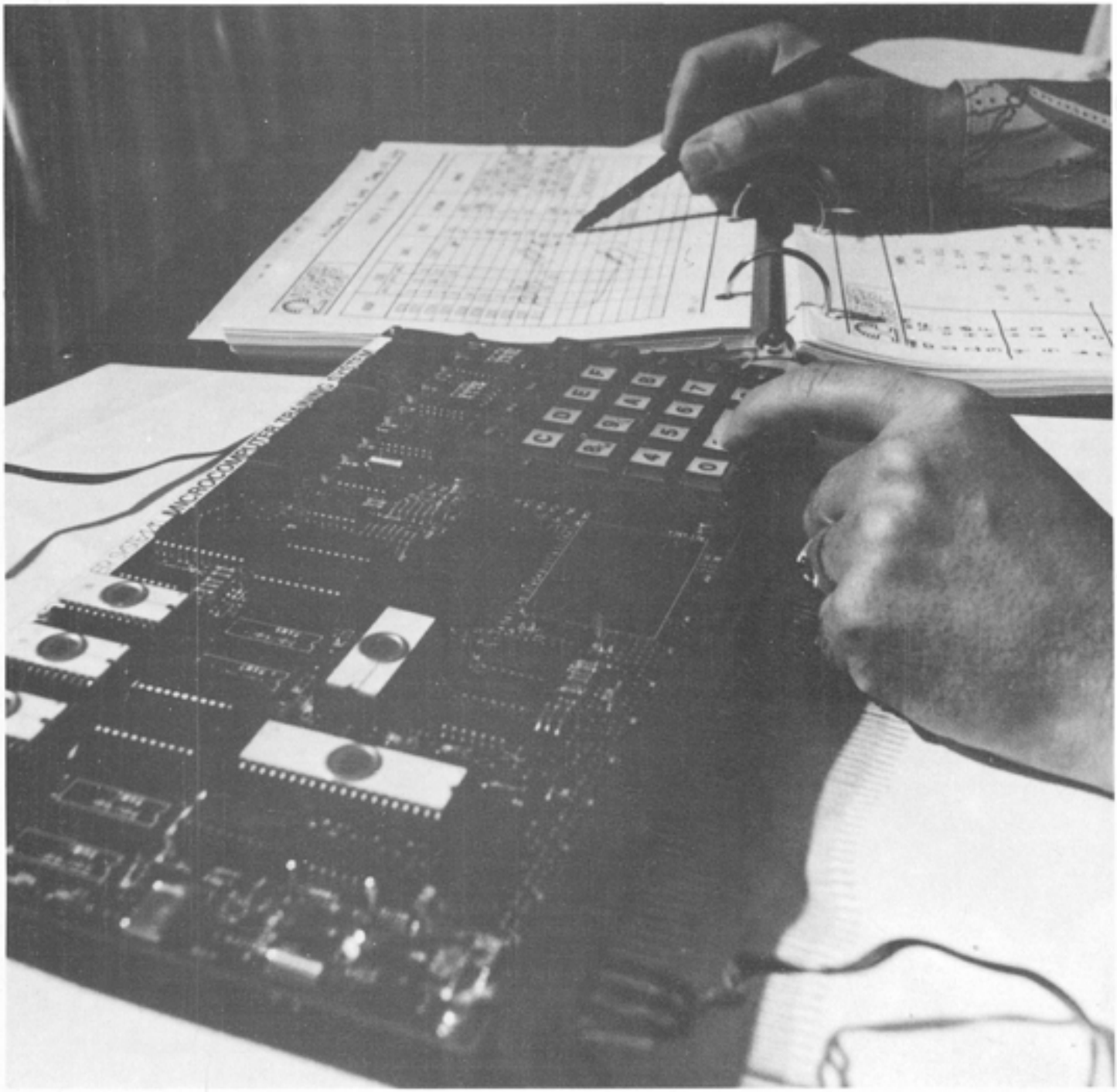
Trouver le bon langage...

· *Henri Lilen*

Pour programmer des milliers de transistors

QUEL bonheur que de pouvoir dialoguer avec l'ordinateur dans un langage proche de la langue courante ! Quand bien même ce serait de l'anglais... C'est tellement plus simple de lui dire : « Exécutez les calculs prescrits et, si le résultat obtenu est tant, passez à la séquence de mise à feu de la rétrofusée. » Car c'est presqu'ainsi qu'on procède avec les langages dits « évolués » ou « haut niveau », dans un style plus concis il est vrai. De tels ordres sont frappés sur un clavier et leur séquence composera un « programme » complet ; celà, en attendant qu'on enseigne à la machine à obéir à la voix...

Pourquoi, alors, se sentirait-on encore concerné par l'antédiluvien « langage



La programmation en « langage machine » Lorsque l'on veut programmer directement en numérique, on passe par le code hexadécimal via un clavier qui comporte 16 touches, de O à F. Avec quelques touches de fonction, comme le fait cet étudiant. En revanche, la programmation en assembleur permet le recours à un langage mnémonique qui est frappé sur un clavier « machine à écrire ». L'assembleur permet ainsi un contrôle plus aisé et une meilleure protection contre les erreurs de programmation.

machine », ou par sa transposition mnémonique, l'« assembleur » ? En première analyse, on peut remarquer que :
— les fameux langages évolués (Fortran, Cobol, Basic, Pascal...) ne sont nullement exempts de défauts. C'est d'ailleurs la raison qui explique leur prolifération. Si un seul d'entre eux était parfait, n'aurait-il pas toutes chances de s'imposer, contre tous ? Bien sûr, ces langages ont représenté un énorme progrès en programmation, mais ils se révèlent parfois tellement « inefficaces » au sens informatique du terme ! Aussi préférera-t-on alors revenir à l'« ancien » mode de programmation, plus proche de la machine, l'assembleur tant décrié par certains.

Une pyramide fondée sur le silicium

Toute la pyramide de l'informatique (quelque 10 % de l'économie mondiale, déjà, en chiffre d'affaires), repose sur du silicium. C'est avec ce silicium que sont fabriqués ces fameux « circuits intégrés » et ces « puces savantes », comme le dit si bien la publicité. Or, dans l'état actuel de la technologie, les transistors élémentaires des circuits électroniques qu'on regroupe aujourd'hui par dizaines de milliers en un circuit intégré travaillent sur le mode binaire : ou ils laissent passer le courant, ou ils l'en empêchent. Ce que l'on traduit, en algèbre de Boole, par les deux symboles « zéro » et « un » (0 et 1). Un additionneur, un circuit devant prendre une décision, un point mémoire, ne connaîtront donc que l'un de ces deux états. Programmer un circuit revient alors à lui imposer l'état « 0 » ou l'état « 1 », conducteur ou bloqué. La difficulté surgit lorsqu'on songe que ce n'est pas un seul transistor qu'il faut programmer, mais des milliers...

Avec un seul 0 ou un seul 1, on ne peut mettre en condition qu'un seul transistor, ou une seule « porte », ou une seule fonction composée de plusieurs transistors. Avec huit « éléments binaires d'information » — c'est ainsi qu'on définit le 0 et le 1, encore appelés « bits », contraction de l'anglais « binary digits », digits binaires — on peut piloter huit portes. Un mot de 8 bits, tel que 01010101 s'appelle un octet. Une première astuce pour multiplier la capacité de commande d'un octet résulte de la constatation suivante ; on peut réaliser 256 combinaisons différentes de 0 et de 1 avec un « mot » de huit bits ; par conséquent cet unique mot pourra être « décodé » et il commandera 256 circuits. Si, de plus, on s'ingénie pour que chaque mot déclenche une séquence organisée de micro-opérations, on pourra rendre chaque octet très efficace. Et c'est ainsi qu'on en arrive au langage machine. Voulez-vous juger de la façon d'écrire un programme en langage machine ? En voici un exemple tout ce qu'il y a de réel ; question : qu'est censé exécuter ce programme ?

Programme en binaire							
0000	0000	0000	0000	1000	0110	0001	1000
0000	0000	0000	0010	1000	1011	0000	1100
0000	0000	0000	0100	1001	0111	0001	0000
0000	0000	0000	0110	0011	1111		

Un tel programme est rigoureusement incompréhensible, et reste très difficile à déchiffrer, même par l'initié. On touche là du doigt l'un des inconvénients majeurs du « langage machine », ce binaire qu'il est d'ailleurs très difficile de poser ou de transcrire sans risque d'erreur. Et si un 1 était devenu par inadvertance un 0 dans un programme composé de mille lignes de ce type : au lieu de freiner la fusée, on l'accélère ! Comment retrouver l'erreur ? Autant chercher une aiguille dans une meule de foin, dit-on souvent !

Le programme ci-dessus est moins ambitieux : sa fonction, très simple, consiste à commander l'addition de 24 et de 12, avec rangement du résultat. Ceux qui lisent le binaire vérifieront que le dernier octet des deux premières lignes est 0001 1000 (= 24) ou 0000 1100 (= 12) : c'est évident...

Alors, pourquoi tant de ronds et de bâtons ? C'est parce que chaque ligne représente une instruction en langage machine. La deuxième ligne, par exemple, est composée de la façon suivante :

Adresse de rangement de l'instruction en mémoire	Opération à exécuter	Opérande
0000 0000 0000 0010 (sur 16 bits)	1000 1011 (sur 8 bits)	0000 1100 (sur 8 bits)

Les seize premiers bits indiquent à quel endroit exact de la mémoire « interne » de l'ordinateur on stockera les seize bits suivants. Pour simplifier, on a choisi ici des adresses faibles : ce sera l'adresse numéro 2 pour la seconde ligne. L'octet suivant, c'est le code de l'opération à exécuter : une addition ; c'est cet octet qui sera « décodé » par l'unité centrale de traitement afin de procurer 256 possibilités ; il se traduit en décimal par le nombre 139, nullement dû au hasard mais imposé par le fabricant du circuit qui en a décidé ainsi, même si cette décision est à l'origine arbitraire. Enfin, le dernier octet est l'opérande, le nombre sur lequel porte l'opération, ici 12 en décimal.

Vous pourriez fort bien exécuter ce programme, rédigé en respectant le code d'un des microprocesseurs les plus répandus, le « 6800 ». Il constitue même l'un des tous premiers exercices d'initiation d'un cours connu de programmation... Mais comme on le voit, il faut 120 bits pour que la machine range en mémoire le résultat de 12 + 24, soit 36. Dans le langage évolué à la fois le plus simple et le plus utilisé, le Basic, on aurait simplement ordonné à la machine : 24 + 12, sans se préoccuper des adresses en mémoire ni du détail de l'opération !

Y aurait-il alors deux types d'ordinateurs, l'un conçu pour utiliser le binaire et l'autre pour les langages évolués ? Nullement ; la machine, en dernière analyse, ne pourra exécuter que des ordres traduits en 0 et 1. Aussi, lorsque le programme est rédigé en langage évolué faut-il qu'il soit converti en binaire afin de pouvoir être exécuté ensuite.

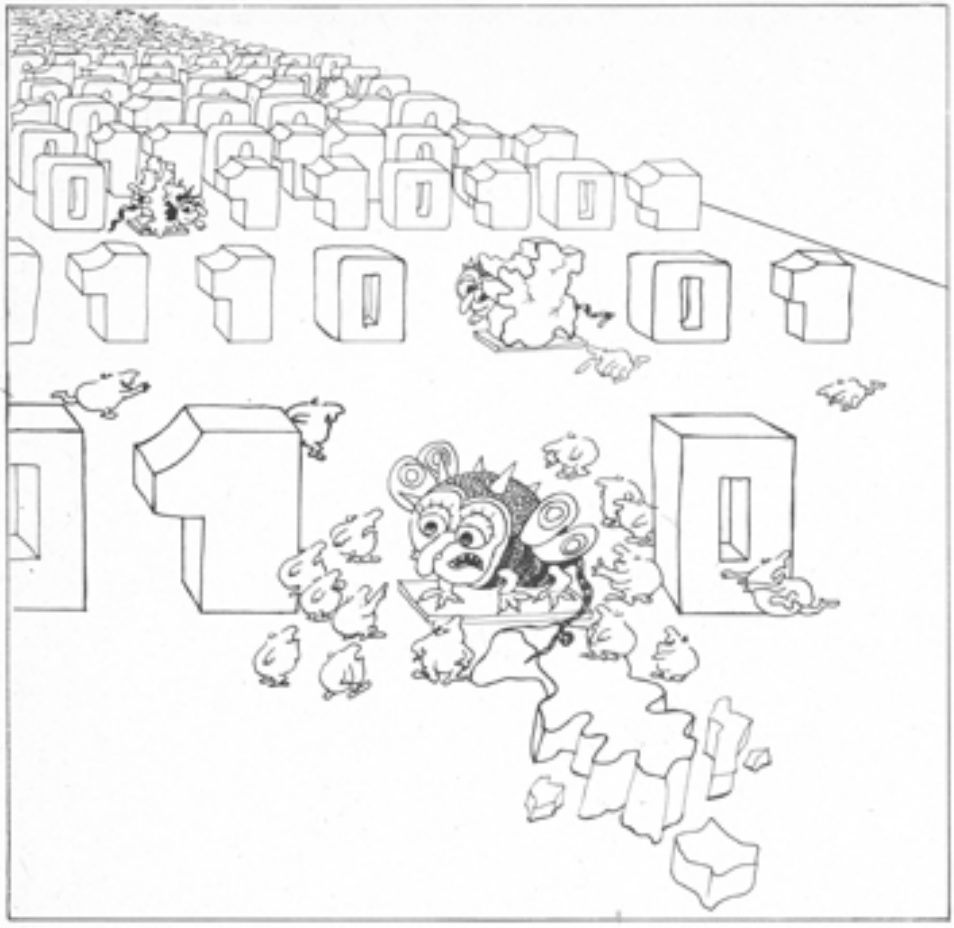
Pour l'utilisateur du Basic, ce processus de « traduction » est transparent, car un « interpréteur », sorte de dictionnaire assez complexe de traduction stocké dans les mémoires de l'ordinateur, se charge de ce processus. C'est lui qui convertira les ordres d'origine, rédigés dans un langage proche de la langue parlée, en binaire, et qui calculera les adresses en mémoire où ils seront stockés.

Avec des langages tels que le Fortran, le Cobol..., la traduction ne sera plus le fait d'un « interpréteur » mais d'un « compilateur » dont le mode d'intervention est quelque peu différent (la traduction se faisant alors en salve, préalablement à toute exécution).

L'interpréteur, tout comme le compilateur, se charge ainsi d'expliquer en détail à la machine les ordres globaux que vous lui donnez. Car l'ordinateur ne sait faire que ce qu'on lui dit de faire, mais ce doit être dit dans les moindres détails ; et toutes les situations doivent avoir été prévues. Or, l'interpréteur et le compilateur sont, eux aussi, des programmes loin d'être parfaits, par conséquent loin d'être en mesure de définir à l'ordinateur la meilleure démarche à suivre.

En revanche, et lorsque vous rédigez un programme en binaire, vous devez connaître très exactement l'architecture de votre machine, le nombre de registres qu'elle contient et leur affectation, l'action des instructions sur les « indicateurs », l'occupation réelle des zones de la mémoire... Vous trouverez parfois des algorithmes ou des raccourcis vous permettant de résoudre élégamment et aux moindres frais un problème complexe, décrit à la machine dans tous ses détails. Les programmes de traduction, eux, sont bien incapables d'être aussi inventifs et efficaces.

L'inconvénient est mineur, affirment bien des informaticiens qui transforment



La hantise des programmeurs : le Bug (erreur de programmation).

en science l'art de la programmation. Voire.

Supposons qu'un fabricant de produits de très large diffusion envisage une série de dix mille produits identiques, des machines à laver à microprocesseur par exemple. Le programme complet qui sera introduit dans les mémoires en circuits intégrés associés au micropresseur occupera, supposons-le, deux kilo-octets, ce qui représente un unique circuit mémoire de seize kilobits. Mais cela, si le programme a été rédigé en langage machine — ou plutôt, dans ce langage appelé « assembleur » qui en est la version mnémonique et qui se transpose directement en binaire, on va y revenir.

Si, en revanche, le même programme — c'est-à-dire un programme menant aux mêmes effets — a été rédigé en langage évolué, il devra être traduit en binaire par un compilateur qui, en faisant de son mieux, trouvera le moyen d'en doubler ou tripler la longueur. Par conséquent, il ne faudra plus un unique circuit intégré, mais trois pour le ranger en mémoire. A 10 F le circuit, la dépense supplémentaire sera de 20 F, somme à laquelle il faut ajouter le circuit imprimé, le câblage, l'alimentation, les charges supplémentaires de stockage, la manutention... Mettons 22 F, qu'on multipliera par le nombre de pièces identiques à produire : 10 000 ; on en arrive à une dépense accrue de 220 000 F... Economiser ce montant ne vaut-il pas l'effort de programmer en langage « bas niveau » (l'assembleur), par opposition aux langages « haut niveau » ?

Un second cas se révèle critique. Si le programme après compilation, donc une fois traduit en binaire, est de deux à trois fois plus long que s'il avait été rédigé en langage bas niveau, il demandera aussi deux à trois fois plus de temps pour être exécuté. Or, certaines applications ne permettent plus ce gaspillage de précieuses microsecondes (millionièmes de seconde). L'ordinateur ne pourra se permettre de demander à la fusée, qui s'apprête à se poser sur une planète, qu'elle veuille bien attendre qu'il ait fini ses calculs pour savoir s'il doit commander la mise à feu des rétrofusées ; la gestion des lignes à haute tension, plus près de nous, ne laisse guère de temps à gaspiller non plus. Et le simple microprocesseur introduit dans une automobile aurait fort à faire s'il voulait exécuter tous les calculs prescrits à chaque étincelle : là encore, les instants seront précieux et un gain sur la longueur du programme sera le bienvenu. Or, c'est ce que l'on obtient le plus souvent immédiatement en court-circuitant le langage évolué et son compilateur, en programmant en « assembleur ».

Hormis ces deux cas, alors oui, la programmation en langage « bas niveau » menant directement au langage machine est probablement à fuir, car elle est bien plus astreignante... et chère. Voici pourquoi.

L'hexadécimal, un moindre mal

En réalité, et depuis longtemps, personne ne code plus un programme en binaire. Si l'on tenait absolument à rester en langage « numérique », on transposerait le binaire en hexadécimal, par exemple. L'hexadécimal, c'est un mode de numération à base 16 ; on comptera ici de 0 à 15 en se servant d'un « mot » d'un seul symbole, et on passera à deux symboles (deux digits) au delà.

Puisque les chiffres de 0 à 9 sont disponibles, ils prendront leur place en hexadécimal ; mais de 10 à 15, on va devoir inventer d'autres symboles : autant utiliser ceux qui existent déjà sur les claviers des machines à écrire. Aussi attribuera-t-on la valeur 10 à A, 11 à B, 12 à C, 13 à D, 14 à E, et 15 à F. Ecrire B en hexadécimal signifie 11 en décimal. Donc, 5 + 6 = B. La valeur décimale 16 se codera, elle, 10 en hexadécimal. C'est déroutant au début, mais guère complexe puisque de très nombreux cours d'initiation à la programmation sur microprocesseurs font appel à cette méthode.

Ainsi, si l'on voulait transposer en hexadécimal le programme d'addition de 24 + 12 applicable au microprocesseur, on obtiendrait (en supposant, pour simplifier, que 24 et 12 sont, eux aussi, en hexadécimal) :

```
00 00 86 24
00 02 8B 12
00 04 97 10
00 06 3F
```

C'est toujours loin d'être évident mais c'est déjà bien plus court, et une erreur sur un digit aura davantage de chances de se faire repérer plus rapidement ! Pour trouver ces valeurs à partir du binaire précédent, il aura suffit de transposer chaque quartet binaire (chaque groupe de quatre bits) en un digit hexadécimal correspondant. A l'exception de 24 et 12 que l'on a supposé être ici déjà en hexadécimal (ce qui n'était pas le cas antérieurement).

Organiser un tel programme est aussi plus simple, puisqu'il n'y aura guère qu'à préparer un tableau de la façon suivante :

Adresse en mémoire	Code de l'opération à exécuter	Opérande ou adresse
00 00	8 6	24
00 02	8 B	12
00 04	9 7	10
00 06	3 F	

Puisqu'un digit hexadécimal représente quatre bits (binaires), deux digits représentent un octet (huit bits). Généralement, on considère qu'une cellule mémoire peut stocker un octet. Par conséquent, on constate que les codes de l'opération à exécuter sont codés sur un octet ; ici, les opérandes 24 et 12 sont également sur un octet. A la troisième ligne de ce programme, on a donné l'ordre de ranger le résultat (« 97 ») en mémoire dans la cellule portant le numéro dix, hexadécimal (c'est son adresse).

Souhaitez-vous quelques détails supplémentaires ? Sinon, sautez les quelques lignes qui suivent ! Voici : les deux colonnes de droite du tableau, qui représentent réellement le programme, verront leur contenu placé dans des cellules mémoires à partir de l'adresse zéro de départ (colonne de gauche). La première ligne occupant deux octets de programme (« 86 » et « 24 »), l'instruction suivante commencera non pas en 0001 mais en 0002. Puisque celle-ci se compose également de deux octets, la troisième ligne ne pourra commencer qu'en 0004 ; et ainsi de suite. Un peu d'attention montre que c'est réellement très simple...

Il est facile de frapper de tels codes sur un clavier « hexadécimal », comportant seize touches. Un tel clavier est plus économique que celui d'une machine à écrire et l'unité centrale de traitement de l'ordinateur qui le scrute saura immédiatement interpréter chaque touche en binaire. C'est là un problème d'organisation et de montage aisé à résoudre.

Ce qui reste toujours aussi peu évident, c'est la signification et les raisons du choix des « octets » de la colonne centrale. Pourquoi « 8 B » ordonne-t-il une addition ? De quoi avec quoi ? Et pourquoi aussi n'écrirait-on pas directement « Additionner » à la place ?

C'est bien ce que l'on a fait ensuite : on a remplacé tous ces codes sans signification évidente par des mots courants, ou plutôt par des mnémoniques non équivoques (si possible), pour en arriver au langage « assembleur ».

L'assembleur : mnémonique et symbolique

Si l'on disposait d'un clavier courant de machine à écrire, connecté à l'ordinateur (c'est alors un « terminal »), on pourrait frapper « additionner », ou plutôt « add » tout court, par souci d'économie, au lieu de 87 ; à la condition toutefois que la machine puisse comparer ce code mnémonique à ce qu'on a déjà confié à sa mémoire afin qu'elle le remplace par son équivalent numérique (binaire). Dès lors, un programme complet pourra être rédigé à l'aide de tels symboles mnémoniques, ce qui le rendra plus lisible.

La traduction mnémonique à numérique sera effectuée par un programme préalablement chargé dans l'ordinateur, l'« assembleur ». Il recevra chaque code mnémonique frappé et recherchera, comme dans « son » dictionnaire, à quel code numériquue il correspond. C'est ce code numérique que l'ordinateur utilisera pour exécuter le programme. La traduction elle-même sera appelée « assemblage » et le langage mnémonique « langage assembleur » ou « langage d'assemblage ».

Exemples d'instructions en langage bas niveau

(elles s'appliquent au microprocesseur « 6800 »)

Instruction	Code en langage assembleur	Code numérique correspondant (langage machine)	
		Hexadécimal	Binaire
Mettre dans l'accumulateur A	LDAA	86	1000 0110
Additionner au contenu de l'accumulateur A	ADDA	8B	1000 1011
Ranger le contenu de l'accumulateur A	STAA	97	1001 0111
Soustraire de l'accumulateur A	SUB	80	1000 0000
Se brancher à telle nouvelle instruction si le résultat de l'opération précédente a donné zéro	BEQ	27	0010 0111
Diminuer de 1	DEC	4C	0100 1100
Il y en a comme cela environ 75, valeur moyenne type			

La seule difficulté — si difficulté il y a — réside dans le fait que tous ces programmes provenant d'outre-Atlantique, les mnémoniques sont anglais ! Ainsi, « charger » une valeur dans un registre se dira « load », ou LD en abrégé, et ranger en mémoire le contenu d'un registre se dira « store » ou ST. Si le registre (une petite mémoire temporaire de l'unité centrale de traitement) est un accumulateur, et plus précisément l'accumulateur A (ce qui s'écrit AA en abrégé), le programme d'addition de 24 + 12 s'écrirait, en le décomposant tout d'abord :

1) — Charger 24 dans l'accumulateur A
2) — Lui ajouter 12
3) — Ranger le résultat à l'adresse 10
4) — Fin du programme

cela donnerait, en assembleur et en suivant de très près désormais les modes d'écriture normalisés :

```
LD AA 24 (LD = « load » = charger)
ADD A 12 (ADD = additionner)
ST AA 10 (ST = « store » = ranger)
SW 1
```

La quatrième et dernière ligne marque la fin du programme et « rend la main » au moniteur gérant le fonctionnement de l'ordinateur.

Avec les adresses en mémoire, on écrirait :

```
0000  LDAA  24
0002  ADDA  12
0004  STAA  10
0006  SW1
```

ce qui mène à un programme réel où il ne manque que quelques symboles précisant, par exemple, que les données sont en hexadécimal.

Ce sera l'assembleur qui décidera que LDAA correspond à 86, soit 1000 0110 en binaire, que ADDA représente 8B, soit 1000 1011, etc. Cela, parce que le fabricant du microprocesseur qu'on a adopté ici a introduit ces codes numériques dans le silicium de son circuit. Par conséquent, l'utilisateur ne pourra employer que les quelque soixante-quinze codes prévus par le fabricant, pas un de plus : leur liste figure dans la documentation que celui-ci lui remettra, codes numériques, codes mnémoniques correspondants, et même durée d'exécution de l'instruction ainsi posée.

C'est par conséquent en assembleur qu'on rédigera des programmes dont on voudra qu'ils soient plus rapidement exécutés ou qu'ils économisent les mémoires. Mais ces avantages se paient…

Le prix de la programmation en langage bas niveau

Une instruction rédigée en assembleur se traduit par une instruction en numérique. En revanche, une instruction en langage évolué pourra donner naissance à plusieurs instructions numériques après compilation. En effet, les instructions en langages évolués sont bien plus « puissantes », on ne leur demande pas de décomposer l'exécution (c'est le compilateur qui le fera), et par conséquent les programmes sont bien plus vite rédigés ; de plus, et parce que les langages évolués sont proches de la langue parlée, leur lisibilité est incomparablement supérieure. Toutes ces raisons concourent à réduire le temps de programmation.

En programmation, d'ailleurs, ce n'est pas le codage du programme en instructions qui prend le plus de temps, mais bien plutôt — et de loin — l'analyse du problème à résoudre et l'analyse de la conception du programme, puis et avec le codage en instructions, la recherche des erreurs que l'on aura pu commettre, leur corrections, et le test du programme complet ainsi que sa documentation. Or, plus le programme « source » (rédigé par l'opérateur) est court et proche de la langue parlée, plus il sera aisé de détecter et corriger des erreurs. A nouveau, l'avantage revient aux langages évolués.

Certains considèrent ainsi que la programmation en langage évolué est dix fois plus rapide qu'en langage assembleur. Or, ce que l'on paie ici, ce sont des heures d'un personnel très qualifié, difficile à trouver (parce qu'on en forme insuffisamment par rapport aux besoins : nous vivons actuellement les débuts d'une véritable crise de main-d'œuvre), et dont le salaire suit d'ailleurs la courbe des prix. La programmation coûtera donc de plus en plus cher : une instruction revient actuellement entre 60 et 120 F et l'on considère souvent qu'un programmeur expérimenté en rédige en moyenne une dizaine par jour (temps d'analyse et corrections compris). Autant, alors travailler en langage évolué…

Pourtant, rappelons les termes actuels du compromis : le langage évolué économise le temps de programmation, le langage bas niveau économise les circuits mémoires et le temps d'exécution.

A l'avenir, le prix des circuits intégrés mémoires continuant à baisser avec l'accroissement de leur densité, cet obstacle disparaîtrait. D'autre part, le temps d'exécution pouvant être réduit par d'autres moyens (meilleurs algorithmes, fonctionnement en « pipe-line », montage d'unités parallèles…), on peut légitimement supposer que l'avenir affirmera la prédominance des langages évolués pour les utilisateurs.

Déjà, certains systèmes deviennent d'ailleurs tellement complexes et puissants qu'on ne peut plus les programmer en assembleur, quand bien même le voudrait-on. Certains traitant d'« objets » par exemple ou encore ce tout récent microprocesseur de 32 bits dénommé APX 432 par la société Intel et qui, en trois circuits intégrés tenant dans le creux de la main, offre une puissance comparable à celle de l'ordinateur IBM 370-158 : il ne se programmera plus qu'en langage évolué, et pas n'importe lequel car ce sera ADA (1).

Ainsi, les termes du choix tel qu'il se trouve aujourd'hui posé ne résisteront probablement pas bien longtemps et la programmation va-t-elle très certainement basculer totalement dans le camp des langages évolués. En attendant d'ailleurs que ces derniers soient à leur tour dépassés et balayés. Mais on n'en est pas encore là ; aussi, de grâce, ne condamnons pas avant terme les langages bas niveaux ! ●

Programmation : où passe le temps ?

Programmer, ce n'est pas seulement rédiger des instructions en quelque langage que ce soit. Cette tâche est peut-être même la plus facile de toutes celles qu'implique la programmation et l'on s'accorde à reconnaître qu'elle occupe moins de 25 % du temps consacré à élaborer un programme. Où passent donc les 75 % restants ?

Avant d'aligner des instructions, il faut procéder à une analyse détaillée et complète du problème à traiter. Puis, on élabore la logique du programme : c'est l'étape de conception. Ce n'est qu'ensuite qu'on rédigera les instructions. Mais le programme dressé comportera certainement des erreurs, et il faudra les détecter et les corriger. Cela fait, il n'est pas encore certain que le programme réponde à ce qu'on attend de lui : on le teste donc. Enfin, il fonctionne… Tout au long de ces étapes, il aura encore fallu « documenter » ce que l'on a fait, c'est-à-dire ajouter toutes les notes utiles qui faciliteront la lecture du programme à une tierce personne, mais aussi à… son rédacteur.

C'est ce qui explique qu'une moyenne de dix instructions par programmeur et par journée de huit heures — toutes ces étapes étant confondues — est considérée comme très satisfaisante pour un programme moyen ou long ; par exemple, de 5 000 instructions. De ce fait, un tel programme demandera 500 jours ouvrables de travail. C'est pourquoi c'est le plus souvent le logiciel qui constitue la partie la plus onéreuse. Une solution : utiliser des progiciels chaque fois que faire se peut.

(1) - ADA a été développé par une équipe française de la CII-HB et adopté par le département américain de la Défense. Adà est le prénom de la comtesse Ada de Lovelace, fille de Lord Byron et amie de l'Anglais Babbage, l'un des précurseurs de l'informatique pour qui elle a rédigé les premiers programmes.