Les machines : architecture des ordinateurs - d'une introduction historique à la définition d'une machine virtuelle universelle -

Olivier Cogis, Jérôme Palaysi, Richard Terrat

Les machines : architecture des ordinateurs

d'une introduction historique à la définition d'une machine virtuelle universelle

Olivier Cogis olivier.cogis@umontpellier.fr
Jérôme Palaysi palaysi@lirmm.fr
Richard Terrat richard.terrat@umontpellier.fr

28 janvier 2020

Résumé

Cet article est une présentation de ce qu'on appelle communément l'architecture des ordinateurs en Informatique. Il est destiné aux étudiants de niveau Licence ou Master en Informatique, notamment à ceux préparant un CAPES d'informatique, comme aux enseignants du secondaire qui souhaitent accompagner l'apparition de la discipline Informatique au lycée. Il suit un plan en 5 parties :

- La genèse des ordinateurs où l'on introduit progressivement et en suivant la voie historique les étapes qui ont conduit aux fondements de l'architecture que l'on connaît actuellement
- L'architecture de base des ordinateurs en exposant les grands principes communs à toutes les réalisations
- La présentation d'un Ordinateur Réduit Facile Évolutif Universel (que nous appelons ORFEU) illustrant ce type d'architecture et son langage d'assemblage (LAMOR). Cet ordinateur et ce langage pouvant également servir de base à l'élaboration de séances d'enseignement.
- Quelques extensions facilitant la programmation d'un ordinateur de base
- Les architectures évoluées du processeur et des mémoires

Il est suivi d'une brève conclusion et de quelques annexes

Table des matières

1 La genèse

1.1 Étymologie

Le *Dictionnaire historique de la langue française*[1] précise que le mot *Ordinateur* fut d'abord employé pour « celui qui institue (en parlant du Christ) ». Entre le XIe et le XVIIe siècle, il désigne celui qui est chargé de « régler les affaires publiques », puis au XIXe siècle, « celui qui met de l'ordre ».

De son côté, le *Dictionnaire des sciences*[2] dirigé par Michel Serres et Nayla Farouki évoque « un vieux mot de latin d'église qui désignait, dans le rituel chrétien, celui qui procède à des ordinations et règle le cérémonial ».

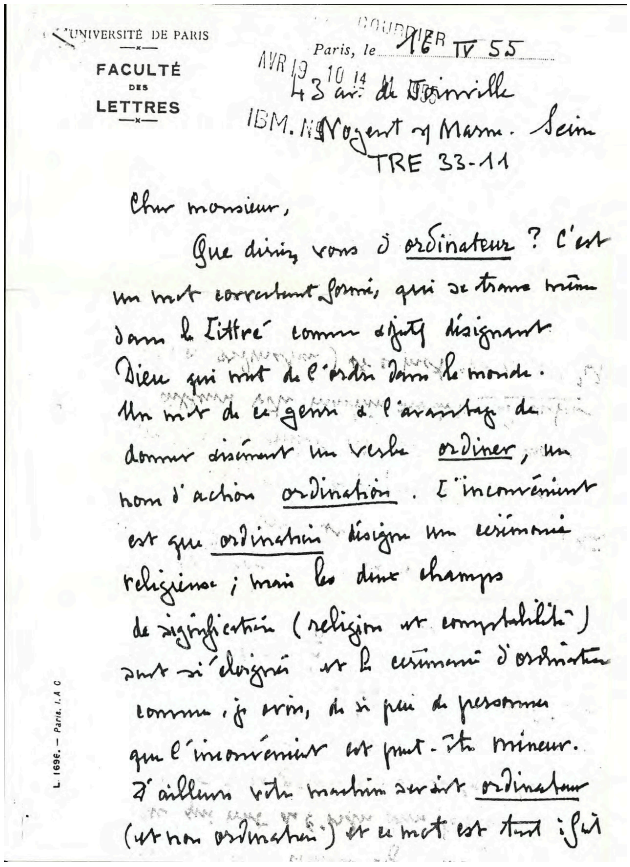
1. Dictionnaire historique de la langue française 2 volumes - NE - Alain Rey - 2016 -Le Robert

2. Le Trésor : dictionnaire des sciences - Michel Serres, Nayla Farouki - 1997 - Flammarion

C'est l'idée de mise en ordre qui semble prévaloir. Ordinateur apparaît dans les dictionnaires du XIXe siècle comme synonyme peu usuel de ordonnateur : celui qui met en ordre.

Le sens nouveau a été proposé par le professeur de philologie Jacques Perret dans une lettre datée du 16 avril 1955 en réponse à une demande de François Girard, responsable du service de publicité d'IBM France, dont les dirigeants estimaient le mot calculateur (*computer*) bien trop restrictif en regard des possibilités de ces machines.

C'est un exemple très rare de la création d'un néologisme authentifiée par une lettre manuscrite et datée (cf. Figure 1 *Lettre de Jacques Perret à François Girard (1955)* page 5)


L'UNIVERSITÉ DE PARIS
FACULTÉ DES LETTRES
Paris, le 16 IV 55
43 av. de Friedland
IBM. Neuilly-sur-Marne. Seine
TRE 33-11

Cher monsieur,

Que dirions-nous d'ordinateur ? C'est un mot correctement formé, qui se trouve même dans le Littré comme adjectif désignant Dieu qui met de l'ordre dans le monde. Un mot de ce genre a l'avantage de donner aisément un verbe ordiner, un nom d'action ordination. L'inconvénient est que ordination désigne une cérémonie religieuse ; mais les deux champs de signification (religion et comptabilité) sont si éloignés et la cérémonie d'ordination comme, je crois, de si peu de personnes que l'inconvénient est peut-être mineur. J'aillerais votre machine serait ordinateur (et non ordinateur) et ce mot est tout à fait

L. 1696. — Paris. I.A.C.

FIGURE 1: Lettre de Jacques Perret à François Girard (1955)

1.2 Quelques définitions

« Équipement informatique comprenant les organes nécessaires à son fonctionnement autonome, qui assure, en

exécutant les instructions d'un ensemble structuré de programmes, le traitement rapide de données codées sous forme numérique qui peuvent être conservées et transmises. »

Académie française neuvième édition[3]

« Machine automatique de traitement de l'information, obéissant à des programmes formés par des suites d'opérations arithmétiques et logiques. »

Dictionnaire Larousse[4]

« Système de traitement de l'information programmable tel que défini par Turing et qui fonctionne par la lecture séquentielle d'un ensemble d'instructions, organisées en programmes, qui lui font exécuter des opérations logiques et arithmétiques. »

Wikipedia[5]

« Machine électronique possédant une grande capacité de mémoire, capable de traiter automatiquement l'information grâce à des programmes codés enregistrés dans sa mémoire. »

Dictionnaire Reverso[6]

1.3 Bref historique

Si l'on met à part les outils antiques de calcul comme les bouliers et les abaques, on peut établir une chronologie de l'architecture des calculateurs, de la renaissance à nos jours et même rêver un peu au futur avec les ordinateurs quantiques :

— Les premières machines à calculer mécaniques

3. https://www.dictionnaire-academie.fr/article/A9O0665
4. https://www.larousse.fr/dictionnaires/francais/ordinateur/56358
5. https://fr.wikipedia.org/wiki/Ordinateur
6. https://dictionnaire.reverso.net/francais-definition/ordinateur

— L'introduction de la mémoire
— Le codage binaire
— Le calcul logique
— La programmation
— Les machines électromécaniques
— L'ordinateur : machine électronique universelle
— Le micro-ordinateur
— L'ordinateur quantique

1.4 Les premières machines à calculer mécaniques

C'est au XVIIème siècle que deux inventions vont se disputer la palme de la première machine à calculer mécanique :

— celle de l'allemand SCHICKARD [7] en 1623
— et la PASCALINE française en 1642 (Figure 2, page 7).

FIGURE 2: La pascaline (1642)

Si la première est sujette à controverse, car seuls des plans en font état et aucune machine de cette époque n'a jamais été retrouvée, la seconde est indiscutable comme en témoigne la quelque dizaine de modèles d'origine existant encore de nos jours.

Ces machines étaient essentiellement capables de faire des additions. Cependant, Blaise PASCAL avait déjà trouvé le moyen de faire des soustractions en utilisant le codage du complément à 10.
Dans cette première approche, le schéma peut se résumer comme celui de la Figure 3, page 8.

7. `https://fr.wikipedia.org/wiki/Wilhelm_Schickard`

FIGURE 3: Machine à calculer numérique

1.5 L'introduction de la mémoire

Les machines précédentes mériteraient plutôt le nom de machines à cumuler.

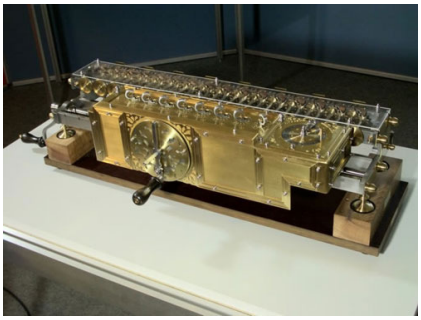
En effet, il n'est possible de visualiser qu'un seul nombre, résultant du cumul des additions et soustractions précédentes. Les opérandes sont oubliés.

Faire un produit ou une division nécessite de coucher sur le papier l'ensemble des résultats intermédiaires qu'il est nécessaire d'effectuer.

L'étape suivante va donc consister à conserver ces résultats intermédiaires pour que le calcul puisse s'effectuer sans passer par une intervention humaine de mémorisation.

C'est l'apparition des mémoires.

On peut alors effectuer les 4 opérations arithmétiques habituelles, ce que feront la machine de LEIBNIZ en 1673, l'Arithmomètre de THOMAS en 1820 (Figure 4, page 8) ... et bien d'autres !



La machine de Leibnitz (1704)

L'arithmomètre de Thomas (1820)

FIGURE 4: Machines à calculer avec mémoires

Le schéma se complète alors ainsi (Figure 5, page 9).

1.6 Le codage binaire

L'idée d'une représentation binaire des nombres entiers n'est pas récente. Elle est évoquée pour la première fois en Europe par Leibniz

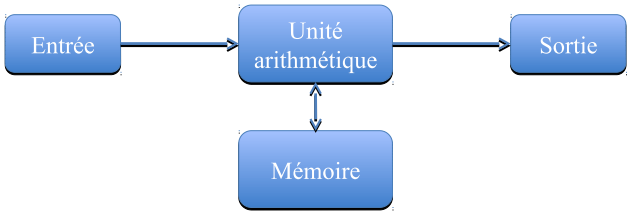


FIGURE 5: Introduction de la mémoire

qui la découvre dans les travaux de Francis Bacon (alphabet bilitère [8]) et surtout dans les documents de la Chine antique. Il fut initié à cette culture par des jésuites qu'il rencontra lors de son séjour en France de 1672 à 1676, et retrouva la structure du système binaire et des hexagrammes [9] dans le « Hi-King » traité philosophique basé sur l'opposition du Yin et du Yang et attribué à l'époque au légendaire empereur Fou-Hi (3ème millénaire avant JC).

Ses travaux sur la logique binaire attestés par une communication en 1703 à l'Académie des Sciences [10] annoncent avec 150 ans d'avance ceux de Boole !

À la fin de sa vie, il aura l'idée de construire une machine à calculer binaire, mais ne pourra mener à bien ce projet de son vivant.

1.7 Le calcul logique

En 1847 George Boole publie « Mathematical Analysis of Logic [11] », puis en 1854 « an Investigation Into the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probabilities [12] » et fonde ce qui est connu comme l'algèbre de Boole ou le calcul booléen, dans lequel il expose que les valeurs de vérité (Vrai ou Faux) des propositions peuvent se calculer comme des expressions proches des expressions arithmétiques.

Le calcul logique devient alors un cousin du calcul arithmétique en utilisant des opérateurs spécifiques tels que OU (qui se calcule *presque* comme une somme) ET (qui se calcule comme un produit) et NON (qui se calcule comme un changement de signe).

Il en résulte un schéma plus complet (Figure 6, page 10).

8. `https://www.apprendre-en-ligne.net/crypto/stegano/bilitere.html`
9. `https://fr.wikipedia.org/wiki/Hexagramme_Yi_Jing`
10. `https://gallica.bnf.fr/ark:/12148/bpt6k3483p/f247`
11. `http://www.gutenberg.org/files/36884/36884-pdf.pdf`
12. `http://www.gutenberg.org/files/15114/15114-pdf.pdf`

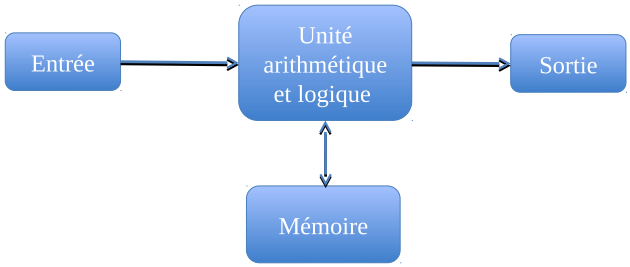


FIGURE 6: Introduction du calcul logique

1.8 La programmation

Même en mémorisant les résultats intermédiaires dans la machine à calculer, la suite des calculs doit néanmoins être mémorisée indépendamment.

L'étape suivante est d'une importance capitale : elle va consister à introduire un nouvel élément qui va se charger de l'enchaînement automatique de la suite des calculs intermédiaires : l'unité de commande.

La mémoire va alors jouer un nouveau rôle : non seulement elle enregistre les résultat des calculs intermédiaires mais elle va aussi devoir enregistrer la suite des opérations à effectuer.

Cette suite d'opérations va être codée de façon à ce que l'unité de commande puisse les réaliser.

C'est la naissance de la programmation.

C'est en 1834, que Charles Babbage [13] pendant le développement d'une machine à calculer destinée au calcul et à l'impression de tables mathématiques (machine à différences) eut l'idée d'y incorporer des cartes du métier Jacquard, dont la lecture séquentielle donnerait des instructions et des données à sa machine.

Par la suite il conçut une machine plus générale : la machine analytique [14] (Figure 7, page 11) qu'il ne put jamais réaliser faute de crédits, mais il passera le reste de sa vie à la concevoir dans les moindres détails et à en construire un prototype. Un de ses fils en construira l'unité centrale (appelée le moulin) et l'imprimante en 1888 et fit une démonstration réussie de calcul de tables à l'académie royale d'astronomie en 1908.

Le premier programme informatique

En octobre 1842, paraît en français, dans un journal suisse, une des-

13. `https://fr.wikipedia.org/wiki/Charles_Babbage`
14. `https://fr.wikipedia.org/wiki/Machine_analytique`

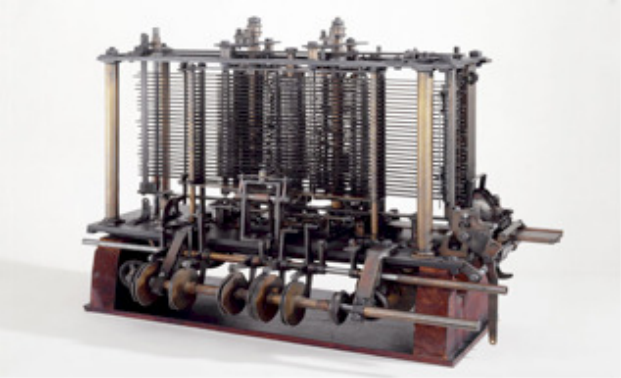


FIGURE 7: La machine analytique (1871)

cription de la machine analytique de Babbage [15] réalisée par le général mathématicien italien Federico Luigi Ménabréa [16], en collaboration avec Ada Lovelace [17], qui a un bon niveau de français.

Elle ajouta à cet article plusieurs notes, l'une d'elle mentionnant un véritable algorithme très détaillé pour calculer les nombres de Bernoulli avec la machine. Ce programme est considéré comme le premier véritable programme informatique au monde.

Elle a également émis, pour la première fois, l'idée que des machines à calculer pouvaient traiter des informations autres que des nombres :

> *« Beaucoup de personnes [...] s'imaginent que parce que la Machine fournit des résultats sous une forme numérique, alors la nature de ses processus doit être forcément arithmétique et numérique, plutôt qu'algébrique ou analytique. Ceci est une erreur*
>
> *La Machine peut arranger et combiner les quantités numériques exactement comme si elles étaient des lettres, ou tout autre symbole général ; en fait elle peut donner des résultats en notation algébrique, avec des conventions appropriées »*

Et même composer de la musique :

> *« La machine pourrait composer de manière scientifique et élaborée des morceaux de musique de n'importe quelle longueur ou degré de complexité »*

15. `http://www.bibnum.education.fr/sites/default/files/babage-menabrea-texte-final.pdf`
16. `https://fr.wikipedia.org/wiki/Luigi_Federico_Menabrea`
17. `https://fr.wikipedia.org/wiki/Ada_Lovelace`

Un nouveau schéma voit alors le jour, qui va servir de modèle à tous les développements ultérieurs (Figure 8, page 12).

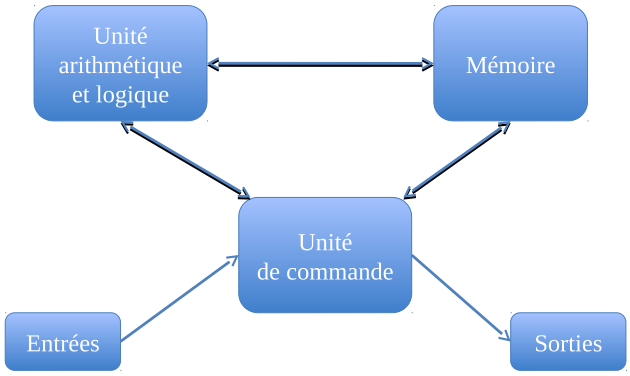


FIGURE 8: L'ordinateur

1.9 Les machines électromécaniques

On n'observera pas de modifications du schéma précédent, mais la fabrication des machines y gagnera en simplicité, en efficacité, en fiabilité et surtout en puissance de calcul.

On attribue généralement à l'ingénieur allemand Konrad Zuse [18] le premier calculateur conçu en 1936 avec mémoire et unité de commande, utilisant l'énergie électrique et le système binaire : le Z1. Par la suite, en 1937 il conçoit le Z3 utilisant pour la première fois la représentation des nombres réels en virgule flottante. Enfin, il fonde aussi le premier langage de haut niveau nommé Plankalkül [19] qu'il ne put utiliser faute de spécifications précises et d'une machine capable de le supporter. Ce langage resta ignoré de tous jusqu'à sa publication en 1972, plus de vingt ans après la sortie du Fortran [20] en 1954. Il ne fut opérationnel qu'en 2000 lorsqu'une équipe de l'université libre de Berlin développa une implémentation à titre historique.

A la même époque, en 1938, l'américain Claude Shannon [21] soutient son mémoire de maîtrise à l'Université du Michigan. Il y expose comment utiliser l'algèbre de Boole pour y construire des machines à calculer à base de commutateurs et de relais. Suivront alors de nombreuses réalisations fondées sur ces principes.

18. `https://fr.wikipedia.org/wiki/Konrad_Zuse`
19. `https://fr.wikipedia.org/wiki/Plankalkul`
20. `https://fr.wikipedia.org/wiki/Fortran`
21. `https://fr.wikipedia.org/wiki/Claude_Shannon`

1.10 L'ordinateur : machine électronique universelle

L'apparition de l'électronique va conduire à un développement spectaculaire, non sur le plan des principes, mais sur celui de la technologie ; l'ensemble des calculs pouvant être effectués par ces machines s'étend au fur et à mesure de la croissance de leur complexité. La question posée est alors celle-ci : peut on perfectionner les machines à calculer au point de pouvoir TOUT calculer avec elles ? C'est à cette question nouvelle et essentielle que vont répondre, la même année (1936) deux chercheurs : l'anglais Alan Turing [22] et l'américain Alonzo Church [23] son directeur de thèse à l'université de Princeton [24].

La réponse tient en deux propositions simples :

1. On peut montrer qu'il existe des nombres et des fonctions NON calculables.
2. Il est possible de construire des machines calculant TOUT ce qui est calculable et ces machines ont toutes une puissance de calcul équivalente [25] : cette proposition est une thèse – appelée « thèse de Church-Turing » [26] – qui ne peut être démontrée, dans la mesure où on ne peut pas prouver la non existence de machines de puissance supérieure.

Le premier ordinateur fonctionnant en langage binaire fut le *Colossus* [27] , conçu lors de la Seconde Guerre Mondiale ; il n'était pas Turing-complet bien qu'Alan Turing ait travaillé au projet. À la fin de la guerre, il fut démonté et caché à cause de son importance stratégique.

L'ENIAC [28], mis en service en 1946, est le premier ordinateur entièrement électronique construit pour être Turing-complet.

Pendant ce projet (en juin 1945 un an avant la démonstration de l'ENIAC) est publié un article fondateur : *First Draft of a Report on the EDVAC* [29] par John von Neumann donnant les bases de l'architecture utilisée dans la quasi totalité des ordinateurs depuis lors. Dans

22. `https://fr.wikipedia.org/wiki/Alan_Turing`
23. `https://fr.wikipedia.org/wiki/Alonzo_Church`
24. `https://fr.wikipedia.org/wiki/Universit%C3%A9_de_Princeton`
25. Dans le sens où toute fonction calculable par l'une est calculable par les autres. On qualifie alors ces machines de «Turing-complète» (`https://fr.wikipedia.org/wiki/Turing-complet`).
26. `https://journals.openedition.org/philosophiascientiae/769`
27. `https://fr.wikipedia.org/wiki/Colossus_(ordinateur)`
28. `https://fr.wikipedia.org/wiki/ENIAC`
29. `https://www.wiley.com/legacy/wileychi/wang_archi/supp/appendix_a.pdf`

cet article Von Neumann veut concevoir un programme enregistré et programmé dans la machine.

La première machine correspondant à cette architecture, dite depuis architecture de von Neumann, ne fut pas l'EDVAC [30] (Figure 9, page 14) qui ne sera livré qu'en Aout 1949, mais une machine expérimentale la *Small-Scale Experimental Machine* (SSEM ou "baby") construite à Manchester en juillet 1948 [31].

Il est intéressant de noter que l'ENIAC fut conçue à l'IAS [32] contre l'avis entre autres d'Albert Einstein et de Kurt Gödel, ces deux éminents membres de l'institut considérant que cette coûteuse réalisation n'apporterait aucune contribution à la science [33]. L'obstination de John von Neumann puisa son énergie dans un anticommunisme fortement encouragé par le gouvernement américain, en pleine période de guère froide, son principal argument en faveur de la construction de la machine étant de gagner la course engagée entre russes et américains pour la fabrication de la future arme nucléaire.

FIGURE 9: L'EDVAC (1949)

1.11 Le microordinateur

L'ENIAC et ses successeurs étaient fabriqués avec des circuits électroniques à base de lampes à vide, de taille importante, à forte consommation d'énergie et compte tenu de leur grand nombre le TMBF [34] de

30. `https://fr.wikipedia.org/wiki/Electronic_Discrete_Variable_Automatic_Computer`
31. `https://fr.wikipedia.org/wiki/Small-Scale_Experimental_Machine`
32. `https://fr.wikipedia.org/wiki/Institute_for_Advanced_Study`
33. Le vrai paradis de Platon - John L. Casti - 2005 - Le Pommier
34. Temps Moyen de Bon Fonctionnement

l'ordinateur ne dépassait pas quelques heures.

Ce type d'ordinateur constitue ce qu'on appelle généralement la première génération (1946-1956).
L'arrivée du transistor en 1947 va considérablement modifier la donne. Tout comme les lampes à vide il va se comporter comme un interrupteur, mais à moindres coût, taille et consommation d'énergie. Les ordinateurs de ce type constituent la seconde génération (1956-1963).

D'abord utilisé comme simple composant sur des circuits imprimés dans les années 60, il va se miniaturiser pour se fondre au sein de circuits intégrés (dont le premier fut conçu en 1958) de taille de plus en plus réduite au point d'être appelé « puce électronique ». Au début, ce sont les mémoires qui constituent les premières puces. Elles sont d'abord utilisées au sein de mécanismes électroniques classiques.
Ces ordinateurs constituent la troisième génération (1963-1971).

Puis, ce sera au tour de l'unité arithmétique et logique et de l'unité de commande de se fondre dans un circuit intégré qu'on appellera le microprocesseur dont le premier exemplaire fut créé en 1971 par la société Intel.

Enfin l'intégration de cet ensemble et des composants permettant de lui connecter des dispositifs d'entrée et de sortie dans un seul boîtier donnera naissance au microordinateur (Figure 10, page 16).

La question de savoir qui a créé le premier micro-ordinateur est source de controverse et dépend du degré d'intégration choisi. Selon certaines sources la première machine vendue toute assemblée prête à l'emploi, serait le Micral[35] conçu en 1972 par l'entreprise française R2E[36] (Figure 11, page 17).

Il s'agit d'un des tous premiers ordinateurs de la quatrième génération (depuis 1971).

1.12 L'ordinateur quantique

Selon la loi de Moore[37] la densité des composants électroniques sur une puce suivrait une croissance exponentielle à raison d'un doublement tous les 2 ans. Cette conjecture énoncée en 1965 a été confirmée dans les faits entre 1970 et 2000.

Cependant depuis cette dernière date des limites infranchissables semblent être atteintes dues à des phénomènes physiques telles que la sensibilité aux rayonnements extérieurs et en dernier ressort à des effets quantiques.

35. https://fr.wikipedia.org/wiki/Micral
36. https://fr.wikipedia.org/wiki/R2E
37. https://fr.wikipedia.org/wiki/Loi_de_Moore

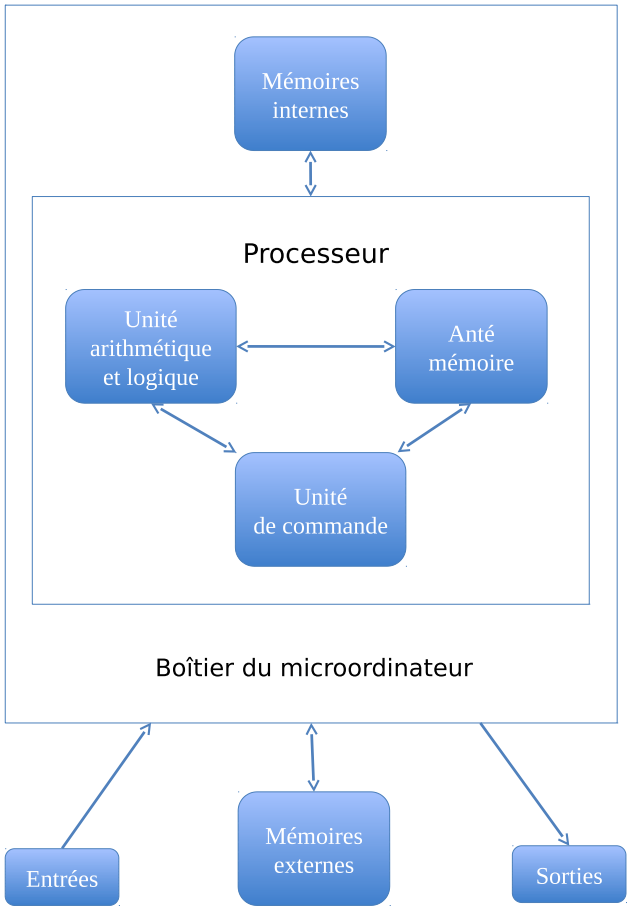


FIGURE 10: Le microordinateur

En effet, lorsque la taille des composants de base atteint l'échelle atomique, ceux ci ne se comportent plus comme le prévoient les lois de la physique classique mais obéissent à celles plus surprenantes de la physique quantique.

L'idée vient alors de construire des ordinateurs avec des composants quantiques [38]. L'un des premiers prototypes a été construit à l'université d'Innsbruck en 2011 [39] (cf. Figure 12 *Prototype d'ordinateur quantique* page 18).

Outre l'avantage de l'extrême densité que l'on peut en tirer, le comportement quantique de ces composants permet de tirer parti de

38. cf. Leçons sur l'informatique - Richard Feynman - Odile Jacob Sciences 1966 - Leçon n°6 p 251-284

39. `https://www.futura-sciences.com/sciences/dossiers/physique-ordinateur-quantique-552/page/7/`

FIGURE 11: Le Micral (1972)

propriétés fort intéressantes telles que la superposition d'états [40].

On conçoit alors qu'un ordinateur quantique puisse posséder une puissance bien supérieure au plus puissant des ordinateurs classiques. On a pu se demander si une telle puissance ne permettrait pas de calculer des fonctions non calculables au sens de la thèse de Church-Turing. La réponse à cette question est négative.

En revanche, il existe des problèmes pour lesquels nous ne connaissons pas d'algorithme polynomial pour machine classique mais des algorithmes polynomiaux pour machines quantiques : c'est par exemple le cas de la factorisation entière en nombre premiers (algorithme de Shor [41])

Deux grands obstacles restent encore à franchir avant de disposer chez soi de son ordinateur quantique :

— La taille gigantesque et le coût exorbitant estimé actuellement à plusieurs millions d'Euros pour quelques qu-octets (octets quantiques : ensemble de huit qu-bits)

— La conception d'algorithmes quantiques dont on ne connaît aujourd'hui que quelques rares exemples

40. `https://fr.wikipedia.org/wiki/Qubit`
41. `https://fr.wikipedia.org/wiki/Algorithme_de_Shor`

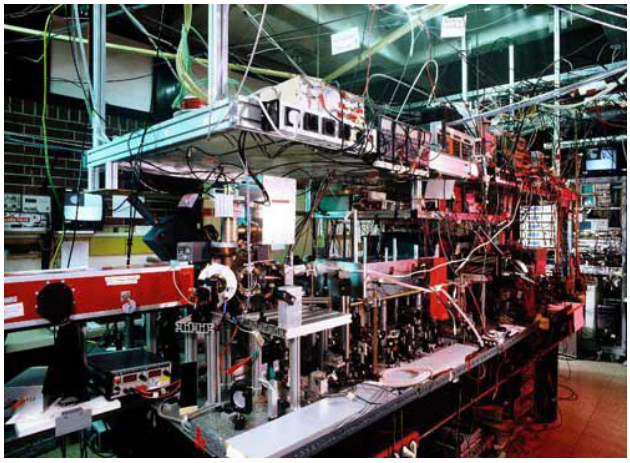


FIGURE 12: Prototype d'ordinateur quantique

2 Architecture de base dite de "Von Neumann"

2.1 Introduction

Le schéma (cf. Figure 8 *L'ordinateur* page 12) nous fournit les éléments centraux de l'architecture générale des ordinateurs, définie en 1945 par John Von Neumann.

Actuellement [42], c'est toujours avec cette architecture que sont construits tous les ordinateurs, du nano-processeur (que l'on trouve par exemple dans une machine à laver) au super-ordinateur (calcul intensif).

Osons un parallèle

Tout comme le cycle de Carnot[43] *fut le premier modèle des moteurs à explosion, la machine de Turing fut le premier modèle des ordinateurs.*

Tout comme le cycle de Carnot a donné naissance au modèle des moteurs à explosion actuels, (via le cycle de Beau de Rochas pour les moteurs à essence[44] *et au cycle Diesel*[45] *pour les moteurs ... Diesel), l'architecture de Von Neumann a donné naissance au modèle des ordinateurs actuels.*

Aujourd'hui, malgré toutes les avancées techniques indéniables, ces modèles sont toujours d'actualité !

42. en 2020

On retiendra les composants suivants :

La mémoire	réalisant la fonction de stockage
L'Unité Arithmétique et Logique	réalisant les calculs élémentaires
L'Unité de Commande	réalisant l'enchaînement des instructions

On désigne généralement par *processeur* l'ensemble de l'unité arithmétique et logique et de l'unité de commande.

2.2 La mémoire

Elle est composée de *cellules*.

Une cellule est référencé par son *adresse*, qui est en général un entier représentant le rang de son emplacement dans la mémoire. Il en résulte qu'il est assez naturel d'organiser l'ensemble de ces cellules comme un grand tableau. Si l'on désigne par M le nom de ce tableau, l'adresse X d'une cellule peut être vue comme l'indice de cette cellule dans le tableau M (cf. Figure 13 *La mémoire* page 21).

M[X] est alors l'*identifiant* de cette cellule dans ce tableau.

Par la suite on utilisera la notation abrégée [X] pour désigner cet identifiant.

Généralement la taille de ce tableau est une puissance de deux. Une adresse est alors constituée d'une suite fixe d'éléments binaires. La longueur de cette suite est appelé le *champ d'adressage* de la mémoire.

Le contenu d'une cellule est exprimé par une chaîne de bits qui peut représenter :

— une donnée : nombre, caractère, couleur, son, etc ...

— une instruction élémentaire du processeur

L'interprétation du contenu d'une cellule comme donnée ou comme instruction dépend de l'utilisation qui en est faite par le processeur. Le contenu d'une même cellule peut être interprété tantôt comme une donnée, tantôt comme une instruction ; cette dualité est exploitée pour la construction et la modification d'instructions pendant l'exécution d'un programme.

Les registres

Il s'agit de cellules distinguées qui jouent un rôle dédié et auxquelles on attribue un nom. il n'y a pas de désignation d'adresse.

Ces cellules sont généralement intégrées au sein de composants tels que l'Unité Arithmétique et Logique et l'Unité de Commande [46].

Opérations de transfert

Lors de l'exécution d'une instruction, des opérations de transfert entre cellules de mémoire et registres doivent être effectués.

On notera ces transferts par l'opérateur "←" de façon analogue à une affectation.

A ← B signifie donc que la valeur de B est transférée vers la cellule (mémoire ou registre) identifiée par A.

B est une expression arithmétique simple comprenant des identifiants de cellules, des valeurs numériques et des opérateurs simples.

Si C est l'identifiant d'une cellule figurant dans l'expression B, la valeur de C est la valeur contenue dans cette cellule.

Exemples :

CO, ACC et RI désignent des registres.

CO ← X	la valeur X est transférée dans le registre CO
CO ← CO + 1	le registre CO est incrémenté de 1
ACC ← [X]	la valeur contenue dans la cellule de mémoire d'adresse X, identifiée par [X], est transférée dans le registre ACC
ACC ← ACC + [X]	la valeur contenue dans la cellule de mémoire d'adresse X, identifiée par [X], est additionnée au contenu du registre ACC
[X] ← ACC	la valeur contenue dans le registre ACC est transférée dans la cellule de mémoire d'adresse X, identifiée par [X]
RI ← [CO]	la valeur contenue dans la cellule de mémoire dont l'adresse est le contenu du registre CO, identifiée par [CO], est transférée dans le registre RI.

2.3 L'unité arithmétique et logique (UAL)

On la représente habituellement par ce schéma : cf. Figure 14 *L'unité arithmétique et logique* page 22

Deux registres sont destinées à stocker les opérandes :

46. `https://fr.wikipedia.org/wiki/Registre_de_processeur`

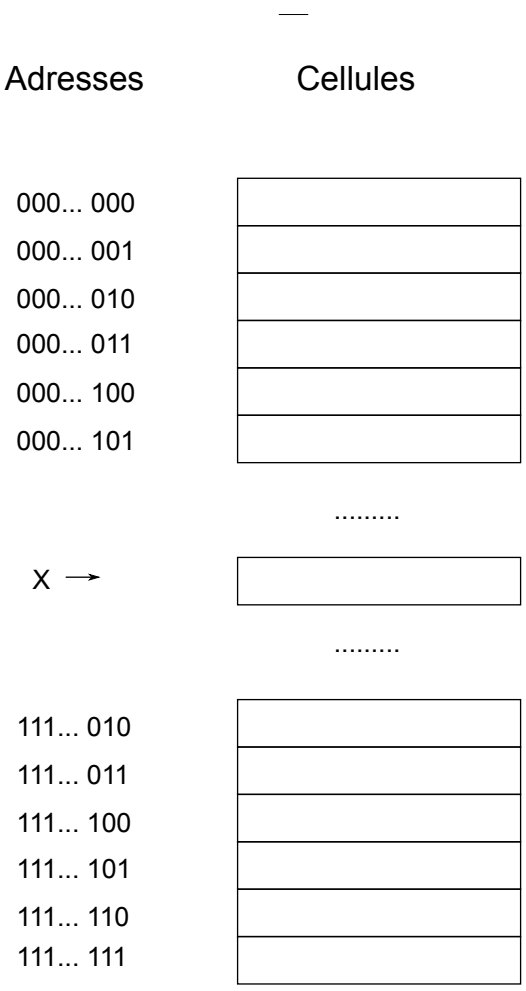


FIGURE 13: La mémoire

— ACC appelée Accumulateur connectée à l'une des entrées et à la sortie de l'UAL ;

— AUX connectée à l'autre entrée de l'UAL.

L'entrée F permet d'activer la fonction que doit réaliser l'UAL.

Cette fonction ne peut donc être qu'une opération binaire sur des opérandes situés dans les registres ACC et AUX, le résultat étant retourné dans le registre ACC.

Les opérations que peuvent réaliser une UAL de base sont typiquement :

— Les opérations arithmétiques usuelles (+, -, x, /) ;

— Les opérations logiques (ET, OU, NON).

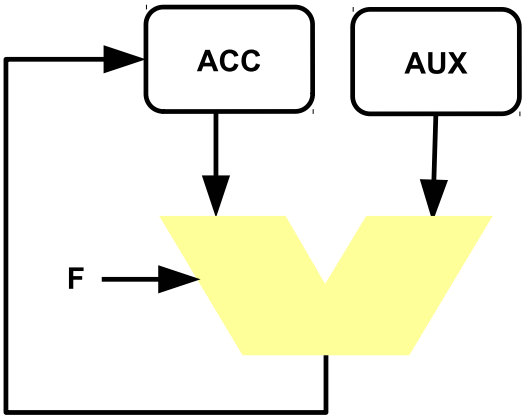


FIGURE 14: L'unité arithmétique et logique

Certaines UAL ont des fonctions spécifiques (virgule flottante, opérations graphiques) et sont même parfois très spécialisées (transformées de Fourrier).

Un même processeur peut comporter plusieurs UAL dont certaines sont spécialisées et ne fonctionnent pas simultanément, ou bien plusieurs UAL pouvant fonctionner en parallèle (architectures multi-cœur).

Toutes les fonctions réalisées par une UAL le sont grâce à des circuits électroniques effectuant des fonctions binaires. Or la théorie nous apprend que toute fonction binaire peut être réalisée avec un seul des opérateurs suivants : NON-ET (NAND) ou NON-OU (NOR). Par un heureux concours de circonstances, chacun de ces opérateurs peut être construit à l'aide de quelques transistors ! Il s'ensuit qu'un simple (pas tout à fait) assemblage de ces composants permet de réaliser toutes les fonctions souhaitées [47].

2.4 L'unité de commande

Il s'agit du composant commandant toutes les autres parties de l'ordinateur (cf. Figure 15 *L'unité de commande* page 23). Son fonctionnement, pourtant très simple, permet de faire de l'ordinateur une machine *Turing-complète* [48].

Le principe réside dans la façon d'enchaîner la suite des opérations constituant un calcul, opérations qui sont enregistrées dans la mémoire interne sous forme d'instructions élémentaires.

47. cf. Leçons sur l'informatique - Richard Feynman - Odile Jacob Sciences 1966 - Leçon n°2 p 31-68

48. `https://fr.wikipedia.org/wiki/Turing-complet`

2.4.1 Description

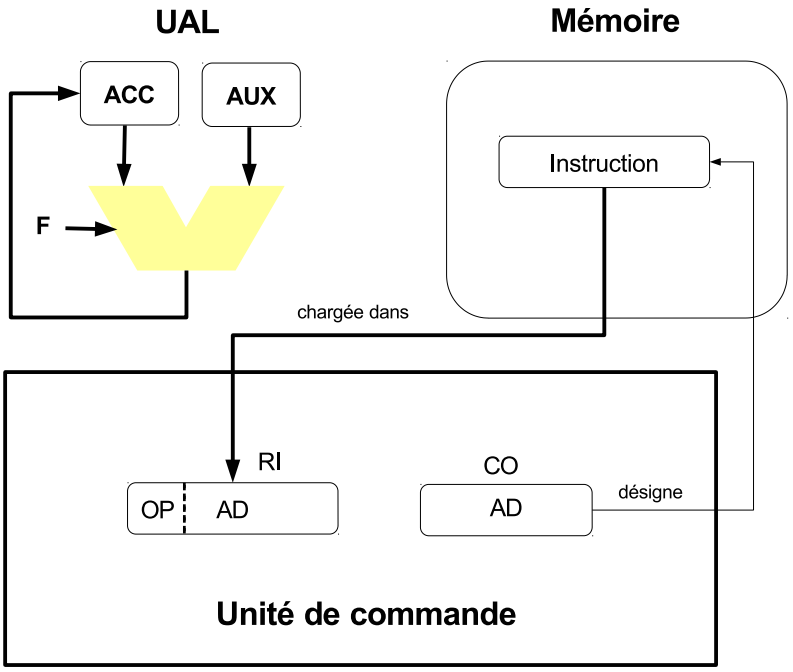


FIGURE 15: L'unité de commande

L'unité de commande est composée principalement de deux registres :

Le registre d'instructions	RI	contient l'instruction en cours d'exécution
Le compteur ordinal	CO	contient l'adresse de la prochaine instruction à exécuter

Le nom de ce dernier registre a d'ailleurs très probablement influencé le choix francophone du mot « ordinateur ».

Le registre RI a la même taille (le même nombre de bits) que les cellules de la mémoire.

Le registre CO a la même taille (le même nombre de bits) que le champ d'adressage de la mémoire.

2.4.2 Fonctionnement

L'unité de commande orchestre l'exécution d'un programme en répétant indéfiniment les trois étapes ci-dessous :

répéter

1. RI ← [CO] : chargement dans le registre d'instruction RI du contenu de la cellule désignée par le compteur ordinal CO.
2. CO ← CO + 1 : incrémentation du compteur ordinal
3. Exécution de l'instruction située dans le registre d'instructions : cf. subsubsection 2.5.2 *Exécution* page 25

FIGURE 16: Algorithme du cycle d'instructions

qu'on appelle le *cycle d'instruction.*

La simplicité d'un tel mécanisme pourrait porter à croire que les instructions exécutées chronologiquement sont rangées consécutivement en mémoire. En réalité, la troisième étape autorise la modification du compteur ordinal au moyen d'instructions spécifiques appelées *rupture de séquence*, ce qui confère à cette architecture toute sa puissance de calcul.

2.4.3 Contrôle de séquence

Pour réaliser ces instructions, il est nécessaire de désigner explicitement la prochaine instruction à exécuter, donc de disposer d'instructions pouvant modifier explicitement le compteur ordinal. Ces instructions sont généralement appelées "ruptures de séquence" ou "sauts" ou "branchements".

Elles peuvent être inconditionnelles ou conditionnelles.

Pour une rupture de séquence inconditionnelle, la modification du compteur ordinal est conditionnée par un test portant, par exemple, sur le contenu de l'accumulateur [49] : si le résultat de ce test est positif le chargement du compteur ordinal est effectué, sinon rien ne se passe.

Dans le cas d'une rupture de séquence inconditionnelle aucun test n'est effectué.

49. Dans le cas d'architectures d'UAL plus sophistiquées, un registre complémentaire appelé généralement *code condition* permet des test plus complets portant, par exemple sur le résultat d'une opération précédente (dépassement de capacité, division par zéro, etc ..)

2.5 Les instructions

2.5.1 Constitution

Une instruction est stockée dans une cellule de mémoire.
Elle est constituée de 2 champs :

— un code opération noté OP qui indique quelle opération doit être effectuée ;

— une adresse notée AD désignant l'opérande de cette opération.

On peut en distinguer plusieurs types :

— les instructions élémentaires, comprenant elles-mêmes :
  — les instructions de transfert entre la mémoire et l'accumulateur de l'UAL : chargement et stockage ;
  — les instructions effectuant des calculs arithmétiques et logiques.

— les instructions de contrôle de séquence, permettant de réaliser les instructions conditionnelles et répétitives

On conçoit que, pour chacune de ces catégories, une instruction doit être d'une extrême simplicité et ne peut agir que sur un seul argument, appelé ici un opérande.

2.5.2 Exécution

L'instruction est contenue dans le registre d'instruction RI.
Son exécution se déroule ainsi (cf. Figure 17 *Exécution d'une instruction* page 26) :

Instructions de transfert

Le champ AD du registre d'instruction RI désigne l'*adresse* de l'opérande.

Instruction de chargement

Le contenu de la cellule désignée par cette adresse est transféré dans l'accumulateur : ACC $\leftarrow$ [AD]

Instruction de stockage

Le contenu de l'accumulateur est transféré dans la cellule désignée par cette adresse : [AD] $\leftarrow$ ACC

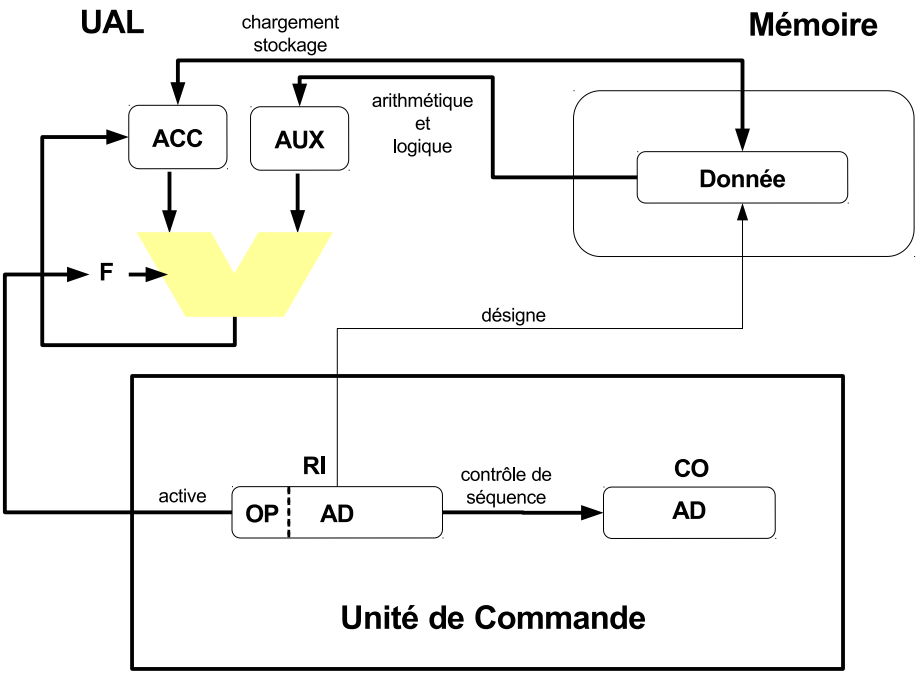


FIGURE 17: Exécution d'une instruction

Instructions arithmétiques et logiques

Le champ AD du registre d'instruction RI désigne l'*adresse* de l'opérande.

Le champ OP du registre d'instruction RI désigne l'opération à faire exécuter par l'UAL.

Le contenu de la cellule désignée par cette adresse est transféré dans le registre auxiliaire : AUX $\leftarrow$ [AD] .

Le code opération active la fonction correspondante dans l'UAL.

Le résultat de l'opération est retourné dans l'accumulateur.

Instructions de contrôle de séquence

Le champ AD du registre d'instruction RI désigne la *valeur* de l'opérande.

Instruction de rupture de séquence inconditionnelle

Cette valeur est transférée dans le compteur ordinal : CO ← AD

Instruction de rupture de séquence conditionnelle

Cette valeur est transférée dans le compteur ordinal si, et seulement si, le test invoqué par l'instruction est positif.

3 ORFEU

ORFEU est un Ordinateur Réduit Facile Extensible Universel.

Il est conçu selon l'architecture fondamentale de Von Neumann, décrite précédemment.

Il servira de guide pour illustrer précisément cette architecture en la concrétisant par des exemples commentés de programmes mettant en œuvre des algorithmes conçus selon divers paradigmes : programmation impérative, objets et récursive.

3.1 L'unité arithmétique et logique

Elle est conçue pour exécuter les instructions élémentaires suivantes :

- Addition
- Soustraction
- ET logique
- OU logique
- OU exclusif
- Décalage arithmétique

Les nombres sont représentés dans ces cellules selon le complément à 2 [50]

Les opérations arithmétiques se font modulo 2^n, n étant la taille (nombre de bits) des cellules de mémoire. Il n'y a aucun contrôle du dépassement de capacité.

Ils sont interprétés modulo 2^n comme des entiers relatifs appartenant à l'intervalle $[-2^{n-1} : 2^{n-1}[$.

Les booléens sont représentés par des éléments binaires (bits) :

- la valeur *FAUX* est représentée par le bit 0
- la valeur *VRAI* est représentée par le bit 1

50. cf. subsubsection 7.1.2 page 61 : "Représentation des nombres"

Les opérations booléennes agissent sur l'ensemble des bits d'une cellule [51].

3.2 La mémoire

La taille des cellules est de 16 bits (ou 2 octets).

La taille de la mémoire est de 4096 cellules.

L'adresse d'une cellule est un nombre entier compris entre 0 et 4095.

Le champ d'adressage est donc de 12 bits.

3.3 Les instructions

Chaque instruction est logée dans une cellule mémoire selon le format suivant :

— code opération (COP) : 4 bits

— adresse de l'opérande (AD) : 12 bits

Le jeu d'instructions

COP	AD	Commentaire	Exécution
Instructions de contrôle de séquence			
0000	X	Non opération	si X = 0 ARRÊT sinon rien*
0001	X	Saut inconditionnel	CO $\leftarrow$ X
0010	X	Saut si négatif	si ACC $<$ 0 CO $\leftarrow$ X sinon rien*
0011	X	Saut si zéro	si ACC = 0 CO $\leftarrow$ X sinon rien*
Instructions de transfert et instructions arithmétiques			
0100	X	Chargement	ACC $\leftarrow$ [X]
0101	X	Stockage	[X] $\leftarrow$ ACC
0110	X	Addition	ACC $\leftarrow$ (ACC + [X]) mod 2^{16}
0111	X	Soustraction	ACC $\leftarrow$ (ACC $-$ [X]) mod 2^{16}
Instructions logiques			
1000	X	Et logique	ACC $\leftarrow$ ACC $\wedge$ [X]

51. cf. subsubsection 7.2.1 page 64 : "Opérations booléennes élémentaires"

1001	X	Ou logique	ACC $\leftarrow$ ACC $\vee$ [X]
1010	X	Ou exclusif	ACC $\leftarrow$ ACC $\oplus$ [X]
1011	X	Décalage arithmétique	ACC $\leftarrow$ (ACC $\times$ $2^{[X]}$) mod 2^{16}

Autres

1100	Non défini
1101	Non défini
1110	Non défini
1111	Non défini

* rien est l'instruction vide qui ne fait ... rien !

TABLE 1: Le jeu d'instructions de base d'ORFEU

Remarques :

- dans les instructions de contrôle de séquence, c'est la valeur de l'adresse X, désignant la cellule de mémoire [X], qui est chargée dans le compteur ordinal ;
- dans les instructions de transfert et les instructions arithmétiques et logiques, [X] identifie la cellule d'adresse X. En partie gauche d'une affectation (instruction STO) il s'agit du nom de cette cellule. En partie droite il s'agit de sa valeur (contenu) ;
- l'instruction de décalage arithmétique est définie précisément ici [52] ;
- au démarrage d'un programme ORFEU, le compteur ordinal CO est initialisé à 0. Cela signifie que la première instruction à exécuter est contenue dans la cellule d'adresse 0.

3.4 Un exemple de programme ORFEU

Le contenu initial de la mémoire, appelé aussi *image mémoire initiale* de cet exemple, est représenté par le Table 2 page 30 : "Image mémoire initiale".

Adresse	Contenu
000000000000	0001000000000110
000000000001	0000000000110010

52. cf. subsubsection 7.3.2 page 66 : "Décalages arithmétiques : instruction DAR"

000000000010	0000000001000000
000000000011	0000000000000000
000000000100	0000000000000001
000000000101	1111111111111111
000000000110	0000000000000001
000000000111	0011000000010100
000000001000	1000000000000100
000000001001	0011000000001101
000000001010	0100000000000011
000000001011	0110000000000010
000000001100	0101000000000011
000000001101	0100000000000001
000000001110	1011000000000101
000000001111	0101000000000001
000000010000	0100000000000010
000000010001	1011000000000100
000000010010	0101000000000010
000000010011	0001000000000110
000000010100	0000000000000000

TABLE 2: Image mémoire initiale

Ici, la première instruction de ce programme est un saut inconditionnel à l'adresse 000000000110 (6 en décimal) Cela signifie que le programme va se poursuivre à partir de cette adresse.

Les cellules de mémoire d'adresses comprises entre 000000000001 (1 en décimal) et 000000000101 (5 en décimal) contiennent des données. En notant les adresses et les données en décimal on constate aisément leurs emplacements (adresses des cellules) et leur valeurs (contenus des cellules) (cf. Table 3 page 30 : "Cellules de données").

Adresse	Contenu
1	50
2	64
3	0
4	1
5	-1

TABLE 3: Cellules de données

Enfin, le programme se termine à l'adresse 000000010100 (20 en

décimal) sur une instruction d'arrêt de l'ordinateur.

Bien évidemment, ce qui est intéressant est ce qui se passe entre le début et la fin de ce programme.

En exécutant les instructions pas à pas, on verra apparaître à la fin la valeur décimale 3 200 dans la cellule d'adresse 000000000011 (3 en décimal).

Et, en y réfléchissant un peu, on doit pouvoir dire ce que fait ce programme.

Pour s'en convaincre, on peut changer, à sa guise, les valeurs initiales de 50 et 64 (en décimal) contenues dans les cellules d'adresses décimales 1 et 2, puis refaire des exécutions pas à pas. Ce n'est certes pas une preuve formelle, mais ce n'en est pas moins fort convaincant.

Enfin, on peut faire une remarque intéressante :

Les instructions de ce programme sont contenues dans 16 cellules contenant chacune 16 bits, soit un total de 256 bits.

Que se passerait-il si l'on changeait un seul bit sur ces 256 ? Y aurait-il la moindre chance pour que ce programme fonctionne encore en donnant un résultat facilement prévisible ?

3.5 LAMOR

D'une façon générale, la conception d'un programme destiné à un ordinateur se fait par l'intermédiaire d'un langage de programmation.

Ce langage est dit *de haut niveau*[53] s'il est capable de décrire un programme destiné à une très large variété d'ordinateurs, indépendamment de leur architecture et notamment de leur jeu d'instructions.

À l'inverse, si ce langage est destiné à l'écriture d'un programme pour un ordinateur particulier, pour un jeu d'instructions bien défini, il est appelé *langage d'assemblage*[54].

LAMOR est le Langage d'Assemblage de la Machine ORFEU

Sa structure est très simple : chaque ligne significative[55] de LAMOR correspond à une cellule d'ORFEU.

Les lignes successives d'un programme correspondent aux cellules successives d'ORFEU.

53. `https://fr.wikipedia.org/wiki/Langage_de_haut_niveau`
54. `https://fr.wikipedia.org/wiki/Assembleur`
55. les lignes non significatives sont vides ou ne contiennent qu'un commentaire

La traduction d'un programme LAMOR vers une image mémoire d'ORFEU peut se faire aisément *à la main* ou plus aisément à l'aide d'un programme appelé un *assembleur*.

Une ligne de programme LAMOR est composée de 4 champs :

- une étiquette (optionnelle)
- un contenu de cellule lui même composé de
  - soit une donnée numérique écrite en décimal
  - soit une instruction composée de
    - un code opération symbolique
    - l'identificateur symbolique d'une adresse
- un commentaire (optionnel)

Chaque champ est séparé par un séparateur constitué d'un ou de plusieurs espaces ou tabulations.

S'il existe une étiquette elle doit être placée en début de ligne.

Un contenu de cellule est nécessairement précédé d'au moins un séparateur

La syntaxe précise de LAMOR est décrite ici : [56]

3.5.1 Codes opération symboliques

À chaque code opération exprimé en binaire dans ORFEU, on fait correspondre un code opération symbolique, selon la Table 4 page 33 : "Codes Opération"

ORFEU	LAMOR	Commentaire
0000	NOP	Non opération
0001	SIN	Saut inconditionnel
0010	SSN	Saut si négatif
0011	SSZ	Saut si zéro
0100	CHA	Charger
0101	STO	Stocker
0110	ADD	Additionner
0111	SUB	Soustraire
1000	ETL	Et logique
1001	OUL	Ou logique
1010	OUX	Ou exclusif
1011	DAR	Décalage arithmétique

56. cf. section 9 page 76 : "La grammaire de LAMOR"

TABLE 4: Codes Opération

3.5.2 Étiquettes et identificateurs symboliques d'un opérande

À chaque adresse figurant dans une instruction, et seulement dans celles-ci, on fait correspondre un identificateur.

Cette correspondance, propre à chaque programme, est construite au moment de l'assemblage dans une *table des adresses* .

3.5.3 Un exemple de programme LAMOR

Le programme de la Listing 1 page 33 : "Un programme LAMOR" est une expression en LAMOR du programme ORFEU de la Table 2 page 30 : "Image mémoire initiale".

Listing 1: Un programme LAMOR

```
# Produit de deux entiers naturels

          SIN       DEBUT
X                   50
Y                   64
Z                   0
UN                  1
MUN                 -1
DEBUT     CHA       X
          SSZ       FIN       #  Tant-Que [X]!=0
          ETL       UN
          SSZ       SUITE     # si [X] pair sauter à SUITE
          CHA       Z
          ADD       Y
          STO       Z         # [Z] <-- [Z] + [Y]
SUITE     CHA       X
          DAR       MUN
          STO       X         # [X] <-- [X] / 2
          CHA       Y
          DAR       UN
          STO       Y         # [Y] <-- [Y] * 2
          SIN       DEBUT     # fin du Tant-Que
FIN       NOP       0
```

La table des adresses de ce programme est donnée par la Table 5 page 34 : "Table des adresses"

Adresse	Identificateur

000000000001	X
000000000010	Y
000000000011	Z
000000000100	UN
000000000101	MUN
000000000110	DEBUT
000000001101	SUITE
000000010100	FIN

TABLE 5: Table des adresses

3.6 Dix exemples de programmes LAMOR

Expression arithmétique (Listing 8.1 page 68)

Ce programme calcule une expression arithmétique simple
C'est un exemple d'instruction séquentielle.

Instruction conditionnelle (Listing 8.2 page 68)

Ce programme calcule le cumul de deux nombres entiers sous condition d'imparité d'un troisième nombre entier

Instruction alternative (Listing 8.3 page 69)

Ce programme calcule le maximum de deux entiers

Itération (Listing 8.4 page 69)

Ce programme calcule la somme des 100 premiers entiers

Algorithme d'Ahmès (Listing 8.5 page 70)

Ce programme calcule le produit de deux nombres entiers naturels, inspiré par l'algorithme d'Ahmès.

Adressage indexé (Listing 8.6 page 70)

Ce programme calcule le maximum des éléments d'un tableau d'entiers.

Afin d'adresser successivement les éléments de ce tableau, on procède à un *calcul d'instructions*

— avant l'itération, la cellule DEBUT contient l'instruction CHA T qui charge le premier élément du tableau, d'adresse T

— en cours d'itération cette cellule est chargée dans l'Accumulateur et est donc considérée comme une donnée

— puis le contenu de l'Accumulateur est incrémenté de la valeur de la cellule d'adresse I et devient donc CHA T+[I]

— cette valeur est stockée dans la cellule d'adresse INS considérée alors comme une donnée

— enfin la cellule d'adresse INS est exécutée en tant qu'instruction : CHA T+[I]

Cet exemple montre que l'on peut

— considérer le contenu d'une cellule tantôt comme donnée, tantôt comme instruction

— modifier l'adresse d'un opérande en cours d'exécution d'un programme

— parcourir un tableau sans avoir besoin de mécanisme supplémentaire

Appel et retour de sous-programme (Listing 8.7 page 71)

Ce programme est constitué de deux parties :

— un programme dit *principal*

— un *sous-programme* de type *fonction*

Le programme principal effectue les opérations suivantes :

— les *paramètres effectifs* [A] et [B] sont transmis par valeur aux *paramètres formels* [X] et [Y] du sous-programme

— l'instruction de retour [IR], qui dépend évidemment du programme appelant, est recopiée dans la dernière cellule du sous-programme [FSP]

— le contrôle passe au sous-programme

— le résultat retourné par le sous-programme via l'accumulateur ACC est stocké dans la cellule [MAX]

Le sous-programme effectue les opérations suivantes :

— le calcul de maximum de [X] et [Y]
— le retour du résultat dans l'accumulateur
— le retour du contrôle au programme principal

D'une façon générale, un sous-programme peut réaliser une *fonction* ou une *procédure* :

— dans le cas d'une fonction, les seules cellules référencées par le sous-programme doivent être celles des paramètres formels
— dans le cas d'une procédure aucun résultat n'est retourné via l'accumulateur

Les paramètres peuvent être transmis :

— par *valeur* : dans ce cas ce sont les valeurs contenues dans les cellules des paramètres effectifs qui sont recopiées dans les cellules des paramètres formels (ici par exemple [X] ← [A])
— par *référence* : dans ce cas ce sont les adresses des cellules des paramètres effectifs qui sont recopiées dans les cellules des paramètres formels ; il faut alors utiliser des cellules intermédiaires dans le programme appelant pour pouvoir transmettre ces adresses

Adressage indirect et pointeurs (Listing 8.8 page 72)

Ce programme est conçu comme une réalisation du paradigme de *programmation objet*.

La cellule [RECT] (pour Rectangle) représente un *pointeur* vers un objet constitué de deux cellules successives commençant à l'adresse OBJET. Ces cellules sont une réalisation de deux *attributs* représentant la hauteur et la largeur de ce rectangle.

Pour accéder à ces attributs il est nécessaire de pouvoir déposer leurs adresses dans les instructions qui les utilisent (ici les instructions [CHAH] et [CHAL] chargeant dans l'accumulateur les valeurs des contenus de ces attributs).

On utilise pour cela un *squelette* de l'instruction CHA contenant l'adresse "0" ; ce squelette est ensuite enrichi des adresses des attributs par l'instruction OUL (ou logique).

On réalise ainsi un *adressage indirect* des cellules des attributs, tel que [[RECT]+ 1], identifiant ici le second attribut de l'objet désigné par le pointeur [RECT].

Pile (Listing 8.9 page 72)

Ce programme est constitué de trois parties :

— un programme principal contenant une pile
— un sous programme *empiler* de type procédure
— un sous programme *dépiler* de type mixte (fonction et procédure)

Une pile est un objet (au sens du paradigme de programmation objet) permettant de stocker les contenus de cellules, selon le principe « dernier arrivé, premier sorti » (en anglais LIFO : last in, first out), ce qui veut dire que les derniers contenus de cellules, ajoutés à la pile, seront les premiers à en sortir [57].

On appelle *sommet de pile* la cellule de la pile désignée par le pointeur [PTP].

Cette cellule ne contient aucune valeur significative. C'est la cellule précédant immédiatement le sommet de pile qui contient :

— la valeur de la dernière cellule empilée qui est aussi
— la valeur de la première cellule qui sera dépilée

On observera que ce pointeur est incrémenté de 1 *après* un empilement et décrémenté de 1 *avant* un dépilement.

En conséquence, [PTP] désigne toujours avant et après l'exécution de chacun de ces sous programmes, le sommet de pile.

Le programme principal :

— empile successivement au somment de la pile les contenu des cellules [A] puis [B]
— dépile successivement le somment de pile vers les cellules [C] puis [D]

On observera qu'à la fin de l'exécution du programme principal :

— la cellule [C] contient la même valeur que la cellule [B]
— la cellule [D] contient la même valeur que la cellule [A]

Sous programme récursif (Listing 8.10 page 74)

Ce programme est conçu comme une réalisation du paradigme de *programmation récursive*.

Il calcule le carré d'un nombre entier naturel n défini par l'algorithme :

57. `https://fr.wikipedia.org/wiki/Pile_(informatique)`

$$carré(n) = \begin{cases} 0 & \text{si } n = 0 \\ carré(n-1) + 2 \times n - 1 & \text{sinon} \end{cases}$$

Il est constitué de deux parties :

— un programme principal contenant une pile

— un sous programme récursif de type fonction effectuant le calcul du carré

Le fonctionnement d'un sous programme récursif nécessite l'utilisation d'une pile.

Au moment de chaque appel, initial et récursif, sont empilés :

— l'adresse de retour

— les paramètres (un seul dans le cas de cet exemple)

— les variables locales qui sont utilisées dans le sous programme *avant* et *après* l'appel récursif (il n'y en a pas dans cet exemple)

Au cours de l'exécution du sous programme les paramètres et variables locales sont accessibles dans la pile et peuvent être temporairement stockés *avant* ou *après* l'appel récursif dans des cellules locales.

À la fin de l'exécution du sous programme :

— le sommet de pile est mis à jour

— l'adresse de retour est dépilée.

— la valeur du résultat est retournée dans l'accumulateur

— le contrôle retourne au programme appelant (programme principal en fin de récursivité ou sous programme après un appel récursif)

4 Extensions de l'architecture de base

L'architecture d'ORFEU [58] a permis de montrer qu'il est possible de réaliser toutes les structures de contrôle des algorithmes itératifs et récursifs dans les paradigmes de programmation procédurale et par objets.

Cette architecture est pourtant assez sommaire. Elle a été présentée dans le cadre de cet ouvrage à des fins pédagogiques afin de rendre compte des mécanismes de base de l'exécution de programmes sur un ordinateur.

58. cf. section 3 page 27 : "ORFEU"

Or certains de ces mécanismes, tels que le parcours d'un tableau, la réalisation de sous-programmes, la représentation des objets, et la récursivité nécessitent des *calculs d'instructions*.

Cela signifie que le programme se modifie en cours d'exécution. En conséquence, la simple lecture du programme initial (celui écrit par le programmeur) ne permet pas de rendre compte de son exécution, ce qui va à l'encontre d'une programmation bien structurée permettant une lisibilité et une maintenance abordable.

Certes, peu d'applications sont aujourd'hui écrites directement en langage d'assemblage et encore moins en langage machine. Cependant c'est encore le cas pour l'écriture de parties les plus profondes des *systèmes d'exploitation* (celles nécessitant la connaissance de la structure détaillée de l'ordinateur) et de protocoles de bas niveau pour les *réseaux*.

Des mécanismes plus élaborés sont donc nécessaires pour parvenir à une meilleure structuration de ces applications. Certains nécessitent simplement l'ajout de registres et d'instructions supplémentaires. D'autres imposent de compléter le *cycle d'instructions*[59].

Quant aux programmes écrits dans des langages de haut niveau, leur compilation va se trouver grandement simplifiée grâce à ces extensions.

4.1 Ajout de registres

Une première extension, la plus simple, consiste en l'ajout de registres, notamment des accumulateurs.

Cette extension permet de conserver des données temporaires au sein de l'unité de commande, évitant ainsi des transferts avec la mémoire.

Un nouveau champ est ajouté au format des instructions pour adresser ces registres.

4.2 Le code condition

Certaines instructions peuvent donner des résultats inattendus ou impossibles à obtenir.

C'est le cas, par exemple de :

— division par 0 (pour les processeurs disposant de la division) ;

— dépassement de capacité lors d'une addition, d'une soustraction ou d'un décalage arithmétique.

59. cf. subsubsection 2.4.2 page 24 : "Fonctionnement"

Si certaines situations peuvent être prévues avant l'exécution de l'instruction (division par 0), d'autres, comme le dépassement de capacité ne sont détectables qu'après.

Ainsi, lors d'une addition Z = X + Y, le dépassement de capacité se produit quand Z est de signe contraire à $X + Y$:

$$x_{n-1} = y_{n-1} \neq z_{n-1}$$

où n est la taille (nombre de bits) de l'accumulateur.

Pour éviter d'inclure systématiquement ces tests dans un programme, avant ou après l'exécution d'une instruction, une sortie supplémentaire est ajoutée à l'Unité Arithmétique et Logique, reliée à un registre appelé le *code condition*.

Ce registre de quelques bits renseigne sur des situations résultant de l'exécution de la dernière instruction telles que :

— tentative de division par 0 ;

— dépassement de capacité.

Il peut ensuite être testé par des instructions *ad hoc* de sauts conditionnels qui doivent donc faire partie du jeu d'instructions du processeur, telles que :

— SSD X : saut à X en cas de tentative de division par 0 ;

— SSC X : saut à X en cas de dépassement de capacité.

4.3 Adressage

Plusieurs mécanismes d'adressage appelés des *modes d'adressage* concourent à la simplification et à la lisibilité des programmes écrits ou compilés en langage machine.

Ces mécanismes nécessitent :

— une extension du format des instructions

— l'adjonction de registres au sein de l'unité de commande

— une extension de l'algorithme du cycle d'instructions

Un nouveau champ est ajouté au format des instructions.

Il comporte des informations relatives à différents modes d'adressage, qui peuvent être combinés entre eux.

On s'intéressera ici aux modes les plus courants :

— adressage indexé

— adressage basé

— adressage indirect

D'autres modes plus élaborés sont utilisés dans des architectures plus évoluées ou au sein de processeurs spécialisés.

4.3.1 Adressage indexé

Un ou plusieurs registres d'index permettent d'adresser directement un élément d'un tableau.

Pendant la phase de recherche de l'opérande, lors de l'exécution d'une instruction [60], son adresse est *calculée* en ajoutant au champ AD du registre d'instruction RI la valeur contenue dans le registre d'index spécifié dans le champ du mode d'adressage.

Ainsi, par exemple, le chargement dans l'accumulateur du contenu d'une cellule désignée par l'adresse X *et* par le registre d'index RI_2 produira :

$$ACC \leftarrow [X + RI_2]$$

ce qui permet d'adresser *directement* la cellule du tableau [X] d'indice RI_2.

Généralement la taille du champ d'un registre d'index (nombre de bits) est la même que celle du champ d'adressage de la mémoire.

4.3.2 Adressage basé

Un ou plusieurs registres de base permettent d'adresser directement un attribut d'un objet.

L'adresse figurant en opérande d'une instruction est alors interprétée *relativement* à l'objet contenant cet attribut.

Pendant la phase de recherche de l'opérande, lors de l'exécution d'une instruction [61], son adresse est *calculée* en ajoutant au registre de base RB la valeur contenue dans le champ AD du registre d'instruction RI.

Ainsi, par exemple, le chargement dans l'accumulateur du contenu d'une cellule désignée par l'adresse X relative à un objet désigné par le registre de base RB_2 produira :

$$ACC \leftarrow [RB_2 + X]$$

ce qui permet d'adresser *directement* la cellule désignée par [X] contenant l'attribut de l'objet désigné par le registre de base RB_2.

La taille du champ d'un registre de base (nombre de bits) est toujours identique à du champ d'adressage de la mémoire.

On remarquera que, finalement, le mode d'adressage basé fonctionne de façon identique à celui de l'adressage indexé. C'est l'utilisation qui en est différente.

60. cf. subsubsection 2.5.2 page 25 : "Exécution"
61. cf. subsubsection 2.5.2 page 25 : "Exécution"

De plus, en combinant ces deux modes d'adressage, on peut accéder *directement* à un élément d'un tableau attribut d'un objet.

4.3.3 Adressage indirect

Ce mode d'adressage va permettre la gestion de *pointeurs* chers aux langages objets.

Aucun registre supplémentaire n'est nécessaire.

L'adresse figurant en opérande d'une instruction est alors interprétée comme celle d'un pointeur désignant un objet.

L'adresse de l'objet est donc désignée par le contenu de la cellule elle même désignée par l'opérande de l'instruction.

Ainsi, par exemple, le chargement dans l'accumulateur du contenu d'une cellule désignant un objet désigné par un pointeur d'adresse X produira :

$$ACC \leftarrow [[X]]$$

Autrement dit, c'est le *contenu du contenu* de la cellule désignée par [X] qui est chargée dans l'accumulateur.

4.3.4 Composition de modes d'adressage

Il est possible de composer les trois modes d'adressage précédemment décrits.

Adressage pré-indexé Par exemple : pour accéder directement à un objet désigné par un pointeur contenu dans un tableau (généralement de pointeurs) on utilisera la composition :

$$\textit{Adressage indirect} \circ \textit{Adressage indexé}$$

signifiant qu'on applique d'abord l'indexation puis l'indirection.

Adressage post-indexé Autre exemple : pour accéder directement à un élément de tableau, ce dernier étant désigné par un pointeur, on utilisera la composition :

$$\textit{Adressage indexé} \circ \textit{Adressage indirect}$$

signifiant qu'on applique d'abord l'indirection puis l'indexation.

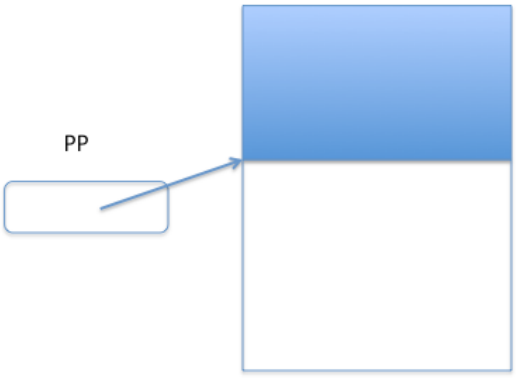


FIGURE 18: La pile d'exécution

4.4 Pile d'exécution

Afin de structurer de longs programmes, il est intéressant de les scinder en plusieurs sous-programmes (appelés aussi routines) selon une structure hiérarchique. Une telle structure permet aussi de réaliser des programmes récursifs. De surcroît un mécanisme bien conçu permet de réaliser des co-programmes (appelés aussi co-routines) qui sont la base des processus.

Pour cela, on ajoute à l'unité de commande un nouveau registre PP : le Pointeur de Pile (Figure 18 page 43 : "La pile d'exécution")

Ce registre contient une adresse désignant une cellule de mémoire.

Deux nouvelles instructions : EMP (Empiler) et DEP (Dépiler) complètent le jeu d'instructions.

EMP X réalise PP $\leftarrow$ PP + 1 ; PP $\leftarrow$ [X]

Le pointeur de pile désigne l'adresse suivante ; puis le contenu de la cellule ADR est transféré dans la cellule dont l'adresse est située dans le registre PP

DEP X réalise [X] $\leftarrow$ PP ; PP $\leftarrow$ PP - 1

Le contenu de la cellule dont l'adresse est située dans le registre PP est transféré dans la cellule d'adresse ADR ; puis le pointeur de pile désigne l'adresse précédente

4.4.1 Sous-programmes

Pour la réalisation de sous-programmes, on ajoute encore 2 instructions :

— APS (Appel de sous-programme) et

— RET (Retour de sous-programme).

L'adresse du sous-programme est désignée par ADR

APS X réalise $PP \leftarrow PP + 1$; $[PP] \leftarrow CO$; $CO \leftarrow X$

Le contenu du compteur ordinal est empilé puis chargé de la valeur ADR

RET réalise $CO \leftarrow [PP]$; $PP \leftarrow PP - 1$

Le contenu du somment de la pile est dépilé dans le compteur ordinal

La pile est utilisée pour y loger bien d'autres données que le compteur ordinal. On y trouve aussi les variables locales et les paramètres des sous-programmes.

accès aux variables locales et paramètres

4.4.2 Coroutines

Les coroutines[62] sont des suites d'instructions pouvant s'exécuter en quasi-parallélisme. Elles constituent le mécanisme de base des fils d'exécution (*Threads*)[63], des *processus*[64] et d'une façon générale de la programmation concurrente[65].

Aucune nouvelle instruction n'est requise pour réaliser des coroutines. Il suffit de considérer que chaque coroutine possède sa propre pile. La commutation de coroutines se résume donc à un simple changement de pile.

4.5 Interruptions

Le cycle d'instruction précédemment décrit[66] ne permet aucune communication de ou vers des composants extérieurs. Ces composants sont d'une part les dispositifs d'entrée, de sortie ou de stockage d'informations, d'autre part des dispositifs particuliers tels qu'une horloge permettant par exemple la commutation automatique de processus en partage de temps ou bien la réalisation de programmes en temps réel.

62. `https://fr.wikipedia.org/wiki/Coroutine`
63. `https://fr.wikipedia.org/wiki/Thread_(informatique)`
64. `https://fr.wikipedia.org/wiki/Processus_(informatique)`
65. `https://fr.wikipedia.org/wiki/Programmation_concurrente`
66. cf. subsubsection 2.4.2 page 24 : "Fonctionnement"

Le mécanisme de base permettant la prise en compte d'événements extérieurs est intimement lié au cycle d'instruction qui doit être modifié dans ce but. C'est le mécanisme d'interruption [67] (sous entendu : de programme). Pour cela on complète l'unité de commande d'un simple bit d'interruption : INT dont la valeur vrai par exemple, indique qu'un événement extérieur est intervenu (faux sinon). Et on réserve une adresse en mémoire : AIT (par exemple l'adresse 0) pour contenir le programme qui devra s'exécuter lors de l'arrivée d'un événement extérieur.

Le cycle d'instructions se complète alors ainsi : Figure 19 page 45 : "Cycle d'instructions avec interruptions".

répéter
    si INT **alors**
        PP ← PP + 1
        [PP] ← CO
        CO ← AIT
        INT ← **faux**
    RI ← [CO]
    CO ← CO + 1
    Exécution de l'instruction située dans le registre d'instructions

FIGURE 19: Cycle d'instructions avec interruptions

La séquence ajoutée au cycle d'instruction de base réalise un appel de sous-programme à l'adresse réservée AIT lorsqu'un événement extérieur a mis le bit INT à vrai.

Ce mécanisme est assez élémentaire ; il ne permet pas de prendre en compte des interruptions multiples, des priorités entre événements et d'une façon générale une gestion complexe des interruptions.

Dans les architectures actuelles, un composant électronique (puce) appelé PIC [68] est dédié à cette tâche qui peut s'avérer très sophistiquée.

4.6 Exceptions

Il s'agit d'une extension de l'utilisation du mécanisme d'interruptions. En effet, lors de l'exécution de certaines instructions, des situations particulières peuvent se produire telles que : dépassement de ca-

67. https://fr.wikipedia.org/wiki/Interruption_(informatique)
68. https://en.wikipedia.org/wiki/Programmable_interrupt_controller

pacité lors d'une addition, division par 0, etc ... Bien entendu, le code condition reflétera ces situations. Mais pour en tenir compte, le programme devra tester ce code de façon systématique après l'exécution de ces instructions. Or, en général ces situations sont exceptionnelles. On utilise alors le mécanisme d'interruptions comme s'il s'agissait d'un événement externe pour les signaler. On parle alors d'interruptions internes ou *exceptions*[69]. Certains langages de programmation de haut niveau permettent le traitement de ces exceptions par le biais de routines spécifiques.

Enfin, ces exceptions sont utilisées dans les systèmes d'exploitation pour la gestion de certaines ressources spécifiques telles que la mémoire virtuelle, la commutation de processus par exemple.

4.7 Modes et sécurité

Dans le cas d'un ordinateur dont l'utilisation peut être partagée par plusieurs utilisateurs, le problème de la confidentialité et de la protection de données personnelles peut se poser. Pour tenir compte de cette particularité, certaines instructions ne peuvent être exécutées que si des conditions spécifiques sont réunies. Le processeur fonctionne alors selon deux ou plusieurs modes distincts : un (ou des) modes utilisateur ou bien un (ou des) modes système (appelés aussi privilégiés).

Dans le mode utilisateur, certaines instructions ne s'exécuteront pas mais provoqueront des exceptions. Dans le mode système, toutes les instructions peuvent être exécutées.

Bien évidemment, il est nécessaire de pouvoir passer d'un mode à un autre. Le passage du mode utilisateur au mode système est réalisé par des instructions spécifiques appelées des appels système qui sont des appels de routines avec changement de mode. Pour que la sécurité soit assurée, il est bien évident que ces routines ne doivent pouvoir être ni écrites ni modifiées par un utilisateur ! Pour que cette condition soit réalisée, ces routines sont logées dans des emplacements de mémoire non accessibles à l'utilisateur. Un dispositif de protection de la mémoire doit donc être présent et toute instruction pouvant modifier ces emplacements protégés ne doit pas pouvoir s'exécuter en mode utilisateur.

69. `https://fr.wikipedia.org/wiki/Système_de_gestion_d'exceptions`

5 Architectures évoluées

5.1 Le processeur

L'architecture de l'unité de commande présentée dans les paragraphes précédents est basée sur l'existence d'un seul processeur et sur un déroulement séquentiel des instructions.

Les évolutions s'orientent dans de nombreuses directions :

- la microprogrammation des instructions [70]
- le jeu réduit d'instructions (architecture RISC [71])
- un ensemble d'UAL, chacune avec un jeu spécialisé d'instructions
  - processeur virgule flottante (FPU [72])
  - processeur graphique (GPU [73])
  - processeur de signal numérique (DSP [74])
  - etc ...
- un parallélisme dans l'exécution d'un ensemble d'instructions grâce à des chaînes de calcul (pipeline [75] ou infoduc)
- un ensemble d'UAL fonctionnant en parallèle (architecture Multi-cœur [76])
- et enfin des architectures de processeurs ou mémoires fonctionnant en parallèle
  - SIMD [77]
  - MISD [78]
  - MIMD [79]
  - Multiprocesseurs [80]

5.1.1 Microprogrammation

Au fur et à mesure de l'évolution de l'unité de commande, les instructions deviennent de plus en plus complexes. La réalisation de ces

70. https://fr.wikipedia.org/wiki/Microprogrammation
71. https://fr.wikipedia.org/wiki/Reduced_instruction_set_computing
72. https://fr.wikipedia.org/wiki/Unité_de_calcul_en_virgule_flottante
73. https://fr.wikipedia.org/wiki/Processeur_graphique
74. https://fr.wikipedia.org/wiki/Processeur_de_signal_numérique
75. https://fr.wikipedia.org/wiki/Pipeline_(architecture_des_processeurs)
76. https://fr.wikipedia.org/wiki/Microprocesseur_multi-c\T1\oeur
77. https://fr.wikipedia.org/wiki/Single_instruction_multiple_data
78. https://fr.wikipedia.org/wiki/Multiple_instructions_single_data
79. https://fr.wikipedia.org/wiki/Multiple_instructions_on_multiple_data
80. https://fr.wikipedia.org/wiki/Multiprocesseur

instructions par des circuits électroniques rend donc de plus en plus complexe la conception des puces qui les supportent.

Or ces instructions peuvent être décrites par des programmes.

La microprogrammation consiste alors à réaliser ces instructions comme des programmes, en stockant leurs codes dans des mémoires permanentes non modifiables (ROM).

L'exécution de chaque instruction se résume donc à l'appel du microprogramme qui la réalise.

5.1.2 Jeu réduit d'instructions

Cette architecture appelée RISC (Reduced Instruction Set Computer) prend un peu le contrepied de la microprogrammation avec le même but de simplifier la conception des puces.

A partir de la constatation que plus une instruction est complexe et moins elle est utilisée, le parti pris de cette architecture consiste à définir un jeu d'instructions minimum, fréquemment utilisées et dont l'exécution est rapide et rendue efficace par une architecture plus simple.

5.1.3 Ensemble d'UAL spécialisées

Leurs noms parlent d'eux-mêmes :

— FPU : Floating Point Unit est une UAL spécialisée dans l'arithmétique des nombres en virgule flottante

— GPU : Graphics Processing Unit est une UAL spécialisée dans les calculs invoqués dans le traitement des images

— DSP : Digital Signal Processor ou Digital Sound Processor est une UAL spécialisée dans le traitement du signal (notamment du son) ; elle possède en particulier des algorithmes micro-programmés de FFT (Fast Fourrier Transform)

À noter qu'on trouve souvent ce type d'UAL dans des dispositifs tels que les appareils audionumériques (préamplificateurs, générateurs de sons d'ambiance, appareils pour mal entendants)

5.1.4 Chaînes de calcul

On part du fait que le cycle d'instructions est composé de plusieurs phases successives.

Imaginons par exemple qu'il en comporte 4 : Ph1, Ph2, Ph3, Ph4.

Une suite de 4 instructions successives I1, I2, I3 et I4 est supposée présente dans un infoduc (un viaduc d'informations).

Une phase du cycle d'instruction est alors exécuté comme suit (l'expression a//b indiquant que les opérations a et b sont exécutées en parallèle) :

Ph4 de I1 // Ph3 de I2 // Ph2 de I3 // Ph1 de I4

suivie du décalage de l'infoduc et de l'introduction de l'instruction I5 , l'instruction I1 ayant alors terminé son exécution

Ce qui revient à dire que globalement pendant un cycle d'instruction, 4 instructions ont été exécutées en parallèle.

Mais il y a un hic ! Il se peut qu'une des instructions en queue d'infoduc ait besoin du résultat d'une instruction placée devant qui n'a pas terminé son cycle ! Ou encore qu'une instruction de rupture de séquence vienne perturber la donne.

Ce dispositif doit donc se compléter d'un test des codes opérations placées dans l'infoduc afin de déterminer si ces instructions peuvent être exécutées en parallèle avec le même résultat que si elles avaient été exécutées en séquence.

Ce test peut se dérouler en parallèle avec les phases décrites précédemment et n'introduit donc pas de délai supplémentaire. Enfin, si la suite des instructions provient de la compilation d'un programme écrit dans un langage de haut niveau, le compilateur peut tenir compte de l'architecture sous jacente pour adapter le code généré à la chaîne de calcul.

5.1.5 Architectures multicœurs

Il s'agit de processeurs possédant plusieurs UAL de même type pouvant fonctionner en parallèle.

Tout comme pour les chaînes de calcul, des contraintes s'imposent pour que les différentes UAL puissent réellement exécuter des instructions en parallèle.

5.1.6 Architectures parallèles

Les chaînes de calcul peuvent en principe exécuter des suites classiques d'instructions, en tenant compte toutefois de la compatibilité nécessaire au parallélisme.

Les architectures parallèles sont en revanche conçues pour des programmes bien spécifiques, dans lesquels l'exécution d'instructions en parallèle est en quelque sorte inhérente à l'algorithme. Ainsi par exemple le calcul d'un produit scalaire $S = u_1.v_1 + u_2.v_2 + \dots u_n.v_n$ peut parfaitement se faire en seulement 2 étapes :

— calcul en parallèle des n produits $u_i.v_i$

— calcul de la somme des n résultats

et, en y réfléchissant un peu, la seconde étape peut aussi être fortement parallélisée !

De cette idée naissent les architectures suivantes :

SIMD : Single Instruction, Multiple Data (Figure 20)

Une seule instruction est exécutée en parallèle sur un ensemble de données. C'est le cas du produit dans l'exemple du produit scalaire. C'est l'architecture parallèle la plus fréquente, travaillant en général sur des calculs vectoriels.

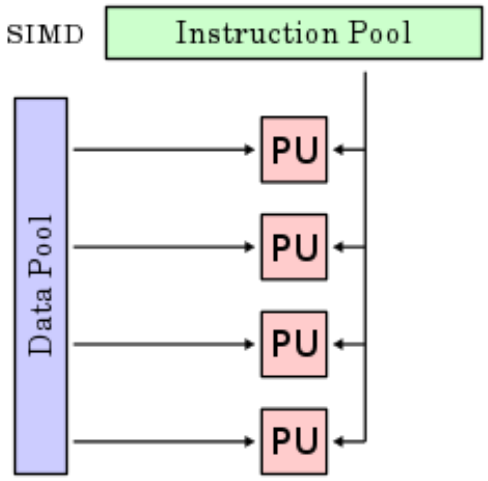


FIGURE 20: L'architecture SIMD

MISD : Multiple Instructions, Single Data (Figure 21)

Plusieurs instructions sont exécutées en parallèle sur une seule donnée C'est la plus rare des architectures parallèles dont l'application la plus courante est la sécurité des calculs dans des environnements fortement bruités (navette spatiale par exemple)

MIMD : Multiple Instructions, Multiple Data (Figure 22)

Le top dans le genre ! Plusieurs instructions sont exécutées en parallèle sur un ensemble de données. En fait, cela revient un peu à dire que plusieurs ordinateurs classiques travaillent en parallèle. Lorsque c'est nécessaire, une synchronisation doit s'effectuer entre les différentes machines afin d'assurer la cohérence globale du programme.

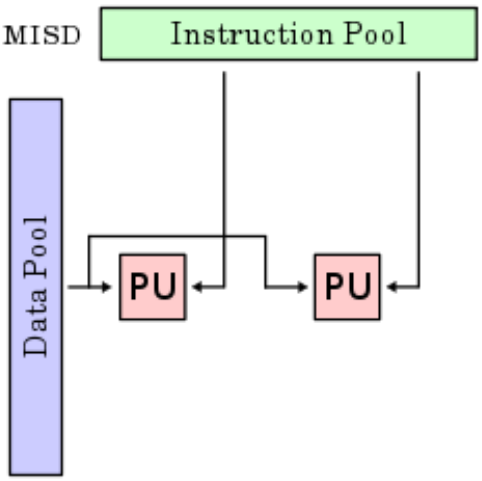


FIGURE 21: L'architecture MISD

Multiprocesseurs Ces architectures sont réservées aux super-ordinateurs chargés d'exécuter des programmes de calcul scientifique très gourmands en puissance et dont les algorithmes sont conçus ad hoc.

5.2 Les Mémoires

On sait que les mémoires des ordinateurs ont pour rôle de stocker données et programmes, aucune distinction de forme n'existant dans ces deux types d'informations. Il existe plusieurs taxinomies possibles pour les décrire :

- selon leurs techniques de fabrication
- selon leurs vitesses d'accès
- selon leurs capacités de stockage
- selon la durée de vie des informations stockées
- selon leur adressage
- selon leurs possibilités d'écriture ou de réécriture
- selon leur fonction dans l'architecture

En général, pour un coût donné, plus une mémoire est rapide, plus sa capacité est faible.

Ainsi il ne semble pas a priori possible de pouvoir bénéficier à la fois d'une grande rapidité d'accès et d'une grande capacité de stockage.

C'est dans le but de concilier ces deux aspects qu'a été conçu la mémoire virtuelle dont le mécanisme sera décrit en fin de cette partie.

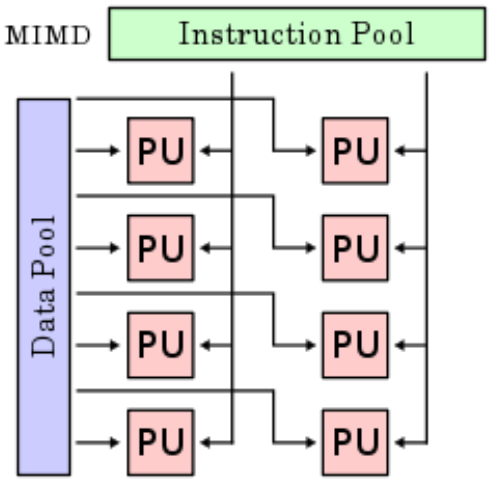


FIGURE 22: L'architecture MIMD

5.2.1 L'antémémoire

Appelée aussi mémoire cache, c'est un intermédiaire entre les registres de l'unité de commande et la mémoire centrale (décrite au paragraphe précédent).

Le temps de conservation des informations est extrêmement court (déroulement d'une suite d'instructions) et elle nécessite une source de tension.

Son rôle est d'accélérer les transferts. Elle est généralement située dans le composant « processeur ».

Elle contient la suite des prochaines instructions à exécuter.

Sa taille est de l'ordre du million d'octets (1Mio).

Il existe souvent plusieurs niveaux d'antémémoires, allant du plus proche de l'unité de contrôle au plus éloigné vers la mémoire centrale.

Elle est également d'une grande utilité dans les architectures parallèles.

5.2.2 La mémoire centrale

Appelée aussi mémoire vive ou mémoire à accès direct ou encore RAM (Random Access Memory), c'est la mémoire historique de l'ordinateur. D'abord mécanique (machine de Babbage) puis électromécanique (à base de relais), ferromagnétique (à tores de ferrite) et enfin électronique (transistors sur circuits intégrés).

Les informations y sont conservées le temps de déroulement d'un

programme et elle nécessite aussi une source de tension. On la trouve dans les micro-ordinateurs sous forme de barrettes de capacités variables (autour de 1 Gio actuellement).

Son rôle est de stocker les données et les programmes en cours d'exécution. Le temps de transfert des informations entre elle et le processeur doit être du même ordre de grandeur que le temps d'exécution d'une instruction. L'antémémoire ne sert qu'à diminuer le temps d'accès (temps mort entre instructions).

5.2.3 La mémoire morte

Au contraire de la mémoire vite, la mémoire morte permet de conserver les informations qui y sont stockées sans alimentation électrique.

Son rôle est multiple :

— stockage du programme de démarrage de l'ordinateur

— stockage d'informations concernant la configuration matérielle

— stockage de programmes permanents (systèmes embarqués)

— tables de constantes

— etc ...

Appelée aussi ROM (Read Only Memory) elle se décline selon différentes techniques de réalisation

PROM (Programmable Read Only Memory) Elle ne peut être programmée qu'une seule fois. Chaque case mémoire est constituée d'un fusible grillé lors de la programmation, par l'application d'une tension adéquate. Pour ce faire, on utilise une mémoire initiale ne contenant que des « 1 » réalisés par des diodes laissant passer un courant électrique entre 2 conducteurs ; puis on « claque » (détruit) certaines de ces diodes coupant le passage du courant et réalisant ainsi un « 0 » .

EPROM (Erasable Programmable Read Only Memory) La programmation se fait en stockant des charges d'électrons au sein d'un transistor. Cette opération nécessite l'utilisation de tensions élevées ($\approx$ 25 V) non disponibles au sein de l'ordinateur, en utilisant un programmateur ad hoc. Les charges restent stockées pendant l'utilisation de la mémoire et celle-ci peut être lue sur un ordinateur en utilisant des tensions classiques ($\approx$ 5 V). Elle peut être effacée en totalité (jamais en partie) en exposant la puce (retirée de l'ordinateur)

à des rayonnements UV libérant les électrons piégés. Elle peut donc être réutilisée.

EEPROM (Electrically Erasable Programmable Read Only Memory) La seule différence avec la précédente réside dans sa possibilité d'effacement par courant électrique donc sans démontage de l'ordinateur. C'est le seul type de ROM qui peut être mis à jour par logiciel

5.2.4 Les mémoires de masse

Les mémoires de masse sont capables de stocker de grandes quantités d'informations pour une durée quasi illimitée. Elles ne nécessitent pas de source de tension pour leur conservation.

Leur capacité varie généralement de quelques Go jusqu'à plus de 1 To actuellement.

Les vitesses et temps d'accès restent nettement inférieures aux mémoires vives ou mortes.

Enfin elles peuvent être situées à l'intérieur du boîtier de l'ordinateur ou s'y connecter de l'extérieur via des interfaces spécifiques (USB par exemple).

Elles ont pour rôle de stocker les logiciels utilisés, notamment les systèmes d'exploitation, et les données pouvant avoir une longue durée de vie, en tous cas nettement supérieure à celle de l'exécution d'un programme.

Dans le cas de supports amovibles elles sont aussi utilisées pour le partage d'informations.

Historiquement, ces mémoires ont été réalisées sur des supports en carton (cartes perforées), en papier (ruban perforé), puis magnétiques (bandes, tambours, disquettes).

Actuellement dans le domaine des micro-ordinateurs, elles se déclinent en :

— Supports magnétiques : Disques durs

— Supports optiques : Disques compacts (CD, DVD)

— Supports électroniques : Clefs USB, Mémoires Flash

Bien que les techniques de réalisation soient différentes, l'organisation des informations se fait toujours selon les mêmes principes.

5.2.5 Organisation des informations

Vu de l'utilisateur, l'unité d'enregistrement est le fichier, considéré généralement comme une suite d'octets. Vu du côté du support, l'unité

est le secteur (également suite d'octets).

L'organisation consiste donc à stocker des fichiers sur des secteurs, sachant qu'il doit être possible de créer, allonger, supprimer, effacer et évidemment lire et écrire ces fichiers.

Anatomie d'un secteur (Figure 23)

FIGURE 23: Anatomie d'un secteur

GAP	Espace entre secteurs
SYN	Synchronisation de lecture
ADR	Adresse du Secteur
DATA	Les Données
CTR	Contrôle (détection, correction d'erreurs)

Placement des fichiers sur les secteurs Au fur et à mesure des créations, allongements et suppressions de fichiers, l'espace utilisable, puis l'espace utilisé deviennent de plus en plus morcelés. En conséquence, un fichier n'est pas nécessairement stocké sur des secteurs continus.

Afin de gérer cette situation, quelques secteurs sont réservés pour y loger une table d'allocation des fichiers. Chaque fichier se voit attribuer un certain nombre de secteurs selon sa taille. Pour ne pas avoir des fragments trop petits, qui verraient s'allonger cette table, les secteurs consécutifs sont constitués en une suite appelée un cluster.

Table d'allocation

Un exemple (Figure 24)
Ici la table comporte 2 fichiers et un espace libre. Elle gère un espace de 8 clusters.
Le fichier PAUL n'est constitué que d'un seul cluster d'adresse 010
Le fichier VIRGINIE comprend 2 clusters non consécutifs d'adresses 011 et 101
Enfin l'espace libre possède 5 clusters

Bien entendu il ne s'agit là que d'un principe. Les tables d'allocation de fichiers sont bien plus complexes et incluent de nombreux

Nom du fichier	Nombre de clusters	Adresses des clusters
PAUL	1	010
VIRGINE	2	011
		101
LIBRE	5	000
		001
		100
		110
		111

FIGURE 24: Table d'allocation

attributs de fichiers (dates de création, de modification, droits d'accès, possibilité ou non de migration, etc...)

De même la réalisation de cette table peut être plus raffinée (utilisation de listes chaînées pour les clusters, etc ...)

A noter que la suppression d'un fichier est une opération qui retire cet élément de la table, chaîne les clusters qu'il utilisait avec les clusters libres, mais n'efface pas son contenu !

5.2.6 La mémoire virtuelle

Dans certaines situations (programmes traitant d'importantes quantités de données, multi-utilisateurs, multi-applications), il serait intéressant de disposer de mémoires de grande capacité (comme les mémoires de masse) et d'accès rapide (comme les mémoires vives). Mais c'est évidemment une utopie !

C'est pour donner vie à cette utopie qu'a été créée la mémoire virtuelle.

Principe

Dans une architecture simple, l'adresse d'une instruction est considérée comme un indice dans un grand tableau constitué de la suite de toutes les cellules de la mémoire vive.

Dans une architecture avec mémoire virtuelle l'adresse d'une instruction est composée de 2 champs considérés comme les indices d'un tableau à 2 dimensions.

Deux réalisations sont possibles :

La mémoire paginée (Figure 25)

Le tableau est rectangulaire, c'est à dire que toutes les colonnes possèdent le même nombre de lignes. Le premier indice désigne un numéro de page (colonne), le second indice un déplacement dans cette page (ligne).

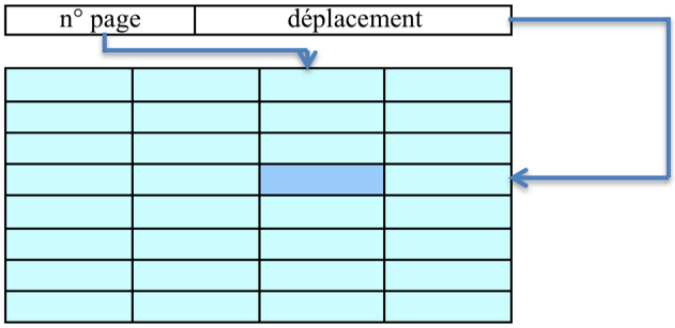


FIGURE 25: la pagination

La mémoire segmentée (Figure 26)

Les colonnes du tableau n'ont pas toutes le même nombre de lignes.
Le premier indice désigne un numéro de segment (colonne), le second indice un déplacement dans ce segment (ligne).

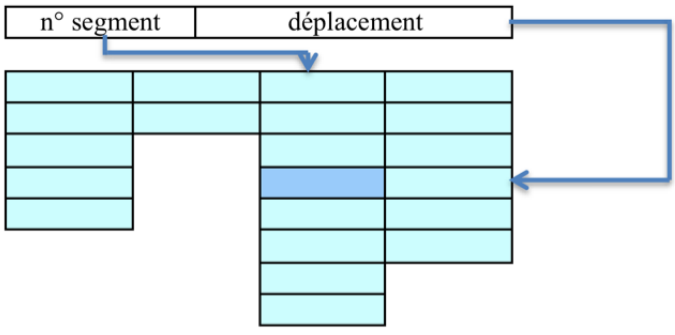


FIGURE 26: la segmentation

Mais tout ceci n'est qu'une vue de l'esprit. Cette mémoire est virtuelle !

Sa taille (nombre total de cellules) peut être bien supérieure à celle de la mémoire vive, mais jamais supérieure à celle de la mémoire de masse.

La relation entre la mémoire virtuelle (qui correspond à l'espace d'adressage des instructions) d'une part, et la mémoire vive et une mémoire de masse d'autre part se fait par l'intermédiaire d'une table (des pages ou des segments).

Chaque entrée de cette table est indicée par le numéro de page (ou de segment).

Une information indique si la page (ou le segment) est ou non présent en mémoire vive.

S'il l'est, la table contient l'adresse en mémoire vive de la première cellule de la page (ou du segment).

Dans tous les cas l'adresse en mémoire de masse d'un fichier contenant une image de la page (ou du segment) figure dans la table. Enfin, dans le cas d'une mémoire virtuelle segmentée, la longueur du segment est présente.

Lorsque l'unité de commande est dans la phase de recherche de l'adresse d'un opérande, la table des pages (ou des segments) est consultée et :

- Si la page (ou le segment) est en mémoire vive, le numéro de page (ou de segment) est remplacé par l'adresse en mémoire vive de la première cellule de la page (ou du segment) et le déplacement est ajouté au résultat obtenu.
- Si la page (ou le segment) n'est pas en mémoire vive, une exception appelée « défaut de mémoire » est déclenchée et ce sera le travail du système d'exploitation de mettre en œuvre une suite d'actions visant à :
  - trouver une place libre en mémoire vive pour loger la page (ou le segment)
  - s'il n'en existe pas, transférer éventuellement une page (ou un segment) de la mémoire vive vers la mémoire de masse et libérer ainsi un espace suffisant.
  - transférer la page (ou le segment) vers l'emplacement libre ou libéré

Bien entendu, au démarrage d'un programme, son code et ses données figurent en mémoire de masse. Le processus peut alors commencer.

6 Conclusion

Si en apparence on ne voit rien de commun entre un microprocesseur qui enchaîne les cycles d'une machine à laver, un micro-ordinateur de bureau et un super-ordinateur chargé de prévoir la météo pour la semaine à venir, en fait la structure de base est identique.

C'est cette structure qu'on a voulu décrire dans ce chapitre, le parti pris étant d'en comprendre les principes et non de décrire avec précision les composants de tel ou tel ordinateur familier.

ANNEXES

7 L'arithmétique et la logique d'ORFEU

On appelle *vecteur*[81] la séquence de n[82] bits contenue dans une cellule de mémoire.

$$\boxed{\mathbf{X} = < x_{n-1}, x_{n-2}, ..., x_1, x_0 >} \quad x_i \in \{0, 1\}$$

7.1 L'arithmétique

7.1.1 Division euclidienne, modulo et congruence

Division euclidienne

Initialement, la division euclidienne est définie ainsi :

$$(a, b) \mapsto (q, r) : a = b \cdot q + r \; et \; 0 \leq r < b$$

avec :

$$a, b, q, r \in \mathbb{N} \; et \; b > 0$$

a est le dividende
b est le diviseur
q est le quotient noté $q = a \div b$
r est le reste noté $r = a \bmod b$

Il n'est pas difficile d'étendre cette définition aux nombres relatifs :

$$a, q \in \mathbb{Z} \; ; \; b, r \in \mathbb{N} \; et \; b > 0$$

Dans ce cas, on obtient par exemple :

$$17 \div 4 = 4$$

$$17 \bmod 4 = 1$$

$$-17 \div 4 = -5$$

$$-17 \bmod 4 = 3$$

81. Un vecteur sera désigné par une lettre majuscule romaine en caractères gras.
82. Pour ORFEU $n = 16$.

Modulo

Il s'agit tout simplement du reste de la division euclidienne :

$$a \bmod b = a - ((a \div b) \cdot b)$$

Congruence

Il s'agit d'une relation d'équivalence définie par :

$$x \equiv y \mod b \ : \ x \bmod b = y \bmod b$$

7.1.2 Représentation des nombres

Les nombres sont représentés selon le complément à 2.

Les opérations arithmétiques se font modulo 2^n, n étant la taille (nombre de bits) des cellules de mémoire.

Les opérations arithmétiques se font modulo la taille n des cellules de mémoire.

Au vecteur de n bits

$$\mathbf{X} = < x_{n-1}, x_{n-2}, ..., x_1, x_0 >$$

on associe le nombre $X = \nu(\mathbf{X})$ défini par

pour $x_{n-1} = 0 : 0 \leq X < 2^{n-1}$

$$X = \sum_{i=0}^{n-1} x_i 2^i$$

pour $x_{n-1} = 1 : -2^{n-1} \leq X < 0$

$$X = -2^n + \sum_{i=0}^{n-1} x_i 2^i$$

soit pour $-2^{n-1} \leq X < 2^{n-1}$

$$\boxed{X = -x_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i}$$

Ainsi, chaque nombre est interprété comme un entier de l'intervalle

$$[-2^{n-1} : 2^{n-1}[$$

En conséquence :

— si $x_0 = 0$ X est impair ; si $x_0 = 1$ X est impair

— si $x_{n-1} = 0$ X est positif ; si $x_{n-1} = 1$ X est négatif ce qui fait parfois dire que x_{n-1} est le *bit de signe*

Réciproquement, on associe à tout nombre

$$X \in [-2^{n-1} : 2^{n-1}[$$

le vecteur de n bits $\mu(X)$

$$\mathbf{X} = < x_{n-1}, x_{n-2}, ..., x_1, x_0 >$$

défini par la décomposition (unique) de X en puissances de 2 selon le schéma décrit ci-dessus.

Remarque

Au vecteur de n bits, pour $m < n$

$$\mathbf{X} = < x_m, x_m, ..., x_m, x_{m-1}, ..., x_1, x_0 >$$

est associé le nombre $X = \nu(\mathbf{X}) \in [-2^m : 2^m[$

$$X = -x_m 2^m + \sum_{i=0}^{m-1} x_i 2^i$$

car

$$-2^{n-1} + \sum_{i=m}^{n-2} 2^i = -2^m$$

7.1.3 Addition : instruction ADD

L'expression

$$Z = \text{ADD } (X, Y)$$

est définie par :

$$\boxed{Z \equiv (X + Y) \mod 2^n \text{ et } Z \in [-2^{n-1} : 2^{n-1}[}$$

Or

$$(X + Y) \in [-2^n : 2^n - 1[$$

si $X + Y \in [-2^{n-1} : 2^{n-1}[$ alors

$$Z = X + Y$$

si $X + Y \in [2^{n-1} : 2^n - 1[$ alors

$$Z = X + Y - 2^n$$

si $X + Y \in [-2^n : -2^{n-1}[$ alors

$$Z = X + Y + 2^n$$

Dans ces deux derniers cas on dit qu'il y a *dépassement de capacité*[83].

Cette situation se produit quand Z est de signe contraire à $X + Y$ donc

$$x_{n-1} = y_{n-1} \neq z_{n-1}$$

7.1.4 Soustraction : instruction SUB

L'expression

$$Z = \text{SUB } (X, Y)$$

est définie par :

$$Z = \text{ADD } (X, -Y)$$

où $-Y$ est l'*opposé*[84] de Y

Calcul d'un opposé

En posant

$$X = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

et

$$X' = -(1 - x_{n-1})2^{n-1} + \sum_{i=0}^{n-2}(1 - x_i)2^i + 1$$

alors

$$X + X' = -2^{n-1} + \sum_{i=0}^{n-2} 2^i + 1 = -2^{n-1} + 2^{n-1} = 0$$

Donc

$$\boxed{-X = -(1 - x_{n-1})2^{n-1} + \sum_{i=0}^{n-2}(1 - x_i)2^i + 1}$$

Sauf pour

$$X = -2^{n-1}$$

car le calcul de X' conduit à

$$X' = -2^{n-1}$$

ce qui provoque un dépassement de capacité.

83. Cette situation n'est pas signalée dans ORFEU.
84. L'opposé d'un nombre est son inverse pour l'addition.

7.2 La logique

7.2.1 Opérations booléennes élémentaires

Les booléens sont représentés par des éléments binaires (bits) :

— la valeur *FAUX* est représentée par le bit 0

— la valeur *VRAI* est représentée par le bit 1

Les opérations ETL, OUL et OUX sont définies comme extensions à la cellule entière des opérations logiques élémentaires $\wedge$, $\vee$ et $\oplus$ portant sur un bit (cf. table 6).

a	b	a$\wedge$b	a$\vee$b	a$\oplus$b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

TABLE 6: Opérations booléennes élémentaires

Ces trois opérations forment un ensemble *fonctionnellement complet* c'est à dire que toute fonction booléenne peut s'exprimer à l'aide de ces opérations.

En effet, la négation $\neg a$ peut s'écrire par l'expression $a \oplus 1$ et il est établi que l'ensemble des opérations $\{\wedge, \neg\}$ ou $\{\vee, \neg\}$ est fonctionnellement complet.

7.2.2 Opérations logiques

Elles s'appliquent directement sur les vecteurs.

L'expression

$$\mathbf{Z} = \text{ETL } (\mathbf{X}, \mathbf{Y})$$

est définie par :

$$\mathbf{X} = < x_{n-1}, x_{n-2}, ..., x_1, x_0 >$$

$$\mathbf{Y} = < y_{n-1}, y_{n-2}, ..., y_1, y_0 >$$

$$\mathbf{Z} = < x_{n-1} \wedge y_{n-1}, x_{n-2} \wedge y_{n-2}, ..., x_1 \wedge y_1, x_0 \wedge y_0 >$$

Il en va de même pour les expressions

$$\mathbf{Z} = \text{OUL } (\mathbf{X}, \mathbf{Y})$$

$$\mathbf{Z} = \text{OUX } (\mathbf{X}, \mathbf{Y})$$

7.3 Arithmétique et Logique

7.3.1 Inverses arithmétiques et logiques

Inverse logique

On définit le *complémentaire* d'un booléen [85] par

$$\neg a = a \oplus 1$$

Par extension, l'inverse d'un vecteur $\mathbf{X}$ est défini par

$$\neg \mathbf{X} = \mathbf{X} \oplus \mathbf{1}$$

où

$$\mathbf{1} = < 1, 1, ..., 1, 1 >$$

Inverse arithmétique

Soit

$$\mathbf{X} = < x_{n-1}, x_{n-2}, ..., x_1, x_0 >$$

$X = \nu(\mathbf{X})$ le nombre X associé à $\mathbf{X}$

$$X = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

$\mathbf{Y} = \neg \mathbf{X}$ l'inverse logique de $\mathbf{X}$

$$\mathbf{Y} = < \neg x_{n-1}, \neg x_{n-2}, ..., \neg x_1, \neg x_0 >$$

$Y = \nu(\mathbf{Y})$ le nombre Y associé à $\mathbf{Y}$

$$Y = -(1 - x_{n-1})2^{n-1} + \sum_{i=0}^{n-2} (1 - x_i)2^i$$

alors pour $X \in]-2^{n-1} : 2^{n-1}[$

$$-X = Y + 1$$

Conclusion

On vérifie donc que l'inverse *arithmétique* d'un nombre X défini par le vecteur $\mathbf{X}$ est obtenu en ajoutant 1 au nombre défini par le vecteur inverse *logique* du vecteur $\mathbf{X}$.

$$\boxed{-X = \nu(\neg \mathbf{X}) + 1}$$

85. appelé aussi inverse

7.3.2 Décalages arithmétiques : instruction DAR

Les décalages sont des opérations *logiques* interprétées comme des opérations *arithmétiques*.

On note $\mathbf{X} \rightleftharpoons d$ le décalage du vecteur $\mathbf{X}$ de

— d bits à *gauche* si $d > 0$

— $|d|$ bits à *droite* si $d < 0$

Bien évidemment aucun décalage n'a lieu si $d = 0$

Décalage à gauche

Si la valeur d de l'opérande est positive, des bits 0 sont insérés aux positions x_0 à x_{d-1}, les bits de positions x_{n-1} à x_{n-d} sont perdus et les d autres bits sont décalés *à gauche* de d positions.

Pour un décalage à gauche de d positions, le vecteur

$$\mathbf{X_0} = < x_{n-1}, x_{n-2}, ..., x_1, x_0 >$$

devient

$$\mathbf{X_d} = \mathbf{X} \rightleftharpoons d = < x_{n-1-d}, x_{n-2-d}, ..., x_1, x_0, 0, ...0 >$$

et représente l'entier

$$X_d = -x_{n-1-d}2^{n-1} + \sum_{i=0}^{n-2-d} x_i 2^{i+d}$$

Il s'ensuit que

$$X_d = X_0 2^d + x_{n-1}2^{n-1+d} - \sum_{i=n-d}^{n-2} x_i 2^{i+d} - x_{n-1-d}2^n$$

donc

$$X_d \equiv X_0 2^d \mod 2^n$$

Et, dans le cas où $X_0 \in [-2^{n-1-d} : 2^{n-1-d}[$

$$X_d = X_0 2^d$$

Décalage à droite

Si la valeur d de l'opérande est négative, la valeur du bit x_{n-1} est recopié dans les bits de positions x_{n-1} à $x_{n-|d|}$, les bits de positions $x_{|d|-1}$ à x_0 sont perdus et les d autres bits sont décalés *à droite* de $|d|$ positions.

Pour un décalage à droite de $|d|$ positions, le vecteur

$$\mathbf{X}_0 = < x_{n-1}, x_{n-2}, ..., x_1, x_0 >$$

devient

$$\mathbf{X_d} = \mathbf{X} \rightleftharpoons d = < x_{n-1}, x_{n-1}, ..., x_{|d|+1}, x_{|d|} >$$

et représente l'entier

$$X_d = -x_{n-1}2^{n-1} + \sum_{i=n-1+d}^{n-2} x_{n-1}2^i + \sum_{i=0}^{n-2+d} x_{i+|d|}2^i$$

$$X_d = -x_{n-1}2^{n-1+d} + \sum_{i=|d|}^{n-2} x_i 2^{i+d}$$

Il s'ensuit que

$$X_d 2^{|d|} = -x_{n-1}2^{n-1} + \sum_{i=|d|}^{n-2} x_i 2^i$$

$$X_d 2^{|d|} = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i - \sum_{i=0}^{|d|-1} x_i 2^i$$

$$X_d 2^{|d|} = X_0 - \sum_{i=0}^{|d|-1} x_i 2^i$$

donc

$$X_0 = X_d 2^{|d|} + \sum_{i=0}^{|d|-1} x_i 2^i$$

or

$$0 \leq \sum_{i=0}^{|d|-1} x_i 2^i < 2^{|d|}$$

X_d est donc le quotient de la division euclidienne du nombre entier relatif X_0 par $2^{|d|}$

$$X_d = X_0 \div 2^{|d|} = X_0 2^d$$

Conclusion

On vérifie bien que l'opération de décalage arithmétique de $\mathbf{X}$ avec d comme valeur de l'opérande donne comme résultat

$$\boxed{\nu(\mathbf{X} \rightleftharpoons d) \equiv X \cdot 2^d \mod 2^n}$$

Et, dans le cas où $d < 0$ ou ($d > 0$ et $X_0 \in [-2^{n-1-d} : 2^{n-1-d}[$)

$$\nu(\mathbf{X} \rightleftharpoons d) = X \cdot 2^d$$

8 Exemples de programmes

8.1 Expression arithmétique

```
# Expression arithmétique simple

            SIN     DEBUT
A                   27
B                   42
C                   30
D                   0
DEBUT       CHA     A
            ADD     B
            SUB     C
            STO     D       # [D] ← [A] + [B] - [C]
            NOP     0
```

8.2 Instruction conditionnelle

```
# Cumul de deux entiers sous condition d'imparité

            SIN     DEBUT
UN                  1
X                   3
Y                   1024
Z                   128
DEBUT       CHA     X
            ETL     UN
            SSZ     FIN     # si [X] pair sauter à FIN
            CHA     Z
```

```
        ADD     Y
        STO     Z       # [Z] ← [Z] + [Y]
FIN     NOP     0
```

8.3 Instruction alternative

```
# Maximum de deux entiers

        SIN     DEBUT
A               27
B               42
MAX             0
DEBUT   CHA     A
        SUB     B
        SSN     BMAX
        CHA     A
        SIN     MAXI
BMAX    CHA     B
MAXI    STO     MAX     # [MAX] ← max ([A], [B])
        NOP     0
```

8.4 Itération

```
# Somme des 100 premiers entiers

        SIN     DEBUT
N               100
UN              1
I               0
SOMME           0
DEBUT   CHA     I       # début d'itération
        SUB     N
        SSZ     FIN     # si [I] = [N] sortie d'itération
        CHA     I
        ADD     UN
        STO     I
        CHA     SOMME
        ADD     I
        STO     SOMME   # [SOMME] ← [SOMME] + [I]
        SIN     DEBUT   # fin d'itération
FIN     NOP     0
```

8.5 Algorithme d'Ahmès

```
# Produit de deux entiers naturels

          SIN       DEBUT
X                   50
Y                   64
Z                   0
UN                  1
MUN                 -1
DEBUT     CHA       X
          SSZ       FIN       # Tant-Que [X] !=0
          ETL       UN
          SSZ       SUITE     # si [X] pair sauter à SUITE
          CHA       Z
          ADD       Y
          STO       Z         # [Z] <- [Z] + [Y]
SUITE     CHA       X
          DAR       MUN
          STO       X         # [X] <- [X] / 2
          CHA       Y
          DAR       UN
          STO       Y         # [Y] <- [Y] * 2
          SIN       DEBUT     # fin du Tant-Que
FIN       NOP       0
```

8.6 Adressage indexé

```
# Maximum d'un tableau d'entiers

          SIN       DEBUT
I                   1
UN                  1
N                   8
MAX                 0
T                   1         # [T+0]
                    -3
                    5
                    -7
                    0
                    -2
                    4
```

```
                    −6          # [T+7]
DEBUT       CHA     T
            STO     MAX         # [MAX] ← [T]
            CHA     I
ITER        SUB     N           # début d'itération
            SSZ     FIN         # si [I] = [N] sortie d'itération
            CHA     DEBUT
            ADD     I
            STO     INS         # calcul d'instruction
INS         NOP     0           # ACC ← [T+I]
            SUB     MAX
            SSN     SUITE
            ADD     MAX
            STO     MAX         # [MAX] ← max([MAX],[T+I])
SUITE       CHA     I
            ADD     UN
            STO     I
            SIN     ITER        # fin d'itération
FIN         NOP     0
```

8.7 Appel et retour de sous programme

```
# Sous programme de calcul du maximum de deux nombres

            SIN     DEBUT
A                   27          # paramètres effectifs
B                   42
MAX                 0
DEBUT       CHA     A           # transmission
            STO     X           # des paramètres
            CHA     B
            STO     Y
            CHA     IR          # transmission de l'
            STO     FSP         # instruction de retour IR
            SIN     DSP         # appel du sous programme
IR          SIN     RET
RET         STO     MAX         # [MAX] ← max ([A], [B])
            NOP     0

# Sous programme

X                   0           # paramètres formels
```

```
Y                   0
DSP     CHA         X       # début de sous programme
        SUB         Y
        SSN         YMAX
        CHA         X
        SIN         FSP
YMAX    CHA         Y
FSP     NOP         0       # retour de la fonction
```

8.8 Adressage indirect et pointeurs

```
# Périmètre d'un rectangle en programmation objet

        SIN         DEBUT
UN                  1
PERI                0       # Périmètre
RECT    NOP         OBJET   # Pointeur
OBJET               24      # Hauteur
                    36      # Largeur
DEBUT   CHA         CHAH
        OUL         RECT
        STO         CHAH
CHAH    CHA         0       # ACC ← [[RECT]]
        STO         PERI
        CHA         CHAL
        OUL         RECT
        ADD         UN
        STO         CHAL
CHAL    CHA         0       # ACC ← [[RECT] + 1]
        ADD         PERI
        DAR         UN
        STO         PERI    # [PERI] ← 2 × [[RECT]] × [[RECT] + 1]
        NOP         0
```

8.9 Pile

```
# Gestion de pile

        SIN         DEBUT
UN                  1
A                   27
```

```
B                45
C                0
D                0

PTP     NOP      PILE     # Pointeur de pile (constitué de 4 cellules)
PILE             0        # cellule 1
                 0        # cellule 2
                 0        # cellule 3
                 0        # cellule 4

DEBUT   CHA      A
        STO      X
        CHA      IR1      # transmission de l'
        STO      FINEMP   # instruction de retour IR1
        SIN      EMPILER  # appel du sous programme EMPILER
IR1     SIN      RET1
RET1    CHA      B
        STO      X
        CHA      IR2      # transmission de l'
        STO      FINEMP   # instruction de retour IR2
        SIN      EMPILER  # appel du sous programme EMPILER
IR2     SIN      RET2
RET2    CHA      IR3      # transmission de l'
        STO      FINDEP   # instruction de retour IR3
        SIN      DEPILER  # appel du sous programme DEPILER
IR3     SIN      RET3
RET3    STO      C
        CHA      IR4      # transmission de l'
        STO      FINDEP   # instruction de retour IR4
        SIN      DEPILER  # appel du sous programme DEPILER
IR4     SIN      RET4
RET4    STO      D
        NOP      0

# empiler

X                0
EMPILER CHA      EMP      # calcul de l'instruction EMP
        OUL      PTP
        STO      EMP
        CHA      X
EMP     STO      0        # empilement de X
        CHA      PTP
```

```
            ADD     UN      # incrémentation du pointeur
            STO     PTP
FINEMP      NOP     0

# depiler

DEPILER     CHA     PTP     # decrémentation du pointeur
            SUB     UN
            STO     PTP
            OUL     DEP     # calcul de l'instruction DEP
            STO     DEP
DEP         CHA     0       # dépilement vers ACC
FINDEP      NOP     0
```

8.10 Sous programme récursif

```
# Calcul du carré d'un nombre en programmation récursive

            SIN     DEBUT
N                   5
CARRE               0
UN                  1
ICHA        CHA     0       # instruction CHA
ISTO        STO     0       # instruction STO
PTP         NOP     PILE    # pointeur de pile
PILE                0
                    0
                    0
                    0
                    0
                    0
                    0
                    0
                    0
                    0
                    0
                    0
                    0
                    0
                    0
DEBUT       CHA     PTP
            OUL     ISTO
            STO     EMPIR
```

```
        CHA     IR
EMPIR   STO     0       # empilement de [IR]
        CHA     PTP     # (instruction de retour)
        ADD     UN
        STO     PTP
        OUL     ISTO
        STO     EMPN
        CHA     N
EMPN    STO     0       # empilement de [N]
        CHA     PTP
        ADD     UN
        STO     PTP
        SIN     DSP     # appel du sous programme
IR      SIN     RET
RET     STO     CARRE
        NOP     0       # fin du programme

# sous-programme

I               0       # variables locales temporaires
S               0
DSP     CHA     PTP     # début de sous programme
        OUL     ICHA
        SUB     UN
        STO     CHI
CHI     CHA     0       # chargement de [I]
        SSZ     FINSP   # si [I] = 0 sauter à FINSP
        SUB     UN
        STO     I
        CHA     PTP
        OUL     ISTO
        STO     EMPIRSP
        CHA     IRSP
EMPIRSP STO     0       # empilement de [IRSP]
        CHA     PTP     # (instruction de retour)
        ADD     UN
        STO     PTP
        OUL     ISTO
        STO     EMPI
        CHA     I
EMPI    STO     0       # empilement de [I]
        CHA     PTP
        ADD     UN
```

```
        STO     PTP
        SIN     DSP     # appel récursif
IRSP    SIN     RETSP
RETSP   STO     S       # retour de sous programme
        CHA     PTP
        OUL     ICHA
        SUB     UN
        STO     RCHI
RCHI    CHA     0       # chargement de [I]
        DAR     UN
        SUB     UN
        ADD     S
FINSP   STO     S       # [S] ← [S]+ 2 × [I] - 1
        CHA     PTP
        SUB     UN
        SUB     UN
        STO     PTP
        OUL     ICHA
        STO     DEPIR
DEPIR   CHA     0       # dépilment de l'instruction
        STO     FSP     # de retour
        CHA     S
FSP     NOP     0       # fin de sous programme
```

9 La grammaire de LAMOR

9.1 La Forme de Backus-Naur (BNF)

Il s'agit d'une grammaire permettant de définir de façon formelle la syntaxe d'un langage de programmation [86].

Les éléments terminaux (ceux du langage LAMOR) figurent ici en lettres majuscules.

Les éléments non terminaux (unités syntaxiques du langage LAMOR) figurent entre chevrons.

Les règles de la grammaire sont définies ci-dessous.

Elles conduisent à une analyse descendante de LAMOR.

L'axiome est : <programme LAMOR>

Règles :

86. `https://fr.wikipedia.org/wiki/Forme_de_Backus-Naur`

<programme LAMOR> : := <ligne>*
<ligne> : := [# <commentaire> | <ligne de programme>] eol
<commentaire> : := <utf 8>*
<ligne de programme> : := [<étiquette>] <sép>+ <cellule> [<fin>]
<cellule> : := <entier relatif> | <code opération> <sép>+ <opérande>
<opérande> : := <identificateur> | <entier naturel>
<fin> : := <sép>+ <commentaire>
<entier relatif> : := [-] <entier naturel>
<entier naturel> : := <chiffre>+
<étiquette> : := <nom>
<identificateur> : := <nom>
<nom> : := <lettre> <lettre ou chiffre>*
<lettre ou chiffre> : := <lettre> | <chiffre>
<utf 8> : := un caractère utf 8 imprimable
<sép> : := espace | tabulation
<chiffre> : := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<lettre> : := A | B | C ... X | Y | Z
<code opération> : := NOP | CHA | STO | ADD | SUB | ETL | OUL | OUX | SIN | SSN | SSZ | DAR

Abréviations :

- [us] désigne un ensemble d'unités syntaxiques optionnelles ;
- us* désigne une unité syntaxique absente ou répétée indéfiniment ;
- us$^+$ désigne une unité syntaxique répétée indéfiniment présente au moins une fois.

Précisions :

- *eol* désigne le ou les caractères de fin de ligne ;
- *espace* et *tabulation* représentent respectivement les caractères de code décimal *utf 8* : 32 et 9 ;
- *un caractère utf 8* signifie n'importe quel caractère imprimable codé en *utf 8* ;
- un entier naturel x figurant comme opérande doit satisfaire : $0 \leq x < 4096$;
- un entier relatif x figurant comme cellule doit satisfaire : $-32768 \leq x < 32768$.

Remarques :

— les entiers sont exprimés en base 10 ;

— les espaces et tabulations sont des caractères significatifs utilisés comme séparateurs entre unités syntaxiques ;

— une étiquette doit commencer en début de ligne.

Lignes valides :

— Lignes non assemblées (non traduites par l'assembleur) :

  — les lignes vides ;

  — les lignes ou commençant par #.

— Lignes assemblées :

  — ce sont les lignes appelées <ligne de programme> ;

  — chacune d'elles est traduite dans une cellule d'Orfeu ;

  — elles revêtent exclusivement une des deux formes :

    — [<étiquette>] <sep>$^+$ <donnée > [<fin>]

    — [<étiquette>] <sep>$^+$ <code opération> <sep>$^+$ <opérande> [<fin>]

9.2 L'automate

LAMOR est un langage rationnel [87].

Il peut donc aussi être décrit par un Automate Fini [88] (cf. Figure 27 *L'automate* page 79).

Sur cette figure, l'état 0 (DEBUT/FIN) est l'état initial et le seul état final.

Il conduit à une analyse ascendante de LAMOR.

10 ORFEU et la calculabilité

On peut constater par les quelques exemples de l'annexe que l'architecture d'ORFEU permet de réaliser toutes les structures de contrôle et toutes les opérations arithmétiques des algorithmes itératifs et récursifs.

87. `https://fr.wikipedia.org/wiki/Langage_rationnel`
88. `https://fr.wikipedia.org/wiki/Automate_fini`

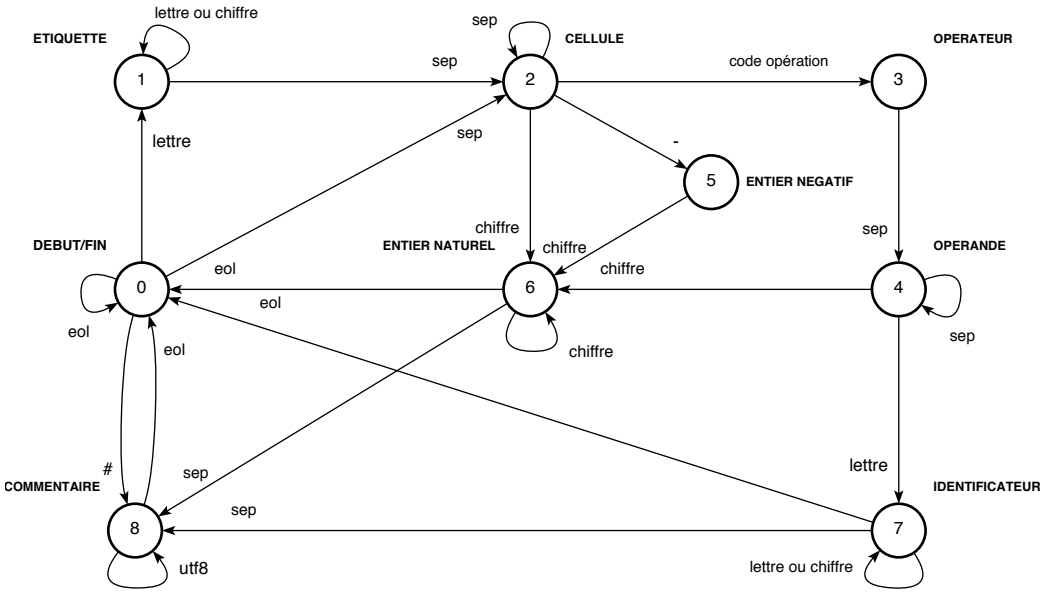


FIGURE 27: L'automate

10.1 Modèles de calcul

On rappelle qu'un modèle de calcul peut être décrit par une machine possédant certaines caractéristiques permettant de calculer toutes les fonctions calculables.

C'est le cas des machines dites *de Minsky* ou encore *machines à compteurs*. Nous extrayons de Wikipedia [89] :

> *« Dans sa version la plus simple une machine à compteurs est composée de deux compteurs (ou registres) et d'un programme. Chaque compteur est un entier naturel (non borné). Le programme est une suite d'instructions de la forme (C1 désigne le premier compteur et C2 le deuxième compteur) :*
>
> — *incrémente C1*

89. `https://fr.wikipedia.org/wiki/Machine_à_compteurs`.

— *décrémente C1*

— *incrémente C2*

— *décrémente C2*

— *si C1=0 alors saut vers l'instruction i1 sinon saut vers l'instruction i2*

— *si C2=0 alors saut vers l'instruction i1 sinon saut vers l'instruction i2*

où i1 et i2 sont des étiquettes (ou numéro de lignes) du programme. »

Or on peut aisément simuler une machine à compteurs par ORFEU, en imaginant une taille non bornée de la mémoire.

Moyennant cette *extension imaginaire* ORFEU est donc lui aussi un modèle de calcul.

10.2 Réduction du jeu d'instructions

ORFEU repose sur une architecture RISC et comprend donc un nombre assez réduit d'instructions.

Et pourtant, il est possible de le réduire encore plus. Cette réduction n'est toutefois pas souhaitable car elle compliquerait considérablement la programmation et nuirait à la lisibilité des programmes.

Elle est cependant intéressante au regard du concept de calculabilité dont tous les modèles se réclament d'une extrême simplicité des outils.

10.2.1 Instructions de contrôle de séquence

Les deux instructions : SIN X et SSN X peuvent être réalisées grâce à l'instruction SSZ X.

instruction SIN X

Elle est réalisable grâce à la séquence :

```
CHA     ZERO
SSZ     X
```

où la cellule d'adresse ZERO contient la valeur 0

instruction SSN X

Elle est réalisable grâce à la séquence :

```
ETL     MINI
```

```
OUX     M1
SSZ     X
```

où la cellule d'adresse MINI contient la valeur binaire 1000000000000000 (soit -32768 en décimal)

et la cellule d'adresse M1 contient la valeur binaire 1111111111111111 (soit -1 en décimal)

instruction NOP X

Cette instruction n'est pas indispensable à la programmation des algorithmes, à l'exception de NOP 0 qui signifie l'arrêt de la machine.

Elle peut être remplacée par l'instruction SIN 0 qui ne pourra alors n'être utilisée que pour cette fonction.

10.2.2 Instructions arithmétiques

Les instructions ADD X et SUB X peuvent être réalisées grâce à une seule instruction sans opérande INC qui réalise :

$$ACC \leftarrow ACC + 1 \mod 16$$

opposé d'un nombre

On sait que l'opposé[90] d'un nombre X peut être calculé par la séquence :

```
CHA     X
OUX     M1
INC
```

où la cellule d'adresse M1 contient la valeur binaire 1111111111111111 (soit -1 en décimal)

instruction ADD X

En s'inspirant de cet algorithme qui réalise S ← S + X mod 16 :

répéter X mod 16 **fois** :
    S ← S + 1 mod 16

L'instruction ADD X est réalisable grâce à la séquence :

```
STO     S       [S] ← ACC
CHA     X
```

90. c'est à dire son inverse pour l'addition

```
        OUX     M1
        INC
        STO     Z       [Z] ← -[X]
SUITE   CHA     Z
        SSZ     FIN     si [Z] = 0 aller à fin
        INC
        STO     Z       [Z] ← [Z] + 1
        CHA     S
        INC
        STO     S       [S] ← [S] + 1
        SIN     SUITE
FIN     CHA     S       ACC ← [S]
```

instruction SUB Y

On sait que soustraire un nombre est équivalent à y ajouter son opposé : X - Y = X + (-Y)

L'instruction SUB Y est réalisable grâce à la séquence :

```
        STO     S       [S] ← ACC
        CHA     Y
        OUX     M1
        INC
        STO     X       [X]← -[Y]
        CHA     S
        ADD     X
```

en replaçant,bien entendu, l'instruction ADD X comme indiqué audessus

10.2.3 Instructions logiques

Les instructions ETL, OUL et OUX forment, avec la constante VRAI, un ensemble fonctionnellement complet permettant donc d'exprimer toute fonction booléenne.

Mais cette complétude peut aussi être obtenue par une seule de ces deux fonctions suivantes :

— NON-ET (NAND) ou fonction de Sheffer généralement notée $\uparrow$

— NON-OU (NOR) ou fonction de Peirce généralement notée $\downarrow$

Ces fonctions sont définies par la table 12.

Les opérations ETL ($\wedge$), OUL($\vee$), OUX($\oplus$) s'écrivent alors :

— A $\wedge$ B :

a	b	a↑b	a↓b
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

TABLE 12: Opérations NON-ET et NON-OU

  - (A ↑ B) ↑ (A ↑ B)
  - (A ↓ A) ↓ (B ↓ B)
- A ∨ B :
  - (A ↑ A) ↑ (B ↑ B)
  - (A ↓ B) ↓ (A ↓ B)
- A ⊕ B :
  - ((A ↑ B) ↑ A) ↑ ((A ↑ B) ↑ B)
  - ((A ↓ A) ↓ (B ↓ B)) ↓ (A ↓ B)

10.2.4 Instruction de décalage arithmétique

DAR X réalise : $ACC \leftarrow ACC \cdot\ 2^{[D]} \bmod 2^{16}$
c'est à dire :

- pour [D] ≥ 0 : une multiplication par $2^{[D]} \bmod 2^{16}$
- pour [D] < 0 : une division euclidienne par $2^{-[D]}$

Ce qui peut se faire, en n'utilisant que les instructions réduites précédemment décrites, par un algorithme de ce type réalisant :

$$S \leftarrow S \cdot\ 2^{[D]} \bmod 2^{16}$$

répéter |D| **fois** :
 si D $>$ 0 :
  S $\leftarrow$ (S + S) mod2^{16}
 sinon :
  Q $\leftarrow$ 0
  R $\leftarrow$ S
  si S $\geq$ 0 :
   tant que R $\geq$ 2 :
    Q $\leftarrow$ Q + 1
    R $\leftarrow$ R - 2 $\{S = 2 \cdot Q + R \wedge R \geq 0\}$
  sinon :
   tant que R $<$ 0 :

Q ← Q - 1
R ← R + 2 $\quad \{S = 2 \cdot Q + R \wedge R < 2\}$
S ← Q

Pour D ≥ 0 le résultat est trivial : $S \leftarrow S \cdot 2^D \bmod 2^{16}$

Pour D < 0 les invariants des itérations (notés entre { et }), conjointement aux conditions de sortie, montrent que le résultat est bien le quotient d'une division euclidienne dans les entiers relatifs, soit : $S \leftarrow S \div 2^{-D}$

10.2.5 Conclusion

Le jeu d'instructions d'ORFEU pourrait donc *théoriquement* être réduit à seulement 5 :

— SSZ X
— CHA X
— STO X
— INC
— NON-ET ou NON-OU