Architecture

Concevoir des algorithmes qui transforment de l'information et les décrire dans des langages de programmation a comme objectif premier de permettre leur exécution par un ordinateur. Un ordinateur peut se décrire à plusieurs échelles. À l'échelle du nanomètre, c'est un assemblage de quelques millions de transistors, alternativement conducteurs et non conducteurs, dont l'évolution aboutit ultimement à l'apparition sur un écran de quelques pixels, auxquels nous accordons une signification. À une échelle un peu plus grande, c'est un processeur entouré de nombreuses cases mémoire, qui lit le contenu de certaines de ces cases et, en fonction du contenu lu, lit le contenu d'autres cases, le modifie en effectuant quelques opérations arithmétiques et logiques, avant de le réécrire en mémoire. À une échelle plus grande encore, c'est une machine capable d'exécuter des programmes écrits dans un langage fruste, le langage machine, mais vers lequel nous pouvons traduire tous les programmes écrits dans des langages évolués. Dans ce chapitre, un zoom arrière, du transistor à l'ordinateur, nous fera comprendre comment chacune de ces structures se construit à partir de structures de taille immédiatement inférieure.

Cours

Objectifs

L'ordinateur est fondé sur plusieurs idées simples : c'est une structure physique utilisant des circuits électroniques câblés une fois pour toutes, capable d'interpréter électriquement, on dit d'« exécuter », un programme constitué d'un ensemble d'instructions codées en binaire et traitées séquentiellement.

Ce jeu d'instructions binaires, qui constituent un *langage machine*, diffère d'une machine à l'autre. Mais tous ces langages reposent sur les mêmes principes et sont, en fait, très proches. C'est, à nouveau, les principes universels,

et non les détails propres à un langage ou un autre, qu'il importe de comprendre ici.

Un langage machine est conçu pour être facilement représentable et exécutable par un ordinateur. Mais les programmes exprimés dans ce langage, qui sont des suites de 0 et de 1, sont difficilement intelligibles pour les êtres humains. Aussi a-t-on introduit deux niveaux d'outils qui permettent d'exprimer les programmes dans des langages plus intelligibles : les langages d'assemblage et les langages de haut niveau.

Le langage d'assemblage reprend la structure du langage machine, mais il est plus facile à lire et à comprendre grâce à l'utilisation de mnémoniques, de symboles et d'étiquettes. Les programmes écrits dans ce langage sont traduits, quasiment ligne à ligne, en langage machine, par un outil appelé un *assembleur*.

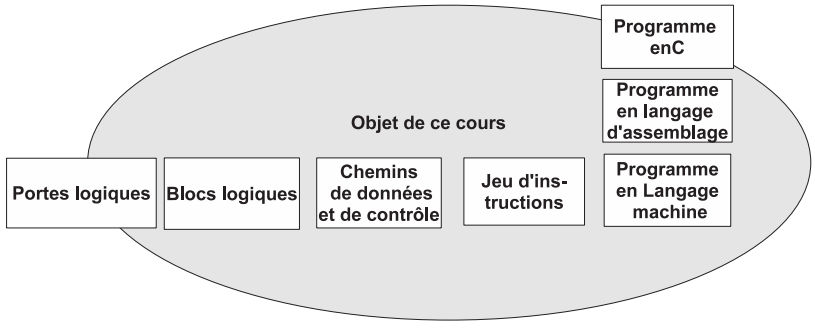
Cependant, un langage d'assemblage reste particulier à un type d'ordinateurs et nécessite donc, de la part du programmeur, une certaine spécialisation. On a donc introduit un second niveau d'outils : les langages de programmation de haut niveau, comme Java ou C, qui sont traduits par un compilateur en langage d'assemblage, puis en langage machine.

Les bases de ce chapitre se trouvent en électronique, en logique, en théorie des langages, en compilation et en algorithmique. Les compétences acquises seront, en retour, utiles en algorithmique, en programmation, en compilation, mais aussi en système et même en réseau.

Dans ce chapitre, nous verrons tout d'abord comment réaliser, avec des transistors, des circuits électroniques simples, en particulier des portes logiques. Nous utiliserons ensuite ces portes pour construire une architecture minimale, capable de traiter toutes sortes de programmes. Nous étudierons les instructions types et les principaux modes d'adressage de la mémoire utilisés dans les programmes écrits en langage machine. Nous terminerons par la description du fonctionnement d'un assembleur et par les rudiments de la traduction en langage d'assemblage de programmes écrits dans un langage de haut niveau. En conclusion, nous présenterons des extensions de ce modèle simplifié de machine, qui permettent de le rendre plus efficace et plus rapide.

Comme dans les autres chapitres, la méthode proposée ici laisse une large part à la création personnelle et à l'assimilation par des travaux pratiques dans lesquels les outils sont complètement ouverts et compréhensibles par tous : c'est pourquoi, pour comprendre ces notions, l'utilisation de simulateurs est préconisée à tous les niveaux. Les simulateurs permettent de réali-

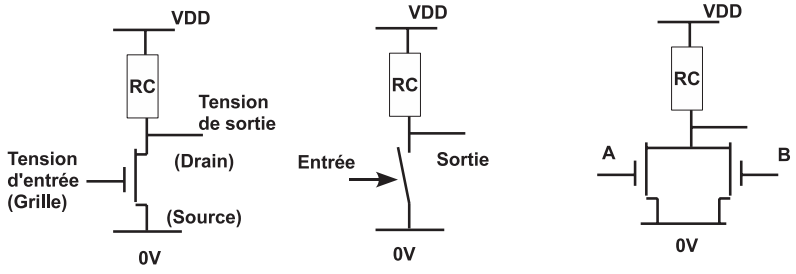
ser à tout moment des modifications personnelles et donc de s'approprier sa propre version de ces outils.



Portes logiques

Les transistors, qui constituent la plus grande partie des circuits d'un ordinateur, sont des dipôles à impédance contrôlée par une tension externe. En les simplifiant à l'extrême, on peut considérer que cette impédance entre les électrodes de drain et de source est soit nulle soit infinie, en fonction de la tension appliquée à l'électrode d'entrée, appelée grille. Ces transistors sont alors équivalents à un court-circuit ou à un circuit ouvert.

Cette vue d'un transistor NMOS, par exemple, est très simplifiée, mais elle permet cependant de comprendre le fonctionnement d'une porte logique. Le premier exemple est un inverseur : il comporte un transistor en série avec une résistance de charge fixe. L'électrode de sortie de ce montage est le drain VDD, qui est connecté à une résistance de charge, elle-même connectée à la tension d'alimentation du circuit. Si la tension de grille est trop faible pour que le transistor soit dans un état « passant », la tension de sortie se rapproche de la tension d'alimentation. Si, en revanche, la tension de grille dépasse un certain seuil, le transistor devient passant et court-circuite la sortie à la masse, ce qui conduit à une sortie de tension nulle. Si l'on définit un état logique 0 comme une tension proche de 0V et un état 1 comme une tension proche de la tension d'alimentation, ce circuit électronique est donc appelé inverseur, car si l'on considère que la tension d'entrée – de grille – est au niveau logique 0 – inférieure à la tension de seuil – la sortie est à un niveau logique 1, et vice versa.

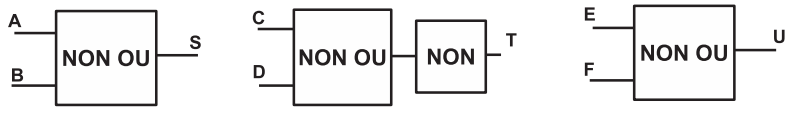


a) porte inverseuse, b) schéma simplifié de a,
c) porte NON OU à 2 entrées, C = NON (A OU B)

Le second circuit est encore plus intéressant, car il comporte deux transistors montés en parallèle et une seule résistance de charge. Pour que le circuit donne une sortie logique à 0, il suffit que l'un ou l'autre des signaux d'entrée A et B soit à un potentiel élevé – 1 logique. Autrement, la sortie sera un potentiel élevé – 1 logique. Ces portes logiques sont appelées NON OU ou NOR. On peut résumer cela à l'aide d'un tableau appelé table de vérité.

A	B	C = NON (A OU B)
0	0	1
0	1	0
1	0	0
1	1	0

Il est possible à partir de ces types de portes de réaliser des portes OU, en connectant la sortie d'une porte NON OU sur un inverseur – porte NON – et des portes ET, en connectant les entrées sur des inverseurs.



a) porte NON OU, b) Porte OU, c) porte ET

Cette partie, montrant le passage d'un circuit électronique à un circuit logique, relève plus d'un cours d'électronique que d'informatique. Cependant, ce survol pose quelques éléments d'évaluation de la complexité, au sens informatique, d'un circuit : la surface physique occupée par un circuit est proportionnelle à la surface d'un transistor multipliée par le nombre d'entrées des portes logiques

employées ; le temps de réponse d'un circuit est proportionnel au temps de basculement d'un transistor multiplié par le nombre de portes logiques se trouvant sur le plus long chemin menant d'une entrée à une sortie du circuit.

Il existe deux grands types de fonctions logiques : les fonctions combinatoires, qui n'ont pas de mémoire et sont indépendantes du temps, et les fonctions séquentielles, qui tiennent compte du passé et dont les sorties dépendent donc non seulement des entrées actuelles mais aussi de l'état précédent du circuit.

Une fonction logique combinatoire est une fonction de $\{0,1\}^n$ dans $\{0,1\}^p$. Elle transforme un ensemble de n bits d'entrée en un ensemble de p bits de sortie et est indépendante du temps. Toute fonction logique combinatoire peut être représentée par sa table de vérité qui définit la sortie d'un circuit, à partir de ses entrées. Cette table indique aussi comment créer un circuit réalisant cette fonction avec des portes ET, OU, et NON. Une méthode élémentaire consiste à faire la somme – c'est-à-dire le OU, la disjonction – des différents cas où la fonction est vraie, chaque cas s'exprimant comme le produit – c'est-à-dire le ET, la conjonction – de l'ensemble des entrées. Les entrées positives 1 se retrouvent dans le produit telles quelles, les entrées négatives 0 sont complémentées dans le produit, avec une porte NON en série. Les formules obtenues sont dites en forme normale disjonctive. Il existe de nombreuses manières d'optimiser cette construction, afin de réduire le nombre de portes : la méthode de Karnaugh, la méthode de Quine-McCluskey, etc., qui ne sont pas présentées ici.

Dans ce qui suit, les notations utilisées seront $\overline{X}$ pour représenter NON X, ET et OU pour représenter le ET et le OU logiques.

Tables de vérité, formules booléennes et circuits sont trois façons de représenter de manière concrète les mêmes objets abstraits : les fonctions booléennes. Des exercices classiques demandent de passer d'un formalisme à l'autre. En particulier, les exercices portant sur le dessin de circuits sont très ouverts, car il y a de nombreuses façons de dessiner des circuits. Il y a même un aspect assez ludique à chercher à obtenir des dessins clairs, réguliers ou compacts…

Création de blocs logiques combinatoires à partir des portes logiques

Voyons maintenant comment utiliser ces portes logiques pour réaliser des circuits combinatoires plus complexes.

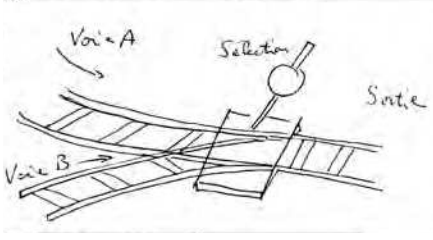
La notion de bus

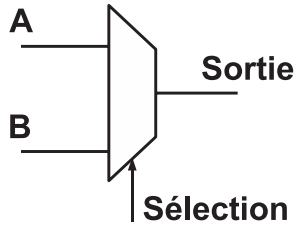
Les signaux transitant sur un fil sont des signaux binaires : chaque signal représente un bit, 0 ou 1. Si l'on désire représenter, par exemple, des nombres

entiers, il est nécessaire de les représenter en base 2, comme nous l'avons vu au premier chapitre, et d'utiliser un fil par chiffre binaire. Avec n fils, on peut représenter les nombres compris entre 0 et $2^n - 1$. Ces n fils, commutés simultanément, constituent un *bus*. Un bus est connecté à un émetteur qui impose une tension sur chaque fil, et à un ou plusieurs récepteurs, qui utilisent les valeurs véhiculées. Un bus ne peut avoir, à un moment donné, qu'un seul émetteur.

Réalisation d'un multiplexeur à deux voies

Un multiplexeur est une structure combinatoire qui sélectionne une valeur parmi plusieurs, en fonction d'une adresse. On peut le comparer à un aiguillage de chemin de fer qui, de même, transfère vers la sortie, une voie ou une autre.





Multiplexeur à 2 voies

Le numéro de la voie choisie est lui-même codé en binaire, et donc représenté par 1 bit pour sélectionner une voie parmi 2, 2 bits pour sélectionner une voie parmi 4, etc. Par exemple, la table de vérité d'un multiplexeur à deux voies est

A	B	Selection	Sortie
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

Et la formule logique d'un tel multiplexeur est :

$$\text{Sortie} = (A \text{ ET } \overline{\text{Selection}}) \text{ OU } (B \text{ ET Selection})$$

On remarquera que le nombre de fils de sortie est égal au nombre de fils des entrées A ou B.

Réalisation d'un additionneur en nombres binaires naturels

Quand on réalise une addition entre deux bits, le résultat peut être 0, 1 ou 2, ce qui se représente en binaire sous la forme 00, 01 ou 10. Le bit de poids faible est appelé « résultat » et le bit de poids fort est appelé « retenue sortante ». Si l'on ajoute deux nombres de plus de 1 bit, on commence par la somme des deux bits de poids faible, qui donne un résultat sur un bit et une retenue. Pour le calcul du bit suivant, la retenue obtenue par la somme des bits de poids faible doit être ajoutée aux bits à ajouter. En appelant ce bit de retenue *Cin* (*Carry in*, retenue entrante), l'addition se fait alors sur A, B et *Cin*, et donc le résultat peut être maintenant 0, 1, 2 ou 3 qui se code toujours sur 2 bits : le bit de résultat et le bit de retenue sortante que nous appellerons *Cout* (*Carry out*, retenue sortante). Donc, pour chacun des bits autres que le tout premier, pour lequel *Cin* vaut 0, la somme est réalisée en additionnant 3 bits de même poids : le bit issu de la première entrée A, celui de la deuxième entrée B et le bit de retenue provenant de l'étage antérieur *Cin*.

Cela se traduit donc très simplement par la table de vérité suivante :

A	B	Cin	C out	Somme
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

D'où l'on déduit immédiatement les équations logiques suivantes :

Somme = ($\overline{A}$ ET $\overline{B}$ ET *Cin*) OU ($\overline{A}$ ET B ET $\overline{Cin}$) OU (A ET $\overline{B}$ ET $\overline{Cin}$) OU (A ET B ET *Cin*)

Cout = ($\overline{A}$ ET B ET *Cin*) OU (A ET $\overline{B}$ ET *Cin*) OU (A ET B ET $\overline{Cin}$) OU (A ET B ET *Cin*)

Le travail sur les tables de vérité et sur les circuits que l'on peut dessiner à la main a des limites – en particulier, celles de la feuille utilisée. Le multiplexeur, l'additionneur donnés précédemment pour une largeur de 1 bit s'étendent à des circuits de 32 bits, ou plus, sous réserve de savoir combiner algorithmiquement les circuits de base, sur un bit, obtenus lors de ces premières étapes. Pour obtenir un multiplexeur à deux entrées sur 32 bits donnant une sortie sur 32 bits, il suffit de dupliquer le multiplexeur précédent 32 fois, une fois pour chaque fil du bus ; mais pour obtenir un multiplexeur à 32 entrées, sur 1 bit, avec une sélection sur 5 bits, donnant le numéro de l'entrée sélectionnée en

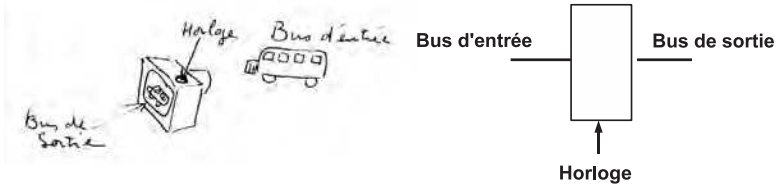
sortie, l'assemblage est plus ardu et demande un peu d'invention. Nous renvoyons le lecteur intéressé par cet aspect de l'architecture, l'algorithmique des circuits, au livre *Arithmétique des ordinateurs* de Jean-Michel Muller.

Blocs de logique séquentielle

La logique séquentielle calcule ses sorties non seulement à partir de ses entrées, mais aussi à partir des états précédents. Tous les circuits séquentiels présentés ici utilisent un signal externe appelé horloge. Ils ne changent d'état que lorsque ce signal d'horloge a une valeur bien précise, on parle alors de synchronisation sur niveau, ou change de valeur, s'il fonctionne avec un front.

La bascule D est la fonction de mémorisation la plus simple : elle est considérée ici comme un composant électronique possédant une entrée pour les données appelée D, une sortie Q et naturellement une horloge H. Son rôle est d'enregistrer la valeur de l'entrée D en fonction du signal d'horloge. Il existe différents types de bascules qui réagissent soit sur un niveau, soit sur un front de l'horloge. Dans une bascule sensible au niveau, la sortie Q prend la valeur de D présente quand, par exemple, H est à 1, et conserve la valeur prise quand H est à 0.

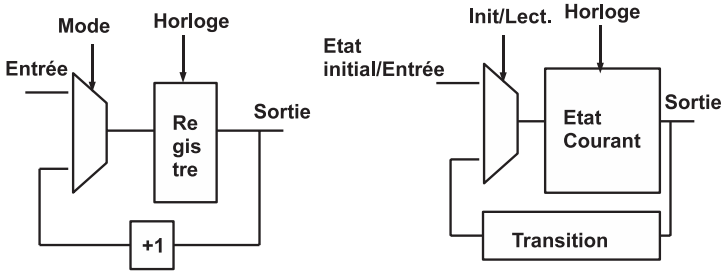
Un *registre* est un ensemble de bascules D connectées chacune en entrée sur l'un des fils d'un bus d'entrée et en sortie sur le fil de même indice du bus de sortie. L'horloge H est commune à toutes les bascules constituant le registre. Bascules D et registres se comportent donc comme des appareils de photo qui enregistrent une image d'entrée lorsque l'on appuie sur le déclencheur H et, pour compléter cette analogie, qui affichent alors l'image acquise sur l'écran LCD de l'appareil.



Un registre.

Le *compteur* est un outil qui permet de réaliser sur action du signal d'horloge, soit le chargement, soit l'incrémentation d'une donnée enregistrée dans un registre, et ce en fonction d'un bit de mode – par exemple, si mode = 0, on effectue un chargement et, si mode = 1, la valeur est incrémentée d'une unité.

Une réalisation possible de ce compteur comporte un registre associé à un additionneur et à un multiplexeur.



a) Compteur, b) Forme générale d'un circuit séquentiel.

La forme de ce circuit est particulièrement représentative des circuits séquentiels avec initialisation, avec un registre conservant l'état d'une variable, et une boucle de retour prenant comme entrée cette valeur pour fournir au prochain top d'horloge une nouvelle valeur à la variable : c'est la fonction de transition. En particulier, les automates à états finis – forme élémentaire de modélisation du calcul algorithmique, voir le deuxième chapitre, paragraphe « États et transitions » – peuvent être représentés à l'aide de circuits de cette forme.

Pour arriver à une forme de circuits séquentiels significativement plus complexes, une méthode consiste à séparer dans la réalisation d'un circuit souhaité la partie algorithmique, modélisée à l'aide d'un automate d'états finis, et dépendant du comportement du circuit souhaité, et la partie calculatoire, qui effectue les calculs sur les données à l'aide d'une unité arithmétique et logique (UAL) associée à quelques registres via des bus. On notera que cette partie est similaire d'un circuit à l'autre. Partie algorithmique – on dit parfois partie contrôle ou PC – et partie calculatoire – on dit parfois partie opérative ou PO – communiquent : la partie algorithmique commande les calculs effectués par la partie calculatoire, les résultats de ces calculs servent à la partie algorithmique en retour. Cette décomposition est très proche de la structure d'un ordinateur – il ne manque que la mémoire et les entrées/sorties –, on la nomme parfois décomposition PC/PO.

La mémoire vive – ou RAM (*Random Access Memory*) – est un système capable d'écrire une valeur d'entrée ou de lire une valeur stockée dans l'une des N cases mémoires qui la constituent. Pour sélectionner la case mémoire désirée en lecture comme en écriture l'utilisateur donne une adresse. Seul le cas où la lecture de la mémoire est combinatoire et non destructive est considéré ici.

La mémoire vive peut dès lors être vue comme un ensemble de registres dont les entrées sont connectées sur le même bus. En écriture, on envoie le signal d'horloge uniquement vers le registre où l'on désire écrire la donnée d'entrée. En lecture, on sélectionne le registre désiré, grâce à un multiplexeur, auquel on communique l'adresse de ce registre.

La machine de Von Neumann

La machine de Von Neumann reprend la décomposition Partie Opérative/ Partie Contrôle des circuits séquentiels proposée précédemment et lui ajoute une mémoire vive (RAM), c'est pourquoi l'on dit parfois Machine RAM. La partie opérative implémente un algorithme d'analyse et d'interprétation du langage machine reconnu par l'ordinateur. Ce modèle de machine informatique, base de tous les processeurs actuels, comporte essentiellement, de façon visible au programmeur :

– une mémoire (RAM), qui contient les programmes et les données ;

– un compteur ordinal (CO) qui indique à chaque instant l'instruction en cours d'exécution. En général, ce compteur est incrémenté à la fin de chaque instruction de façon que la machine exécute l'instruction suivante. Pour effectuer un branchement, ou rupture de séquence, il suffit de modifier la valeur de ce compteur ;

– un ou plusieurs registres accumulateurs destinés à stocker des valeurs temporaires lors d'un calcul ;

– une unité arithmétique et logique (UAL), qui permet d'effectuer les calculs entre les données contenues en mémoire et l'accumulateur, ou entre les registres généraux si l'on utilise une version de ces machines appelée RISC (*Reduced Instruction Set Computer*).

De façon un peu moins visible, la machine comporte aussi pour la partie contrôle :

– un registre adresse mémoire qui permet de sauvegarder l'adresse d'une donnée pendant la durée d'exécution d'une instruction,

– tous les bus d'interconnexion qui permettent le traitement des données,

– un registre instruction qui conserve l'instruction en cours de traitement, et se trouve à l'interface entre la partie opérative et la partie contrôle,

– un système de décodage lié à l'automate de contrôle dont le rôle est de gérer le séquencement de chaque instruction,

– l'ensemble de la machine reçoit une horloge de période habituellement fixe et dont le signal peut être à deux ou plusieurs phases.

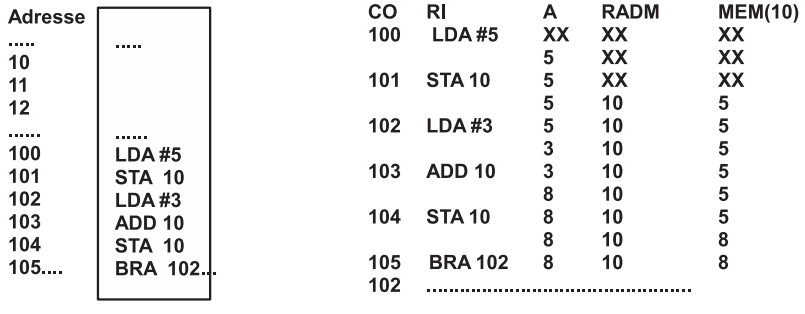
Pour simplifier, nous nous limitons au cas d'une machine à un seul registre accumulateur. Un programme informatique vu par une machine de Von Neu-

mann est de type impératif, c'est-à-dire que l'ordre d'exécution est imposé. Dans la grande majorité des cas, le passage d'une instruction à la suivante est séquentiel, sauf dans les cas où un branchement est demandé.

Il y a une très nette différence entre mémoire RAM et registres accumulateurs, même si l'un et l'autre sont capables de contenir des valeurs binaires : pour limiter le nombre d'instructions à une trentaine ou une soixantaine, par exemple, on ne permet qu'un seul accès à la mémoire de données par instruction. S'il faut réaliser des calculs entre plusieurs cases mémoire, la solution consiste à passer par un ou plusieurs registres qui serviront de stockage temporaire et dont l'accès est implicite. Par exemple, pour réaliser TOTO + TITI ⇒ TATA, où TOTO, TITI et TATA sont des cases mémoire, on programmera LDA TOTO – charger TOTO dans le registre A ou accumulateur –, ADD TITI – lui ajouter TITI –, STA TATA – sauvegarder le résultat dans TATA. Dans les années 70, certaines machines – dites machines CISC (*Complex Instruction Set Computer*) comme le VAX de DEC – ont utilisé des types d'instructions à trois adresses. Cela conduit à des nombres gigantesques d'instructions, plus de 300, avec des modes d'adressage très délicats, jusqu'à 7 pour chacun des trois opérandes, et donc à des matériels en définitive beaucoup plus lents et difficiles à maintenir.

Premières instructions, premiers modes d'adressage

Les données traitées par un programme sont de deux types : les constantes, dont la valeur est explicite dans le programme, et les variables, qui sont mises en mémoire à une adresse explicite dans le programme. Cette figure



Un programme et son exécution.

représente la mémoire, le compteur ordinal, le registre instruction, l'accumulateur, le registre adresse mémoire et la valeur à l'adresse mémoire 10 avec leur évolution cycle par cycle. Chaque instruction dans ce schéma dure deux cycles :

lecture de l'instruction, décodage et exécution. Ce programme commence à l'adresse 100 où il charge la constante de valeur 5 dans l'accumulateur A. Le compteur ordinal passe à 101, et la valeur de A est recopiée à l'adresse 10. Puis, à l'instruction suivante, la constante 3 est mise dans A. Le compteur ordinal passe à 103 où l'on ajoute à A le contenu de la case 10. Puis, à l'instruction 104, le contenu de A est recopié dans la case mémoire 10.

Les instructions LDA #3 et ADD 10, par exemple, utilisent deux modes d'adressage différents : le premier est le mode IMMEDIAT, qui permet d'utiliser une constante, dont la valeur 3 est explicite dans l'instruction. Le second est l'adressage DIRECT où c'est l'adresse 10 de la donnée qui est explicite dans l'instruction.

La dernière instruction est une opération de branchement (*Branch Always*) qui permet de se brancher à l'instruction dont l'adresse est donnée en paramètre, ici 102. Donc, à chaque tour de cette boucle, comprise ici entre les instructions 102 et 105, la valeur immédiate 3 sera ajoutée à la case mémoire 10. Cette instruction fait partie des instructions de rupture de séquence. Ici, le mode d'adressage direct est utilisé. Le mode immédiat n'aurait aucun sens, puisque, par nature, un branchement demande de spécifier une adresse.

Chemins de données et unité de contrôle

Une analyse rapide du programme précédent montre que, du point de vue des données manipulées, le circuit électronique de l'ordinateur gère deux flux venant de la mémoire : celui lié au programme, en particulier lors du cycle de lecture de l'instruction, et celui lié aux données du programme. Ce circuit ne doit plus seulement effectuer un calcul à chaque étape, mais d'abord lire un morceau du programme qui indique les calculs à effectuer. Un ordinateur n'est pas une calculatrice : une calculatrice calcule, un ordinateur exécute un programme qui calcule. De plus, pour ajouter à la possible confusion de cette partie, les deux niveaux, programme et calcul, sont mis en œuvre au même endroit, au sein du même circuit.

L'ordinateur doit donc être capable :

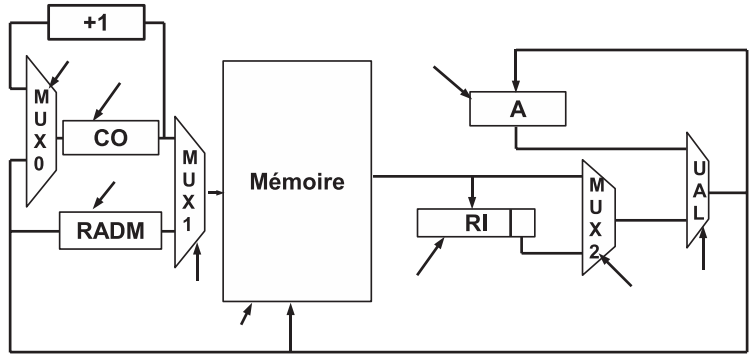
– de lire une case mémoire dont l'adresse est donnée par le compteur ordinal CO pour obtenir l'instruction de code à exécuter ;

– de lire ou d'écrire une case mémoire dont l'adresse est donnée par le programme pour la manipulation des valeurs du programme ;

– de transférer la sortie mémoire vers le registre instruction ou vers l'unité arithmétique et logique ;

– d'écrire une valeur que l'on peut avoir calculée dans A (LDA, ADD), dans la mémoire (STA) dans le CO (BRA) ou dans RADM.

Cette partie de l'ordinateur est qualifiée de « chemin de données ». De nombreuses solutions sont possibles pour la construire, par exemple



Le chemin de données initial.

Pour arriver à réaliser l'ensemble des opérations voulues, il est nécessaire d'ajouter les éléments combinatoires suivants : le multiplexeur MUX1 piloté par un bit de contrôle SELMUX1, pour sélectionner l'adresse mémoire choisie – instruction ou donnée, respectivement SELMUX1 = 0 et 1 –, le MUX2 – commandé par SELMUX2 – pour choisir entre une sortie mémoire ou le paramètre de l'instruction – respectivement SELMUX2 = 0 et 1 –, et l'unité arithmétique et logique. Celle-ci est une fonction combinatoire pour réaliser une fonction FUAL – pour l'instant, soit un choix entre ses deux entrées : A pour entrée accumulateur, B pour entrée issue de MUX2, soit une addition entre ses deux entrées, mais qui sera étendue plus tard à d'autres fonctions.

Le compteur ordinal CO est du même type que le compteur examiné ci-avant avec la possibilité de passer de CO à CO + 1, pour les exécutions en séquence, ou d'instancier CO avec une nouvelle valeur, pour les ruptures de séquence. Le registre instruction RI sera virtuellement découpé en deux champs distincts, le numéro d'instruction et le paramètre.

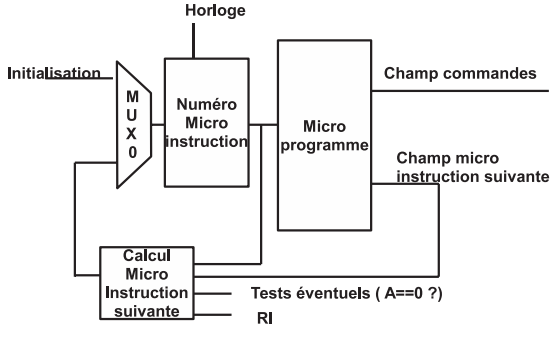
Les différents éléments du chemin de données reçoivent des commandes permettant soit une modification combinatoire, soit une modification liée au signal d'horloge. Des exemples de modifications combinatoires sont la commande FUAL, qui indique la fonction de l'unité arithmétique et logique (UAL) à exécuter, par exemple ici FUAL = A pour obtenir A, FUAL = B pour obtenir B ou FUAL = ADD pour obtenir A + B ; la commande CO + 1 qui permet de choisir d'incrémenter le CO ou de le charger à partir du bus de données, les valeurs de ces deux commandes étant imposées par l'unité arithmé-

tique et logique – on remarque que le bit de commande CO + 1 n'est autre que le bit SELMUX0, il est plus aisé de comprendre que le CO est incrémenté si ce bit est à 1 et que HCO = 1 – ; et les commandes SELMUX1 et SELMUX2. Dans le cas d'une modification liée au signal d'horloge, l'unité de contrôle fournit des bits permettant de valider le signal d'horloge au niveau du CO, de RADM, de la mémoire RAM, du registre instruction et de l'accumulateur – respectivement HCO, HRADM, HMEM, HRI, HACCU.

En contrepartie, le chemin de données – par l'intermédiaire de RI et A – fournit à l'unité de contrôle des informations sur le code de l'instruction courante et la valeur de l'accumulateur.

Le chemin de données vu ici est un ensemble totalement inerte, que l'on peut comparer à une marionnette, qui ne peut rien faire si l'on n'actionne pas ses fils. L'unité de contrôle est la partie qui permet de donner vie à cette marionnette. « Faire un pas en avant » est une instruction, que le marionnettiste décompose en plusieurs micro-instructions : lever le pied droit, l'avancer, le reposer, lever le pied gauche, l'avancer et le reposer, qui correspondent chacune au mouvement d'un fil de la marionnette. Quand le marionnettiste lit le programme « exécuter une valse », instruction par instruction, comme dans les livres initiant à l'art de la danse, il les réalise, l'une après l'autre, en décomposant chacune en plusieurs micro-instructions.

De même, chaque instruction du langage machine est décomposée par l'unité de contrôle en plusieurs micro-instructions. L'unité de contrôle est un circuit séquentiel comme celui présenté ci-avant. Plus en détail, ce circuit a la forme suivante



L'unité de contrôle.

Elle a pour entrées l'horloge, le numéro de l'instruction à réaliser, le numéro de l'état en cours et parfois des valeurs extérieures. Une des solutions les plus

compréhensibles pour réaliser un tel système est la machine dite microprogrammée, dont l'ensemble des micro-instructions à réaliser est enregistré dans une mémoire annexe, en général non modifiable, en fonction du numéro de l'état actuel, qualifié par la suite de « numéro de micro-instruction ». Le passage à l'état suivant est en général connu, sauf dans le cas du décodage d'une nouvelle instruction, où il faut utiliser le contenu du registre instruction RI pour décider de la micro-instruction à lancer.

Les instructions de base et leurs micro-instructions

Dans tout ce qui suit, les bits de validation d'horloge – HRI, HCO, HMEM, HA – sont considérés comme actifs s'ils sont à une valeur 1 logique. Tous les circuits reçoivent implicitement une horloge commune CK. De ce fait, l'horloge sur l'une des parties synchrones ne sera active que si l'horloge est active et que le signal de validation correspondant est à 1.

Au démarrage de la machine, nous supposerons que tous les registres sont mis à zéro. Il faut alors lancer l'opération de lecture du registre instruction, qui est commune à l'exécution de toutes les instructions. Cette opération nécessite une micro-instruction, appelée *décodage*, qui transfère le contenu de la mémoire d'adresse CO dans le registre d'instruction RI – Mem (CO) ⇒ RI. Les autres micro-instructions dépendent de l'instruction qui se trouve désormais dans le registre RI. Ces micro-instructions sont stockées en ROM (*Read-Only Memory*) – mémoire morte, c'est-à-dire non modifiable, dont la valeur est définie à la construction de l'ordinateur –, dans la partie contrôle de l'ordinateur, à une adresse *f*(*RI*) qui dépend de l'instruction. Cette adresse peut, par exemple, être trois fois le numéro de l'instruction plus une constante. Cette opération s'écrit simplement SELMUX1 = 0, HRI = 1, micro-instruction = *f*(*RI*). Si, par exemple, la première instruction à traiter est l'instruction LDA #5 et si le code du LDA# est *n*, la micro-instruction à traiter après la micro-instruction de décodage aura pour adresse 3 ∗ *n* + constante.

Après la micro-instruction de décodage, il peut y avoir une ou plusieurs micro-instructions désignées par le numéro de l'instruction : une seule si l'instruction contient l'ensemble des informations nécessaires, plusieurs si l'instruction contient aussi l'adresse en mémoire des données à charger.

Détaillons ces micro-instructions pour quelques instructions classiques :

LDA # paramètre – chargement immédiat – la première micro-instruction correspondant à cette instruction réalise : Paramètre ⇒ A, CO + 1 ⇒ CO, aller à la micro-instruction de décodage

Ce qui se traduit immédiatement par

SELMUX2 = 1, FUAL = B, HA = 1, CO + 1 = 1, HCO = 1, aller à la micro-instruction de décodage

LDA paramètre – chargement de A en mode direct –, la première micro-instruction doit transférer le paramètre dans le registre RADM :

Paramètre ⇒ RADM, micro-instruction suivante, soit

SELMUX2 = 1, FUAL = B, HRADM = 1, micro-instruction suivante

la seconde micro-instruction réalise le stockage du résultat sur le bus dans l'accumulateur : M(RADM) ⇒ A, CO + 1 ⇒ CO, aller à la micro-instruction de décodage, soit

SELMUX1 = 1, SELMUX2 = 0, FUAL = B, HA = 1, CO + 1 = 1, HCO, aller à la micro-instruction de décodage

ADD #paramètre – addition en mode immédiat –, la première micro-instruction ajoute le paramètre au registre A

Paramètre + A ⇒ A, CO + 1 ⇒ CO, aller à la micro-instruction de décodage, soit

SELMUX2 = 1, FUAL = ADD,HA = 1,CO + 1 = 1, HCO, aller à la micro-instruction de décodage

ADD paramètre – chargement de A en mode direct –, la première micro-instruction doit transférer le paramètre dans le registre RADM : Paramètre ⇒ RADM, micro-instruction suivante, soit

SELMUX2 = 1, FUAL = B, HRADM = 1, micro-instruction suivante

la seconde micro-instruction réalise l'addition et le stockage du résultat dans l'accumulateur : M(RADM) + A ⇒ A, CO + 1 ⇒ CO, HCO, aller à la micro-instruction de décodage, soit

SELMUX1 = 1, SELMUX2 = 0, FUAL = ADD, HA = 1, CO + 1 = 1, HCO, aller à la micro-instruction de décodage

STA paramètre – stockage de A en mode direct – la première micro-instruction doit transférer le paramètre dans le registre RADM :

Paramètre ⇒ RADM, micro-instruction suivante, soit

SELMUX2 = 1, FUAL = B, HRADM = 1, micro-instruction suivante

la seconde micro-instruction réalise le stockage du résultat en mémoire : A ⇒ M(RADM), CO + 1 ⇒ CO, aller à la micro-instruction de décodage, soit

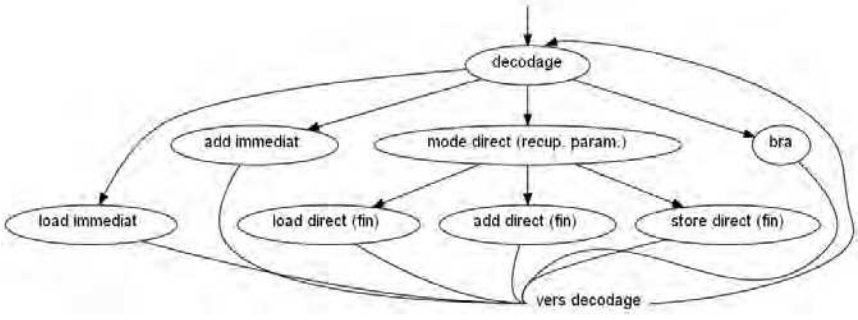
SELMUX1 = 1, SELMUX2 = X (0 ou 1, c'est indifférent), FUAL = A, HMem = 1, CO + 1 = 1, HCO, aller à la micro-instruction de décodage

BRA paramètre – branchement inconditionnel direct –, se traduit par

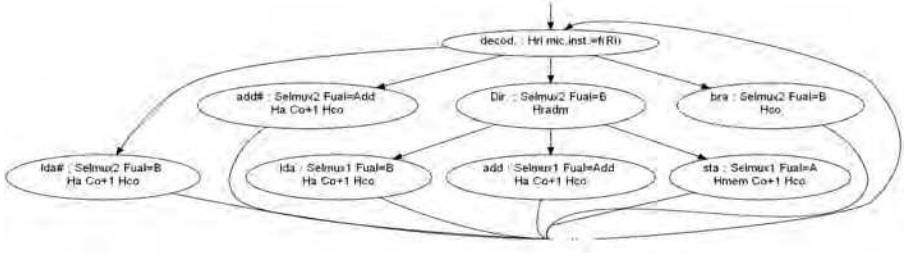
Paramètre ⇒ CO, aller à la micro-instruction de décodage

SELMUX2 = 1, FUAL = B, CO + 1 = 0, HCO, aller à la micro-instruction de décodage

Les microprogrammes d'interprétation peuvent être représentés par un graphe sous la forme d'un automate appelé graphe de contrôle.



Automate de contrôle (version symbolique).



Automate de contrôle (version avec les micro-instructions).

Réalisation de tests

Cette première version de machine n'a pas la possibilité de tester une valeur. Pour y parvenir, il est nécessaire de créer des instructions de branchement conditionnel. Plusieurs solutions existent pour cela, beaucoup de processeurs utilisent des *indicateurs* connectés à l'UAL qui sauvegardent des résultats de tests réalisés dans les instructions précédentes et les éventuels débordements de capacité observés. La solution présentée ici est beaucoup plus intuitive et consiste à tester le contenu du registre A. On peut ainsi créer une instruction :

BRZ paramètre, qui effectue un branchement à l'adresse indiquée si A vaut 0, et incrémente le CO dans les autres cas.

Si A = 0, paramètre ⇒ CO

Sinon CO + 1 ⇒ CO

Les micro-instructions correspondent à très peu près à celles du BRA, sinon que la valeur du bit CO + 1 doit être modifiée en fonction du test de A à 0.

Pour cette instruction seulement, si $A = 0$, CO + 1 = 0, en revanche, si $A \neq 0$, CO + 1 = 1.

On peut naturellement créer des instructions BRN branchement si A est négatif et même, pourquoi pas, BRP branchement si A est strictement positif.

Extension des modes d'adressage

Les modes d'adressage présentés jusqu'ici, le mode immédiat et le mode direct, ne permettent pas au programme de calculer l'adresse d'une variable ou d'un branchement. Pour écrire, par exemple, un programme qui effectue la somme des éléments d'un tableau, il est nécessaire d'accéder à des cases mémoire dont l'adresse est calculée par le programme. Deux types d'adressage sont possibles pour cela : l'adressage *indirect* et l'adressage *indexé*.

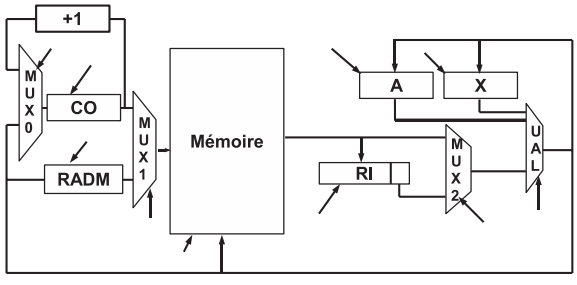
Dans le cas de l'adressage indirect, l'adresse réelle est calculée et mise dans une case mémoire d'adresse connue, c'est donc un pointeur. L'adressage indirect se note avec des crochets [ ]. Détaillons deux exemples dans lesquels on donne un nom à la valeur du paramètre.

LDA [pointeur] met dans A la valeur dont l'adresse est contenue dans la case mémoire pointeur.

BRA [retour] branche à l'adresse contenue dans la case mémoire réservée à l'adresse retour.

Dans le cas du LDA, cet adressage effectue deux appels mémoire successifs, ce qui peut ralentir l'exécution du programme.

Dans le cas de l'adressage indexé, qui est souvent préféré, une opération d'indirection est réalisée non plus avec une case mémoire, mais avec un registre accumulateur supplémentaire : le registre X, ou registre d'index.



Le chemin de données avec registre d'index.

Il est nécessaire pour cela d'ajouter ce registre au chemin de données ci-avant, et d'ajouter aussi des instructions complémentaires pour charger et sau-

vegarder ce registre – LDX en modes direct et immédiat, STX en mode direct –, ainsi que des instructions pour modifier X – ADDX direct et immédiat, par exemple.

Cela nécessite naturellement d'autre part une modification de l'unité arithmétique et logique UAL qui accepte maintenant 3 entrées, et dont les fonctions vont être A, X, B, A + B, X + B et, pour la soustraction, A – B et X – B.

Une instruction utilisant l'adressage indexé aura encore deux champs, l'instruction et le paramètre. Le paramètre pourra être utilisé pour ajouter un déplacement connu et constant au contenu du registre X. LDA paramètre, X réalisera alors comme fonction principale Mem([X] + paramètre) ⇒ A. Souvent, le paramètre vaut 0, c'est directement X qui donne l'adresse de la donnée à charger ; les cas où le paramètre est différent de 0 peuvent correspondre à des données structurées lorsque le champ visé n'est pas le premier.

Introduction au langage d'assemblage

Le langage d'assemblage est un langage plus intelligible que le langage machine, bien qu'il n'en diffère que par l'utilisation de mnémoniques, de symboles et d'étiquettes. Les programmes écrits dans ce langage sont traduits, quasiment ligne à ligne, en langage machine, par un outil appelé un *assembleur*.

Structure d'une ligne de code, instructions et directives, commentaires

Un programme en langage d'assemblage comporte des lignes très structurées constituées des éléments suivants.

Un premier champ qui commence en début de ligne. Il est appelé étiquette et représente l'adresse symbolique de la ligne. La ligne en cours définit cette étiquette si le champ n'est pas blanc. Ce champ est un symbole – commençant par une lettre – qui ne peut être défini au plus qu'une fois dans un module de programme donné.

Le deuxième champ comporte une instruction ou une directive d'assemblage. S'il n'est pas reconnu, il doit envoyer à l'utilisateur une erreur d'assemblage. S'il s'agit d'une instruction, elle pourra être exécutée – si elle a un sens – par la machine. Ce champ peut aussi être une directive d'assemblage. Auquel cas, la ligne sera gérée non pas par la machine mais par l'assembleur lui-même. Quelques directives fondamentales ont été choisies ici : ORG (*Origin*) définit l'adresse en mémoire à laquelle on veut implanter un programme ou des données ; RMW (*Reserve Memory Word*) réserve un certain nombre de mots en mémoire ; FCW (*Form Constant Word*) réserve et initialise un mot mémoire ; EQU (*Equate*) attribue une valeur à un symbole ; END (fin du texte source) définit la fin du texte source écrit en langage d'assemblage.

Le troisième champ est destiné au paramètre de l'instruction, ou de la directive, et au mode d'adressage. Ce champ peut représenter une adresse de donnée, dans le cas de l'adressage direct, une valeur constante, donnée en décimal ou en hexadécimal si elle est précédée par un $, le nom d'un symbole, qui sera remplacé par sa valeur dans la table des symboles, ou * qui représente alors le compteur d'implantation, c'est-à-dire l'adresse mémoire à laquelle le programme d'assemblage est en train d'écrire la valeur d'une instruction. Il est possible d'ajouter ou de soustraire des termes de ce type, avec la syntaxe : TERMEopTERME où op représente + ou –, sans espaces ni parenthèses, par exemple : Nom EQU*-TAB.

Le quatrième champ, optionnel, est réservé aux commentaires, qui commencent en général par le caractère *.

Par habitude, le mode d'adressage qui devrait se trouver lié à l'instruction, deuxième champ, est en réalité mis dans le troisième champ.

Les instructions du langage d'assemblage et directives de l'assembleur dépendent de la machine et de l'assembleur. Pour ce chapitre, les instructions et directives, qui sont celles de la Machine 2 présentée dans le livre *Du transistor à l'ordinateur* de Claude Timsit, et qui sont cohérentes avec les outils fournis dans `http://www.e-campus.uvsq.fr/dutransistoralordinateur/`, sont les suivantes

Instructions	Code	Exemple	Action
LDA immédiat	0	LDA #Param	Param=>A, CO++
LDA direct	1	LDA Param	Mem(Param)=>A, CO++
LDA indexé	9	LDA Param,X	Mem([X]+Param)=>A,CO++
STA direct	3	STA Param	[A]=>Mem(Param),CO++
STA indexé	10	STA Param,X	[A]=>Mem([X]+Param),CO++
LDX immédiat	7	LDX #Param	Param=>X, CO++
LDX direct	8	LDX Param	Mem(Param)=>X, CO++
ADD immédiat	4	ADD #Param	[A]+Param=>A, CO++
ADD direct	6	ADD Param	[A]+Mem(Param)=>A, CO++
ADD indexé	15	ADD Param,X	[A]+Mem([X]+Param)=>A,CO++
ADDX immédiat	12	ADDX #Param	[X]+Param=>X, CO++
ADDX direct	19	ADDX Param	[X]+Mem(Param)=>X, CO++
SUB immédiat	13	SUB #Param	[A]-Param=>A, CO++
SUB direct	14	SUB Param	[A]-Mem(Param)=>A, CO++
SUB indexé	16	SUB Param,X	[A]-Mem([X]+Param)=>A,CO++
SUBX immédiat	17	SUBX #Param	[X]-Param=>X, CO++
SUBX direct	20	SUBX Param	[X]-Mem(Param)=>X, CO++
STX direct	18	STX Param	[X]=>Mem(Param),CO++
BRA direct	2	BRA Param	Param=>CO
BRA indexé	11	BRA Param,X	[X]-Param=>CO
BRA indirect	24	BRA [Param]	Mem(Param)=>CO
BRZ direct	5	BRZ Param	SiA==0 Param=>CO, sinon CO++
BAL direct	21	BAL Param	Param=>CO, CO++=>SVCO
STSV direct	22	STSV Param	[SVCO]=>Mem(Param),CO++
STSV indexé	24	STSV Param,X	[SVCO]=>Mem([X]+Param),CO++

Directive	Action		
ORG	définit une origine		
exemple	debut	ORG	$100
définit une étiquette exemple et l'associe à l'adresse hexa 100			
RMW	réserver un ou plusieurs mots mémoire		
exemple	T	RMW	10
Réserve 10 mots en mémoire à partir de l'adresse actuelle			
FCW	réserve et initialise un mot mémoire		
exemple	N	FCW	5
Réserve une case mémoire et l'initialise à 5			
END	fin d'assemblage		
Ce n'est que la fin du texte source. Le paramètre doit être 0.			
A ne pas confondre avec la dernière instruction exécutable			
EQU	définit un symbole (#define)		
exemples	TOTO	EQU	5
	TOTO ne réserve pas de mémoire mais		
	sera remplacé par la valeur 5 à chaque apparition		
	TOTO	EQU	*
	dans les calculs d'adresses * a un sens particulier		
	(compteur d'implantation		

Instructions et directives reconnues par l'assembleur M2 de « Du transistor à l'ordinateur »

On retrouve différents types d'instruction, communs à la plupart des langages machine : des instructions de transfert de données – LDA, STA, LDX, STX, STSV –, des instructions de calcul – ADD, SUB, ADDX, SUBX – et des instructions de rupture de séquence – BRA, BRZ, BAL.

Supposons que nous voulions faire la somme des N premières valeurs du tableau d'entiers T, considéré comme déjà chargé. Le code de ce programme peut s'écrire de façon symbolique, en langage d'assemblage, comme suit :

```
DONNEES ORG $100 *Origine des données a l'adresse hexadécimale $100
T     RMW   10 *Réservation de 10 mots (10 en décimal)
N     EQU   3 *N est une constante valant 3
I     RMW   1 *I est par exemple l'indice de boucle
SOMME RMW 1 *SOMME est le résultat attendu
      * On considère que quelqu'un a chargé le tableau T.
PROG  ORG $0 *Adresse de début d'exécution
      LDA #0 * for (I=0;I<N;I=I+1) // on commence par mettre I
à 0
      STA I * I=0
      LDA #0
      STA SOMME * SOMME=0  //et SOMME à 0
      LDX #T *   T est l'adresse du tableau
TEST  LDA I * Boucle  //Puis on effectue le test de boucle
      SUB #N * comparaison de I à N
      BRN SUITE
      BRA FIN
SUITE LDA SOMME *    //et le calcul du corps de boucle
      ADD 0,X * { SOMME=SOMME+T[I];}
      STA SOMME
      ADDX #1 * //incrémentation de l'indice de boucle et de
l'adresse
      LDA I
      ADD #1
      STA I
      BRA TEST * Retour en début de boucle
FIN   ...................
```

Dans ce programme, T est une constante représentant l'adresse du tableau. Si l'on réalise LDA T, on met donc, dans A, le contenu du premier élément de tableau (Mem(T) ⇒ A). L'instruction LDX #T effectue, en revanche, le transfert de la constante, soit ici l'adresse du tableau dans X (Adresse T ⇒ X), ce qui permet de réaliser simplement des opérations indexées : X peut être incrémenté (ADDX #1) et LDA 0,X réalise alors Mem(0 + contenu de X) ⇒ A.

On peut d'autre part remarquer que LDX #T suivi de LDA 0,X est équivalent à LDA T. En revanche, dans ce cas, aucun calcul ne peut être fait sur T.

Une autre façon de faire consiste à réaliser une indirection par rapport à la mémoire : LDA #T; STA pointeur; LDA[pointeur]. Pour des raisons de performance, la solution utilisant l'adressage indexé est très souvent préférée, car elle économise un appel mémoire.

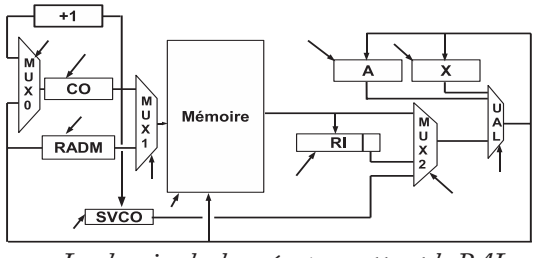
La programmation en langage d'assemblage est utile au programmeur, car elle permet d'utiliser des symboles de mnémoniques d'instructions et d'étiquettes, mais le programme ne peut être interprété tel qu'il est par le matériel qui s'attend à des instructions codées en valeurs binaires. Le paragraphe « Le fonctionnement d'un assembleur » montre le fonctionnement d'un programme appelé « assembleur » dont le rôle est, après quelques vérifications de syntaxe et d'adresses mémoire, de traduire ce programme en langage machine exécutable par l'ordinateur.

Appel de sous-programmes

Un sous-programme, ou fonction, peut être appelé de plusieurs endroits différents par un programme. L'appel du sous-programme est un simple branchement, mais, pour revenir au bon endroit dans le programme appelant, il est nécessaire de conserver l'adresse à laquelle ce retour doit s'effectuer. C'est ce que fait l'instruction *Branch and Link* (BAL).

Pour mettre en évidence toutes les opérations nécessaires dans un appel de sous-programme, nous avons fait le choix de sauvegarder l'adresse de retour dans un registre spécialisé, appelé SVCO – sauvegarde du compteur ordinal CO –, qui doit être ensuite entièrement géré par logiciel : ce registre est transféré en mémoire, par une instruction STSV.

Cela conduit à une légère modification du chemin de données, du fait de l'introduction du registre SVCO, avec son signal de validation d'horloge HSVCO et d'une voie supplémentaire sur MUX2, avec l'extension de SELMUX2 à 2 bits.



Le chemin de données permettant le BAL.

L'exécution de l'instruction BAL paramètre réalise donc : CO + 1 ⇒ SVCO ; paramètre ⇒ CO et l'exécution de STSV paramètre : SVCO ⇒ M(paramètre) ;

CO + 1 ⇒ CO. D'autres modes d'adressage sont naturellement concevables pour ces deux instructions.

Un exemple de fonction simple: la fonction suivante réalise le calcul d'une multiplication par trois. On charge A avec une valeur, puis on appelle cette fonction et on récupère, à l'issue de son exécution, le résultat dans A.

```
Donnees ORG 2 *Origine des données
Resultat RMW 1
RET   RMW 1 *Adresse de retour
Tmp   RMW 1 *Temporaire pour le calcul

Prog ORG 12 *Origine du programme
.....
      LDA #5 *chargement de A
      BAL FOISTROIS *Appel de fonction
      STA Resultat *Utilisation du résultat
.....
Fin   BRA Fin

FOISTROIS STSV RET *Sauvegarde de SVCO
      STA Tmp *Calcul
      ADD Tmp
      ADD Tmp * A vaut maintenant trois fois sa valeur initiale
      BRA [RET] *Retour à l'adresse contenue dans RET, résultat dans A
```

Dans cet exemple, un seul paramètre d'entrée a été utilisé. On peut alors le mettre dans A. Le résultat de la fonction est renvoyé dans A. Si plusieurs paramètres d'entrée étaient nécessaires, on les aurait rangés dans une pile logicielle avant l'appel de la fonction. De même, l'appel à la fonction FOISTROIS étant unique, le registre SVCO a suffi à conserver l'adresse de retour. Dans des programmes plus complexes, possédant plusieurs appels de fonctions imbriqués, en particulier pour les programmes récursifs, le registre SVCO doit être sauvegardé. Une solution habituelle consiste à le mettre dans une pile logicielle.

On remarque la dernière instruction du programme principal Fin BRA Fin dont le seul rôle est d'empêcher le programme de continuer son exécution. L'horloge continuant toujours à fonctionner, il faut en effet limiter strictement l'exécution en « bloquant » le compteur ordinal.

La fonction FOISTROIS peut être appelée de n'importe où et même plusieurs fois dans le programme principal. C'est pourquoi le BAL sauvegarde

l'adresse de retour. Celle-ci est mise dans un pointeur appelé ici RET. Pour revenir à l'adresse contenue dans ce pointeur, il est donc nécessaire de réaliser l'adressage indirect BRA[RET].

Réalisation et utilisation d'une pile logicielle

La machine définie ci-dessus permet de traiter de très nombreux problèmes, en particulier d'implanter très simplement une pile logicielle. Rappelons qu'une pile est une zone mémoire de taille fixe dans laquelle il est possible d'empiler une valeur, si la pile n'est pas pleine, ou de dépiler une valeur, si la pile n'est pas vide. L'analogie avec une pile d'assiettes dans une mémoire est immédiate. Une variable de type pointeur, appelée « sommet de pile », permet très simplement de réaliser ces opérations. Si l'on simplifie le problème en supprimant les tests de débordement, ces fonctions sont très facilement réalisées.

```
N     EQU 10 *taille de pile
Pile ORG *
      RMW N *réservation mémoire
Spile RMW 1 *pointeur de sommet de pile

PROG EQU * *Le programme commence ici
……
INIT EQU * *INIT est implanté ici
      LDA #Pile
      STA Spile

EmpileA EQU * *empiler A
      LDX Spile
      STA 0,X
      ADDX #1
      STX Spile

DepileA EQU * *dépiler A
      LDX Spile
      SUBX #1
      STX Spile
      LDA 0,X
…
```

Entrées/sorties simples

Un ordinateur sans entrée ni sortie n'a aucun intérêt, car ce n'est qu'un système fermé qui consomme de l'énergie ! Pour que ce système effectue des opérations utiles, il faut le doter de périphériques. Ceux-ci sont généralement répartis entre périphériques d'entrée – par exemple, clavier, souris, scanner, etc. – et périphériques de sortie – écrans, voyants lumineux, imprimantes. Deux grands modes de fonctionnement permettent de gérer les périphériques : le mode *programmé*, dans lequel un transfert entre le processeur et le périphérique est à l'initiative du processeur, et un mode plus complexe, *par interruption*, dans lequel le périphérique envoie un signal au processeur pour signaler une opération d'entrée/sortie. Pour l'analyse du processus d'interruption, nous renvoyons aux livres *Du transistor à l'ordinateur*, cité ci-avant, et *Systèmes d'exploitation* d'Andrew Tanenbaum.

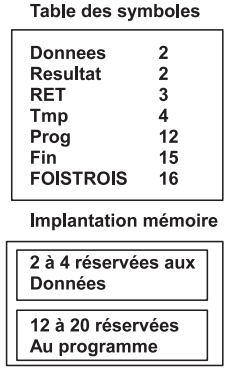
Dans le mode programmé, le processeur lit et écrit directement dans des « registres » d'entrées/sorties. Les deux méthodes habituelles utilisées pour cela sont soit de prévoir des registres supplémentaires, dits registres d'entrées/sorties, et des instructions supplémentaires servant à effectuer des transferts entre la mémoire et ces registres d'entrées/sorties, soit de donner à ces registres une adresse dans l'espace mémoire, les instructions servant à effectuer ces communications étant alors des LDA et des STA classiques. Ces deux méthodes sont employées, par exemple par les processeurs Intel pour la première et par les processeurs Motorola pour la seconde. La première conduit à un ensemble d'instructions supplémentaires. La seconde demande de neutraliser quelques cases mémoire pour les remplacer par des registres, mais la programmation en devient beaucoup plus simple. Les simulateurs Machine 3 et Machine 4 proposés dans *Du transistor à l'ordinateur*, utilisent cette seconde méthode.

Si, par exemple, l'affichage décimal est installé à l'adresse 100, un affichage décimal du contenu du registre A se fera simplement par STA 100. De même, si une entrée de valeur entière est assignée à l'adresse 101 – appelée clavier décimal dans les simulateurs Machine 3 et Machine 4 –, LDA 101 permettra de la lire.

Le fonctionnement d'un assembleur

Les rôles de l'assembleur sont de vérifier que le programme écrit en langage d'assemblage est correct et, si possible, de le traduire en langage machine interprétable directement par la machine. Il faut donc que l'assembleur crée une photographie de ce que sera la mémoire juste avant l'exécution du programme. Cette image de la mémoire, l'implantation mémoire, sera ensuite chargée dans le processeur, avec – si l'on désire tracer les variables en mémoire – une table

des symboles, qui associe aux étiquettes et autres symboles une valeur – cette valeur sera une adresse pour les étiquettes. Un assembleur fonctionne généralement en deux passes : lors de la première passe, on remplit au fur et à mesure la table des symboles et l'on essaye de créer l'implantation mémoire, ou, si l'on n'y arrive pas, tout au moins de réserver de la place dans cette implantation mémoire. Ce cas se produit, par exemple, si l'on effectue une référence en avant, en particulier quand on traduit une instruction BRA Fin, dans laquelle Fin est une étiquette située, dans le texte source, après l'instruction en cours de traduction. Ne connaissant pas l'adresse de l'instruction dont l'étiquette est Fin, il faut attendre la fin de la lecture totale du fichier source pour pouvoir terminer la traduction de l'instruction.



Les tables de l'assembleur dans l'exemple traité.

```
1 Donnees  ORG 2 *Origine des données
2 Resultat RMW 1
3 RET      RMW 1*Adresse de retour
4 TmpRMW 1 *Temporaire pour le calcul
5
6
7 Prog ORG 12 *Origine du programme
......
8     LDA #5 *chargement de A
9     BAL FOISTROIS *Appel du sous-programme FOISTROIS
10    STA Resultat *Utilisation du résultat
......
11 Fin     BRA Fin
```

```
12 FOISTROIS STSV RET *Sauvegarde de SVCO
13    STA Tmp *Calcul
14    ADD Tmp
15    ADD Tmp
16    BRA [RET] *Retour à l'adresse contenue dans RET
17    END  0
```

La première directive ORG, ligne 1, donne une valeur au compteur d'implantation, qui définit l'adresse à partir de laquelle il faut modifier l'implantation mémoire, ici c'est 2. Une case mémoire d'adresse 2 est donc réservée, et l'on attribue la valeur 2 au symbole `Resultat`. Ce 2 représente ici une adresse, la valeur de `Resultat` n'étant pas déterminée ici. Avant de faire cette opération, l'assembleur vérifie que l'étiquette `Resultat` n'a pas été déjà définie. Si c'était le cas, il y aurait un essai de double définition qui serait considéré comme une erreur. Le compteur d'implantation est incrémenté du nombre de cases désiré, ici 1, la mémoire est réservée et l'on passe aux lignes suivantes qui permettent de définir les emplacements de RET et de `Tmp` et d'effectuer la réservation nécessaire.

À la ligne 7, on assigne à la valeur `Prog` la valeur 12, mais, avant de continuer à implanter des données ou des instructions dans la mémoire, il faut vérifier que celle-ci est encore libre. À la ligne 8, on traduit l'instruction LDA #5 : c'est ici possible, le code de LDA# et la valeur 5 étant connus. La case mémoire n'étant pas déjà utilisée, elle est réservée et chargée à la bonne valeur – par exemple, pour Machine 2, à 0*1024 + 5. La ligne 9 représente un cas typique de référence en avant : si l'on sait traduire BAL, son paramètre fait appel à une étiquette TROISFOIS que la table des symboles ne connaît pas. Une case mémoire est cependant réservée pour le cas où, lors de la deuxième passe, celle-ci sera connue. En continuant ainsi jusqu'à la ligne 12, l'étiquette TROISFOIS va être définie, etc. La ligne 17 représente la fin du fichier source. Quand l'assembleur y arrive, il lance la seconde passe qui permet de remplir les cases de la mémoire d'implantation qui n'ont pas pu l'être auparavant – ici, celle qui correspond à la ligne 9 – en utilisant la table des symboles qui s'est remplie au cours de la première phase. Si aucune erreur de syntaxe, de non-définition ou de double définition d'étiquette, d'implantation, etc. n'est trouvée, le résultat de l'assemblage peut être transféré dans la mémoire de la machine destinée à exécuter le code – opération de chargement réalisée par un programme souvent nommé Chargeur ou *Loader* – pour être exécuté.

Rudiments de compilation manuelle

La compilation consiste à transformer un code écrit dans un langage de haut niveau, ici Java, en un code en langage d'assemblage. Nous montrons ici comment traduire un programme simple, à la main, afin de définir une méthodologie stricte, pour créer des programmes en langage d'assemblage. Les idées principales sont de conserver strictement le nom des variables et de respecter strictement les algorithmes écrits, en les traduisant directement ligne à ligne, sans chercher à optimiser. La machine présentée ici a volontairement un seul registre accumulateur et un seul registre d'index, qui sont considérés comme des temporaires. Toute variable doit donc être implantée en mémoire. Les directives d'assemblage permettent de réaliser la réservation de la mémoire, de définir des constantes, etc.

Un exemple de programme avec une boucle

Un exemple, valant un long discours, est présenté ici. Pour faciliter la lecture, les constantes et variables ne faisant pas partie explicitement du code source sont en majuscules. On donne ainsi l'adresse de l'afficheur décimal, les adresses correspondant aux données et au programme, mais aussi SAVX variable qui va permettre une utilisation simple de l'adresse de l'élément de tableau et de son incrémentation. Il y a de très nombreuses façons de traduire cette boucle en langage d'assemblage, celle présentée ici est simple et ne fait aucune optimisation, ce qui facilite la maintenance. L'utilisation de SAVX dans ce programme n'est pas nécessaire, car X n'est utilisé que comme pointeur sur l'élément de tableau en cours de traitement. Elle devient nécessaire quand on utilise plusieurs tableaux ou plusieurs boucles imbriquées. Dans ce cas, on créera autant de variables de sauvegarde de X que nécessaire. Dans le cas de programmes récursifs, il faudra utiliser la pile pour sauvegarder les adresses de retour des appels récursifs ainsi que la valeur des autres registres, voire les variables locales intermédiaires.

```
                                ECRAN EQU 100 *adr. de l'aff. décimal
                                DEBDATA EQU 2 *adr. d'impl. des données
                                DEBPROG EQU 12 *adr. d'impl. du programme
static final int N=5;           N EQU 5 * traitement du #define
static int i;                   Data ORG DEBDATA *origine des données
static int resultat;            i RMW 1 *réservation des variables
static int [] Tab=new int[N]; resultat RMW 1
                                Tab RMW N *reservation du tableau
                                SAVX RMW 1 *sauvegarde de l'index
```

```
public static void main(String[] args){
 for (i=0;i<N;i=i+1) {
   Tab[i]=i;
   resultat=resultat+i;
 }
 System.out.println(resultat);
}
```

```
Prog ORG DEBPROG *début du programme
LDA #0 *initialisation de la boucle
STA i
LDX #Tab *adresse du tableau
STX SAVX *pointeur courant ds le tableau
TEST LDA #N *test de i
SUB i
BRZ FINBCL * traitement de i<N
BRN FINBCL
SUITEBCL LDX SAVX
LDA i
STA 0,X *Tab[i]=i;
LDA resultat
ADD i
STA resultat *resultat=resultat+i

LDA i *modifications indice et index
ADD #1
STA i
LDX SAVX
ADD #1
STA SAVX
BRA  TEST *branchement au test
FINBCL LDA resultat
STA ECRAN *impression
..
```

Exercices corrigés et commentés

Circuits combinatoires

Petits circuits classiques

Exercice 1

Donner les tables de vérité des circuits suivants :

Majorité à 3 entrées, une sortie : S = *Majorité*(E_2, E_1, E_0) le bit S est le gagnant du vote à la majorité.

Encodeur à 3 entrées, 2 sorties : la sortie donne l'indice de la première entrée à 1, ou 3 si aucune entrée n n'est à 1. Discuter du cas où l'on peut garantir qu'il y a toujours une et une seule entrée à 1.

Décodeur à 3 entrées. Le circuit a 8 sorties dont une seule est à 1 (vrai), celle dont l'indice correspond au nombre binaire donné en entrée sur trois bits. Autrement dit : $S_i = ((E_2E_1E_0)_{\text{base 2}} = i)$.

Correction

E2	E1	E0	Ma j(E2,E1,E0)	Enc(E2,E1,E0)	Dec(E2,E1,E0)
0	0	0	0	11	00000001
0	0	1	0	00	00000010
0	1	0	0	01	00000100
0	1	1	1	01	00001000
1	0	0	0	10	00010000
1	0	1	1	10	00100000
1	1	0	1	10	01000000
1	1	1	1	10	10000000

Pour l'encodeur, lorsqu'il est garanti qu'une et une seule entrée est à 1, la première ligne peut être quelconque, ce qui libère un code en sortie. Les lignes où apparaissent deux 1 peuvent aussi avoir une valeur quelconque. Pour la réalisation d'un circuit, cette hypothèse supplémentaire permet, en outre, de simplifier les formules réalisant l'encodage.

Exercice 2. Unité arithmétique et logique

1. Rappeler le circuit d'un additionneur 1 bit, et de l'additionneur *n* bits.

2. À partir du circuit précédent, construire un circuit faisant des soustractions sur *n* bits.

3. Donner le dessin d'une unité arithmétique et logique pouvant faire les 4 opérations suivantes : Addition/Soustraction/Et (bit à bit)/ Opérande Gauche, au choix selon une commande (F1 F0).

Correction (indications)

1. Se reporter au paragraphe sur les circuits booléens pour retrouver les formules pour l'additionneur 1 bit. Le circuit d'addition *n* bits s'obtient en cascadant la cellule d'addition 1 bit *n* fois, le *Cin* de la première cellule – pour le bit de poids faible – valant 0, le *Cout* de

cette cellule étant relié au *Cin* de la cellule suivante, et ainsi jusqu'à la dernière cellule, pour laquelle *Cout* sert de retenue globale de l'additionneur.

2. Comme $A - B = A + B + 1$, pour obtenir un soustracteur, il suffit de prendre un additionneur avec en entrée A et B. Pour le +1, la retenue entrante de la première cellule d'addition peut être positionnée à 1 au lieu de 0. C'est tout.

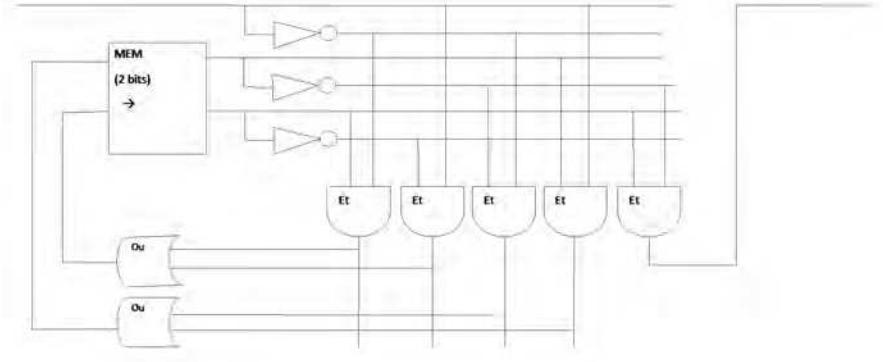
3. À partir de ce qui précède, pour réaliser une UAL faisant 4 opérations, on peut utiliser un additionneur n bits, un ET n bits, et deux multiplexeurs – un premier : 2 vers 1 sur n bits pour les entrées et un second 3 vers 1 sur n bits pour les sorties –: dans un premier étage du circuit, un multiplexeur 2 vers 1 sur n bits avec en entrée B et B sélectionne la valeur nécessaire à la réalisation du calcul demandé – si c'est une addition B, si c'est une soustraction B, sinon peu importe. Dans un second étage du circuit, le ET bit à bit de A et de B est effectué en parallèle de l'addition de A et de la sortie du multiplexeur précédent – avec en *Cin* = 0 si c'est une addition et *Cout* = 1 si c'est une soustraction. Dans un troisième étage du circuit, le second multiplexeur 3 vers 1, avec en entrée B, le résultat du ET précédent et de l'addition précédente, sélectionne en fonction de l'opération demandée le bon résultat.

Logique séquentielle

Circuit d'un petit automate

Exercice 3

1. En supposant que le circuit suivant



Circuit d'un petit automate.

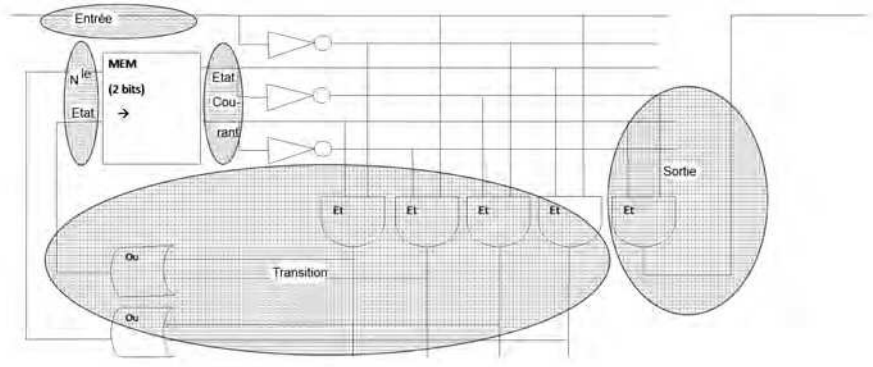
représente un automate, identifier sur le schéma les différents éléments : état courant, nouvel état, entrée, sortie, circuit de sortie et circuit de transition.

2. Donner les tables de vérité des deux circuits (sortie, transition).

3. Dessiner l'automate correspondant.

Correction

1.

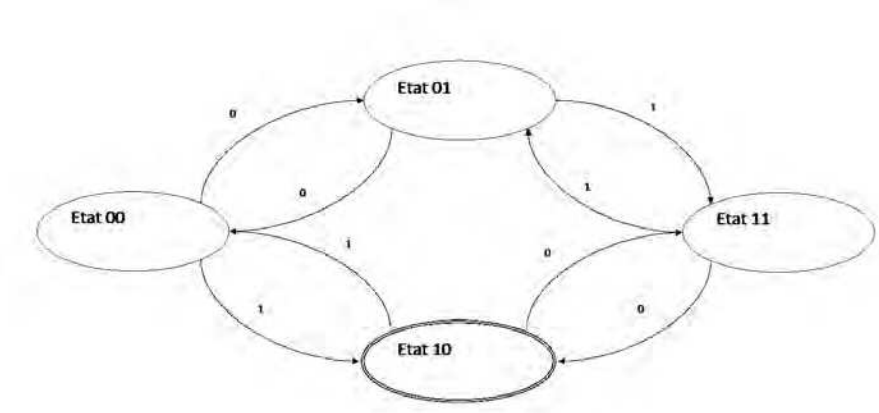


Circuit d'un petit automate

2. En nommant les fils correspondant à l'état courant de haut en bas ET0, ET1 – resp. NE0, NE1 pour le nouvel état –, les tables de vérité des circuits de transition et de sortie sont données par le tableau

ET1	ET0	Entrée	NE1	NE0	Sortie
0	0	0	0	1	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	0	0	1
1	1	0	1	0	0
1	1	1	0	1	0

3. L'automate défini a quatre états : 00 (E1=0, E0=0), 01 (E1=0, E0=1), 10 (E1=1, E0=0), 11 (E1=1, E0=1). L'état 10 est final (sortie à 1). Les transitions sont les suivantes



Modes d'adressage

Exercice 4

Le tableau ci-dessous représente l'état de certaines cases mémoire et des registres A et X avant et après les instructions données sur la colonne de gauche. Modifier les cases du tableau au fur et à mesure de l'exécution du programme en expliquant les modifications :

	X 3	A 5	M(1) 1	M(2) 4	M(3) 8	M(4) 3	M(5) 1	M(6) 2	M(8) 0
LDA3	–	–	–	–	–	–	–	–	–
STA2,X	–	–	–	–	–	–	–	–	–
LDA#2	–	–	–	–	–	–	–	–	–
ADD3	–	–	–	–	–	–	–	–	–
STA4	–	–	–	–	–	–	–	–	–
SUB1,X	–	–	–	–	–	–	–	–	–
LDX[6]	–	–	–	–	–	–	–	–	–
STX[3]	–	–	–	–	–	–	–	–	–

Correction

	X	A	M(1)	M(2)	M(3)	M(4)	M(5)	M(6)	M(8)
	3	5	1	4	8	3	1	2	0
LDA3	–	–	–	–	–	–	–	–	–
		8							
STA2,X	–	–	–	–	–	–	–	–	–
							8		
LDA#2	–	–	–	–	–	–	–	–	–
		2							
ADD3	–	–	–	–	–	–	–	–	–
		10							
STA4	–	–	–	–	–	–	–	–	–
						10			
SUB1,X	–	–	–	–	–	–	–	–	–
		0							
LDX[6]	–	–	–	–	–	–	–	–	–
	4								
STX[3]	–	–	–	–	–	–	–	–	–
									4

Ligne 1 A= M(3)=8
Ligne 2 M(2+[X])= M(5)= A= 8
Ligne 3 A= 2
Ligne 4 A =A+ M(3)= 10
Ligne 5 M(4)=A=10
Ligne 6 A=A- M(1+[X])=A- M(1+3)= 10-10=0
Ligne 7 X=M(M(6))= M(2)=4
Ligne 8 M(M(3))= M(8)=4

Un exemple complet de fonction

Exercice 5. Traduire en langage d'assemblage le programme suivant qui réalise la multiplication non signée de deux mots de 10 bits. Contraintes : on considérera que les paramètres d'entrée de la fonction sont mis dans la pile, et que le résultat de la fonction est remis en A.

```
3 /************************************
4 * Multiplication non signée 10bits *
5 ************************************/
6 public static int mul(unsigned int Na,unsigned int Nb) {
7 int i;
8 intj = 1;
9 int Nc = 0; //résultat
10
11  for (i = 0; i < 10; i = i + 1) {
```

```
12   if ((Nb & j) != 0) {Nc = Nc + Na;}// Addition
13   Na = Na + Na;   //Décalage de Na à gauche
14   j = j + j; //Calcul du masque de Nb suivant
15  }
16  return (Nc);
17 }

18 public static void main (String [] args) { //test de mul
19 int a = 5;
20 int b = 6;
21 int c;
22      c = mul(a,b);
23      System.out.println(c); // à simuler par un STA 100
24 }
```

On considérera que les instructions AND, OR et XOR n'existent qu'en mode direct. Elles réalisent l'opération bit à bit entre les 10 bits de poids faibles du mot lu et les 10 bits de poids faibles de A, avec un résultat dans A.

Pour tester les différents bits de Nb, il suffit de créer un masque égal initialement à 1, puis, à chaque itération, de doubler le masque, lignes 12 et 14. On réalisera donc *test* = *masque* AND B suivi d'un branchement conditionnel, ligne 12.

La fonction de décalage n'existant pas à proprement parler dans la machine simulée, la méthode consiste, pour réaliser le décalage à gauche, à ajouter la valeur à décaler à elle-même, lignes 13 et 14.

Correction

```
*********************************************************
* Programme en langage d'assemblage            *
* de test de multiplication compatible Machine 3    *
* Les deux valeurs sont empilées par le programme appelant *
* Celui-ci appelle la fonction MUL par un BAL  *
* Le résultat se retrouve dans le registre A au retour de fonction *
* Les numéros de ligne correspondent au texte source     *
*********************************************************
*DONNEESPP ORG 2
a RMW 1
b RMW 1
c RMW 1
SPILE RMW 1 *pointeur de sommet de pile
PILE RMW 2 *une toute petite pile
```

```
PROG ORG 15
MAIN EQU * *Début du main
     LDA #5
     STA a */ligne19
     LDX #PILE *mise de l'adresse de la pile en SPILE
     STX SPILE
     STA 0,X *a est mis en pile (ligne 22)
     LDA #6
     STA b */ligne20
     STA 1,X *b est mis en pile (ligne 22)
     ADDX #2    *Déplacement du sommet de pile
     STX SPILE

     BAL mul *Appel de mul(ligne 22)

     STA c *rangement du résultat (ligne 22)
     STA ECRAND *simulation du printf
Fin BRA Fin     *arret du programme par blocage (ligne24)

DONNEESSP EQU * *on considère qu'elles sont dans le tas.
*On aurait pu les mettre dans la pile. Simplifions.
i     RMW 1
j     RMW 1
Na   RMW 1
Nb   RMW 1
Nc   RMW 1
RET   RMW 1 *relais d'adresse de retour
SAVX RMW 1 *pour sauvegarde du registre X (A est la valeur de retour)

mul     STX SAVX
     STSV RET *Tout est sauvegardé, on peut y aller

     LDX SPILE
     SUB #2 *Pour avoir un accès simple aux valeurs empilées
     LDA 0,X *Passage d'arguments a dans NA puis b dans Nb
     STA Na
     LDA 1,X
     STA Nb
     STX SPILE *On ne touche plus à la pile
```

```
      LDA #1 */ligne 8
      STA j
      LDA #0 */ligne 9
      STA Nc
Initbcl LDA #0 * bien qu'inutile, respect méthodologie de traduction
      STA i */ligne 11
TEST LDA i
      SUB #10
      BRN SUITE *méthode sûre de traduction. D'autres sont possibles
      BRA FINBCLE
SUITELDA Nb
      AND j *on considère que cette instruction existe
      BRZ SUITE1 *aller en ligne 13
      LDA Nc
      ADD Na
      STA Nc *fin de la ligne 12
SUITE1 LDA      Na
      ADD Na
      STA Na *ligne 13
      LDA j
      ADD j
      STA j *ligne 14

INCI   LDA i *incrémentation de i et rebouclage
      ADD #1
      STA i
      BRA TEST

FINBCLE LDA Nc *Régénération des registres et retour appelant
      LDX SAVX
      BRA [RET]
```

Choix d'algorithme : calcul de la puissance

Exercice 6. Deux algorithmes sont proposés pour calculer R : la puissance n-ème d'un nombre X ($R \leftarrow X^n$).

```
Algorithme 1 :
R<-1
Pour i allant de 1 à n faire :
```

```
R<-R*X

Algorithme 2:
R<-1
Tant que n != 0 faire:
Si n est impair :
R<-R*X;N<-N-1
X<-X^2
N<-N/2
```

1. Choix de l'algorithme à implémenter. Choisir entre les deux algorithmes celui que vous comptez implémenter, donnez vos raisons.

2. Implémentation sous la forme d'un circuit à flot de données. Donner un circuit type « flot de données » réalisant l'exécution du calcul.

3. Implémentation sous la forme d'un circuit PC/PO (Partie Contrôle/Partie Opérative). En explicitant vos hypothèses sur la partie opérative PO, donner le dessin de l'automate de la partie contrôle PC d'un circuit PC/PO réalisant le calcul.

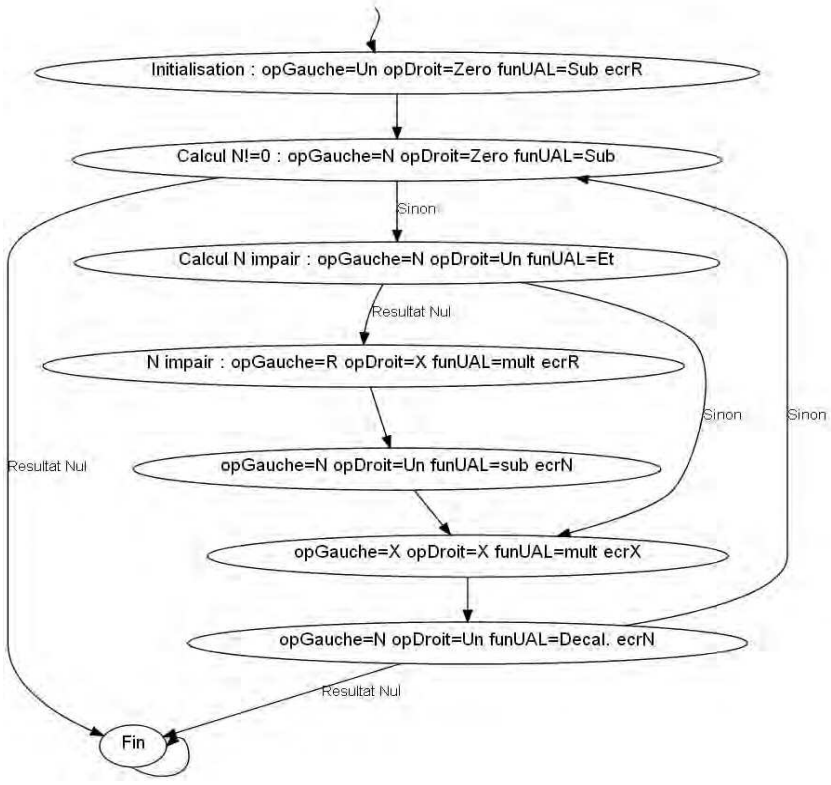
Correction (indications)

1. Pour comparer les deux algorithmes, observons leurs complexités. La complexité en place du premier algorithme sera liée à la complexité en place de la multiplication. Idem pour le second algorithme. Cependant, le second algorithme nécessite deux multiplications. La complexité en temps du premier algorithme sera proportionnelle à n et à la complexité en temps de la multiplication. Pour le second algorithme, la complexité en temps sera proportionnelle à $log(n)$ et à la complexité en temps de la multiplication. Pour la mise en œuvre, le premier algorithme semble plus simple.

2. Pour l'algorithme 2, on peut prendre le circuit donné au paragraphe « Blocs de logique séquentielle », avec comme état courant le triplet (N, R, X), comme entrée (N, 1, X), comme sortie (X, N==0) et comme fonction de transition (N, R, X) ← (si (N est pair) N-1/2 sinon N/2, si (N est pair) R*X sinon R, X/2).

3. On suppose une partie opérative PO possédant au moins 5 registres N, R, X, Zero et Un – Zero et Un sont des registres contenant les valeurs 0 et 1 –, une unité arithmétique et logique UAL permettant de faire des décalages à droite, des soustractions, des ET et des multiplications.

La partie contrôle PC est donnée par l'automate suivant.



Automate pour le calcul $R \leftarrow X^n$

Exercices non corrigés

Circuits logiques de base

Exercice 1

1. Dresser la table de vérité d'une porte OU, dresser la table de vérité du circuit NON (NON-OU(X,Y)) et comparer.

2. Dresser la table de vérité d'une porte ET, dresser la table de vérité du circuit NON-OU(NON(X),NON(Y)) et comparer.

Exercice 2

1. Concevoir un multiplexeur à 4 entrées de deux bits chacune. À partir d'un simulateur de portes, testez le circuit que vous avez conçu.

2. Réaliser un additionneur sur un bit, *A*, *B* et *Cin*, sur le simulateur proposé, dupliquez-le et testez-le sur des données de 3 bits.

Exercice 3

Réaliser et tester une mémoire vive de ce type sur simulateur à partir de bascules D et de portes.

Réalisation d'instructions de base

Exercice 4

Comment peut-on réaliser les instructions SUB #paramètre – soustraction immédiate – et SUB paramètre – soustraction en mode direct ?

Pile logicielle

Exercice 5

Ajouter au programme ci-avant les tests de débordement de pile pleine et pile vide.

Exercice 6

Le programme décrit au paragraphe « Introduction au langage d'assemblage » ne sauvegarde pas la valeur de X. Le modifier de telle sorte que cette valeur soit restituée après le retour de fonction.

Exercice 7

Créer et utiliser une fonction P1 PLUS P2 qui considère que P1 et P2 sont mis en pile et que le résultat renvoyé dans A est la somme de ces valeurs. Il est préférable de sauvegarder X !

Entrées/sorties simples

Exercice 8

Réaliser et tester un programme qui lit au clavier une valeur, appelle la fonction TROISFOIS vue ci-avant et renvoie le résultat sur l'afficheur décimal, puis reboucle sur la lecture.

Compilation manuelle

Exercice 9

Compilation d'une conditionnelle

Traduire en langage d'assemblage, en faisant ressortir l'aspect systématique, le programme suivant de calcul de la valeur absolue d'un nombre. Le résultat de la fonction sera mis dans A.

```
public static void main(String[] args)
  {
     int N,Resultat;
       // lire N au clavier;
```

```
        if (N>0) {Resultat=N;}
        else {Resultat=-N;}
        //Resultat est imprime;
}
```

Questions d'enseignement

Pour être accepté par les élèves, l'enseignement de l'architecture doit être le plus ludique possible. Cela nécessite, à notre sens, de fournir des simulateurs à tous les niveaux, de telle sorte que l'élève puisse s'affronter à la machine de simulation qui est, de ce fait, la meilleure alliée de l'enseignant.

Tout d'abord, si on en a le temps, il est utile de bien comprendre comment fonctionne un transistor : ce n'est qu'un tuyau souple qui est pincé, donc fermé, quand la tension de grille est inférieure à un certain seuil et devient conducteur, donc passant, quand la tension de seuil est dépassée. Puis il est utile de réaliser une porte avec ce transistor – un inverseur, par exemple – en montrant avec la loi d'Ohm la tension de sortie en fonction de la tension d'entrée. Ces notions permettent de montrer qu'il doit y avoir une relation entre la tension d'alimentation, la résistance de charge et la tension de pincement du transistor. Les portes réalisées sont très simples mais ont le mérite de bien montrer les problèmes technologiques liés à la réalisation de toutes les portes logiques.

La notion de porte étant acquise, on peut utiliser un simulateur et donner des challenges aux élèves, qu'ils doivent réaliser et tester sur le simulateur.

Puis l'on passe à la simulation de circuits plus complexes possédant des bus à plusieurs bits. Dans ce cas, il faut se méfier des simulateurs qui ne permettent pas l'usage de bus, car le fait d'avoir à dupliquer et à organiser les dessins devient vite rébarbatif. La simulation d'un compteur, par exemple, est une étape importante, qui permet de découvrir un tout premier circuit séquentiel.

L'utilisation d'une machine de Von Neumann simplifiée simulée permet de se familiariser avec les modes d'adressage de base, ce qui se révèle souvent difficile, et d'appréhender le fonctionnement intime de l'ordinateur grâce à la visualisation des bus et des registres. Là encore, de très nombreux défis sont possibles : si la machine simulée présente des sorties en ASCII et des entrées clavier simulées, un défi est d'écrire un programme qui affiche le prénom de l'élève, ce qui est une occasion de revoir le code ASCII, et si le simulateur possède une sortie décimale, ou même s'il n'offre que la possibilité d'analyser une case de la mémoire simulée, un défi est d'afficher la somme des n premiers entiers, ce qui demande de réaliser un programme comprenant une boucle.

Des programmes plus savants, comme la multiplication proposée en exercice corrigé, permettront de bien sentir le lien entre algorithme et architecture.

Nous ne saurions trop insister sur la nécessité de l'aspect ludique de cet enseignement, qui a fait ses preuves et permet, à la condition de laisser le temps de compréhension et d'appréhension, « vous terminerez à la maison », de faire passer ces notions qui semblent un peu complexes au premier abord.

Compléments

Quelques extensions du modèle d'architecture

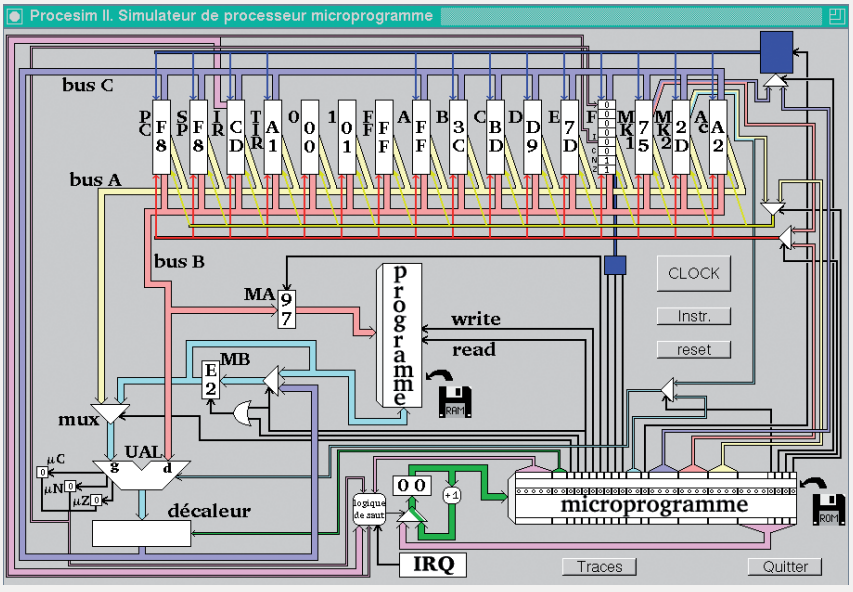
Pour gérer des entrées/sorties sans que le programme passe son temps à vérifier si elles ont été réalisées, on utilise un processus d'interruption qui, quand un signal physique arrive, réveille un « programme d'interruption ». La tâche est alors mise en sommeil pendant la durée du traitement de l'interruption. Cela nécessite naturellement une sauvegarde, au moins partielle, du contexte de la tâche interrompue, de telle sorte que celle-ci soit susceptible d'être reprise après la fin de l'interruption. Cela nécessite l'ajout de registres de sauvegarde du compteur ordinal CO spécifique aux interruptions, de même qu'un mécanisme permettant de masquer le signal d'interruption et de supprimer la demande d'interruption et les instructions spécifiques qui lui seront attachées. Une version de ce dispositif est présentée dans *Du transistor à l'ordinateur*, cité ci-avant, et les simulateurs associés à cette machine ainsi que l'assembleur se trouvent en `http://www.e-campus.uvsq.fr/dutransistoralordinateur/`.

Mis à part ce dispositif d'interruption, qui n'a pu être développé ici par manque de place, la machine décrite représente la base de toutes les machines réalisées à ce jour. Elle est capable de traiter toutes sortes de programmes. Cependant, pour obtenir des performances meilleures, il est possible d'ajouter un certain nombre d'améliorations :

– un nombre plus important de registres généraux de façon à permettre à l'utilisateur de conserver dans ses registres des données intéressantes sans être obligé d'effectuer des opérations incessantes de transferts de et vers la mémoire de données,

– un ou plusieurs registres de pointeurs de pile spécialisés, qui peuvent dans certains cas posséder des fonctions matérielles d'auto-incrémentation et d'auto-décrémentation.

Un exemple de simulateur de machine de ce type est le suivant.



Exemple de simulateur de machine RISC à banc de registres.

On y retrouve la séparation entre une partie opérative, avec un chemin de données comportant le banc de registres, la mémoire et l'UAL, et une partie contrôle, avec, dans le quart en bas à droite, le microprogramme commandant la partie opérative avec le numéro de la micro-instruction en cours d'exécution – 00 sur la copie d'écran. Une vidéo du fonctionnement de cette machine est présentée en `http://www.noe-kaleidoscope.org/public/people/DenisB/Enseignement/Architecture/Demo-Procesim.htm`. Même si ce fait est difficilement visible à la lecture des programmes en langage d'assemblage, les machines actuelles utilisent de plus une structure permettant un accès simultané à deux mémoires : une mémoire de programme et une mémoire de données, appelées *machines de Harvard*, ce qui divise par deux le nombre de cycles nécessaires pour un LDA par exemple. Cette transformation s'opère souvent de façon invisible à l'utilisateur grâce à l'utilisation d'un cache de données et d'un cache de programme qui permettent non seulement un accès simultané, mais aussi un temps d'accès simulé à la mémoire beaucoup plus court.

L'unité arithmétique et logique UAL décrite ici est extrêmement simple et ne réalise que des opérations d'addition, de soustraction et quelques fonc-

tions logiques, ET, OU, etc. Il est naturellement possible de l'étendre en ajoutant des opérations de décalage, mais aussi des opérations câblées de multiplication et de division, voire de calcul flottant. Pour plus d'information sur ce domaine, nous renvoyons au livre *Structured Computer Organization* d'Andrew Tanenbaum.

Enfin, les micro-instructions réalisées ici de façon séquentielle peuvent parfaitement être engendrées de façon câblée, et même être pipelinées, c'est-à-dire que plusieurs instructions peuvent se dérouler simultanément dans la machine : pendant qu'une instruction est en phase d'exécution, il y a de nombreux cas où l'instruction suivante peut être décodée par exemple. Le gain en temps est alors d'un facteur 2 ; en revanche, ce type de réalisation demande parfois un doublement d'une partie du chemin de données.

Pour aller encore plus vite : des machines parallèles

Les circuits d'un processeur ne peuvent aller plus vite que la vitesse permise par la technologie. Or, il est nécessaire pour certaines applications – comme le calcul scientifique ou le calcul embarqué, par exemple les applications radar – d'aller encore beaucoup plus vite, et donc de paralléliser les traitements : si l'on considère que, sur un chantier, un maçon monte un mur en 6 heures, quatre maçons peuvent soit aller quatre fois plus vite pour monter un mur, soit monter quatre murs dans le même temps. Naturellement, certains problèmes se laissent paralléliser mieux que d'autres.

De nombreux types d'architecture de machines utilisant cette idée ont été conçus. Cela nécessite que les processeurs travaillent sur une ou plusieurs tâches communes, partagent les données et le code, et surtout se synchronisent entre eux.

La complexité de telles machines dépasse de beaucoup le but de ce chapitre d'introduction, et la complexité des méthodes utilisées par les compilateurs est plus grande encore. Nous renvoyons le lecteur intéressé au livre *Computer Architecture : A Quantitative Approach*, de John L. Hennessy et David A. Patterson.

Qu'est-ce qu'un système d'exploitation ?

Un ordinateur tel que celui que nous avons décrit dans ce chapitre serait très difficile à utiliser s'il n'était pas équipé de programmes qui permettent

– d'exécuter « en même temps » plusieurs programmes, appartenant ou non à plusieurs utilisateurs,

– de gérer les entrées/sorties à un haut niveau, par exemple, non pas allumer un pixel de l'écran, mais écrire une lettre dans une fenêtre,

– d'organiser les informations présentes sur le disque en une arborescence de fichiers,

– d'utiliser une mémoire plus grande que la mémoire physique de l'ordinateur,

– d'utiliser un ensemble d'instructions plus grand que celui du processeur de l'ordinateur.

Ces différents programmes, qui présentent au programmeur une machine plus évoluée que la machine réelle, forment le *système d'exploitation* : Windows, Unix, Linux, MacOS…

La notion clé pour comprendre le fonctionnement d'un système d'exploitation est celle d'*interruption* : une interruption lance des tâches en fonction d'un signal externe ou interne à la machine.

Un système multitâches ou multi-utilisateurs

Les codes de plusieurs programmes ou tâches prêtes à être exécutées se trouvent dans la mémoire vive. Imaginons alors qu'un générateur externe au processeur envoie périodiquement des interruptions. À chaque interruption, le programme d'interruption examine, parmi l'ensemble des tâches à exécuter en attente du processeur, celle qui est prioritaire. Au lieu de revenir directement à la tâche interrompue, il range le contexte de celle-ci dans l'ensemble des contextes des tâches en attente. Cela permet donc à chaque interruption de choisir la tâche qui doit utiliser le processeur.

Tous les systèmes d'exploitation comme UNIX, Windows, etc. utilisent ce principe pour partager une ressource fondamentale, le processeur, entre les différentes tâches en cours d'exécution.

L'algorithme choisissant, entre les différentes tâches, celle qui est exécutée à un instant donné dépend de la priorité affectée à telle ou telle tâche et des critères d'optimisation choisis.

Avoir plusieurs tâches pour un même utilisateur n'est souvent utile que si elles peuvent communiquer entre elles ; ces communications supposent un partage des données, mais aussi des moyens de synchronisation. Par exemple, dans un programme de calculette, il peut y avoir une tâche dévolue à la lecture du clavier, une tâche de traitement et une tâche d'affichage qui fonctionnent simultanément. Tant que les données ne sont pas entrées au clavier, il n'est pas nécessaire de lancer la tâche de calcul qui se met donc en attente et ne consomme aucune ressource processeur pendant ce temps.

Les grosses machines peuvent être partagées entre plusieurs utilisateurs. Dans ce cas, le même principe s'applique aux différentes tâches de tous les utilisateurs simultanés. L'un des problèmes à résoudre est alors l'étanchéité

totale qu'il doit y avoir entre les données de deux utilisateurs simultanés. Ils doivent se sentir parfaitement indépendants les uns des autres et dans un environnement sûr – les autres utilisateurs n'apprécieraient pas que vous modifiiez leurs données sans leur consentement !

Apports du système au niveau des entrées/sorties

Nous avons vu qu'il est possible de lire et d'écrire dans des registres de périphériques. Cependant, si nous considérons un disque dur par exemple, celui-ci est constitué d'un système mécanique constitué du disque lui-même et d'un bras muni de têtes de lecture – assez analogue à celui d'un pick-up pour disques vinyles – auquel on donne l'adresse – position en `r` – d'un bloc de données à lire ou à écrire. Le temps de réponse d'un tel système est de l'ordre de la dizaine de millisecondes, ce qui est très lent par rapport à la vitesse de la mémoire de l'ordinateur – de l'ordre de la nanoseconde. Le processeur envoie alors souvent une requête au disque puis, au lieu d'attendre le résultat de cette requête, met en sommeil cette tâche, en attente de la réponse, et effectue une autre tâche plus prioritaire.

Pendant ce temps, le disque effectue le travail qui lui a été demandé et transfère à son rythme les données entre le support – magnétique ou optique – et la mémoire du processeur. Quand cela est terminé, il signale au processeur, grâce à une interruption, la fin du transfert, ce qui a pour effet de réveiller la tâche endormie.

Le système de gestion de fichiers

Une partie du système d'exploitation appelée « système de gestion de fichiers » effectue pour l'utilisateur

– la vérification de l'existence d'un fichier de nom donné,

– la transformation entre le nom du fichier et son adresse physique sur l'ensemble des périphériques connectés sur une machine – ce qui permet de masquer les propriétés physiques de tel ou tel périphérique,

– la gestion des droits d'accès — le système vérifie que le programme utilisateur a le droit de lire, d'écrire ou d'exécuter le fichier sélectionné,

– certaines optimisations de performance des accès demandés.

La mémoire virtuelle

Une des fonctions du système d'exploitation est de laisser croire à chacun des utilisateurs que la mémoire qui lui est attribuée est bien plus grosse que la mémoire physique de la machine. L'idée est simple : stocker le programme et les données de chaque utilisateur, par exemple sur les disques durs de la

machine qui sont de très grande taille, et aller rechercher ces données au moment où l'on en a besoin.

Mais, comme nous l'avons vu, les disques durs sont très lents par rapport à la vitesse de la mémoire interne du processeur. Le temps d'accès à la mémoire est de l'ordre de la nanoseconde, alors que celui d'un bon disque dur est de l'ordre de la milliseconde, d'où un rapport de vitesses de l'ordre du million. Et on ne peut naturellement pas se permettre une chute de performance d'un facteur un million.

Une solution est de copier en mémoire vive, non une case mémoire unique, mais des blocs de données contiguës, l'hypothèse étant que, lorsque l'on vient d'utiliser une donnée, il est probable que l'on utilise les données contiguës dans un avenir proche. Cette hypothèse s'appelle la *localité spatiale*. Une autre hypothèse est que, lorsqu'on utilise une case mémoire, par exemple une instruction, les programmes étant très souvent constitués de boucles, il est probable que l'on réutilise cette même case mémoire dans un avenir proche. Cette hypothèse s'appelle la *localité temporelle*.

Sachant que les disques durs transfèrent des blocs de données de taille fixe, appelés *pages*, on peut découper la mémoire, ou une partie de celle-ci, en « cadres » de la taille d'une page, et utiliser un mécanisme qui effectue la transformation de l'adresse logique, appelée le plus souvent *adresse virtuelle*, en une adresse physique en mémoire. Cela peut être réalisé par un dispositif matériel appelé MMU (*Memory Management Unit*) qui transforme les adresses logiques en adresses physiques et qui, en cas de défaut, lance un programme d'interruption spécifique qui gère les pages, entre les disques et les cadres.

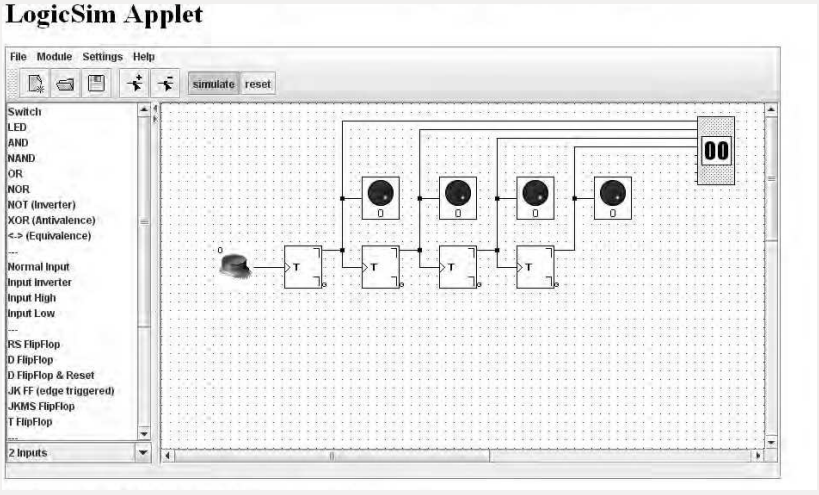
L'extension de l'ensemble des instructions

Une machine donnée détecte et exécute directement un jeu d'instructions assez réduit. En revanche, elle détecte aussi les instructions inexistantes éventuelles. Cette détection engendre un signal d'interruption qui lance un programme spécifique.

Cette méthode permet, par exemple, de traiter par logiciel des « instructions » trop complexes pour la machine câblée. C'est ainsi qu'un processeur ne possédant pas d'opérateur flottant, par exemple, peut détecter les instructions flottantes et les faire traiter par un programme spécifique qui se substitue alors au processeur de base. Comme pour toute interruption, cette opération s'effectue à l'insu, ou presque, de la tâche principale. Du point de vue de l'utilisateur, seul le temps d'exécution sera donc différent entre deux machines possédant tel type d'instruction câblée ou programmée.

Quelques simulateurs permettant de se familiariser avec la logique et les processeurs

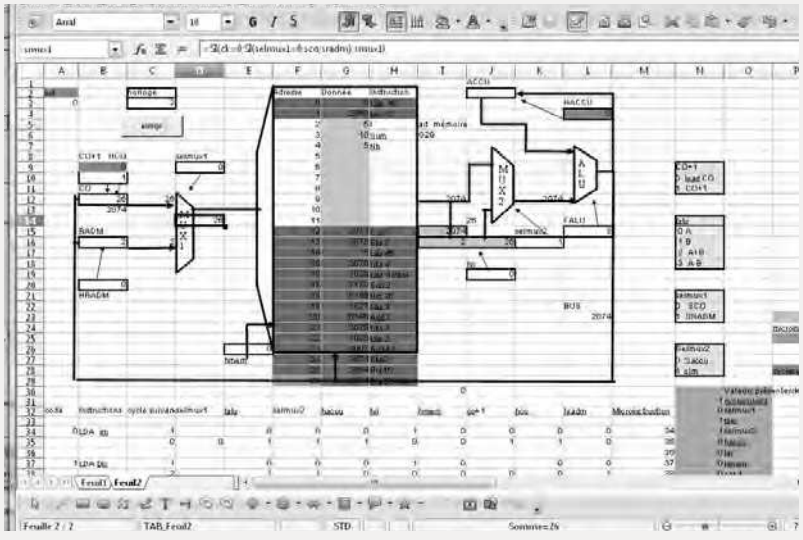
De nombreux simulateurs de logique simple, commerciaux et non commerciaux, existent sur le Web. On citera des simulateurs écrits en Java comme LogicSim `http://www.tetzl.de/java_logic_simulator.html`, facile d'emploi, mais limité en taille à de petits circuits de quelques dizaines de portes.



Exemple de simulateur de portes logiques écrit en Java (LogicSim de Tetzl).

Des simulateurs écrits avec des tableurs: `http://www.e-campus.uvsq.fr/dutransistoralordinateur/`.

Au niveau des simulateurs de machine on trouve des simulateurs écrits en C, en Java ou avec des fonctions de tableurs `http://www.e-campus.uvsq.fr/dutransistoralordinateur/`.



Exemple de simulateur de machine sur tableur.

Pour en savoir plus

Joffroy Beauquier et Béatrice Bérard, *Systèmes d'exploitation : concepts et algorithmes*, Ediscience international, 1994.

John L. Hennessy et David A. Patterson, *Computer Architecture : A Quantitative Approach*, Morgan Kaufmann, 1996.

Jean-Michel Muller, *Arithmétique des ordinateurs*, Masson, 1989.

Andrew Tanenbaum, *Structured Computer Organization*, Prentice-Hall, 2005.

Andrew Tanenbaum, *Systèmes d'exploitation*, Pearson Education, 2008.

Claude Timsit, *Du transistor à l'ordinateur*, Hermann, 2010.