sxtw` de l'exemple du début, contre `mov` et `uxtw`.

Class		Two's Complement Encoding	
		Sign	
Positive	Largest	0	11..11
		.	.
	Smallest	0	00..01
Zero		0	00..00
Negative	Smallest	1	11..11
		.	.
	Largest	1	00..00
Integer indefinite		1	00..00
		Signed Byte Integer:	← 7 bits →
		Signed Word Integer:	← 15 bits →
		Signed Doubleword Integer:	← 31 bits →
		Signed Quadword Integer:	← 63 bits →

Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.

- Un nombre fixe de bits ne code qu'un nombre fini de valeurs : il faut choisir **lesquelles**, et accepter d'**arrondir** les autres.
- **Virgule fixe** : k bits pour la partie entière, $n - k$ pour la partie fractionnaire. Simple (c'est un entier qu'on lit autrement), mais une seule échelle : impossible d'avoir la masse du Soleil et celle de l'électron dans le même format.
- **Virgule flottante** : la notation scientifique en base 2,

$$x = (-1)^s \times (1 + f) \times 2^e, \quad 0 \leq f < 1$$

un **signe** s , une **mantisse** $1 + f$ dont seule la fraction f est stockée, et un **exposant** e .
La virgule « flotte » : même précision relative à toute échelle.

- **IEEE 754** (1985, révisée en 2008 et 2019) fixe les formats, les arrondis et les cas particuliers : le même programme donne le même résultat sur toutes les machines. Avant, chaque constructeur avait les siens.

	Bits	Signe	Exposant	Mantisse	Biais	Décimales
float, simple précision	32	1	8	23	127	environ 7
double, double précision	64	1	11	52	1023	environ 16

- Le champ **exposant** E stocke e **décalé** du biais, $E = e + 127$, donc toujours positif.
- Le champ **mantisse** M , sur 23 bits, stocke la fraction : $f = M/2^{23}$. Le 1 devant la virgule est **implicite**, d'où 24 bits de précision pour 23 stockés. En simple précision : $x = (-1)^s \times (1 + M/2^{23}) \times 2^{E-127}$.
- Exemple, $-6,25 = -1,5625 \times 2^2$, et $0,5625 = 0,1001$ en binaire : signe 1, $E = 2 + 127 = 129 = 1000\ 0001$, $M = 1001$ suivi de 19 zéros :

1 10000001 100100000000000000000000 = 0xC0C80000

- Le double est le défaut de C; le float celui des GPU et de l'apprentissage automatique, qui descendent à 16 et 8 bits.

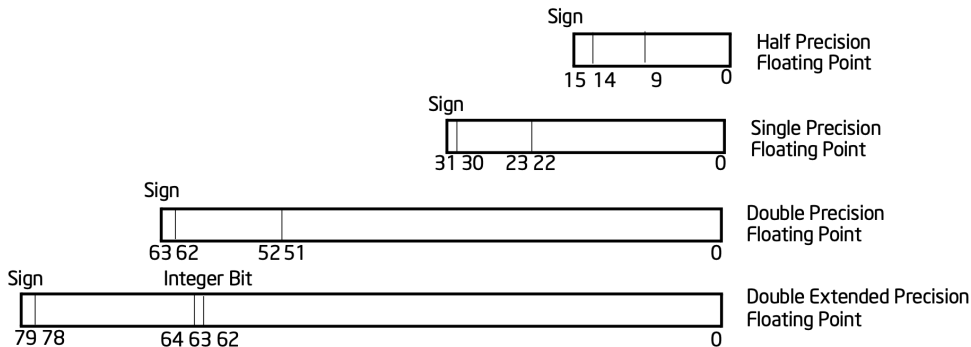
Exposant (simple)	Mantisse	Valeur
0000 0000	0	zéro , avec son signe : +0 et -0
0000 0000	$\neq 0$	dénormalisé : $f \times 2^{-126}$, sans le 1 implicite, précision réduite
de 0000 0001 à 1111 1110	quelconque	normalisé : $(1 + f) \times 2^{E-127}$
1111 1111	0	infini , $\pm\infty$: débordement, 1/0
1111 1111	$\neq 0$	NaN , <i>Not a Number</i> : 0/0, $\sqrt{-1}$, $\infty - \infty$

- Un calcul ne **plante jamais** : il produit un infini ou un NaN, qui se propage. NaN n'est égal à rien, pas même à lui-même : $x \neq x$ teste si x est un NaN.
- **Arrondi** au plus proche après chaque opération : 0,1 n'est pas représentable (comme 1/3 en décimal), et $0.1 + 0.2 \neq 0.3$ est faux. L'addition n'est **pas associative** dès que les échelles diffèrent : le compilateur ne réordonne pas sans -ffast-math.
- Les **entiers** sont exacts jusqu'à 2^{24} en float et 2^{53} en double : au-delà, $x + 1 = x$.

Format	Bits	Exp.	Mant.	Biais	Max	6,25 devient	Usage
fp32, binary32	32	8	23	127	$3,4 \times 10^{38}$	0x40C80000	référence, accumulations
fp16, binary16	16	5	10	15	65 504	0x4640	GPU, inférence, ARMv8.2
bf16, bfloat16	16	8	7	127	$3,39 \times 10^{38}$	0x40C8	entraînement (TPU, AMX)
fp8 E4M3	8	4	3	7	448	0x4C = 6	poids, activations
fp8 E5M2, bf8 chez AMD	8	5	2	15	57 344	0x46 = 6	gradients
fp4 E2M1	4	2	1	1	6	0x7 = 6	poids, échelle par bloc

- Même recette, moins de bits : un réseau de neurones tolère le bruit, et diviser la taille par deux double ce qui tient en cache et le débit des unités matricielles.
- **Étendue contre précision** : bf16 garde les 8 bits d'exposant de fp32, c'est fp32 tronqué à ses 16 bits de poids fort, conversion gratuite. fp16 est plus précis mais déborde à 65 504 : les gradients demandent un facteur d'échelle. E4M3 n'a pas d'infini ; E5M2 est fp16 tronqué.
- Les fp8 et fp4 des GPU récents (OCP 2023, Blackwell) portent une **échelle commune** par tenseur ou par bloc de 32 valeurs, et les produits sont **accumulés en fp32** : on gagne sur la mémoire et les multiplications, pas sur les sommes.

- Des **registres à part** : `xmm0` à `xmm15` sur x86-64, `d0` à `d31` (ou `s0` en simple) sur AArch64. Pas de flottant dans `rax` ou `x0`.
- Des **instructions à part**, avec la précision dans le nom : `addsd` et `mulsd` (*scalar double*) contre `addss`; `fadd d0, d1, d2` contre `fadd s0, s1, s2`. Les mêmes unités font du SIMD : `addpd` (*packed*) et `fadd v0.2d`.
- Des **conversions** explicites, parce que les bits ne veulent pas dire la même chose : `cvtss2sd` et `cvttsd2ss` (*truncate*), `scvtf` et `fcvtzs`. Le C les insère à chaque mélange `int` et `double`.
- **Comparer** a trois issues, pas deux : inférieur, supérieur ou égal, et **non ordonné** quand un opérande est NaN : `ucomisd` positionne ZF, PF, CF; `fcmp` positionne N, Z, C, V. Le saut doit en tenir compte.
- Pour les **appels** : les arguments flottants vont dans `xmm0` à `xmm7` ou `d0` à `d7`, le résultat dans `xmm0` ou `d0` : voir la partie ABI.



Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.

Data Type	Length	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Half Precision	16	11	2^{-14} to 2^{15}	3.1×10^{-5} to 6.50×10^4
Single Precision	32	24	2^{-126} to 2^{127}	1.18×10^{-38} to 3.40×10^{38}
Double Precision	64	53	2^{-1022} to 2^{1023}	2.23×10^{-308} to 1.79×10^{308}
Double Extended Precision	80	64	2^{-16382} to 2^{16383}	3.37×10^{-4932} to 1.18×10^{4932}

Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.

Class		Sign	Biased Exponent	Significand	
				Integer ¹	Fraction
Positive	$+\infty$	0	11..11	1	00..00
	+Normals	0	11..10	1	11..11
		.	.	.	.
		.	.	.	.
		0	00..01	1	00..00
	+Denormals	0	00..00	0	11..11
		.	.	.	.
		0	00..00	0	00..01
	+Zero	0	00..00	0	00..00
Negative	−Zero	1	00..00	0	00..00
	−Denormals	1	00..00	0	00..01
		.	.	.	.
		.	.	.	.
		1	00..00	0	11..11
	−Normals	1	00..01	1	00..00
		.	.	.	.
		1	11..10	1	11..11
	$-\infty$	1	11..11	1	00..00

Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.

Class		Sign	Biased Exponent	Significand	
				Integer ¹	Fraction
NaNs	SNaN	X	11..11	1	0X..XX ²
	QNaN	X	11..11	1	1X..XX
	QNaN Floating-Point Indefinite	1	11..11	1	10..00
	Half-Precision:		← 5Bits →		← 10 Bits →
	Single-Precision:		← 8 Bits →		← 23 Bits →
	Double-Precision:		← 11 Bits →		← 52 Bits →
	Double Extended-Precision:		← 15 Bits →		← 63 Bits →

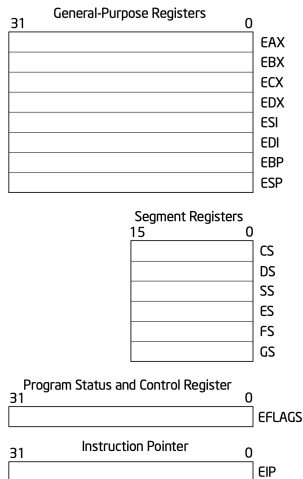
Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.

Bits	Ce cours	Intel	Arm	ABI System V
8	octet (<i>byte</i>)	byte	byte	byte
16	2 octets	word	halfword	halfword
32	4 octets	doubleword	word	word
64	8 octets	quadword	doubleword	doubleword
128	16 octets	double quadword	quadword	

Le même terme *word* désigne 16 bits chez Intel et 32 bits chez Arm : d'où les suffixes *word*, *dword*, *qword* de l'assembleur x86, et les registres *w* (32 bits) et *x* (64 bits) d'AArch64.

Architecture x86-64

- Premier représentant : le **8086** d'Intel, **1978**, architecture 16 bits.
- Passage à 32 bits (80386, 1985), puis à **64 bits** sous l'impulsion d'**AMD** (2003) : Intel misait alors sur une nouvelle architecture, l'Itanium, aujourd'hui disparue.
- AMD64, x86-64, x64, EM64T, Intel 64 : **synonymes**.
- Deux modes principaux de fonctionnement :
 - Mode réel** (*real mode*) l'environnement du 8086 des origines, avec des extensions. Le processeur **démarre toujours** dans ce mode.
 - Mode protégé** le mode **natif**, et son extension 64 bits le **mode long** : celui qui permet à un système d'exploitation d'être multi-tâches et d'avoir une mémoire « à plat ».



eax accumulateur : opérandes et résultats

ebx pointeur vers les données du segment ds

ecx compteur pour les boucles et les chaînes

edx pointeur d'entrées-sorties

esi pointeur source (chaînes)

edi pointeur destination (chaînes, segment es)

esp pointeur de pile (segment ss)

ebp pointeur de données sur la pile

General-Purpose Registers

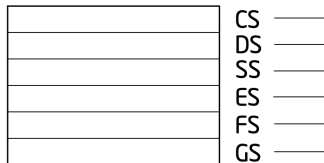
31	16 15	8 7	0	16-bit	32-bit
	AH	AL		AX	EAX
	BH	BL		BX	EBX
	CH	CL		CX	ECX
	DH	DL		DX	EDX
	BP				EBP
	SI				ESI
	DI				EDI
	SP				ESP

Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.

Register Type	Without REX	With REX
Byte Registers	AL, BL, CL, DL, AH, BH, CH, DH	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8B - R15B
Word Registers	AX, BX, CX, DX, DI, SI, BP, SP	AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W
Doubleword Registers	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D
Quadword Registers	N.A.	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15

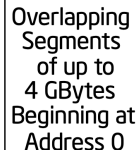
Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.

Segment Registers

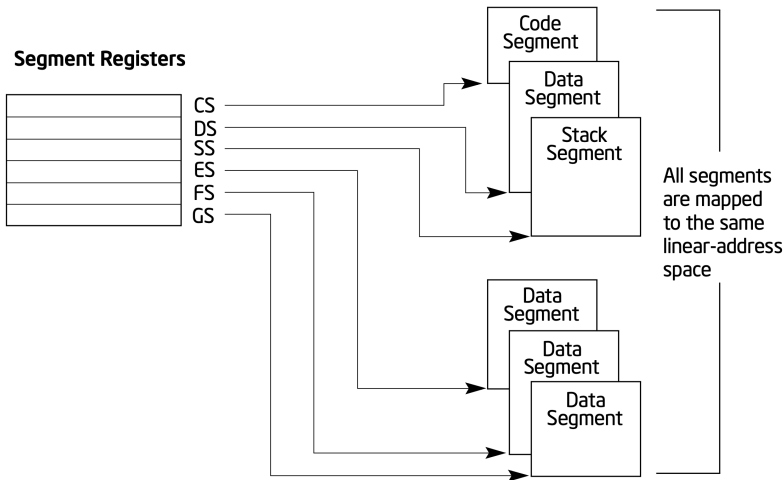


The segment selector in each segment register points to an overlapping segment in the linear address space.

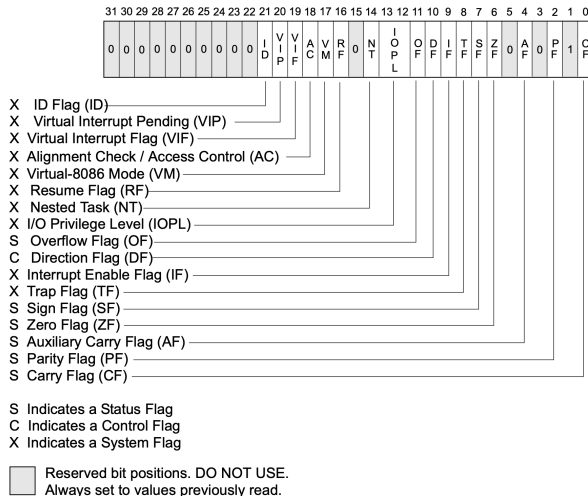
Linear Address Space for Program



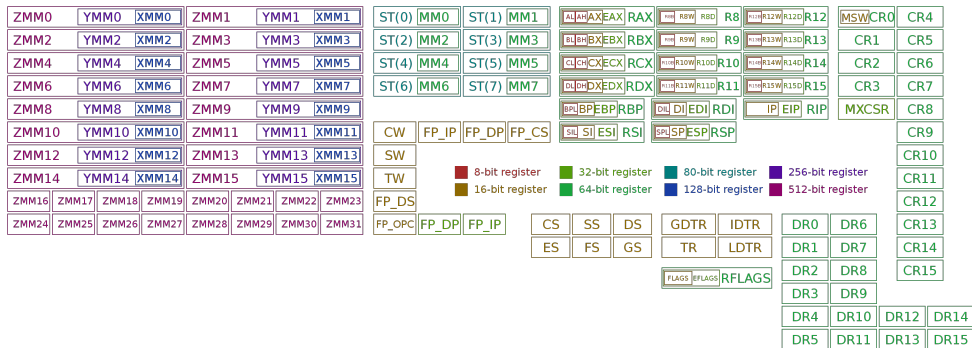
Overlapping
Segments
of up to
4 GBytes
Beginning at
Address 0



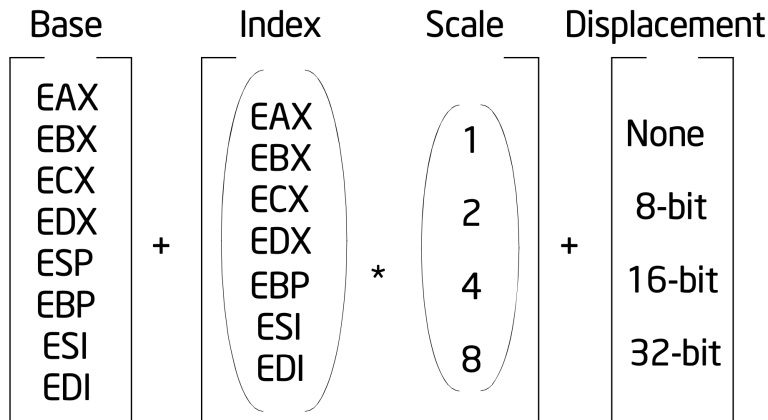
Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.



Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.

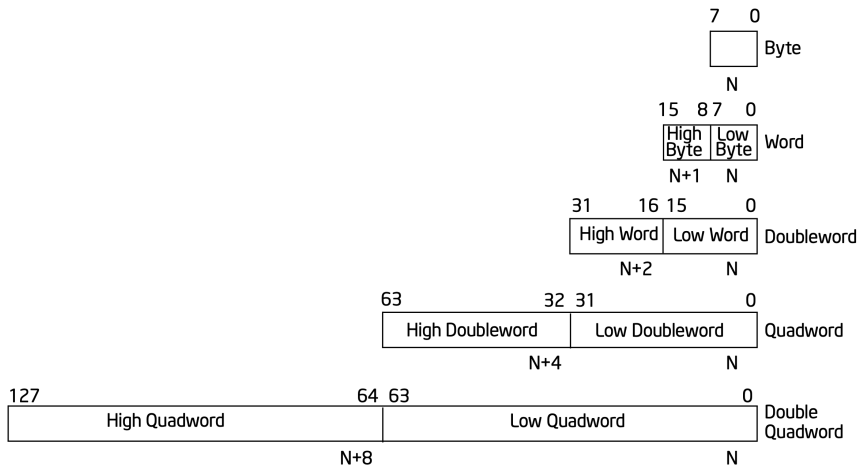


Source : Wikipedia. Instructions : documentation Intel ou <https://sandpile.org>.

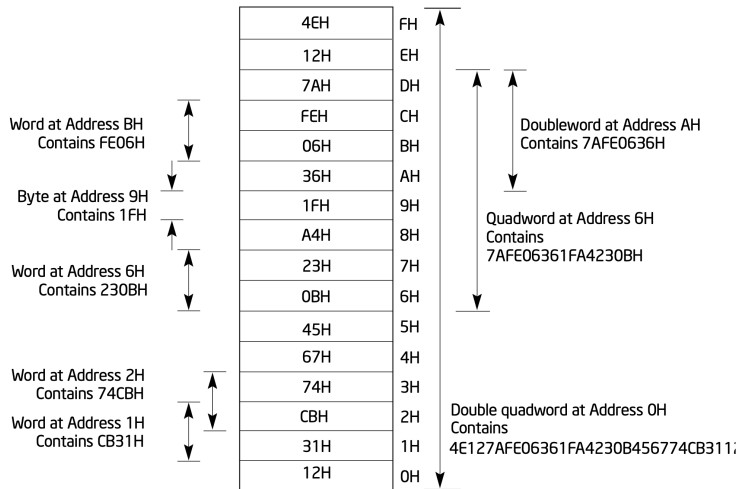


$$\text{Offset} = \text{Base} + (\text{Index} * \text{Scale}) + \text{Displacement}$$

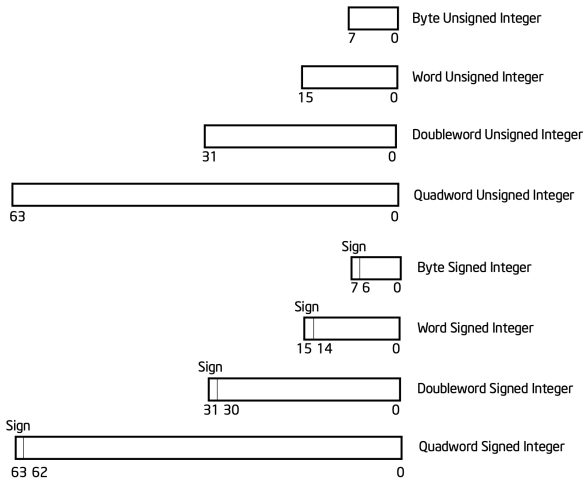
Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.



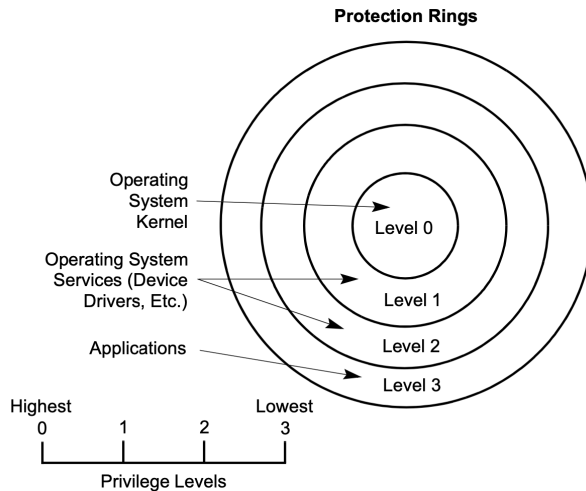
Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.



Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.



Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.



Source : Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1.

	Intel (nasm, .intel_syntax)	AT&T (gas)
Ordre	destination, source	source, destination
Exemple	mov eax, 1	movl \$1, %eax
Mémoire	[base + index*taille + offset]	offset(base, index, taille)

<https://stackoverflow.com/questions/13793609/nasm-vs-gas-practical-differences>

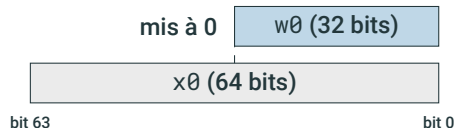
Architecture ARMv8-A

- Conçue chez Acorn Computers en **1983–1985** : *Acorn RISC Machine*, puis *Advanced RISC Machines*, stylisé arm.
- *Arm Ltd.* (1990) : coentreprise Acorn, Apple et VLSI Technology.
- Pendant trente ans, *Arm Ltd.* **n'a rien fabriqué** : elle vendait uniquement de la **propriété intellectuelle** :
 - des **architectures externes**, sous licence à des compagnies qui conçoivent leurs propres cœurs et SoC (Apple, Fujitsu, Qualcomm);
 - des **cœurs** implémentant ces architectures (Cortex-A, Cortex-M, Neoverse), intégrés par d'autres dans leurs SoC (STMicroelectronics, Broadcom).
- Ce n'est **plus vrai** depuis le 24 mars 2026 : Arm vend son premier processeur, l'**Arm AGI CPU** (136 cœurs Neoverse V3, pour les centres de données d'IA, Meta comme premier client) et devient concurrente de ses propres licenciés.

- **ARM1** (1985) : structure interne 32 bits, adressage sur 26 bits, donc 64 Mo de mémoire principale au plus.
- Évolution vers un adressage sur 32 bits ; jeu **Thumb** (instructions sur 16 puis 16 et 32 bits, *Thumb-2*) pour la densité du code ; **Jazelle** pour le bytecode Java.
- **ARMv7** : trois profils, **A** (applications, avec MMU), **R** (temps réel), **M** (microcontrôleurs, Thumb-2 seulement).
- **ARMv8-A** (2011) : architecture **64 bits**. Deux états d'exécution : **AArch64** (jeu A64) et **AArch32** (jeux A32 et T32, compatibles ARMv7).
- Révisions annuelles : ARMv8.1 (atomiques LSE), **ARMv8.2** (flottants demi-précision, extension vectorielle **SVE** optionnelle), ..., ARMv9 (2021).

- Architecture relativement simple, **faible consommation** : dominants dans l'**embarqué**, la téléphonie mobile, les tablettes.
- Postes de travail et serveurs : puces **Apple M1** à M4 (ARMv8.4-A et plus), serveurs **Neoverse** (AWS Graviton, Ampere).
- Supercalculateur **Fugaku** : Fujitsu **A64FX**, **ARMv8.2-A** avec SVE 512 bits, 48 cœurs de calcul, 158 976 processeurs, 7 630 848 cœurs.
- **Raspberry Pi 4 B** : SoC Broadcom BCM2711, quatre cœurs Cortex-A72 (ARMv8-A) à 1,5 GHz.
- Carte **Nucleo** des TP : STM32F746, cœur **Cortex-M7**, ARMv7E-M, 32 bits, Thumb-2 : **pas** AArch64.

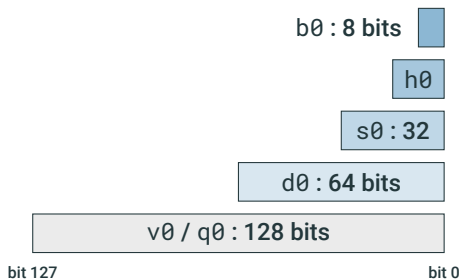
- **31 registres** de 64 bits, x_0 à x_{30} .
- w_0 à w_{30} : les **32 bits de poids faible**. Écrire dans w_n **met à zéro** les 32 bits de poids fort.
- Le numéro 31 code, selon l'instruction, le **pointeur de pile** sp ou le **registre zéro** xzr/wzr (lit 0, ignore les écritures).
- pc n'est **pas** un registre général : lu seulement par adr , $adrp$, b , bl .
- Par convention : $x_{30} = lr$ (adresse de retour, écrite par bl), $x_{29} = fp$.



Registres spéciaux : sp , xzr , pc , $nzcv$.

Source : Arm Architecture Reference Manual for A-profile architecture (ARM DDI 0487).

- **32 registres** de 128 bits, v_0 à v_{31} , partagés entre le **flottant scalaire** et le **SIMD** (NEON, *Advanced SIMD*).
- Vues scalaires : q_0 (128), d_0 (64, double), s_0 (32, float), h_0 (16), b_0 (8).
- Vues vectorielles : $v_0.2d$, $v_0.4s$, $v_0.8h$, $v_0.16b$, ...
- **SVE** (ARMv8.2-A, optionnel) : registres z_0 à z_{31} de **128 à 2048 bits** selon l'implémentation (512 sur l'A64FX), prédicats p_0 à p_{15} .



Source : Arm Architecture Reference Manual for A-profile architecture (ARM DDI 0487).

- PSTATE contient quatre drapeaux : **N** négatif, **Z** zéro, **C** retenue, **V** débordement signé.
- Seules les instructions **suffixées** s les modifient : `adds`, `subs`, `ands`; `cmp` $\equiv$ `subs xzr`, `tst` $\equiv$ `ands xzr`.
- Branchement conditionnel `b.cond`; aussi `cbz`, `cbnz`, `tbz`, `tbnz` sans drapeaux.
- **Sélection conditionnelle** sans branchement : `csel`, `cset`, `cinc`, `csneg`.

Code	Sens	Drapeaux
<code>eq / ne</code>	égal / différent	Z
<code>hs / lo</code>	$\geq / <$ non signé	C
<code>mi / pl</code>	négatif / positif	N
<code>vs / vc</code>	débordement / non	V
<code>hi / ls</code>	$> / \leq$ non signé	C, Z
<code>ge / lt</code>	$\geq / <$ signé	N, V
<code>gt / le</code>	$> / \leq$ signé	N, V, Z
<code>al</code>	toujours	

Source : Arm Architecture Reference Manual for A-profile architecture (ARM DDI 0487).

```
ldr  x0, [x1]                // x0 <- mem[x1]
ldr  x0, [x1, #16]           // mem[x1 + 16] (imm. 12 bits, x8)
ldur x0, [x1, #-8]           // mem[x1 - 8] (imm. 9 bits signé)
ldr  x0, [x1, x2]            // mem[x1 + x2]
ldr  x0, [x1, x2, lsl #3]     // mem[x1 + x2*8]
ldr  x0, [x1, w2, sxtw #3]    // mem[x1 + SignExtend(w2)*8]
ldr  x0, [x1, #16]!          // pré-indexé: x1 <- x1 + 16, puis charge
ldr  x0, [x1], #16           // post-indexé: charge, puis x1 <- x1 + 16
ldp  x0, x1, [sp], #16       // paire: dépile x0 et x1
ldrb w0, [x1]                // octet étendu par zéro (ldrsb: mot, par signe)
adrp x0, msg                  // page relative au pc (+/-4 Go)
add  x0, x0, :lo12:msg       // + bas de l'adresse
```

Comparer avec x86 : $[base + index \times échelle + déplacement]$ dans n'importe quelle instruction.

- **Immédiats limités** : une constante 64 bits se construit en plusieurs instructions.

```
mov  x0, #0x1234                // movz: 16 bits
movk x0, #0x5678, lsl #16        // insère 16 bits de plus
movk x0, #0x9abc, lsl #32
movk x0, #0xdef0, lsl #48        // x0 = 0xdef09abc56781234
```

- Les immédiats de add/sub tiennent sur 12 bits (décalables de 12); ceux de and/orr/eor sont des **masques codés**.
- Les **adresses** sont relatives au pc (adr ± 1 Mo, adrp ± 4 Go) ou dans un registre.
- **Branchement** b ± 128 Mo, b.cond ± 1 Mo.
- En contrepartie : décodage simple, pipeline régulier.

Niveau	Usage	Entrée	Équivalent x86
EL0	applications		anneau 3
EL1	noyau de l'OS	svc	anneau 0
EL2	hyperviseur	hvc	mode racine VMX
EL3	moniteur sécurisé (TrustZone)	smc	

- Chaque niveau a ses **registres système** (sp_el1, elr_el1, spsr_el1, ...) et sa **table de vecteurs** d'exceptions.
- Interruptions, fautes et appels système **montent** d'un niveau; eret redescend.
- TrustZone sépare deux **mondes**, sécurisé et normal, sur le même cœur.

Source : Arm Architecture Reference Manual for A-profile architecture (ARM DDI 0487).

- **32 bits**, jeu **Thumb-2** seulement, **pas** d'AArch64.
- **16 registres** : r0 à r12, sp (r13), lr (r14), pc (r15); xpsr pour les drapeaux.
- Pas de MMU : **adresses physiques**, MPU optionnelle.
- Deux modes, *Thread* et *Handler* ; interruptions par le **NVIC**, vecteurs en début de Flash.
- STM32F746 : FPU simple précision, 216 MHz, 1 Mo de Flash, 320 Ko de RAM.

r0-r3	arguments, résultats
r4-r11	sauvés par l'appelé
r12	ip, temporaire
r13	sp
r14	lr
r15	pc

Convention AAPCS 32 bits : mêmes principes qu'en 64 bits, moitié moins de registres.

**ABI : System V AMD64, AAPCS64,
Apple arm64**

- Une **ABI** (*Application Binary Interface*) définit la manière dont les programmes binaires s'interfacent :
 - les **conventions d'appel** de fonction : où vont les arguments, les résultats, qui sauve quels registres ;
 - la **représentation des types** : taille, alignement, signe ;
 - le **cadre de pile** ;
 - la normalisation des noms par le compilateur (*name mangling*), donc ce que l'éditeur de liens peut trouver.
- Une ABI par couple **OS et processeur** :

	x86-64	AArch64
Linux, BSD	System V AMD64	AAPCS64 (Arm)
macOS, iOS	System V AMD64	AAPCS64, variante Apple
Windows	Microsoft x64	AAPCS64, variante Microsoft

- **LP64** partout ici : int 32 bits, long et pointeurs 64 bits (Windows : LLP64).

	Type C	sizeof	Align.	Linux x86-64	Linux AArch64
Entier	char	1	1	signé	non signé
	short	2	2		
	int, enum	4	4		
	long, long long	8	8		
	__int128	16	16		
Pointeur	T *, T (*)()	8	8		
Flottant	float	4	4	IEEE simple	IEEE simple
	double	8	8	IEEE double	IEEE double
	long double	16	16	x87, 80 bits	IEEE, 128 bits
Vectoriel	__m128 / int32x4_t	16	16	SSE	NEON
	svint32_t	variable	16		SVE

Apple arm64 : char **signé** et long double = double (8 octets). Ne jamais supposer le signe de char : écrire signed char ou unsigned char.

Registre	Usage	Sauvé par l'appelé
rax	temporaire ; 1 ^{er} registre de retour	non
rbx		oui
rcx	4 ^e argument entier	non
rdx	3 ^e argument ; 2 ^e registre de retour	non
rsp	pointeur de pile	oui
rbp	pointeur de cadre (optionnel)	oui
rsi	2 ^e argument	non
rdi	1 ^{er} argument	non
r8, r9	5 ^e et 6 ^e arguments	non
r10, r11	temporaires	non
r12-r15		oui
xmm0, xmm1	arguments et retours flottants	non
xmm2-xmm7	arguments flottants	non
xmm8-xmm15	temporaires	non

Registre	Usage	Sauvé par l'appelé
x0-x7	arguments entiers et pointeurs; résultats dans x0, x1	non
x8	adresse du résultat indirect (structure retournée en mémoire)	non
x9-x15	temporaires	non
x16, x17	ip0, ip1 : temporaires de l'éditeur de liens (PLT, veneers)	non
x18	registre de plateforme : réservé sous macOS et Windows, libre sous Linux	
x19-x28		oui
x29	fp, pointeur de cadre	oui
x30	lr, adresse de retour	oui
sp	pointeur de pile, aligné sur 16 octets à tout accès	oui
v0-v7	arguments et résultats flottants et SIMD	non
v8-v15	sauvés par l'appelé : 64 bits de poids faible seulement	partiel
v16-v31	temporaires	non

Huit registres d'arguments entiers contre six en x86-64; les drapeaux `nzcv` ne sont jamais préservés.

Source : *Procedure Call Standard for the Arm 64-bit Architecture (AAPCS64)*, Arm; *Writing ARM64 code for Apple platforms*, Apple.

Position	Contenu	Cadre
$8n+16(\text{rbp})$	argument mémoire n (8 octets)	Précédent
	...	
$16(\text{rbp})$	argument mémoire 0 (8 octets)	Courant
$8(\text{rbp})$	adresse de retour (empilée par <code>call</code>)	
$0(\text{rbp})$	valeur précédente de <code>rbp</code>	
$-8(\text{rbp})$	variables locales	
	...	
$0(\text{rsp})$	taille variable	Courant
$-128(\text{rsp})$	zone rouge (<i>red zone</i>)	

`rsp` aligné sur 16 octets avant `call`. Zone rouge : 128 octets sous `rsp` utilisables sans l'ajuster.

- **bl n'empile rien** : l'adresse de retour va dans `lr`. C'est à la fonction appelante non-feuille de la sauver.
- **Enregistrement de cadre** (*frame record*) : la paire `{fp, lr}` empilée, `fp` pointant dessus ; les cadres forment une liste chaînée.
- `sp` toujours **aligné sur 16 octets** quand il sert de base d'accès (faute sinon).
- **Pas de zone rouge** sous Linux.

```
f: stp x29, x30, [sp, #-32]!  
   mov x29, sp  
   str x19, [sp, #16]  
   // ... corps ...  
   ldr x19, [sp, #16]  
   ldp x29, x30, [sp], #32  
   ret
```

sp+32	arguments mémoire, cadre précédent
sp+24	(libre)
sp+16	x19 sauvé
sp+8	lr sauvé
sp+0 = fp	fp précédent

long f(long a, long b, double x, long c) :

	a	b	x	c	résultat
System V AMD64	rdi	rsi	xmm0	rdx	rax
AAPCS64	x0	x1	d0	x2	x0

- Entiers et flottants comptés **séparément** : 6 entiers et 8 flottants en registres sur x86-64, 8 et 8 sur AArch64.
- Un entier de moins de 8 octets occupe un registre entier; sous Linux AArch64 ses bits de poids fort sont **non spécifiés** (Apple exige leur extension).
- Une **structure** de 16 octets au plus passe en registres; au-delà, par copie sur la pile en x86-64, par **pointeur** vers une copie en AArch64.
- Retour d'une grande structure : adresse fournie par l'appelant dans `rdi` (x86-64) ou `x8` (AArch64).

	Linux (AAPCS64)	macOS, iOS (Apple)
x18	registre général	réservé , ne jamais l'écrire
x29	pointeur de cadre optionnel	obligatoire , cadres chaînés
char	non signé	signé
long double	128 bits	64 bits (= double)
Arguments variadiques	comme les autres	toujours sur la pile
Arguments sur la pile	tranches de 8 octets	alignement naturel , tassés
Appel système	x8 = numéro, svc #0	x16 = numéro, svc #0x80
Erreur d'appel système	x0 = -errno	C positionné, x0 = errno
Format binaire	ELF	Mach-O, pages de 16 Ko

Conséquence pratique : du code assembleur ou un *printf* variadique écrit pour l'un ne fonctionne pas tel quel sur l'autre.

Source : *Procedure Call Standard for the Arm 64-bit Architecture (AAPCS64)*, Arm; *Writing ARM64 code for Apple platforms*, Apple.

	Instruction	Numéro	Arguments	Résultat
Linux x86-64	syscall	rax	rdi rsi rdx r10 r8 r9	rax; rcx, r11 détruits
Linux AArch64	svc #0	x8	x0-x5	x0
macOS x86-64	syscall	rax = 0x20000000 + n	rdi rsi rdx r10 r8 r9	rax, erreur si CF
macOS AArch64	svc #0x80	x16	x0-x5	x0, erreur si C

- Les **numéros diffèrent** entre OS et entre architectures : `write` vaut 1 sur Linux x86-64, 64 sur Linux AArch64, 4 sur macOS.
- Linux garantit la stabilité de ses appels système; macOS ne la garantit que pour la **libSystem** : en pratique on passe par la bibliothèque C.
- `man 2 syscall`, <https://filippo.io/linux-syscall-table/>,
`arch/arm64/include/asm/unistd.h`.

x86-64 (syntaxe Intel) :

```
.globl _start
_start:
    mov    rax, 1          # write
    mov    rdi, 1          # stdout
    lea    rsi, [rip+msg]
    mov    rdx, 13
    syscall
    mov    rax, 60         # exit
    xor    rdi, rdi
    syscall
msg: .ascii "Hello world!\n"
```

AArch64 :

```
.globl _start
_start:
    mov    x8, #64         // write
    mov    x0, #1          // stdout
    adr    x1, msg
    mov    x2, #13
    svc    #0
    mov    x8, #93         // exit
    mov    x0, #0
    svc    #0
msg: .ascii "Hello world!\n"
```

Même structure, registres et numéros différents. `as + ld`, sans `libc`; `qemu-aarch64` exécute le second sur un PC.