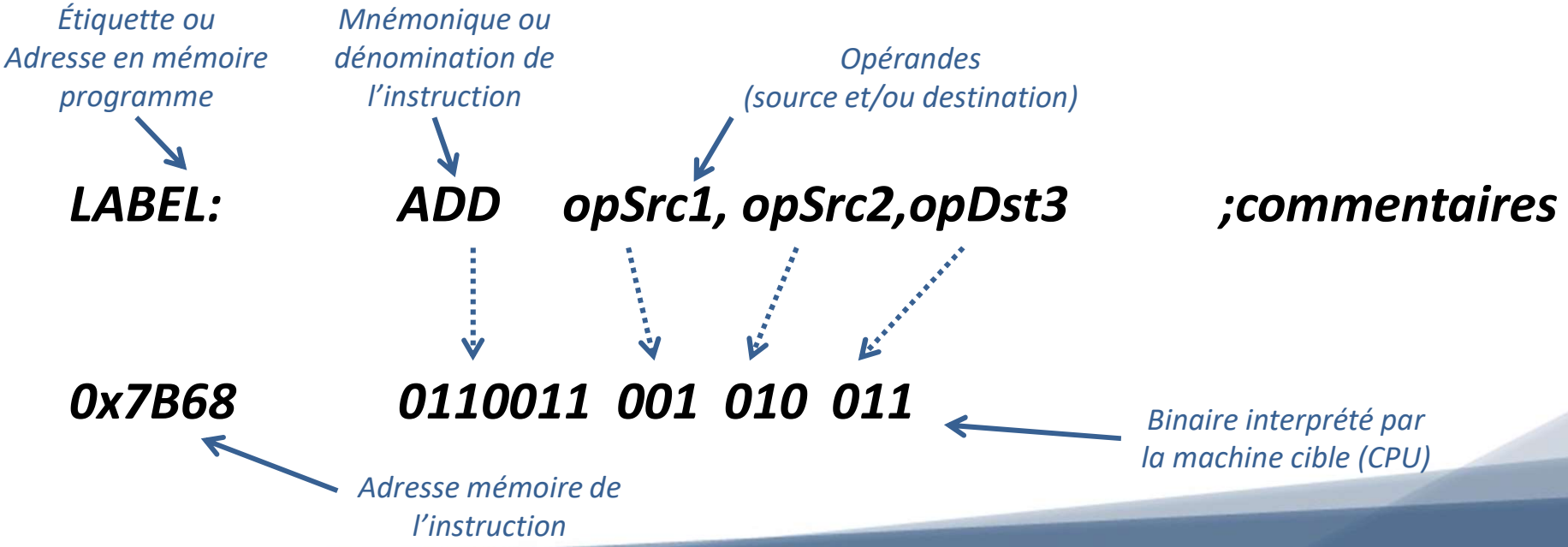
LANGAGE D'ASSEMBLAGE

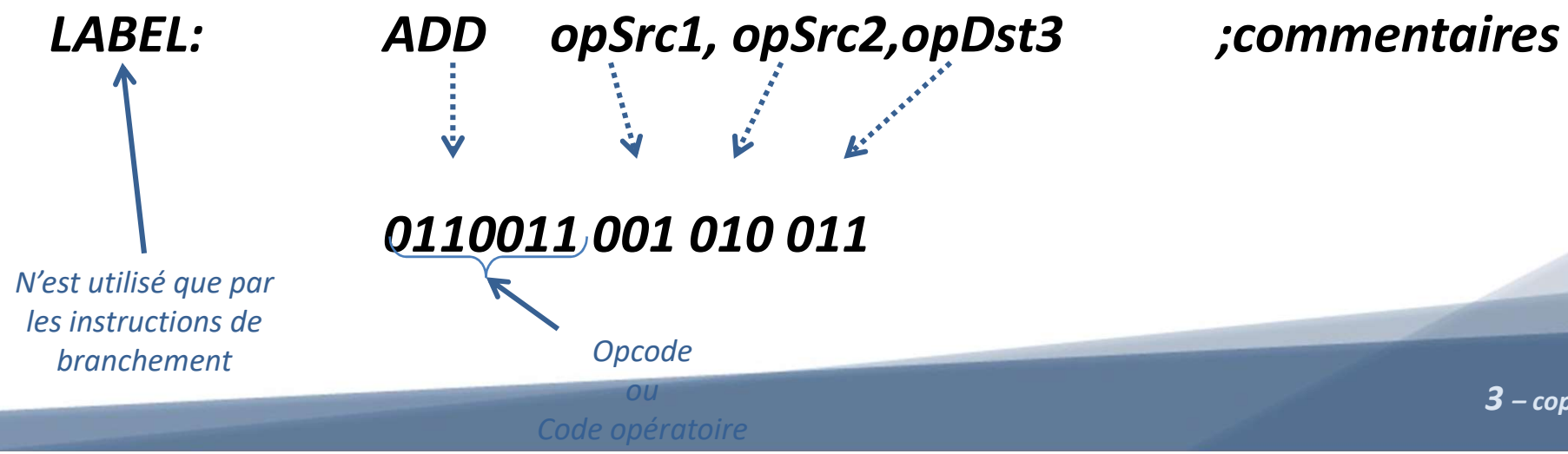
Architecture et Technologie des Ordinateurs

Hugo Descoubes - Juin 2013

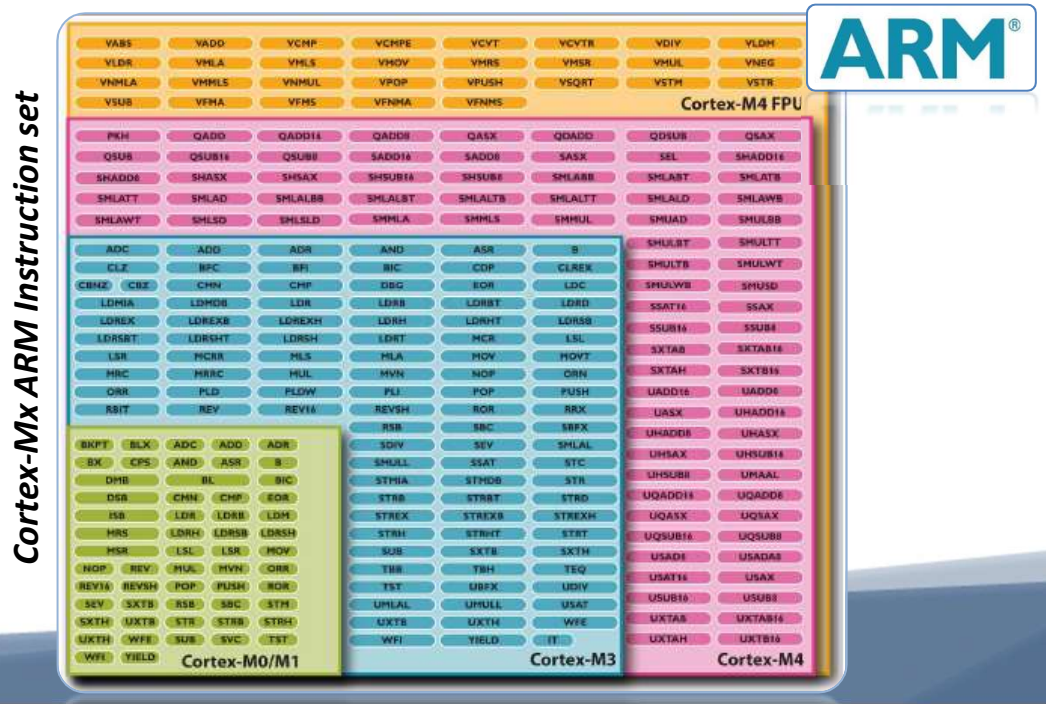
Un langage d'assemblage ou assembleur ou ASM est un langage de programmation bas niveau représentant, sous forme lisible pour un être humain, le code binaire exécutable par un processeur (ou code machine). *Prenons l'exemple d'une instruction assembleur élémentaire raccrochée à aucune architecture connue :*



Hormis label et commentaires, en général à tout champ d'une instruction assembleur correspond un champ dans le code binaire équivalent. Ce code binaire ne peut être compris et interprété que par le CPU cible. Rappelons également que comme tout langage de programmation, la syntaxe ne fait appel qu'à des références symboliques. La résolution des symboles étant gérée à l'édition des liens (exemple des labels, des adresses mémoire des variables …) :



L'assembleur est le langage de programmation le moins universel au monde. *Il existe autant de langage d'assemblage que de familles de CPU. Prenons l'exemple des jeux d'instructions Cortex-M de ARM. La société Anglaise ARM propose à elle seule 3 familles de CPU, cortex-M, -R, -A possédant chacune des sous familles. Ne regardons que la famille cortex-M :*



Observons les principaux acteurs dans le domaine des CPU's. Chaque fondeur présenté ci-dessous propose une voire plusieurs architectures de CPU qui lui sont propres et possédant donc les jeux d'instructions associés (CPU server et mainframe non présentés) :

- ***GPP CPU architectures :*** *Intel (IA-32 et Intel 64), AMD (x86 et AMD64), IBM (PowerPC), Renesas (RX CPU), Zilog (Z80), Motorola (6800 et 68000) …*

- ***Embedded CPU architectures (MCU, DSP, SoC) :*** *ARM (Cortex –M – R -A), MIPS (Rx000), Intel (Atom, 8051), Renesas, Texas Instrument (MSPxxx, C2xxx, C5xxx, C6xxx), Microchip (PICxx) , Atmel (AVR), Apple/IBM/Freescale (PowerPC) …*

***Tout CPU est capable de décoder puis d'exécuter un jeu d'instruction qui lui est propre (ou instruction set ou ISA ou Instruction Set Architecture**). Dans tous les cas, ces instructions peuvent être classées en grandes familles :*

- ***Calcul et comparaison :** opérations arithmétiques et logiques (en C : +, -, *, /, &, |, ! ...) et opérations de comparaison, (en C : >=, <=, !=, == …). Les formats entiers courts seront toujours supportés nativement. En fonction de l'architecture du CPU, les formats entiers long (16bits et plus) voire flottants peuvent l'être également.*

- ***Management de données :** déplacement de données dans l'architecture matérielle (CPU vers CPU, CPU vers mémoire ou mémoire vers CPU)*

- ***Contrôle programme :*** *saut en mémoire programme (saut dans le code). Par exemple en langage C : if, else if, else, switch, for, while, do while, appels de procédure. Nous pouvons rendre ces sauts conditionnels à l'aide d'opérations arithmétiques et logiques ou de comparaisons.*

Certaines architectures, comme les architectures compatibles x86-64 (Intel et AMD), possèdent des familles spécialisées :

- ***String manipulation :*** *manipulation au niveau assembleur de chaînes de caractères.*

- ***Divers :*** *arithmétique lourde (sinus, cosinus…), opérations vectorielles (produit vectoriel, produit scalaire…) …*

Les jeux d'instructions et CPU associés peuvent être classés en 2 grandes familles, RISC et CISC*, respectivement Reduce et Complex Instruction Set Computer. Les architectures RISC n'implémentent en général que des instructions élémentaires (CPU's ARM, MIPS, 8051, PIC18 …). A l'inverse, les architectures CISC (CPU's x86-64, 68xxx …) implémentent nativement au niveau assembleur des traitements pouvant être très complexes (division, opérations vectorielles, opérations sur des chaînes de caractères …).*

En 2012, la frontière entre ces deux familles est de plus en plus fine. Par exemple, le jeu d'instructions des processeurs spécialisés DSP RISC-like TMS320C66xx de TI compte 323 instructions. Néanmoins, les architectures compatibles x86-64 sont des architectures CISC. Nous allons rapidement comprendre pourquoi.

Avantages architecture CISC :

- *Empreinte mémoire programme faible, donc plus d'instructions contenues en cache. Néanmoins sur CPU CISC, en moyenne près de 80% des instructions compilées sont de types RISC.*
- *Compatibles x86-64, rétrocompatibilité des applications développées sur anciennes architectures.*

Inconvénients architecture CISC :

- *Architecture CPU complexe (mécanismes d'accélération matériels, décodeurs, Execution Units …), donc moins de place pour le cache.*
- *Jeu d'instructions mal géré par les chaînes de compilation (mécanismes d'optimisation)*

Inconvénients architecture RISC :

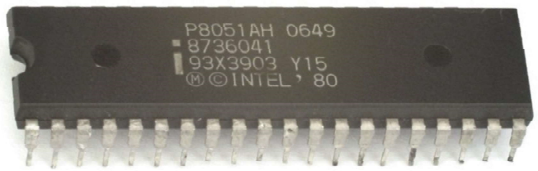
- *Empreinte mémoire programme élevée, donc moins d'instructions contenues en cache et mémoire principale.*

Avantages architecture RISC :

- *Architecture du CPU moins complexe (mécanismes d'accélération matériels, décodeurs, unités d'exécution …).*
- *En général, tailles instructions fixes et souvent exécution en un ou deux cycles CPU.*
- *Jeu d'instructions plus simple à appréhender pour le développeur et donc le compilateur. Jeu d'instructions très bien géré par les chaînes de compilations (mécanismes d'optimisation). Beaucoup d'architectures RISC récentes, travaillent avec de nombreux registres de travail généralistes, facilite le travail du compilateur.*

Observons le jeu d'instructions complet d'un CPU RISC 8051 proposé par Intel en 1980. En 2012, cette famille de CPU, même si elle reste très ancienne, est toujours extrêmement répandue et intégrée dans de nombreux MCU's ou ASIC's (licence libre). Prenons quelques exemples de fondeurs les utilisant : NXP, silabs, Atmel …

8051 Intel CPU (only CPU)
(1980)



MCU Silabs with 8051 CPU
(2012)

Assembleur – Architectures CPU – ISA Extensions

- Jeu d'instruction RISC 8051
- Jeu d'instruction CISC 8086

8051 Instruction set

ACALL	Absolute Call	**MOV**	Move Memory
ADD, ADDC	Add Accumulator (With Carry)	**MOVC**	Move Code Memory
AJMP	Absolute Jump	**MOVX**	Move Extended Memory
ANL	Bitwise AND	**MUL**	Multiply Accumulator by B
CJNE	Compare and Jump if Not Equal	**NOP**	No Operation
CLR	Clear Register	**ORL**	Bitwise OR
CPL	Complement Register	**POP**	Pop Value From Stack
DA	Decimal Adjust	**PUSH**	Push Value Onto Stack
DEC	Decrement Register	**RET**	Return From Subroutine
DIV	Divide Accumulator by B	**RETI**	Return From Interrupt
DJNZ	Decrement Register and Jump if Not Zero	**RL**	Rotate Accumulator Left
INC	Increment Register	**RLC**	Rotate Accumulator Left Through Carry
JB	Jump if Bit Set	**RR**	Rotate Accumulator Right
JBC	Jump if Bit Set and Clear Bit	**RRC**	Rotate Accumulator Right Through Carry
JC	Jump if Carry Set	**SETB**	Set Bit
JMP	Jump to Address	**SJMP**	Short Jump
JNB	Jump if Bit Not Set	**SUBB**	Subtract From Accumulator With Borrow
JNC	Jump if Carry Not Set	**SWAP**	Swap Accumulator Nibbles
JNZ	Jump if Accumulator Not Zero	**XCH**	Exchange Bytes
JZ	Jump if Accumulator Zero	**XCHD**	Exchange Digits
LCALL	Long Call	**XRL**	Bitwise Exclusive OR
LJMP	Long Jump		

Observons le jeu d'instructions complet d'un CPU 16bits CISC 8086 proposé par Intel en 1978. Il s'agit du premier processeur de la famille x86. En 2012, un corei7 est toujours capable d'exécuter le jeu d'instruction d'un 8086. Bien sûr, la réciproque n'est pas vraie.

8086 Intel CPU
(1978)

Assembleur – Architectures CPU – ISA Extensions

- Jeu d'instruction RISC 8051
- Jeu d'instruction CISC 8086

Original 8086 Instruction set

AAA	ASCII adjust AL after addition
AAD	ASCII adjust AX before division
AAM	ASCII adjust AX after multiplication
AAS	ASCII adjust AL after subtraction
ADC	Add with carry
ADD	Add
AND	Logical AND
CALL	Call procedure
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
CMP	Compare operands
CMPSB	Compare bytes in memory
CMPSW	Compare words
CWD	Convert word to doubleword
DAA	Decimal adjust AL after addition
DAS	Decimal adjust AL after subtraction
DEC	Decrement by 1
DIV	Unsigned divide
ESC	Used with floating-point unit

HLT	Enter halt state
IDIV	Signed divide
IMUL	Signed multiply
IN	Input from port
INC	Increment by 1
INT	Call to interrupt
INTO	Call to interrupt if overflow
IRET	Return from interrupt
Jcc	Jump if condition
JMP	Jump
LAHF	Load flags into AH register
LDS	Load pointer using DS
LEA	Load Effective Address
LES	Load ES with pointer
LOCK	Assert BUS LOCK# signal
LODSB	Load string byte
LODSW	Load string word
LOOP/LOOPx	Loop control
MOV	Move
MOVSB	Move byte from string to string
MOVSW	Move word from string to string
MUL	Unsigned multiply

Assembleur – Architectures CPU – ISA Extensions

- Jeu d'instruction RISC 8051
- Jeu d'instruction CISC 8086

Original 8086 Instruction set

NEG	Two's complement negation
NOP	No operation
NOT	Negate the operand, logical NOT
OR	Logical OR
OUT	Output to port
POP	Pop data from stack
POPF	Pop data from flags register
PUSH	Push data onto stack
PUSHF	Push flags onto stack
RCL	Rotate left (with carry)
RCR	Rotate right (with carry)
REPxx	Repeat MOVS/STOS/CMPS/LODS/SCAS
RET	Return from procedure
RETN	Return from near procedure
RETF	Return from far procedure
ROL	Rotate left
ROR	Rotate right
SAHF	Store AH into flags
SAL	Shift Arithmetically left (signed shift left)
SAR	Shift Arithmetically right (signed shift right)
SBB	Subtraction with borrow

SCASB	Compare byte string
SCASW	Compare word string
SHL	Shift left (unsigned shift left)
SHR	Shift right (unsigned shift right)
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOSB	Store byte in string
STOSW	Store word in string
SUB	Subtraction
TEST	Logical compare (AND)
WAIT	Wait until not busy
XCHG	Exchange data
XLAT	Table look-up translation
XOR	Exclusive OR

Prenons un exemple d'instruction CISC 8086. Les deux codes qui suivent réalisent le même traitement et permettent de déplacer 100 octets en mémoire d'une adresse source vers une adresse destination :

CISC

```
MOV     CX,100
MOV     DI, dst
MOV     SI, src
REP     MOVSB
```

RISC

```
        MOV     CX,100
        MOV     DI, dst
        MOV     SI, src
LOOP:
        MOV     AL, [SI]
        MOV     [DI], AL
        INC     SI
        INC     DI
        DEC     CX
        JNX     LOOP
```

Attention, si vous lisez de l'assembleur x86-64, il existe deux syntaxes très répandues. La syntaxe Intel et la syntaxe AT&T utilisée par défaut par gcc (systèmes UNIX).

Intel Syntax		**AT&T Syntax**	
MOV	ebx,0FAh	MOV	$0xFA, %ebx

Syntaxe AT&T :

- *Opérandes sources à gauche et destination à droite*
- *Constantes préfixées par $ (adressage immédiat)*
- *Constantes écrites avec syntaxe langage C (0x + valeur = hexadécimal)*
- *Registres préfixés par %*
- *Segmentation : [ds:20] devient %ds:20, [ss:bp] devient %ss:%bp …*

- *Adressage indirect [ebx] devient (%ebx), [ebx + 20h] devient 0x20(%ebx), [ebx+ecx*2h-1Fh] devient -0x1F(%ebx, %ecx, 0x2) …*
- *Suffixes, b=byte=1o, w=word=2o, s=short=4o, l=long=4o, q=quad=8o, t=ten=10o, o=octo=16o=128bits (x64)*
- *…*

Prenons un exemple de code écrit dans les 2 syntaxes :

Intel Syntax

```
        MOV     CX,100
        MOV     DI, dst
        MOV     SI, src
LOOP:
        MOV     AL, [SI]
        MOV     [DI], AL
        INC     SI
        INC     DI
        DEC     CX
        JNX     LOOP
```

AT&T Syntax

```
        movw    $100, %cx
        movw    dst, %di
        movw    src, %di
LOOP:
        movb    (%si), %al
        movb    %al, (%di)
        inc     %si
        inc     %di
        dec     %cx
        jnx     LOOP
```

Par abus de langage, les CPU compatibles du jeu d'instruction 80x86 (8086, 80386, 80486..) sont nommés CPU x86. Depuis l'arrivée d'architectures 64bits ils sont par abus de langage nommés x64. Pour être rigoureux chez Intel, il faut nommer les jeux d'instructions et CPU 32bits associés IA-32 (depuis le 80386 en 1985) et les ISA 64bits Intel 64 ou EM64T (depuis le Pentium 4 Prescott en 2004).

L'une des grandes forces (et paradoxalement faiblesse) de ce jeu d'instruction est d'assurer une rétrocompatibilité avec les jeux d'instructions d'architectures antérieures. En contrepartie, il s'agit d'une architecture matérielle très complexe, difficile à accélérer imposant de fortes contraintes de consommation et d'échauffement.

Extensions x86 et x64 n'opérant que sur des formats entiers :

CPU Architecture	Nom extension	Instructions
8086 Original x86	-	**AAA, AAD, AAM, AAS, ADC, ADD, AND, CALL, CBW, CLC, CLD, CLI, CMC, CMP, CMPSzz, CWD, DAA, DAS, DEC, DIV, ESC, HLT, IDIV, IMUL, IN, INC, INT, INTO, IRET, Jcc, LAHF, LDS, LEA, LES, LOCK, LODSzz, LODSW, LOOPcc, MOV, MOVSzz, MUL, NEG, NOP, NOT, OR, OUT, POP, POPF, PUSH, PUSHF, RCL, RCR, REPcc, RET, RETF, ROL, ROR, SAHF, SAL, SALC, SAR, SBB, SCASzz, SHL, SAL, SHR, STC, STD, STI, STOSzz, SUB, TEST, WAIT, XCHG,XLAT, XOR**
80186/80188	-	BOUND, ENTER, INSB, INSW, LEAVE, OUTSB, OUTSW, POPA, PUSHA, PUSHW
80286	-	ARPL, CLTS, LAR, LGDT, LIDT, LLDT, LMSW, LOADALL, LSL, LTR, SGDT, SIDT, SLDT, SMSW, STR, VERR, VERW
80386	-	BSF, BSR, BT, BTC, BTR, BTS, CDQ, CMPSD, CWDE, INSD, IRETD, IRETDF, IRETF, JECXZ, LFS, LGS, LSS, LODSD, LOOPD, LOOPED, LOOPNED, LOOPNZD, LOOPZD, MOVSD, MOVSX, MOVZX, OUTSD, POPAD, POPFD, PUSHAD, PUSHD, PUSHFD, SCASD, SETA, SETAE, SETB, SETBE, SETC, SETE, SETG, SETGE, SETL, SETLE, SETNA, SETNAE, SETNB, SETNBE, SETNC, SETNE, SETNG, SETNGE, SETNL, SETNLE, SETNO, SETNP, SETNS, SETNZ, SETO, SETP, SETPE, SETPO, SETS, SETZ, SHLD, SHRD, STOSD
80486	-	BSWAP, CMPXCHG, INVD, INVLPG, WBINVD, XADD
Pentium	-	**CPUID**, CMPXCHG8B, RDMSR, RDPMC, WRMSR, RSM
Pentium pro	-	CMOVA, CMOVAE, CMOVB, CMOVB, CMOVE, CMOVG, CMOVGE, CMOVL, CMOVLE, CMOVNA, CMOVNAE, CMOVNB, CMOVNBE, CMOVNC, CMOVNE, CMOVNG, CMOVNGE, CMOVNL, CMOVNLE, CMOVNO, CMOVNP, CMOVNS, CMOVNZ, CMOVO, CMOVP, CMOVPE, CMOVPO, CMOVS, CMOVZ, RDPMC, SYSENTER, SYSEXIT, UD2
Pentium III	**SSE**	MASKMOVQ, MOVNTPS, MOVNTQ, PREFETCH0, PREFETCH1, PREFETCH2, PREFETCHNTA, SFENCE
Pentium 4	**SSE2**	CLFLUSH, LFENCE, MASKMOVDQU, MFENCE, MOVNTDQ, MOVNTI, MOVNTPD, PAUSE
Pentium 4	**SSE3 Hyper Threading**	LDDQU, MONITOR, MWAIT
Pentium 4 6x2	**VMX**	VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON
X86-64	-	CDQE, CQO, CMPSQ, CMPXCHG16B, IRETQ, JRCXZ, LODSQ, MOVSXD, POPFQ, PUSHFQ, **RDTSC**, SCASQ, STOSQ, SWAPGS
Pentium 4	**VT-x**	VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON

Les extensions x87 ci-dessous n'opèrent que sur des formats flottants. Historiquement, le 8087 était un coprocesseur externe utilisé comme accélérateur matériel pour des opérations flottantes. Ce coprocesseur fut intégré dans le CPU principal sous forme d'unité d'exécution depuis l'architecture 80486. Cette unité est souvent nommée FPU (Floating Point Unit).

CPU Architecture	Nom extension	Instructions
8087 Original x87	-	F2XM1, FABS, FADD, FADDP, FBLD, FBSTP, FCHS, FCLEX, FCOM, FCOMP, FCOMPP, FDECSTP, FDISI, FDIV, FDIVP, FDIVR, FDIVRP, FENI, FFREE, FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FILD, FIMUL, FINCSTP, FINIT, FIST, FISTP, FISUB, FISUBR, FLD, FLD1, FLDCW, FLDENV, FLDENVW, FLDL2E, FLDL2T, FLDLG2, FLDLN2, FLDPI, FLDZ, FMUL, FMULP, FNCLEX, FNDISI, FNENI, FNINIT, FNOP, FNSAVE, FNSAVEW, FNSTCW, FNSTENV, FNSTENVW, FNSTSW, FPATAN, FPREM, FPTAN, FRNDINT, FRSTOR, FRSTORW, FSAVE, FSAVEW, FSCALE, FSQRT, FST, FSTCW, FSTENV, FSTENVW, FSTP, FSTSW, FSUB, FSUBP, FSUBR, FSUBRP, FTST, FWAIT, FXAM, FXCH, FXTRACT, FYL2X, FYL2XP1
80287	-	FSETPM
80387	-	FCOS, FLDENVD, FNSAVED, FNSTENVD, FPREM1, FRSTORD, FSAVED, FSIN, FSINCOS, FSTENVD, FUCOM, FUCOMP, FUCOMPP
Pentium pro	-	FCMOVB, FCMOVBE, FCMOVE, FCMOVNB, FCMOVNBE, FCMOVNE, FCMOVNU, FCMOVU, FCOMI, FCOMIP, FUCOMI, FUCOMIP, FXRSTOR, FXSAVE
Pentium 4	**SSE3**	FISTTP

*Les extensions présentées ci-dessous inplémentent toutes des instructions dites **SIMD (Single Instruction Multiple Data)** :*

- ***MMX :** MultiMedia eXtensions*
- ***SSE :** Streaming SIMD Extensions*
- ***AVX :** Advanced Vector Extensions*
- ***AES :** Advanced Encryption Standard*

CPU Architecture	Nom extension	Instructions	
Pentium MMX	MMX	EMMS, MOVD, MOVQ, PACKSSDW, PACKSSWB, PACKUSWB, PADDB, PADDD, PADDSB, PADDSW, PADDUSB, PADDUSW, PADDW, PAND, PANDN, PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTD, PCMPGTW, PMADDWD, PMULHW, PMULLW, POR, PSLLD, PSLLQ, PSLLW, PSRAD, PSRAW, PSRLD, PSRLQ, PSRLW, PSUBB, PSUBD, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PSUBW, PUNPCKHBW, PUNPCKHDQ, PUNPCKHWD, PUNPCKLBW, PUNPCKLDQ, PUNPCKLWD, PXOR	
Pentium III	SSE	Float Inst.	ADDPS, ADDSS, CMPPS, CMPSS, COMISS, CVTPI2PS, CVTPS2PI, CVTSI2SS, CVTSS2SI, CVTTPS2PI, CVTTSS2SI, DIVPS, DIVSS, LDMXCSR, MAXPS, MAXSS, MINPS, MINSS, MOVAPS, MOVHLPS, MOVHPS, MOVLHPS, MOVLPS, MOVMSKPS, MOVNTPS, MOVSS, MOVUPS, MULPS, MULSS, RCPPS, RCPSS, RSQRTPS, RSQRTSS, SHUFPS, SQRTPS, SQRTSS, STMXCSR, SUBPS, SUBSS, UCOMISS, UNPCKHPS, UNPCKLPS
		Integer Inst.	ANDNPS, ANDPS, ORPS, PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULHUW, PSADBW, PSHUFW, XORPS
Pentium 4	SSE2	Float Inst.	ADDPD, ADDSD, ANDNPD, ANDPD, CMPPD, CMPSD, COMISD, CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPD2PI, CVTPD2PS, CVTPI2PD, CVTPS2DQ, CVTPS2PD, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSS2SD, CVTTPD2DQ, CVTTPD2PI, CVTTPS2DQ, CVTTSD2SI, DIVPD, DIVSD, MAXPD, MAXSD, MINPD, MINSD, MOVAPD, MOVHPD, MOVLPD, MOVMSKPD, MOVSD, MOVUPD, MULPD, MULSD, ORPD, SHUFPD, SQRTPD, SQRTSD, SUBPD, SUBSD, UCOMISD, UNPCKHPD, UNPCKLPD, XORPD
		Integer Inst.	MOVDQ2Q, MOVDQA, MOVDQU, MOVQ2DQ, PADDQ, PSUBQ, PMULUDQ, PSHUFHW, PSHUFLW, PSHUFD, PSLLDQ, PSRLDQ, PUNPCKHQDQ, PUNPCKLQDQ
	SSE3	Float Inst.	ADDSUBPD, ADDSUBPS, HADDPD, HADDPS, HSUBPD, HSUBPS, MOVDDUP, MOVSHDUP, MOVSLDUP

Les instructions et opérandes usuellement manipulées par grand nombre de CPU sur le marché sont dites scalaires. Nous parlerons de ***processeur scalaire*** *(PIC de Microchip, 8051 de Intel, AVR de Atmel, C5xxx de TI…). Par exemple sur 8086 de Intel , prenons l'exemple d'une addition : scalaire + scalaire = scalaire :*

```
add         %bl,%al
```

A titre indicatif, les instructions MMX, SSE, AVX, AES … sont dîtes vectorielles. Les opérandes ne sont plus des grandeurs scalaires mais des grandeurs vectorielles. Nous parlerons de ***processeur vectoriel*** *(d'autres architectures vectorielles existent). Prenons un exemple d'instruction vectorielle SIMD SSE4.1, vecteur . vecteur = scalaire :*

```
dpps        0xF1, %xmm2,%xmm1
```

*Cette instruction vectorielle peut notamment être très intéressante pour des applications de traitement numérique du signal : **dpps signifie dot product packet single**, soit produit scalaire sur un paquet de données au format flottant en simple précision (IEEE-754). Observons le descriptif de l'instruction ainsi qu'un exemple :*

DPPS — Dot Product of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 40 /r ib DPPS *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Selectively multiply packed SP floating-point values from *xmm1* with packed SP floating-point values from *xmm2*, add and selectively store the packed SP floating-point values or zero values to *xmm1*.

http://www.intel.com

Etudions un exemple d'exécution de l'instruction dpps :

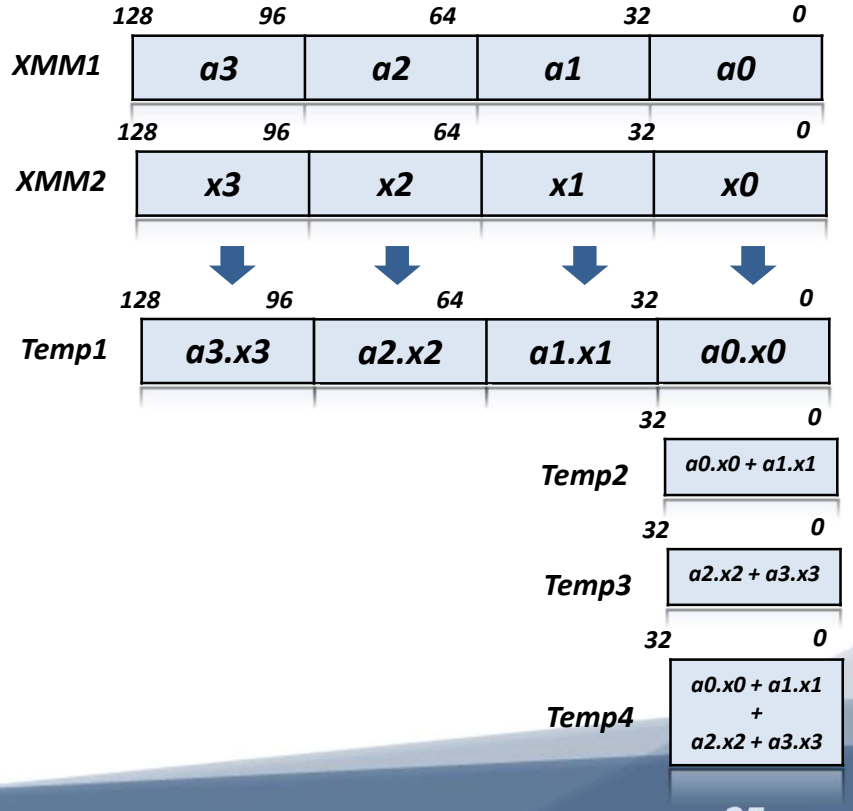
dpps 0xF1, %xmm2,%xmm1

Operation

```
DP_primitive (SRC1, SRC2)
IF (imm8[4] = 1)
    THEN Temp1[31:0] ← DEST[31:0] * SRC[31:0];
    ELSE Temp1[31:0] ← +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[63:32] ← DEST[63:32] * SRC[63:32];
    ELSE Temp1[63:32] ← +0.0; FI;
IF (imm8[6] = 1)
    THEN Temp1[95:64] ← DEST[95:64] * SRC[95:64];
    ELSE Temp1[95:64] ← +0.0; FI;
IF (imm8[7] = 1)
    THEN Temp1[127:96] ← DEST[127:96] * SRC[127:96];
    ELSE Temp1[127:96] ← +0.0; FI;

Temp2[31:0] ← Temp1[31:0] + Temp1[63:32];
Temp3[31:0] ← Temp1[95:64] + Temp1[127:96];
Temp4[31:0] ← Temp2[31:0] + Temp3[31:0];
```

XMMi (i = 0 à 15 with Intel 64)
128bits General Purpose Registers
for SIMD Execution Units

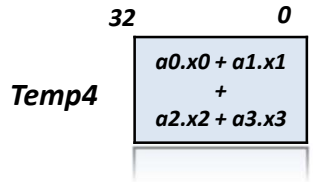


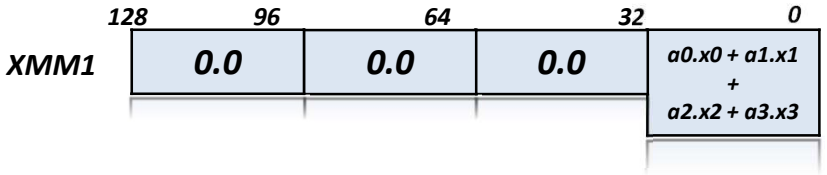
http://www.intel.com

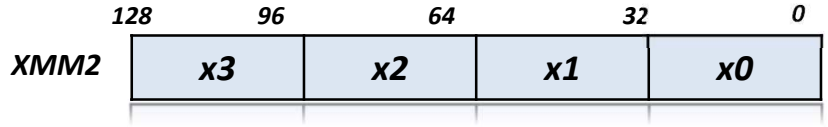
Etudions un exemple d'exécution de l'instruction dpps :

dpps 0xF1, %xmm2,%xmm1

XMMi (i = 0 à 15 with Intel 64)
128bits General Purpose Registers
for SIMD Execution Units







- IF (imm8[0] = 1)
  THEN DEST[31:0] ← Temp4[31:0];
  ELSE DEST[31:0] ← +0.0; FI;
- IF (imm8[1] = 1)
  THEN DEST[63:32] ← Temp4[31:0];
  ELSE DEST[63:32] ← +0.0; FI;
- IF (imm8[2] = 1)
  THEN DEST[95:64] ← Temp4[31:0];
  ELSE DEST[95:64] ← +0.0; FI;
- IF (imm8[3] = 1)
  THEN DEST[127:96] ← Temp4[31:0];
  ELSE DEST[127:96] ← +0.0; FI;

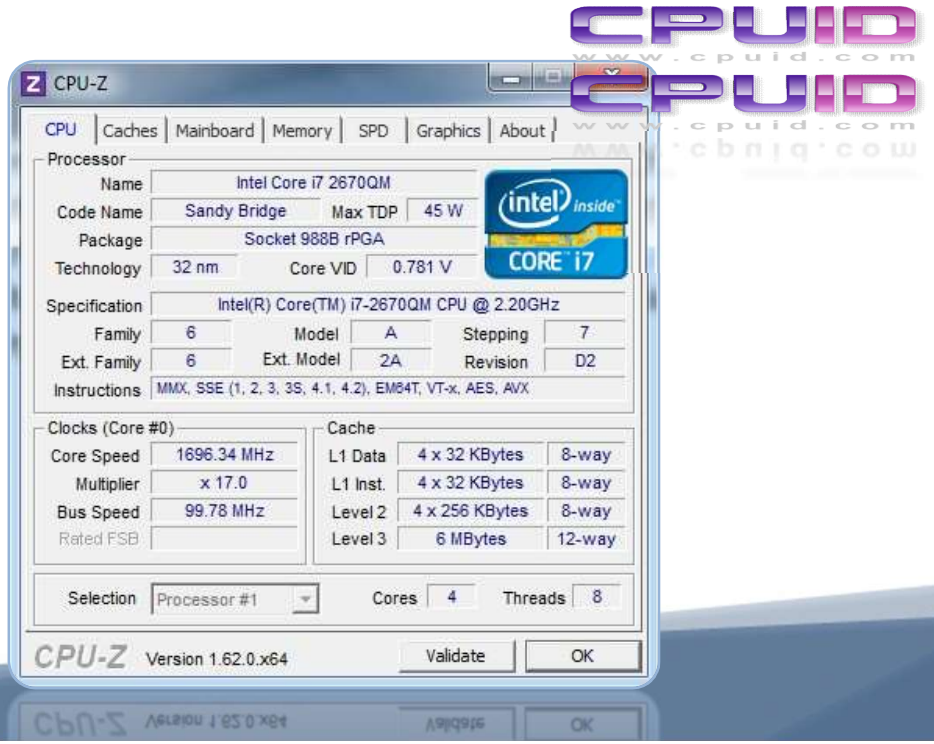
DPPS (128-bit Legacy SSE version)
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] (Unmodified)

http://www.intel.com

Les extensions x86-64 présentées jusqu'à maintenant ne présentent que les évolutions des jeux d'instructions apportées par Intel. Les extensions amenées par AMD ne seront pas présentées (MMX+, K6-2, 3DNow, 3DNow!+, SSE4a..).

CPU Architecture	Nom extension	Instructions
Core2	**SSSE3**	PSIGNW, PSIGND, PSIGNB, PSHUFB, PMULHRSW, PMADDUBSW, PHSUBW, PHSUBSW, PHSUBD, PHADDW, PHADDSW, PHADDD, PALIGNR, PABSW, PABSD, PABSB
Core2 (45nm)	**SSE4.1**	MPSADBW, PHMINPOSUW, PMULLD, PMULDQ, **DPPS**, DPPD, BLENDPS, BLENDPD, BLENDVPS, BLENDVPD, PBLENDVB, PBLENDW, PMINSB, PMAXSB, PMINUW, PMAXUW, PMINUD, PMAXUD, PMINSD, PMAXSD, ROUNDPS, ROUNDSS, ROUNDPD, ROUNDSD, INSERTPS, PINSRB, PINSRD/PINSRQ, EXTRACTPS, PEXTRB, PEXTRW, PEXTRD/PEXTRQ, PMOVSXBW, PMOVZXBW, PMOVSXBD, PMOVZXBD, PMOVSXBQ, PMOVZXBQ, PMOVSXWD, PMOVZXWD, PMOVSXWQ, PMOVZXWQ, PMOVSXDQ, PMOVZXDQ, PTEST, PCMPEQQ, PACKUSDW, MOVNTDQA
Nehalem	**SSE4.2**	CRC32, PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM, PCMPGTQ
Sandy Bridge	**AVX**	VFMADDPD, VFMADDPS, VFMADDSD, VFMADDSS, VFMADDSUBPD, VFMADDSUBPS, VFMSUBADDPD, VFMSUBADDPS, VFMSUBPD, VFMSUBPS, VFMSUBSD, VFMSUBSS, VFNMADDPD, VFNMADDPS, VFNMADDSD, VFNMADDSS, VFNMSUBPD, VFNMSUBPS, VFNMSUBSD, VFNMSUBSS
Nehalem	**AES**	AESENC, AESENCLAST, AESDEC, AESDECLAST, AESKEYGENASSIST, AESIMC

L'instruction CPUID arrivée avec l'architecture Pentium permet de récupérer très facilement toutes les informations relatives à l'architecture matérielle du GPP (CPU's, Caches, adressage virtuel..). L'utilitaire libre CPU-Z utilise notamment ce registre pour retourner des informations sur l'architecture :



Sous Linux, vous pouvez également consulter le fichier ***/proc/cpuinfo*** *listant les informations retournées par l'instruction CPUID :*

```
vmlinux@vmlinux: ~
vmlinux@vmlinux:~$ cat /proc/cpuinfo | more
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 42
model name      : Intel(R) Core(TM) i7-2670QM CPU @ 2.20GHz
stepping        : 7
cpu MHz         : 2107.531
cache size      : 6144 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic se
pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm
e3 lahf_lm
bogomips        : 4215.06
clflush size    : 64
cache_alignment : 64
```

De même, lorsque l'on est amené à développer sur un processeur donné, il est essentiel de travailler avec les documents de référence proposés par le fondeur, Intel dans notre cas. Vous pouvez télécharger les différents documents de référence à cette URL :
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

ENSICAEN
Computer Science

Merci de votre attention !