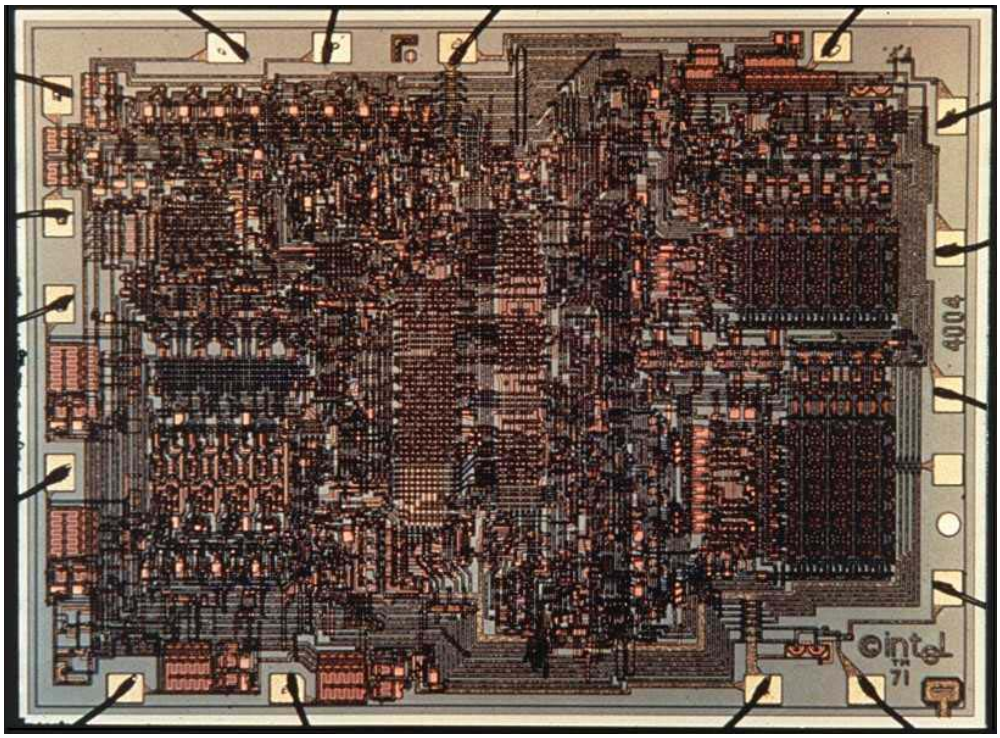
LE MICROPROCESSEUR

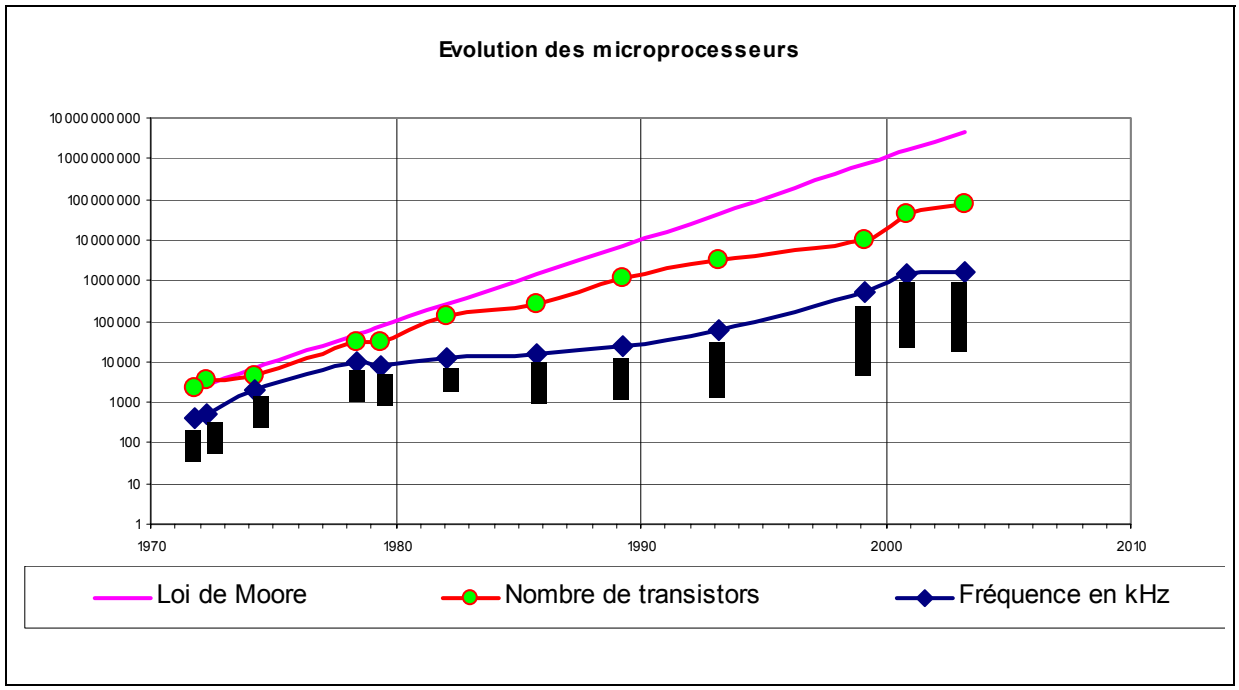
Dans ce chapitre, nous allons construire ce qui constitue le cœur de tout micro-ordinateur : un microprocesseur. Nous utiliserons les composants déjà étudiés, qui pouvaient sembler *a priori* parfaitement dénués de tout intérêt, pour les unir en un ensemble simple, mais complet et utile : nous câblerons humblement, mais dans le détail, un microprocesseur imaginaire 8 bits (certes inspiré du Z80 qui permit, avec quelques autres, l'éclosion de la micro-informatique au début des année 80), tel que ceux qui équipaient les tout premiers micro-ordinateurs grand public. Le but de notre microprocesseur n'étant pas la performance, mais la pédagogie, nous n'aborderons que dans les principes les évolutions qui ont conduit à l'émergence des bêtes de course d'aujourd'hui (mais qui seront à leur tour considérées comme préhistoriques d'ici quelques années…).
De la même manière que pour l'UAL, l'ensemble des circuits que nous utiliserons ne sont pas construits sur des puces distinctes reliées par des fils, mais sont intégrés sur le même morceau de silicium, pour former une puce unique appelée microprocesseur.

UN PEU D'HISTOIRE :

Nous sommes en novembre 1971.

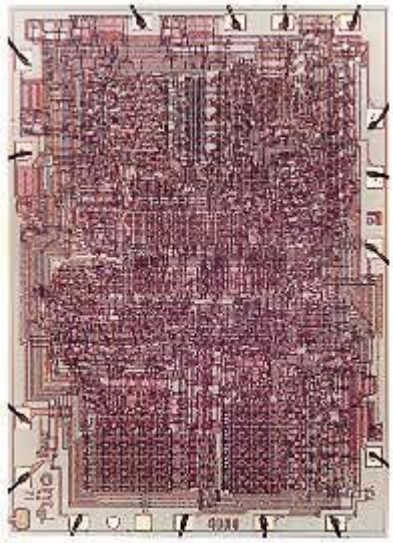
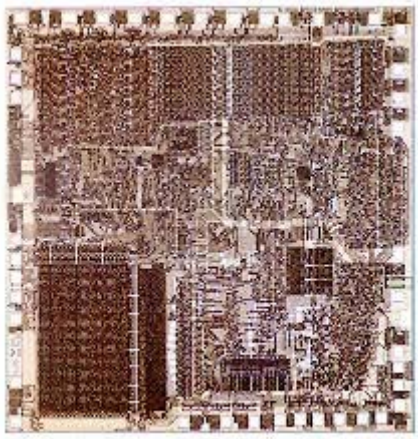
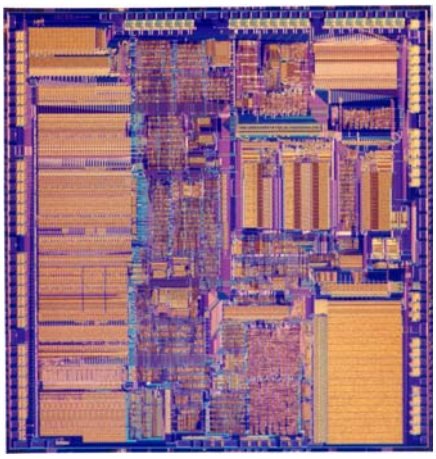
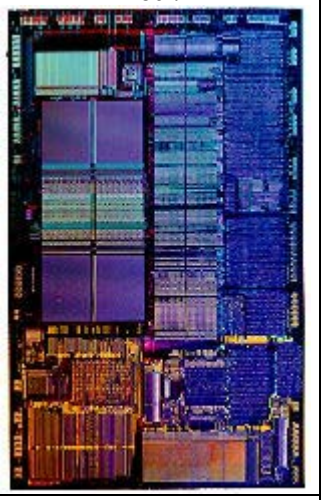
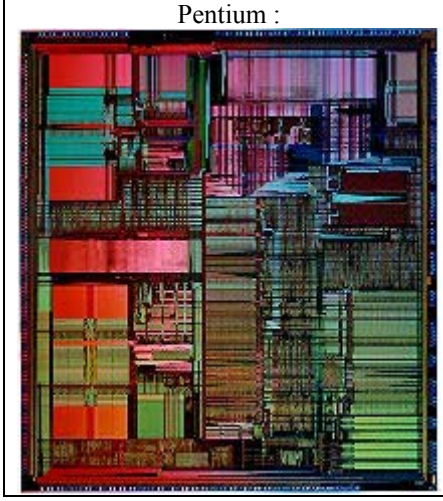
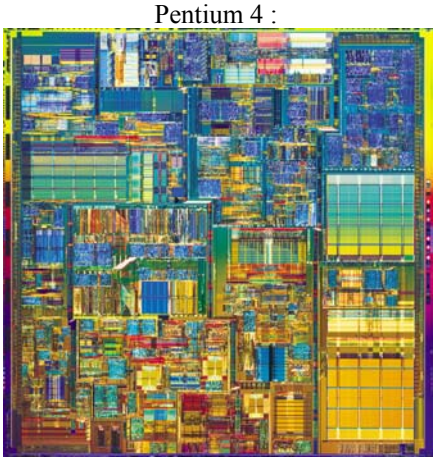


Voici le premier microprocesseur d'Intel : la puce 4004, avec ses 108 kHz et son bus de 4 bits. Le prix initial était de $200. Sa capacité de calcul de 60000 opérations par seconde. Il était constitué de 2300 transistors et basé sur une technologie de 10 microns. Il pouvait adresser 640 octets. La matrice de la puce mesurait 3 × 4 mm.
Le graphe suivant montre l'évolution qui a eu lieu depuis :



La loi de Moore est une prévision décrite par Gordon Moore, un des fondateurs d'Intel, qui prévoyait en 1965 que le nombre de transistors dans un microprocesseur doublerait tous les 18 mois. Si l'expérience montre qu'il s'agit plutôt de 24 mois, il n'empêche que la croissance exponentielle s'est depuis toujours confirmée.

Apres le 4004, Intel a sorti le 8080. Quelques ingénieurs de chez Intel ont quitté alors cette société pour fonder Zilog et sortir le Z80, copie améliorée du 8080. Intel a continué avec les 8086-8088 (qui équipa les premier IBM PC), 80286, 80386 (32bits multitâches), 80486 (coprocesseur mathématique intégré), puis la série des Pentium.

4004 : 	8088 : 	
286 : 	386 : 	486 : 
Pentium : 	Pentium 4 : 	

LE CONTROLE

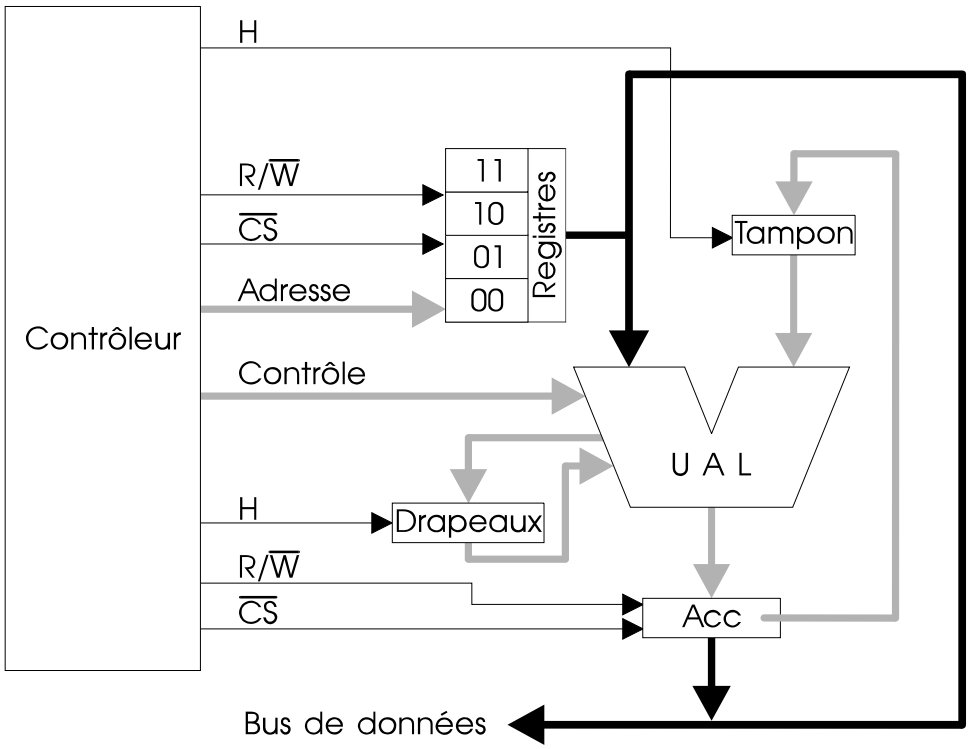
Cas d'une addition

Nous devons faire fonctionner une UAL. Dans le cas qui nous intéresse, cela consiste à :

a) inscrire les bonnes opérandes, issues de deux registres, à ses entrées,
b) inscrire alors le bon code sur son bus de contrôle,

c) récupérer le résultat à sa sortie,
d) enregistrer ce résultat dans un registre, de manière à ce qu'il ne soit pas perdu à la prochaine opération.

L'architecture du microprocesseur juste nécessaire à ces opérations peut être représentée comme suit :



L'UAL est souvent représentée comme dans le schéma ci-dessus : une sorte de V empâté. Elle reçoit :

- en opérande1 : un registre appelé *Acc* (comme ACCumulateur), matérialisé par une mémoire vive statique à une seule adresse (d'où l'absence de bus d'adresse, inutile). Une bascule D flip-flop (nommée ici tampon), dont les entrées sont prises directement au niveau des cellules mémoire de Acc (en by-passant les commandes de lecture/écriture de Acc), permet la mise à jour de l'opérande1 en copiant les données de Acc lorsqu'elle est sollicitée, et maintient ces information sur opérande1 sinon (quelle que soient alors les modifications apportées à Acc par la suite).
- en opérande2 : une série de registres, matérialisée par une mémoire vive statique. Chaque adresse de cette mémoire contient un registre. Les chiffres indiquent ces adresses en binaire (dans ce cas, le bus d'adresses de cette mémoire comprend deux bits).

Elle fournit :

- son résultat à Acc. De par sa position privilégiée, Acc est le registre spécialisé dans les calculs, qu'ils soient arithmétiques ou logiques : en effet, Acc est toujours opérande, et toujours résultat.
- des drapeaux, enregistrés ou pas par une bascule D (appelée ici *Drapeaux*) selon l'ordre qui lui est donné. Cette bascule sera appelée souvent *registre F* (F comme Flag = drapeau), même s'il ne s'agit pas à proprement parler d'un registre.

Les traits gras et grisés indiquent les bus distincts du bus de données. Cela est évident pour la bascule D. Pour l'Acc, la mémoire diffère d'une mémoire vive classique (schématisée dans le chapitre "Mémoire vive") en ce que le bus de données en entrée de mémoire (servant pour l'écriture) et celui en sortie (servant pour la lecture) ne sont pas reliés, mais distincts.
Par ailleurs, aucun des registres ne comporte de signal *Prêt*. Nous verrons plus tard pourquoi.
Le but du contrôleur est de générer les bonnes valeurs sur les bons fils au bon moment, de manière à ce que l'opération réussisse. Il a 12 bits à générer :

- H_{tampon} : 1 bit
- $R/W_{mémoire}$: 1 bit
- $CS_{mémoire}$: 1 bit
- $Adresse_{mémoire}$: 2 bits
- $Contrôle_{UAL}$: 4 bits

- $H_{drapeaux}$: 1 bit
- R/W_{Acc} : 1 bit
- CS_{Acc} : 1 bit

Ces bits constituent un "mot" (ici de 12 bits), appelé une *micro-instruction*, dont voici la forme binaire :

	Micro-instruction												
Composant	Tampon Op1	Mémoire				UAL				F	Acc	
Signal	H	R/~~W~~	~~CS~~	Adresse		Contrôle				H	R/~~W~~	~~CS~~
Bits (exemple)	0	1	0	1	1	0	0	1	0	0	1	1

Supposons que nous voulions additionner les valeurs situées aux registres Acc et 11. Il nous faut :
- Transférer Acc vers opérande1 via le tampon.
- Lire les cellules mémoire du registre 11 pour les mettre sur l'entrée Opérande2 de l'UAL
- Indiquer à l'UAL qu'il faut réaliser une addition
- Ecrire le résultat dans Acc

On voit que si ces opérations sont réalisées simultanément, on va en même temps lire dans Acc (par le tampon), et y écrire une valeur différente (le résultat de l'opération). On ne maîtrise donc pas du tout la valeur qui est transférée à l'opérande1, et donc pas non plus le résultat : on a un système bouclé. En revanche, rien n'empêche, pendant qu'on transfère Acc vers l'opérande1, de transférer le registre 11 vers l'opérande2, et de programmer l'UAL. Il faut donc procéder en plusieurs temps :
- Temps 1 :
  - transférer Acc vers opérande1 via le tampon.
  - transférer le registre 11 vers l'opérande2.
  - indiquer à l'UAL qu'il faut réaliser une addition
- Temps 2 :
  - figer l'opérande1
  - enregistrer les drapeaux
  - écrire le résultat de l'UAL dans Acc
- Temps 3 :
  - isoler toutes les mémoires (mémoire et Acc) de manière à être prêt pour l'opération suivante.

Il suffit donc de faire générer successivement les bonnes micro-instructions au contrôleur :
Temps 1 : 1 1 0 11 0111 0 0 1, soit 110110111001
Temps 2 : 0 1 0 11 0111 1 0 0, soit 010110111100
Temps 3 : 0 1 1 11 0111 0 0 1, soit 111110111001

C'est beau, non ? C'est beau, mais faux ! En effet, nous devons tenir compte des temps de propagation des signaux : le basculement des transistors (tiens, c'est vrai : nous étions partis de transistors !…) de l'état 1 vers l'état 0, ou l'inverse, n'est pas instantané, même s'il est court. Ainsi, au sujet de l'opération du temps 3, on peut se poser la question : lequel des signaux $\sout{CS}_{mémoire}$=1 ou $\sout{CS}_{Acc}$=1 sera exécuté en premier ? Si c'est $\sout{CS}_{Acc}$=1, c'est un moindre mal, mais si c'est $\sout{CS}_{mémoire}$=1, l'UAL verra une valeur erratique en opérande2, et fournira donc un résultat fantaisiste à l'Acc avant que celui-ci ne quitte l'état d'écriture. Le résultat sera loufoque.
Pour les mêmes raisons, lorsqu'on active une mémoire, il est préférable de mettre à jour le signal R/~~W~~ avant le signal ~~CS~~ : en effet, si ces mises à jour sont programmées de manière simultanée, on ne sait pas *a priori* lequel sera pris réellement en compte en premier par la mémoire. Si c'est le signal ~~CS~~, il risque d'y avoir une opération "pirate" d'écriture (ou lecture suivant le cas) avant l'opération de lecture (respectivement d'écriture) souhaitée. Il est même possible que le traitement réel diffère selon le bit considéré !

La suite des opérations devient donc :
- Temps 1 :
  - transférer Acc vers opérande1 via le tampon.
  - transférer le registre 11 vers l'opérande2.
  - indiquer à l'UAL qu'il faut réaliser une addition
- Temps 2 :
  - enregistrer les drapeaux

- écrire le résultat de l'UAL dans Acc

- Temps 3 :
  - isoler Acc, de manière conserver la valeur quel que soit le devenir du bus de données (lié à l'opérande2).
- Temps 4 :
  - isoler la mémoire, de manière à rendre disponible le bus de données aux opérations suivantes.

Les bonnes micro-instructions à faire générer successivement au contrôleur deviennent :
Temps 1 : 1 1 0 11 0111 0 0 1, soit 110110111001
Temps 2 : 0 1 0 11 0111 1 0 0, soit 010110111100
Temps 3 : 0 1 0 11 0111 0 0 1, soit 010110111001
Temps 4 : 0 1 1 11 0111 0 0 1, soit 011110111001

Nous venons donc de réaliser une opération à partir de données non pas fugitives, mais enregistrées, et nous avons également assuré la pérennité du résultat.
Nous aurions très bien pu réaliser n'importe quelle opération disponible dans l'UAL, en changeant les seuls bits du bus de contrôle de l'UAL. De même, l'opérande2 aurait pu être un autre registre, par simple modification des deux bits d'adresse du registre. Par contre, dans tous les cas, l'opérande1 est Acc, et le résultat est stocké dans Acc. Qu'on se le dise…

Cas de deux transferts

De la mémoire vers l'accumulateur

Le cas précédent était tout de même très confortable : les valeurs que nous voulions additionner étaient exactement là où on les attendaient, notamment une des deux opérandes se situait justement dans l'accumulateur ! Cela se passe rarement de cette manière dans la réalité : il faut alors préalablement amener l'opérande1 dans l'accumulateur. En supposant que l'opérande1 soit située dans le registre 01, nous allons le transférer dans l'accumulateur.
Nous pourrions nous servir seulement du bus de données (lire le registre 01 pour générer sa valeur sur le bus de données, alors que l'accumulateur écrit ce qu'il y a sur le bus dans sa mémoire), mais l'accumulateur ne sait enregistrer que ce qui sort de l'UAL. Nous allons donc passer par celle-ci, qui possède justement (quelle coïncidence !) une opération où le résultat est l'opérande2 (code de contrôle 0000b). De la même manière que précédemment, nous définissons la valeur des signaux que doit générer le contrôleur. La procédure est la suivante :

- Temps 1 :
  - transférer le registre 01 vers l'opérande2
  - indiquer à l'UAL qu'il faut réaliser l'opération 0000
  - écrire le résultat de l'UAL dans Acc
- Temps 2 :
  - isoler Acc, de manière à conserver la valeur, quel que soit le devenir du bus de données (lié à l'opérande2).
- Temps 3 :
  - isoler la mémoire, de manière à rendre disponible le bus de données aux opérations suivantes.

On remarque que, contrairement à précédemment, on se permet d'écrire le résultat dans Acc dès le début : en effet, on n'effectue ici aucune lecture des valeurs de Acc, ce qui écarte tout risque de bouclage de données.
Cependant, un autre aspect de la non-instantanéité des basculement de transistors est à prendre en compte : les composants que nous utilisons (mémoires, additionneurs, UAL proprement dite...) présentent de multiples transistors câblés en cascade. Pour donner un exemple à taille humaine, dans l'incrémenteur (tel que celui vu dans le chapitre "Circuits arithmétiques"), le résultat de la colonne des unités sera calculé avant celui des "dizaines" (à cause de la retenue), lui-même calculé avant celui des "centaines", etc… Ainsi, même si à tout instant la sortie de l'incrémenteur présente un résultat, celui ci sera faux tant que les signaux d'entrée ne se seront pas propagés à travers tous ses transistors. Seulement alors le résultat sera fiable. Il y a donc une réelle propagation de l'information valide.
Dans notre cas, l'Acc en mode écriture verra plusieurs valeurs affichées à son entrée pendant le temps 1. Cela n'est ici pas gênant, dans la mesure où le mode d'écriture prendra fin lors du temps 2. Or, la valeur qui sera retenue par l'Acc sera la dernière qu'il aura vue pendant tout son passage en mode écriture, donc la bonne. En effet, la durée séparant deux temps est calculée à partir des temps de propagation de l'information valide à travers les composants, pour justement assurer la fiabilité des données.

Les bonnes micro-instructions à faire générer successivement au contrôleur deviennent :
Temps 1 : 0 1 1 01 0000 0 0 0, soit 011010000000
Temps 2 : 0 1 1 01 0000 0 0 1, soit 011010000001
Temps 3 : 0 1 0 01 0000 0 0 1, soit 010010000001

Les drapeaux n'ont volontairement pas été mis à jour : ceux-ci ne le sont que sur des opérations arithmétiques ou logiques. Or, nous avons affaire ici à un simple transfert.

De l'accumulateur vers la mémoire

Après le calcul, si nous voulons libérer réellement l'UAL pour d'autres opérations, il va falloir libérer l'accumulateur, qui ne sait pas enregistrer plusieurs valeur : nous allons donc transférer le résultat depuis l'accumulateur vers le registre 01. L'Acc sachant transférer sa valeur sur le bus de données, nous pouvons maintenant utiliser celui-ci directement :

- Temps 1 :
  - transférer l'Acc vers le bus de données.
  - écrire le contenu du bus dans le registre 01
- Temps 2 :
  - isoler le registre 01, de manière conserver la valeur quel que soit le devenir du bus de données.
- Temps 3 :
  - isoler l'Acc, de manière à rendre disponible le bus de données aux opérations suivantes.

Les bonnes micro-instructions à faire générer successivement au contrôleur deviennent :
Temps 1 : 0 0 0 01 0000 0 1 0, soit 000010000010
Temps 1 : 0 0 1 01 0000 0 1 0, soit 001010000010
Temps 1 : 0 0 1 01 0000 0 1 1, soit 001010000011

Opération globale

En réalisant ces trois opérations dans le bon ordre et à la suite :

- Transfert du registre 01 vers l'Acc
- Addition de l'Acc avec le registre 11, résultat dans Acc
- Transfert de l'Acc vers le registre 01,

nous réalisons au final l'addition des registres 01 et 11, en mettant le résultat dans le registre 01. Nous avons additionné la valeur du registre 11 dans le registre 01.
Aussi, maintenant que ces trois opérations ont été construites, on aimerait bien pouvoir les réutiliser à la demande. Dans un premier temps, et afin de faciliter les explications, nous allons les nommer :

- le transfert du registre *X* vers Acc sera noté : LD Acc,*X* (de "LoaD" : charger)
- une addition sans retenue de Acc et le registre *X*, (avec le résultat dans Acc) sera notée : ADD Acc,*X*.
- le transfert de Acc vers le registre *X* sera noté : LD *X*,Acc

De plus, pour plus de commodité, nous nommons les registres. Ainsi, par convention :

- l'adresse 00 est appelée registre B
- l'adresse 01 est appelée registre D
- l'adresse 10 est appelée registre H
- l'adresse 11 est appelée registre M

Ainsi, la suite d'opérations que nous avons réalisée ci-dessus se note :
LD Acc, D
ADD Acc, M
LD D,Acc

Remarquez que les suites de micro-instructions générées par le contrôleur pour les opérations *LD Acc,D* et *LD Acc,M* (par exemple) ne diffèrent que par les bits correspondant au bus d'adresse de la mémoire de registres. De même, en modifiant, dans les suites de micro-instructions générées par le contrôleur, les bits correspondant au bus de contrôle de l'UAL, on effectue d'autres opérations que l'addition. Ainsi, nous noterons les différentes opérations (où *X* est un registre au choix) :

Bus de contrôle de l'UAL	Résultat obtenu	Notation	Signification de la notation
0000	X		
0001	X + 1	INC X	INCrement
0010	X - 1	DEC X	DECrement
0011	NON X	NON X	Non
0100	Acc AND X	AND Acc,X	ET
0101	Acc OR X	OR Acc,X	OU
0110	Acc XOR X	XOR Acc,X	XOU
0111	Acc + X	ADD Acc,X	ADDition
1000	Acc + X + retenue	ADC Acc,X	ADdition avec Carry (Carry = retenue)
1001	X + retenue	INCC X	INCrement avec Carry
1010	Rotation gauche X	RL X	Rotation Left
1011	Rotation gauche X avec retenue	RLC X	Rotation Left avec Carry
1100	Rotation droite X	RR X	Rotation Right
1101	Rotation droite X avec retenue	RRC X	Rotation Right avec Carry
1110	Décalage gauche X	SL X	Shift Left
1111	Décalage droite X	SR X	Shift Right

Remarques :

- Acc est l'opérande1 des opérations à 2 opérandes
- Acc reçoit toujours le résultat
- La notation de chaque opération est appelée *mnémonique* : il permet de se souvenir du sens d'une écriture, le lien entre l'écriture d'une opération et l'opération elle-même n'étant qu'une convention.
- Le cas 0000 ne porte pas de nom : il ne s'agit pas d'une opération, mais de l'utilisation "frauduleuse" de l'UAL pour un transfert de l'entrée vers la sortie.

Aussi, nous avons vu quels signaux envoyer, nous avons vu comment nommer les paquets de micro-instructions, mais nous n'avons toujours pas vu comment les générer. Non mais dites-moi, dites-moi : que cache donc le contrôleur derrière son aspect blafard et ingrat ?

LE CADENCEMENT

Il faudrait enregistrer définitivement les micro-instructions qui définissent chaque instruction, et pouvoir les rappeler à tout moment. Nous avons dit "Enregistrer définitivement" ? Nous devons donc naturellement utiliser une ROM (mémoire morte). Prenons par exemple le cas des instructions :

- ADD Acc, M
- LD Acc, D
- LD D,Acc
- AND Acc,H

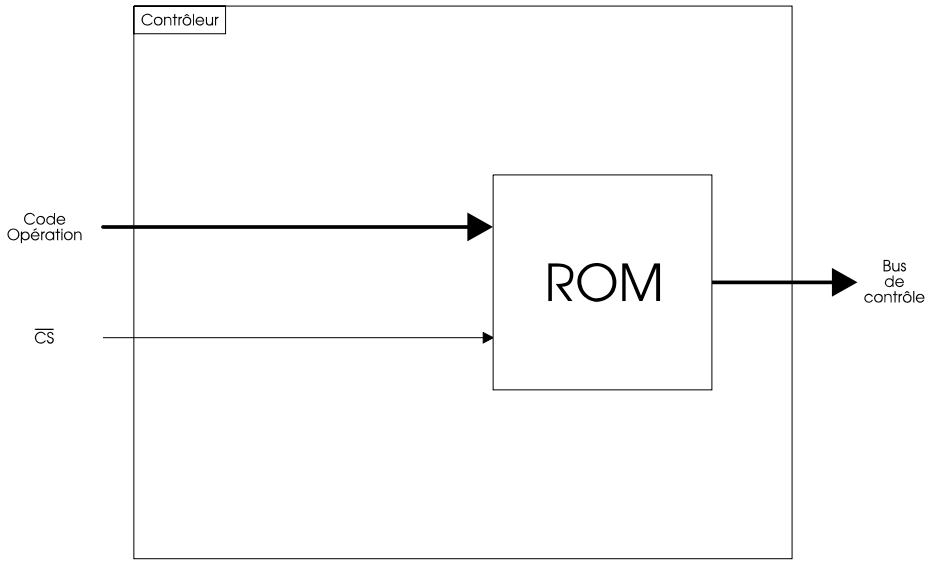
Nous pouvons les enregistrer (à la construction) dans une ROM :

Opération	Adresse ROM	Tampon Op1	Mémoire				UAL				Drapeaux	Acc	
		H	R/~~W~~	~~CS~~	Adresse		Contrôle				H	R/~~W~~	~~CS~~
ADD Acc,M	0000	1	1	0	1	1	0	1	1	1	0	0	1
	0001	0	1	0	1	1	0	1	1	1	1	0	0
	0010	0	1	0	1	1	0	1	1	1	0	0	1
	0011	0	1	1	1	1	0	1	1	1	0	0	1
LD Acc,D	0100	0	1	1	0	1	0	0	0	0	0	0	0
	0101	0	1	1	0	1	0	0	0	0	0	0	1
	0110	0	1	0	0	1	0	0	0	0	0	0	1
LD D,Acc	0111	0	0	0	0	1	0	0	0	0	0	1	0
	1000	0	0	1	0	1	0	0	0	0	0	1	0
	1001	0	0	1	0	1	0	0	0	0	0	1	1
AND Acc,H	1010	1	1	0	1	0	0	1	0	0	0	0	1
	1011	0	1	0	1	0	0	1	0	0	1	0	0
	1100	0	1	0	1	0	0	1	0	0	0	0	1
	1101	0	1	1	1	0	0	1	0	0	0	0	1
Etc... (autres opérations)	...	...	...	...	...	...	...	...	...	...	...	...	...

Ainsi, la ROM comprend les micro-instructions correspondant à chaque opération combinant les différentes opérations de l'UAL aux différents registres : par exemple, *LD B,D*, *XOR Acc,M*, etc... On nomme généralement ces opérations des *instructions*.
L'ensemble des micro-instructions contenues dans la ROM s'appelle le *micro-programme*.

Commande des composants du microprocesseur

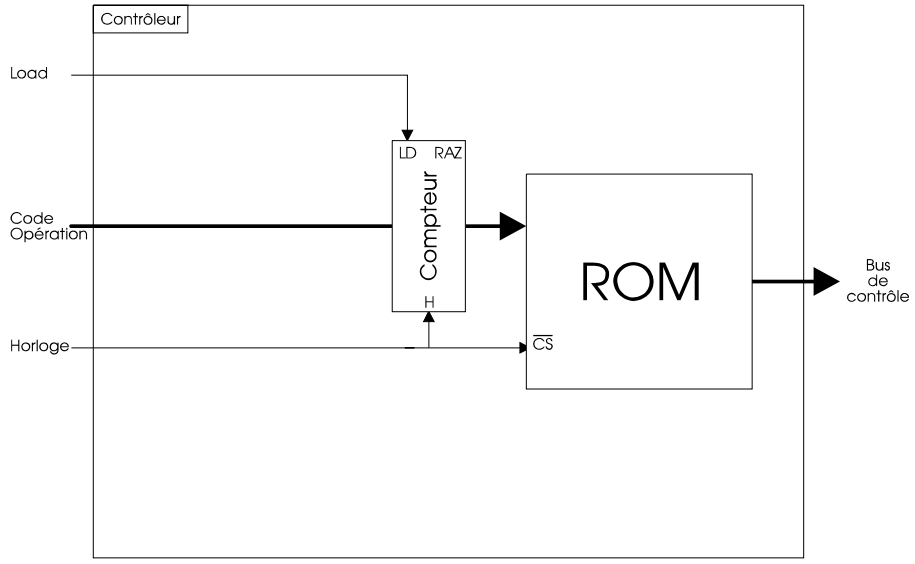
L'adresse de la première micro-instruction de chaque instruction est appelée *Code Opération* (souvent abrégé *Code Op*) : dans notre exemple, 0000 pour *ADD Acc,M* / 0100 pour *LD Acc,D* / 0111 pour *LD D,Acc* / 1010 pour *AND Acc,H*.
Il est possible d'envoyer les micro-instructions à la partie "opérationnelle" (comportant l'UAL, les registres, le bus de données, les drapeaux, l'accumulateur, par opposition à la partie "gestionnaire" devant assurer le cadencement) du microprocesseur, de manière à commander ses actions (le schéma représente le pavé « contrôleur » du schéma du microprocesseur) :



Génération de l'ensemble d'une instruction

On voit que ce schéma a de l'idée, mais qu'il est incomplet : il est grand mais contient beaucoup de blanc, cela présage du remplissage… Plus objectivement, la ROM ne générera que la première micro-instruction de

l'instruction, alors qu'on veut envoyer successivement toutes celles d'une instruction : nous devons donc interposer entre le bus "Code Opération" et la ROM un incrémenteur automatique et séquentiel du Code Opération. L'incrémentation peut être assurée par un compteur ; l'aspect séquentiel (cadençage dans le temps) sera assuré par une horloge. Le circuit devient :



Il suffit d'envoyer le Code Opération de l'instruction à exécuter (par exemple 0111 pour *LD D,Acc*), et de lancer un créneau positif (valeurs 0, puis 1, puis 0) à l'entrée LD du compteur. Alors, si on assure la génération de l'horloge :

- dès la fin du créneau sur LD et dès que l'horloge indique 0, la ROM envoie la micro-instruction à sa sortie,
- à chaque valeur 1 de l'horloge, l'adresse à lire en ROM est incrémentée en sortie du compteur,
- à chaque valeur 0 de l'horloge, la ROM envoie la micro-instruction à sa sortie, et ainsi de suite.

Nous voyons alors que l'horloge doit être la plus rapide possible (pour raccourcir le temps global de traitement), mais tout en laissant le temps aux différentes commandes le temps d'être effectives avant le créneau suivant. La fréquence de l'horloge sera donc dimensionnée en fonction du temps de propagation du plus long signal du système (le plus de transistors en cascade). Ce dimensionnement comprenant les mémoires et ces mémoires ayant des performances parfaitement connues (après tout, c'est nous qui construisons le microprocesseur), cela explique l'absence, au sein de notre microprocesseur, de signal *Prêt* : au créneau suivant de l'horloge, on sait par construction que la donnée est prête.
Un autre paramètre important entre en compte dans la définition de la fréquence d'horloge : pour chaque commutation, un transistor consomme du courant, donc produit de la chaleur. Or, sur les circuits modernes, il se trouve une très forte densité de transistors à la surface du silicium. La conséquence est que la somme des échauffements de chaque transistor fait monter de manière sensible la température du circuit, avec les risques d'endommagement que cela présente. Les parades sont les suivantes :

- diminuer la densité d'intégration, mais le but recherché est en général inverse.
- diminuer l'échauffement par la limitation du nombre global de commutations par unité de temps. Deux moyens permettent d'y arriver :
  - l'utilisation du code de Gray (vu au § « La numérotation binaire »), qui limite le nombre de basculements dans les cas fréquents de suites de nombres.
  - enfin, la limitation de la fréquence d'horloge. De fait, c'est généralement l'échauffement du microprocesseur qui limite la fréquence des microprocesseurs. Pratiquer l'overclocking, qui consiste à augmenter soi-même la fréquence du microprocesseur en reparamétrant la carte-mère, présente le risque de « griller » le microprocesseur.

Branchements sur d'autres instructions

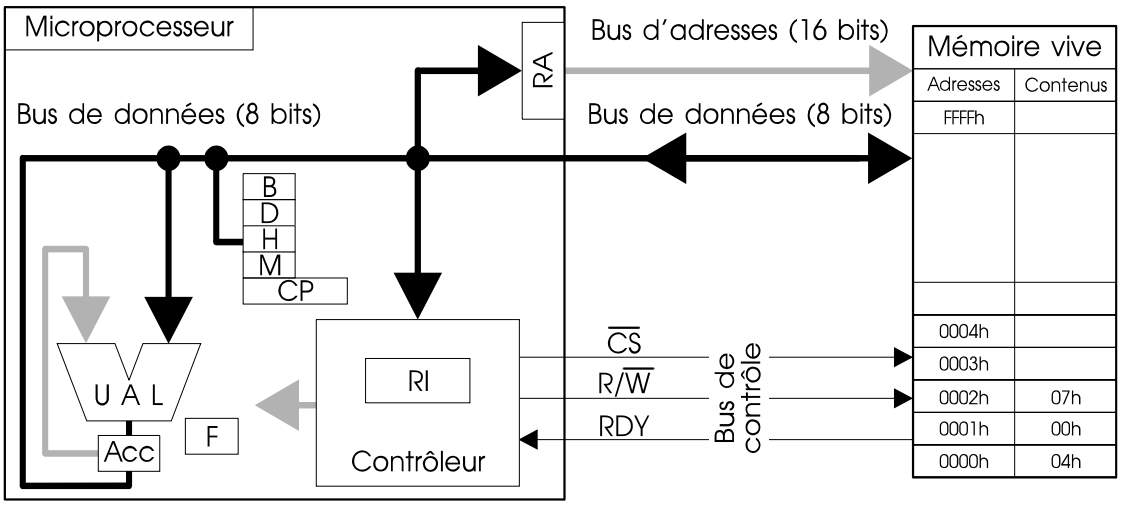
Il reste cependant quelques inconvénients :

a) le processus n'a pas de fin. Notamment, il ne s'arrêtera pas comme par enchantement à l'adresse 1001 (fin de l'instruction *LD D,Acc*), comme nous l'aurions souhaité. Il faudrait donc un système qui permette l'arrêt de l'incrémentation.
b) si l'exécution d'une instruction prend fin, il faudrait pouvoir alors basculer sur une autre instruction à la demande (et pas forcément celle à la suite dans la ROM). Supposons que nous voulions exécuter successivement les instructions suivantes (dans l'ordre) :
   - LD Acc, D
   - ADD Acc, M
   - LD D,Acc

Pour répondre au point *a*, il suffit de se servir du signal RAZ du compteur pour revenir à l'adresse 0 de la ROM : nous rajouterons un bit de données dans notre ROM (à la suite du mot de 12 bits déjà constitué), destiné à indiquer au compteur s'il faut revenir ou pas à l'adresse 0 : ce bit sera à 0 dans toutes les micro-instructions, sauf dans la dernière de chaque instruction, où ce bit sera à 1. Ce bit de donnée de la ROM étant relié à l'entrée RAZ du compteur, chaque instruction se terminera par un retour à l'adresse 0 de la ROM.
Pour répondre au point *b*, nous décalerons les données inscrites dans notre ROM de manière à libérer les premières adresses (car nous venons de voir que toute instruction se termine par un retour à l'adresse 0). Ces premières adresse comprendront les signaux nous permettant d'aller chercher dans une mémoire vive (externe au microprocesseur) le prochain Code Opération à exécuter. Pour cela, nous aurons besoin encore de signaux (ou bits de données) supplémentaires dans la ROM, et de quelques registres supplémentaires dans notre microprocesseur.

Architecture du microprocesseur

La liaison entre le microprocesseur et la mémoire vive est construite selon le schéma suivant :



Les liaisons électriques entre le microprocesseur (qui tient sur un composant) et la mémoire vive (qui tient sur un composant distinct) est réalisée à partir des *pattes* des composants et de simples pistes gravées sur la carte électronique. La mémoire vive comportant 65536 octets (depuis l'adresse 0 jusqu'à l'adresse 65535), son adressage s'effectue sur 16 bits. La RAM étant généralement amovible et extensible, on ne connaît *a priori* pas ses performances à la construction du microprocesseur. Elle comporte donc un signal RDY.
Le registre CP est un registre 16 bits, construit par l'accolement de deux registres 8 bits (un de poids fort, l'autre de poids faible). Physiquement, nous avons simplement ajouté deux registres à la mémoire vive statique du microprocesseur (fixons leurs adresses à 0100 pour CP_{fort} et 1100 pour CP_{faible}), du même type que les registres B, D, H et M. Fonctionnellement, nous utiliserons toujours ces deux registres ensemble, l'un (appelé CP_{faible}) représentant l'octet de poids faible d'un registre 16 bits "virtuel" que nous appelons CP, l'autre (appelé CP_{fort}) en représentant l'octet de poids fort. CP contient l'adresse dans la RAM du Code Opération à y lire (CP = Compteur du Programme, le programme étant la suite de Codes Opération placés dans la RAM).
RA est également un registre 16 bits "virtuel" (c'est-à-dire construit par l'accolement fonctionnel de 2 registres 8 bits). Par contre, ses deux sous-registres 8 bits ne font pas partie des registres de données, mais sont construits de la même manière que l'accumulateur. En effet, ils ne fonctionnent que dans un sens : leurs bus de données

entrantes et sortantes ne sont pas reliés, mais restent distincts. Le bus de données entrantes (dans RA_{faible} et RA_{fort}) est le bus de données du microprocesseur, celui de données sortantes est relié au bus d'adresses de la RAM. De plus, de même que l'accumulateur, la commande de ces registres est directe (un signal ~~CS~~ et R/~~W~~ par registre provenant du contrôleur). RA est le registre de communication d'adresses à la RAM lors des opérations d'écriture ou lecture à celle-ci (RA = Registre d'Adresses).
RI est le nom du compteur que nous avons déjà introduit au sein du contrôleur. Celui-ci est destiné à recevoir les Codes Opération lus successivement dans la RAM, de manière à les laisser à disposition du contrôleur (RI = Registre d'Instructions).

Fonctionnement de l'architecture

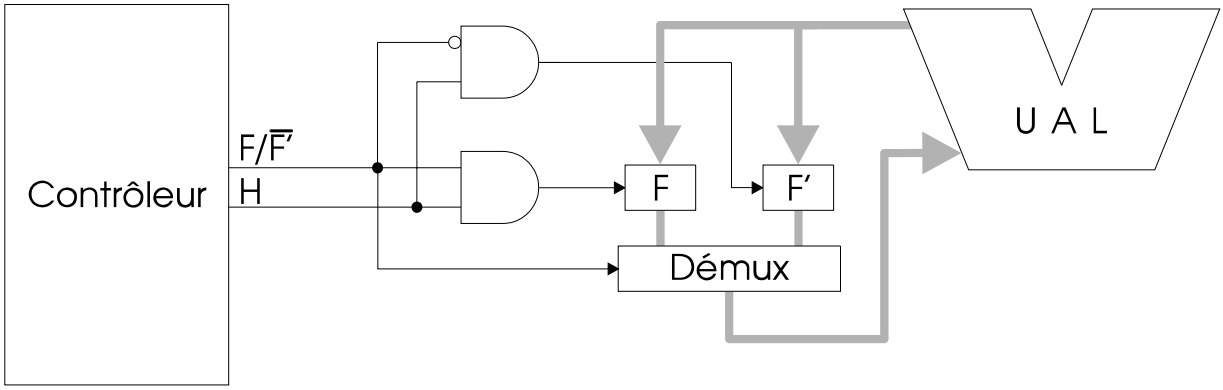
Le processus qui répond au point *b* consistera donc à inscrire dans RI le contenu de l'adresse CP en RAM, c'est-à-dire à faire réaliser par le contrôleur les instructions suivantes :

- LD RA_{faible},CP_{faible}
- LD RA_{fort},CP_{fort}
- Envoyer un ordre de lecture de la RAM
- Dès que la RAM est prête, écriture du contenu du bus de données dans RI
- Entamer le processus de lecture de la ROM du contrôleur (le microprogramme), à partir de l'adresse spécifiée par RI.

Ces commandes, constituant ce qu'on appelle le *cycle fetch*, seront placées à partir de l'adresse 0 de la ROM du contrôleur.
Ainsi, le processus qui répond au point *a* consistera simplement à commander, dans la dernière ligne en ROM de chaque Code Opération, la remise à zéro de RI, ce qui déclenchera, via le cycle fetch, l'inscription dans RI du contenu de la prochaine adresse CP en RAM : en plus de l'ajout d'un bit de donnée dans la ROM, directement relié à l'entrée RAZ du compteur RI, cela nécessite aussi d'incrémenter CP entre-temps. Puisque cette nécessité est systématique (le programme se lisant en déroulant les adresses de la RAM, on n'y lira pas deux fois de suite la même adresse), nous pouvons l'inclure dans le cycle fetch, par exemple à sa fin. Or, il se trouve que les accès à la RAM sont bien plus longs que les opérations internes au microprocesseur : nous avons largement le temps d'effectuer quelques opérations internes au microprocesseur (même utilisant le bus de données) en attendant que la RAM daigne nous répondre (en utilisant à son tour le bus de données). Nous allons donc mettre cette attente (c'est-à-dire celle du signal RDY de la RAM) à profit pour incrémenter CP. Le cycle fetch devient alors :

- LD RA_{faible},CP_{faible}
- LD RA_{fort},CP_{fort}
- Envoyer un ordre de lecture de la RAM
- INC CP_{faible}
- INCC CP_{fort}
- Dès que la RAM est prête, écriture du contenu du bus de données dans RI
- Entamer le processus de lecture de la ROM du contrôleur à partir de la valeur entrée dans RI.

Nous utilisons l'UAL pour effectuer des opérations de gestion du microprocesseur. Cela présente un inconvénient majeur : la modification et l'utilisation du registre des drapeaux. Or (nous verrons pourquoi dans le chapitre suivant), il est impératif de ne pas modifier les drapeaux lors d'opérations de gestion du microprocesseur : ceux-ci doivent être à la seule disposition de l'utilisateur. Pour autant, les opération de gestion du microprocesseur ont besoin de drapeaux : nous utilisons par exemple l'instruction INCC ci-dessus. Nous allons donc ajouter un deuxième registre de drapeaux : F'. F' sera le registre de drapeaux pour les opérations de gestion du microprocesseur, F pour les opérations de l'utilisateur. Nous allons donc rajouter une deuxième bascule de drapeaux en parallèle de la première, le choix entre l'utilisation de F et de F' se faisant par des portes AND pour l'écriture (réalisant un multiplexeur), par un démultiplexeur pour la lecture, ces dispositifs étant commandés directement par un bit supplémentaire de la ROM (que nous appellerons F/~~F'~~ : registre F sélectionné quand F/~~F'~~ est à 1, registre F' sélectionné s'il est à 0). Cela conduit à l'enrichissement suivant :



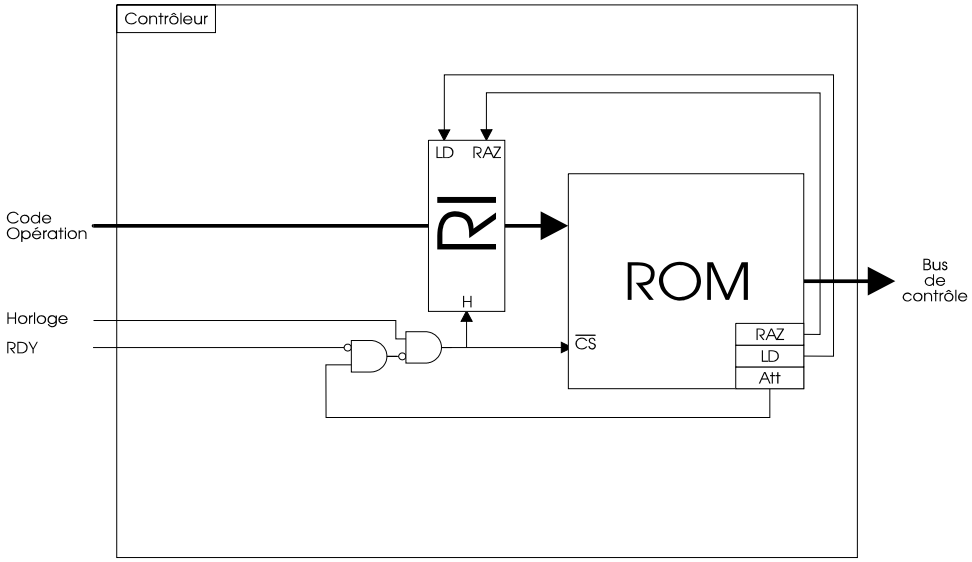
De la sorte, les opération sur les drapeaux sont inchangées, si ce n'est que le signal F/~~F'~~ indique sur quel registre on travaille. Bien sûr, il faudra prendre soin de ne pas modifier ce signal lorsque H est à 1 (auquel cas une des deux bascules D verra un front montant à son entrée, ce qui provoquera un enregistrement inopiné). Le cycle fetch devient alors :

- LD RA_{faible},CP_{faible}
- LD RA_{fort},CP_{fort}
- Envoyer un ordre de lecture de la RAM
- Sélectionner les drapeaux F'
- INC CP_{faible}
- INCC CP_{fort}
- Sélectionner les drapeaux F
- Dès que la RAM est prête, écriture du contenu du bus de données dans RI
- Entamer le processus de lecture de la ROM du contrôleur à partir de la valeur entrée dans RI.

Notre processus présente encore une carence : comment faire attendre le contrôleur tant que la RAM n'a pas encore généré son signal RDY=1 ? Le seul moyen d'empêcher l'incrémentation successive du compteur (et donc l'exécution des micro-instructions suivantes) est de bloquer l'horloge. Nous allons donc rajouter un bit aux micro-instructions de la ROM, appelé Att (comme Attente), dont la fonction sera de bloquer le signal d'horloge si Att est à 1 et que RDY est à 0. La table de vérité est :

Att	RDY	Horloge bloquée ?
0	0	0
0	1	0
1	0	1
1	1	0

On en déduit le câblage :



La structure et le contenu de la ROM deviennent :

Instruction	Adresse ROM (binaire)	Tampon Op1	Mémoire							UAL				Drapeaux		Acc		RA fort		RA faible		RI		RAM		
		H	R/~~W~~	~~CS~~	Adresse					Contrôle				H	F/~~F~~	R/~~W~~	~~CS~~	R/~~W~~	~~CS~~	R/~~W~~	~~CS~~	LD	RAZ	~~CS~~	R/~~W~~	Att
Fetch	000000	0	1	0	1	1	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	1	0	0
	000001	0	1	0	1	1	0	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	1	0	0
	000010	0	1	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	1	0	0
	000011	0	1	0	0	1	0	0	0	0	0	0	0	0	1	1	0	1	1	1	0	0	1	0	0
signaux RA	000100	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0
lecture RAM	000101	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	1	0
CPfa->Acc	000110	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	0
	000111	0	1	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	1	0
Acc->tampon	001000	1	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0
INC -> Acc	001001	0	0	1	1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	1	0	0	0	1	0
Acc -> CPfa	001010	0	0	0	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	1	0	0	0	1	0
	001011	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0
CPfo->Acc	001100	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0
	001101	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	1	0
Acc->tampon	001110	1	1	1	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	1	0	0	0	1	0
INCC -> Acc	001111	0	0	1	0	1	0	0	1	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0
Acc -> CPfo	010000	0	0	0	0	1	0	0	1	0	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0
	010001	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0
	010010	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	1	1
bus->RI	010011	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	0	1	0	1	0
ADD Acc,M	010100	1	1	0	0	0	1	1	0	1	1	1	0	1	0	1	0	1	0	1	0	0	1	1	0
	010101	0	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	1	0	1	0	0	1	1	0
	010110	0	1	0	0	0	1	1	0	1	1	1	0	1	0	1	0	1	0	1	0	0	1	1	0
	010111	0	1	1	0	0	1	1	0	1	1	1	0	1	0	1	0	1	0	1	0	1	1	1	0
LD Acc,D	011000	0	1	1	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	0
	011001	0	1	1	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	1	0	0	1	1	0
	011010	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	1	0	1	1	1	0
LD D,Acc	011011	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	1	0	1	0	0	1	1	0
	011100	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0	1	0	1	0	0	1	1	0
	011101	0	0	1	0	0	0	1	0	0	0	0	0	1	1	1	0	1	0	1	0	1	1	1	0
AND Acc,H	011110	1	1	0	0	0	1	0	0	1	0	0	0	1	0	1	0	1	0	1	0	0	1	1	0
	011111	0	1	0	0	0	1	0	0	1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0
	100000	0	1	0	0	0	1	0	0	1	0	0	0	1	0	1	0	1	0	1	0	0	1	1	0
	100001	0	1	1	0	0	1	0	0	1	0	0	0	1	0	1	0	1	0	1	0	1	1	1	0
Etc...																										

Avez-vous bien tout vérifié ?! Pour plus de clarté, quelques indications figurent (en italique) dans la colonne "Instruction" du cycle fetch.

AUTRES OPERATIONS

Nous avons vu comment réaliser des opérations arithmétiques et logiques et des transferts de registres sur 8 bits. Nous allons voir une partie de la palette possible des opérations d'un microprocesseur.

Enrichissement des opérations déjà vues

Traitements sur 16 bits

Bien que nous ayons une UAL et un bus de données qui travaillent sur 8 bits, nous avons vu qu'il était possible de réaliser des opérations d'UAL et des transferts de registre virtuels de 16 bits. Quand ? Lorsque nous avons :

- chargé CP dans RA
- incrémenté CP

CP et RA étant des registres de gestion du microprocesseur, et non des registres opératoires comme B, D, H et M, ils sont inutilisables pour l'utilisateur. Nous allons donc "doubler" les registres B, D, H et M pour créer les registres 16bits virtuels BC, DE, HL et MN. Physiquement, C, E, L et N ne sont jamais que des registres 8 bits supplémentaires que nous utiliserons de manière privilégiée avec B, D, H et M respectivement. Ces relations seront privilégiées par deux moyens :

- les adresses données aux octets de poids fort seront 0XXX, les adresses données aux octets de poids faible seront 1XXX. Ainsi, B ayant pour adresse 0000, C aura pour adresse 1000 ; D ayant pour adresse 0001, E aura pour adresse 1001, etc... On retrouve le pourquoi des adresses de CP_{fort} et CP_{faible}. Nous verrons en quoi cela crée un lien privilégié dans le paragraphe "Taille de la ROM".
- nous allons créer des instructions portant sur ces paires (BC, DE, HL et MN), à l'exclusion de toute autre association (par exemple BM, LD...). L'écriture des micro-instructions associées n'ayant plus de secrets pour nous, nous nous contenterons de simplement décrire l'opération à réaliser, sans entrer dans le détail binaire et fastidieux (vraiment ?) des micro-instructions, bien que tout cela finisse par le micro-programme. Par exemple, nous pouvons créer en ROM les instructions (Xx et Yy étant BC, DE, HL ou MN) :
  - INC Xx :
    ```
    LD Acc,x
    INC Acc
    LD x,Acc
    LD Acc,X
    INCC Acc
    LD X,Acc
    ```
  - ADD Xx,Yy (avec le résultat dans Xx, car Acc n'a que 8 bits) :
    ```
    LD Acc,x
    ADD Acc,y
    LD x,Acc
    LD Acc,X
    ADC Acc,Y
    LD X,Acc
    ```
  - AND Xx,Yy (avec le résultat dans Xx) :
    ```
    LD Acc,x
    AND Acc,y
    LD x,Acc
    LD Acc,X
    AND Acc,Y
    LD X,Acc
    ```
  - LD Xx,Yy
    ```
    LD x,y
    LD X,Y
    ```
  - etc…

Bien sûr, pour le même type d'instruction, celle réalisée sur 16 bits est au moins deux fois plus longue à exécuter que son homologue 8 bits, mais ça dépanne. C'est ce qui fait la différence avec un microprocesseur 16 bits, qui calcule simultanément sur 16 bits, alors que le nôtre, bien que réalisant des opérations sur 16 bits, ne calcule simultanément que sur 8 bits.

Opérations avec des valeurs numériques constantes

Plutôt que de vouloir transférer la valeur d'un registre dans un autre, ou additionner deux registres, on peut vouloir transférer une valeur numérique en tant que telle (par exemple A6h) dans un registre, ou additionner par exemple le registre D et la valeur numérique A6h (toujours avec le résultat dans Acc). Dans ce type d'opération, la valeur numérique à prendre en compte est stockée en RAM à l'adresse immédiatement supérieure à celle contenant le Code Opération de l'instruction. Par exemple (on note généralement *n* un nombre tenant sur un octet, *nn* un nombre tenant sur deux octets) :

- LD D,n : nous pouvons écrire les micro-instruction de cette instruction à la suite de la ROM présentée dans le tableau précédent. Dans ce cas, le Code Opération de LD D,n sera 100010b. Supposons que nous rencontrions ce code opératoire à l'adresse 03h de la RAM. Alors l'adresse 04h contiendra la valeur n. Les micro-instructions seront écrites de manière à exécuter les actions suivantes :
  - LD RA,CP
  - $R/\overline{W}_{RAM}$ =1, $\overline{CS}_{RAM}$ =0, Att_{RAM} =1
  - Ecriture du bus de données dans D, Att_{RAM} =0
  - $R/\overline{W}_{RAM}$ =1, $\overline{CS}_{RAM}$ =1
  - INC CP
  - RAZ_{RI} = 1

- LD DE,nn : instruction fondée sur le même principe, mais s'opérant en deux fois plus de temps :
  - Transfert de l'octet à l'adresse CP dans D
  - INC CP
  - Transfert de l'octet à l'adresse CP dans E
  - INC CP
  - RAZ_{RI} = 1

- ADC Acc,n :
  - Transfert de Acc sur sa bascule
  - LD RA,CP
  - $R/\overline{W}_{RAM}$ =1, $\overline{CS}_{RAM}$ =0, Att_{RAM} =1
  - Att_{RAM} =0
  - Commande de l'addition avec retenue à l'UAL, $F/\overline{F'}$ =1
  - Enregistrement du résultat dans Acc
  - $R/\overline{W}_{RAM}$ =1, $\overline{CS}_{RAM}$ =1
  - RAZ_{RI} = 1

- etc…

Les différents modes d'adressage des données

Les instructions que nous utilisons doivent indiquer sur quelle donnée effectuer les opérations. La manière de le faire est appelée *mode d'adressage*. Nous allons passer en revue les plus courants. Ceux-ci sont utilisables pour l'ensemble des instructions que nous avons vues (transferts, calcul).

Adressage immédiat

L'adressage immédiat est celui qui consiste à spécifier immédiatement la valeur à utiliser lors de l'exécution de l'instruction, par exemple : *LD Acc,A6h*.

Bien qu'utile, ce mode d'adressage présente l'inconvénient de figer la valeur au moment de l'écriture du programme en RAM. D'autres modes d'adressage ont donc été mis au point, qui n'indiquent pas directement la valeur à utiliser, mais l'endroit où la trouver.

Adressage par registre

L'adressage par registre consiste à définir la valeur à utiliser en indiquant le registre où elle se trouve, par exemple : *LD Acc,D*.

L'inconvénient de ce mode d'adressage est que le nombre de registre du micro-processeur est limité, alors que nous aurons souvent de nombreuses données à manipuler. Son avantage est qu'il est extrêmement rapide : l'opération étant purement interne au microprocesseur, elle ne nécessite aucune utilisation de la RAM.

Adressage direct (ou absolu)

L'adressage direct consiste à définir la valeur à utiliser en indiquant l'adresse en RAM où elle se trouve. Par exemple : *LD Acc,(nn)* charge dans Acc la valeur située à l'adresse nn de la RAM (la notation *(X)* signifiant

contenu de l'adresse X en RAM). Si nn contient A4FEh, et que le contenu de la RAM à l'adresse A4FEh est 45h, alors 45h sera chargé dans Acc.
L'inconvénient de ce mode d'adressage est qu'il nécessite un accès supplémentaire en RAM, ce qui ralentit le traitement : un pour aller chercher la valeur nn qui suit le Code Opération en RAM, puis un second pour aller chercher le contenu de l'adresse nn. L'avantage est qu'il est très souple.

Adressage indirect

L'adressage indirect consiste un indiquer un registre contenant l'adresse de la donnée, par exemple : *LD Acc,(HL)*.

Adressage indexé

L'adressage indexé consiste à modifier l'adresse indiquée par un adressage indirect par une valeur d'index. Par exemple : *LD Acc,(HL+X)* charge dans Acc la valeur située à l'adresse indiquée par HL de la RAM, majorée de X. L'intérêt de ce mode d'adressage est sensible lorsqu'il faut réaliser des opérations sur toute une plage d'adresses de la RAM (transfert d'un bloc d'adresses vers un autre, par exemple).

Sauts

Nous avons vu que les Codes Opération étaient lus dans la RAM dans l'ordre de leurs adresses. Cela peut être contraignant. Il serait pratique de pouvoir "sauter" directement à une autre adresse, sans lire toutes les adresses intermédiaires. Facile : il suffit de modifier la valeur de CP et de déclencher un cycle fetch.

Sauts absolus

La première manière de commander un saut est de dire : «je veux aller à l'adresse nn» (rappel : les adresses dans notre RAM sont sur deux octets). Cela revient à appliquer l'instruction :

```
LD CP,nn
```

Cela est juste, mais pour des raisons pratiques, les instructions de saut sont décrites par le mnémonique *JP* (pour JumP = sauter). Ainsi, la notation est :

```
JP nn
```

Notons que bien que cette instruction porte sur une "opérande" de type constante, les micro-instruction en ROM de l'instruction CP se garderont bien d'incrémenter à nouveau CP, vu qu'il est directement à la bonne valeur.

Sauts relatifs

La deuxième manière de commander un saut est de dire : «je veux aller à l'adresse n octets plus haut (ou plus bas) par rapport à l'adresse actuelle» (le décalage étant sur un octet). Cela revient à créer l'instruction :

```
ADS CP,n
```

ADS ? Nous nommons provisoirement par ce mnémonique une addition prenant en compte le bit "de signe" de l'opérande2 : si le bit de poids le plus fort est à 0, l'opération réalisée est une addition. Si ce bit est à 1, l'opération est une addition avec le complément à un de n. En pratique :

- si n est entre 0 et 127 compris, n est additionné à CP.
- si n est entre 128 et 255 compris, n est additionné à CP puis CP_{fort} est décrémenté.

Cela est résumé par le tableau d'exemples ci-dessous :

n	CP final
128	CP - 128
129	CP - 127
…	...
254	CP - 2
255	CP - 1
0	CP
1	CP +1
2	CP +2
…	...
126	CP +126
127	CP +127

Certes, cette opération ne figure pas dans la table du bus de contrôle de l'UAL, mais, diriez-vous, rien n'empêche de la créer. En fait, il s'agit d'une addition sur 8 bits quasiment normale ! En effet, réalisons l'opération 5Dh+254d :

5D + 254d = 5Dh + FEh = 01 5Bh.

Si on ne regarde que l'octet de poids faible :

5Dh+254d = 5Bh
c'est-à-dire 5Dh + 254d = 5Dh – 2d
donc 254d = -2d

Les instructions de sauts relatifs sont décrites par le mnémonique *JR* (pour Jump Relatif). Ainsi, la notation est :

JR n

Sauts conditionnels

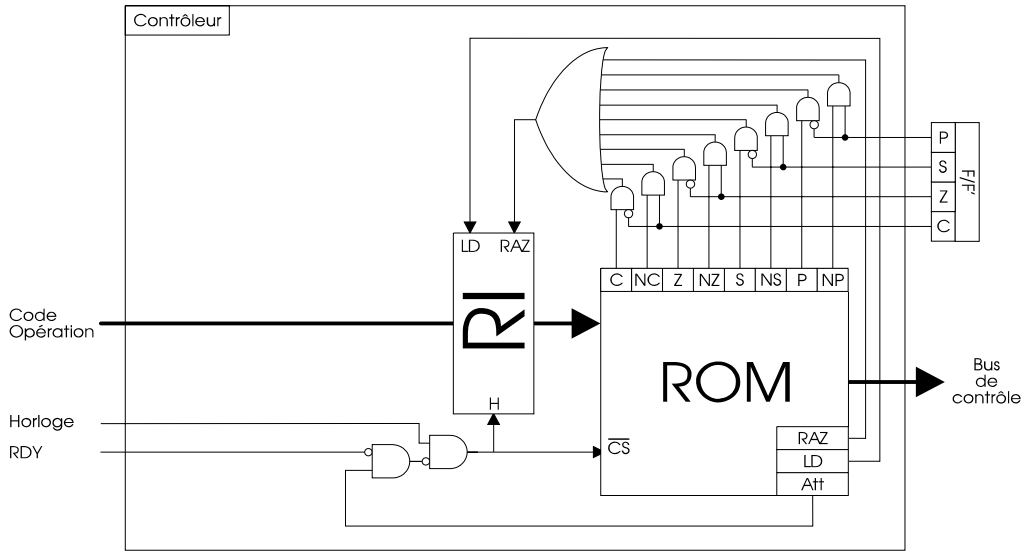
Il peut être intéressant également d'avoir des sauts conditionnels : "si telle condition est remplie (par exemple D=B), j'exécute le saut JP 2B56h". En réalisant des conditions sur la valeur d'un des drapeaux au choix, nous verrons que toutes les conditions de comparaison entre registres sont possibles. Nous utiliserons la notation suivante :

- JP X nn : saut absolu à l'adresse nn si le drapeau X est à 1.
- JP NX nn : saut absolu à l'adresse nn si le drapeau X est à 0 (si X=0, alors NX = NON X est vrai).
- JR X n : saut relatif de n octets si le drapeau X est à 1.
- JR NX n : saut relatif de n octets si le drapeau X est à 0.

Pour rappel, les drapeaux sont : C pour la retenue, S pour le signe, Z pour le zéro et P pour la parité. Par exemple, JP NZ 2B56h réalisera un saut absolu à l'adresse 2B56h si Z est à zéro. Sinon, le Code Opération suivant dans la RAM sera exécuté.

Comment d'innocentes micro-instructions peuvent-elles réaliser des tests qui modifieront leur propre déroulement ? Le seul moyen est de mettre à jour le compteur RI. Ça, nous savons faire : il suffit d'ajouter des bits à la ROM, alimentant un circuit combinatoire générant le signal RAZ de la bascule RI. Nous allons donc ajouter un bit par test, soit huit bits (tests Z, NZ, C, NC, P, NP, S, NS) reliés par des portes à l'entrée RAZ du compteur RI. En effet :

- si le test est négatif, il faut aller chercher le Code Opération suivant : c'est le but du cycle fetch, déclenché par l'activation de RAZ.
- sinon, le micro-programme continue, où il modifie CP suivant la valeur de nn (saut absolu) ou n (saut relatif). Le câblage sera le suivant :



Les micro-instruction de l'instruction JP NZ nn (qui prend 3 octets en RAM : 1 pour le Code Opération, 2 pour nn) exécuteront les tâches suivantes :

- INC CP
- INC CP (ainsi, dans le cas où le test est négatif, CP est calé sur l'instruction suivante)

- Mise à 1 du bit NZ de la micro-instruction (lequel déclenche le fetch si NZ est faux)
- JP nn
- Mise à 1 du bit RI_{RAZ}

Nous savons donc réaliser des sauts conditionnels testant les drapeaux. Comment réaliser des comparaisons entre données (par exemple, exécuter un saut si D=B) ? Nous allons utiliser le fait que toute opération réalisée par l'UAL (sauf l'opération 0000b) modifie les drapeaux en fonction de son résultat (même si l'opération, comme ADD, n'utilise pas les drapeaux en tant qu'opérande comme le fait ADC). En supposant que notre UAL sache réaliser des soustractions, le programme suivant réalise notre vœu :

- LD Acc,D
- SUB Acc,B
- JP Z nn

En effet, comme toute autre opération, la soustraction sans retenue SUB (SBC = soustraction avec retenue) modifie les drapeaux selon son résultat. Ici :

- si D = B, alors Acc-B = 0, donc Z=1.
- si D ≠ B, Acc-B ≠ 0, donc Z = 0.

L'inconvénient de la méthode est qu'elle modifie l'accumulateur. En effet, tout ce que nous voulons est comparer deux valeurs, et non récupérer un résultat. A partir de nouvelles micro-instructions en ROM, nous pouvons aisément définir une instruction identique à SUB, mais omettant de transmettre son résultat à Acc. Son nom est *COMP* (compare).

Pour réaliser un saut si D > B, sans modifier Acc :

- LD Acc,B
- COMP Acc,D
- JP S nn

Pour réaliser un saut si D ≥ B, sans modifier Acc :

- LD Acc,D
- COMP Acc,B
- JP NS nn

Etc…

Sous-programmes

Instructions CALL et RET

Supposons que nous écrivions un programme en RAM comprenant souvent une même séquence d'instructions. Par exemple :

Adresse RAM	Instruction
ADR1	I1
	I2
	A
	B
	C
	D
	E
	I3
	I4
	I5
	A
	B
	C
	D
	E
	I6
	I7
	A
	B
	C
	D
	E
	I8
	RET

La séquence d'instructions ABCDE est répétée 3 fois. Il serait intéressant de l'écrire une fois, puis de l'exécuter à la demande. C'est ce qui est réalisé ci-dessous :

Adresse RAM	Instruction
ADR1	I1
	I2
ADR2	CALL ADR6
ADR2 + 3	I3
	I4
	I5
ADR3	CALL ADR6
ADR3 + 3	I6
	I7
ADR4	CALL ADR6
ADR4 + 3	I8
ADR5	RET
ADR6	**A**
	B
	C
	D
	E
ADR7	RET

L'instruction *CALL nn* (de CALL = appeler) est similaire à *JP nn*, si ce n'est qu'elle mémorise la valeur CP+3 au moment de son exécution (car CALL nn occupe 3 octets, 1 pour CALL et 2 pour nn, l'instruction suivante se situe donc à CP+3). RET, que nous avons de manière simpliste déjà présentée comme une instruction de fin de programme, a en fait pour effet d'inscrire dans CP la valeur mémorisée par la dernière instruction CALL, ce qui provoque un retour de l'exécution du programme à l'instruction située en RAM juste après la dernière instruction CALL.
Ainsi, l'exécution du programme ci-dessus suit l'ordre suivant :

- ADR1 à ADR2

- ADR6 à ADR7
- ADR2 + 3 à ADR3
- ADR6 à ADR7
- ADR3 + 3 à ADR4
- ADR6 à ADR7
- ADR4 + 3 à ADR5
- Retour au programme appelant.

La série d'instructions A-B-C-D-E-RET est appelée *sous-programme*, les instructions depuis ADR1 jusqu'à ADR5 étant appelée *programme principal*, ou *programme appelant*. Un sous-programme peut appeler un autre sous-programme : il s'agit alors de sous-programmes dits *imbriqués*, tels des poupées russes.
Le RET de "fin de programme" (adresse ADR5) n'a aucune différence avec les autres : les notions de sous-programme et de programme principal ne sont pas absolues, mais relatives. Un programme est toujours le sous-programme d'un autre (à une exception près : le programme exécuté au démarrage d'un ordinateur est toujours le programme principal). Ainsi, le RET de l'adresse ADR5 ne fait que renvoyer au sous-programme appelant supposé.

La pile

C'est bien joli, mais cela ne dit pas comment le RET de l'adresse ADR7 sait quelle valeur inscrire dans CP : un coup c'est ADR2 + 3, l'autre coup c'est ADR3 + 3, ensuite c'est ADR4 + 3... Où RET va-t-il chercher ces valeurs ?
Comme nous l'avons dit, l'enregistrement de l'adresse de retour est réalisé par l'instruction CALL, dans un endroit de la RAM appelé *pile*.
Imaginez que vous soyez plongeur dans un restaurant : vous avez un certain nombre d'assiettes dans le bac de votre évier. Au fur et à mesure que vous lavez et rincez une à une les assiettes, vous les empilez sur le rebord de l'évier en attente de séchage. Pendant votre travail arrive un collègue qui se propose d'essuyer les assiettes : à moins de chercher la difficulté, il prendra toujours l'assiette du dessus de la pile. La dernière assiette mise sur la pile sera aussi la première à en partir : ce système est une pile de type *LIFO* (Last In, First Out = dernier entré, premier sorti, comme à l'école…). C'est exactement de cette manière que sont stockées en RAM les adresses de retour au programme appelant : la RAM est le rebord de l'évier, les adresses de retour sont les assiettes, CALL est le plongeur (qui réalise l'*empilage*), RET l'essuyeur (qui réalise le *dépilage*).
En effet, le microprocesseur contient un registre (encore !) dont nous n'avons pas encore parlé : *SP* (Stack Pointer = pointeur de pile). Ce registre, initialisé au démarrage de l'ordinateur à une valeur pointant sur une adresse libre de la RAM, contient l'adresse en RAM de la prochaine adresse de retour à inscrire en RAM : pour prolonger l'équivalence gastronomique, SP correspond au nombre d'assiettes dans la pile (par rapport à une référence). Ainsi :

- l'empilage (CALL) consiste à :
  - réaliser l'équivalent de *LD (SP),CP+3*. Attention : CP doit être pris ici au sens de "adresse de l'instruction CALL". En réalité, dans le microprogramme de l'instruction CALL, c'est CP+2 qui sera enregistré : en effet, CP a déjà été incrémenté une fois par défaut lors du cycle fetch.
  - réaliser deux incrémentations de SP (car les adresses enregistrées dans la pile prennent 2 octets), afin de pointer sur le nouveau sommet de la pile (pour préparer une éventuelle prochaine instruction CALL).
- le dépilage (RET) consiste à :
  - réaliser deux décrémentations de SP, afin de pointer sur le nouveau sommet de la pile.
  - réaliser l'équivalent de *LD CP,(SP)*. Notez que la valeur contenue à l'adresse SP (l'adresse de retour, donc) n'est pas pour autant effacée : elle n'est seulement plus reconnue comme adresse de retour, et sera remplacée par une autre lors d'une prochaine instruction CALL.

Pour reprendre notre exemple, supposons arbitrairement qu'à l'appel de ce programme, SP contienne la valeur C000h.

Instruction exécutée		Contenu de SP	Contenu de la pile		Commentaire
Adresse	Instruction		Adresse	Valeur	
Appelant	CALL ADR1	C002	C002	0000h	CALL est la méthode de lancement de tout programme par le système
			C000	Appelant+3	
ADR1	I1	C002	C002	0000h	Aucune modification
			C000	Appelant+3	
ADR2	CALL ADR6	C004	C004	0000h	Empilage de la valeur ADR2+3
			C002	ADR2+3	
			C000	Appelant+3	
ADR6	A	C004	C004	0000h	Aucun changement
			C002	ADR2+3	
			C000	Appelant+3	
ADR7	RET	C002	C002	ADR2+3	Dépilage de la valeur ADR2+3
			C000	Appelant+3	
ADR2+3	I6	C002	C002	ADR2+3	Aucun changement
			C000	Appelant+3	
ADR3	CALL ADR6	C004	C004	0000h	Empilage de la valeur ADR3+3
			C002	ADR3+3	
			C000	Appelant+3	
ADR6	A	C004	C004	0000h	Aucun changement
			C002	ADR3+3	
			C000	Appelant+3	
ADR7	RET	C002	C002	ADR3+3	Dépilage de la valeur ADR3+3
			C000	Appelant+3	
ADR3+3	I6	C002	C002	ADR3+3	Aucun changement
			C000	Appelant+3	
ADR4	CALL ADR6	C004	C004	0000h	Empilage de la valeur ADR4+3
			C002	ADR4+3	
			C000	Appelant+3	
ADR6	A	C004	C004	0000h	Aucun changement
			C002	ADR4+3	
			C000	Appelant+3	
ADR7	RET	C002	C002	ADR4+3	Dépilage de la valeur ADR4+3
			C000	Appelant+3	
ADR4+3	I6	C002	C002	ADR4+3	Aucun changement
			C000	Appelant+3	
ADR5	RET	C000	C000	Appelant+3	Dépilage de la valeur Appelant+3

Les instructions connexes

De la même manière que JP et JR, CALL et RET peuvent s'associer avec les tests de drapeaux (ex : CALL NZ nn), et CALL peut indiquer un adressage relatif (ex : CALL n), mais l'adresse sera toujours convertie en une adresse absolue avant d'être inscrite dans la pile. En effet, dans le cas contraire, le RET de fin de sous-programme :

- ne saurait pas si l'adresse dans la pile doit être interprétée comme relative ou absolue,
- ne saurait pas exploiter une adresse relative, car il n'en connaîtrait pas la référence (l'adresse du début du sous-programme).

Par ailleurs, de manière complètement indépendante des appels à sous-programme, il est possible d'utiliser la pile pour enregistrer de manière temporaire un registre : *PUSH X* permet d'empiler la valeur du registre X (au lieu de CP+3), *POP X* permet de la dépiler dans X (au lieu de CP). Ces instructions gèrent les incrémentations et décrémentations de SP, et sont donc parfaitement compatibles avec les instructions CALL/RET, à condition de ne pas interposer un CALL ou RET entre un empilage et un dépilage.
Au final, CALL et PUSH CP+3 / JMP sont équivalents, RET et POP CP le sont aussi.

OPTIMISATION DE LA ROM

Utilisation des adresses

La constitution des Codes Opération pour adresser les instructions en ROM tel que nous l'avons fait présente un inconvénient majeur : un Code Opération tenant sur 1 octet, il devrait pouvoir indiquer 255 (256 moins le fetch) instructions. Or, nous voyons déjà que dans notre ROM, sur les adresses allant de 00000000b à 00100001b, seules les adresses 00000000b (fetch), 00010100b (ADD Acc,M), 00011000b (LD Acc,D), 00011011b (LD D,Acc) et 00011110b (AND Acc,H) désignent des Codes Opération. Il y a donc un réel gâchis de Codes Opérations possibles, à tel point que cela limiterait sérieusement le nombre d'instructions disponibles si on n'y faisait rien. Il y a à ce problème plusieurs solutions possibles, pouvant être appliquées séparément ou au contraire simultanément, et dont voici quelques exemples :

Compléter le Code Opération

Supposons que l'instruction la plus longue de la ROM utilise 32 micro-instructions. On peut alors utiliser une ROM de 256 × 32 = 8192 adresses, codées donc sur 8 + 5 = 13 bits. Les Codes Opérations de notre exemple seraient conservés, mais il faudrait systématiquement leur adjoindre 5 zéros pour définir l'adresse complète de la première micro-instruction à exécuter en ROM pour chaque instruction. Cet ajout de 5 zéros serait réalisé par l'utilisation, pour RI, d'une bascule 13 bits et par un simple câblage de ses 5 bits de poids faible en entrée à 0. Dans notre exemple, la ROM à 24 bits de données vue précédemment (lorsque nous avons introduit le cycle fetch) deviendrait :

Code Opération	Instruction	ROM	
		Adresse complète	Donnée
00000000	Fetch	00000000 00000	Donnée de l'adresse 00000000 dans la ROM initiale avec fetch
		00000000 00001	Donnée de l'adresse 00000001 dans la ROM initiale avec fetch
		00000000 00010	Donnée de l'adresse 00000010 dans la ROM initiale avec fetch
		…	…
		00000000 10011	Donnée de l'adresse 00010011 dans la ROM initiale avec fetch
		00000000 10100	Que des zéros
		…	…
		00000000 11111	Que des zéros
00000001	ADD Acc,M	00000001 00000	Donnée de l'adresse 00010100 dans la ROM initiale avec fetch
		00000001 00001	Donnée de l'adresse 00010101 dans la ROM initiale avec fetch
		00000001 00010	Donnée de l'adresse 00010110 dans la ROM initiale avec fetch
		00000001 00011	Donnée de l'adresse 00010111 dans la ROM initiale avec fetch
		00000001 00100	Que des zéros
		…	…
		00000001 11111	Que des zéros
00000010	LD Acc,D	00000010 00000	Donnée de l'adresse 00011000 dans la ROM initiale avec fetch
		00000010 00001	Donnée de l'adresse 00011001 dans la ROM initiale avec fetch
		00000010 00010	Donnée de l'adresse 00011010 dans la ROM initiale avec fetch
		00000010 00011	Que des zéros
		…	…
		00000010 11111	Que des zéros
Etc…			

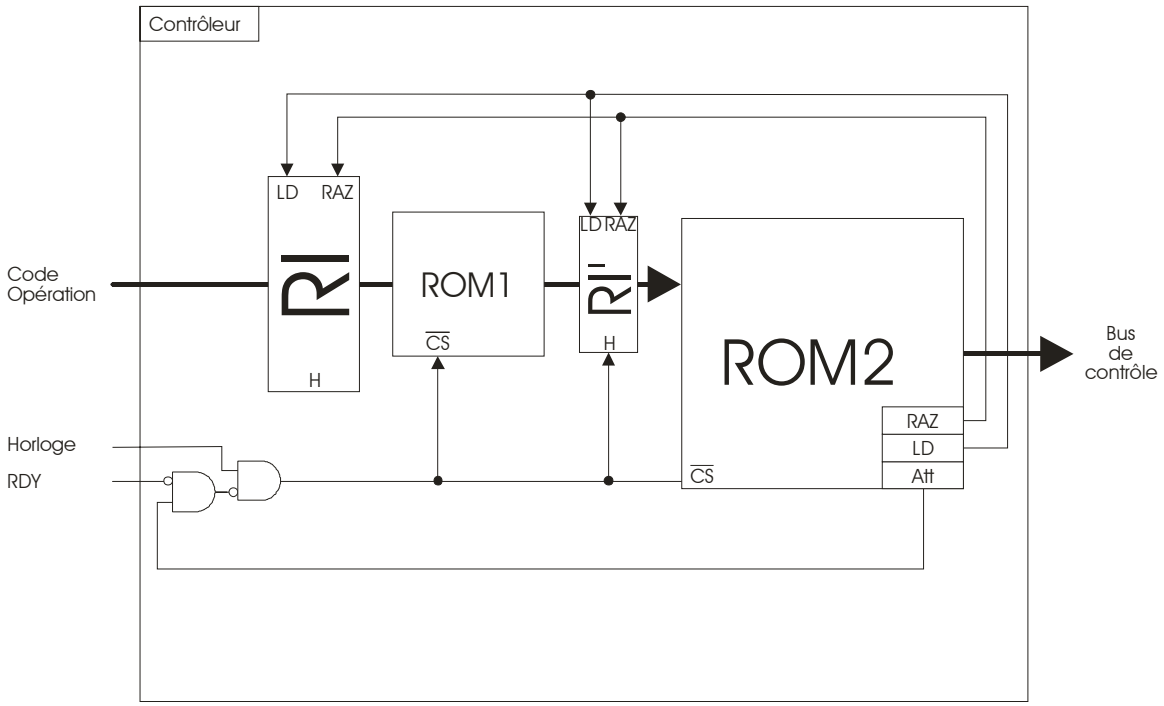
L'inconvénient majeur de cette technique est le taux d'utilisation très bas des adresses de la ROM : tous les Codes Opérations possibles sont utilisés, mais au détriment des adresses de la ROM qui deviennent largement sous-utilisées, vu que celle-ci est gérée comme si toutes les instructions avaient autant de micro-instructions que celle qui en a le plus. Ainsi, dans l'exemple précédent, sur les 95 premières adresses de la ROM, 69 sont inutiles (comme les adresses 00000000 10100 à 00000000 11111), ce qui fait un taux d'utilisation de seulement 27% ! Une solution intermédiaire consisterait à ne pas rajouter 5, mais par exemple 3 bits, la plupart des instructions nécessitant au plus 8 micro-instructions. La ROM ci-dessus deviendrait :

Code Opération	Instruction	ROM	
		Adresse complète	Donnée
00000000	Fetch	00000000 000	Donnée de l'adresse 00000000 dans la ROM initiale avec fetch
		00000000 001	Donnée de l'adresse 00000001 dans la ROM initiale avec fetch
		00000000 010	Donnée de l'adresse 00000010 dans la ROM initiale avec fetch
		…	…
		00000000 111	Donnée de l'adresse 00000111 dans la ROM initiale avec fetch
00000001		00000001 000	Donnée de l'adresse 00001000 dans la ROM initiale avec fetch
		…	…
		00000001 111	Donnée de l'adresse 00001111 dans la ROM initiale avec fetch
00000010		00000010 000	Donnée de l'adresse 00010000 dans la ROM initiale avec fetch
		00000010 001	Donnée de l'adresse 00010001 dans la ROM initiale avec fetch
		00000010 010	Donnée de l'adresse 00010010 dans la ROM initiale avec fetch
		00000010 011	Donnée de l'adresse 00010011 dans la ROM initiale avec fetch
		00000010 100	Que des zéros
		00000010 101	Que des zéros
		00000010 110	Que des zéros
		00000010 111	Que des zéros
00000011	ADD Acc,M	00000011 000	Donnée de l'adresse 00010100 dans la ROM initiale avec fetch
		00000011 001	Donnée de l'adresse 00010101 dans la ROM initiale avec fetch
		00000011 010	Donnée de l'adresse 00010110 dans la ROM initiale avec fetch
		00000011 011	Donnée de l'adresse 00010111 dans la ROM initiale avec fetch
		00000011 100	Que des zéros
		00000011 101	Que des zéros
		00000011 110	Que des zéros
		00000011 111	Que des zéros
00000011	LD Acc,D	00000011 000	Donnée de l'adresse 00011000 dans la ROM initiale avec fetch
		00000011 001	Donnée de l'adresse 00011001 dans la ROM initiale avec fetch
		00000011 010	Donnée de l'adresse 00011010 dans la ROM initiale avec fetch
		00000011 011	Que des zéros
		…	…
		00000011 111	Que des zéros
Etc...			

Dans cet exemple, le taux d'utilisation de la ROM a nettement augmenté : sur les 31 premières adresses, 13 sont inutilisées (taux d'utilisation de 58%). Mais ce n'est tout de même pas optimal. De plus, cette amélioration se fait au détriment du taux d'utilisation des Codes Opérations : certains d'entre eux sont désormais interdits, comme 00000001 et 00000010 dans notre exemple qui n'identifient aucune instruction mais sont utilisés, ou plutôt gaspillées, par le cycle fetch.

2 ROM

Plutôt que d'utiliser le procédé précédent, si nous insérons une deuxième ROM entre la bascule RI et la ROM des micro-instructions (RI étant relié à son bus d'adresses, son bus de données étant relié aux bus d'adresses de la ROM des micro-instructions via un deuxième compteur), nous pouvons, à chaque adresse cette ROM, faire correspondre par son contenu une adresse de la deuxième.



Notons que la bascule RI ne reçoit plus l'horloge : en effet, la scrutation des micro-instructions dans ROM2 est réalisée par l'incrémentation de RI'. RI ne garde comme fonction d'un compteur que la remise à zéro (pour le cycle fetch) et le chargement de son bus d'entrée (pour lire un nouveau Code Opération). Dans notre exemple, la ROM supplémentaire (ROM1) contiendrait :

Adresse	Donnée	Instruction concernée dans ROM2
00000000	00000000	Fetch
00000001	00010100	ADD Acc,M
00000010	00011000	LD Acc,D
00000011	00011011	LD D,Acc
00000100	00011110	AND Acc,H
00000110	Etc…	…

Le Code Opération inscrit en RAM n'est plus l'adresse en ROM2 de micro-instructions, mais celle de cette ROM1 intermédiaire. Etant vidé de sens physique, le Code Opération peut désormais prendre n'importe quelle valeur, et donc rien ne nous empêche d'utiliser toutes les valeurs d'un octet. Nous avons donc une parfaite utilisation de l'octet de Code Opération : 256 instructions (dont fetch) identifiées pour 256 possibilités. De plus, toutes les adresses de la ROM des micro-instructions sont utilisées.
Inconvénients :

- il faut deux ROM, ce qui est plus cher (cependant, ROM1 reste de petite taille)
- ces deux ROM sont en cascade, ce qui allonge le temps de propagation des signaux, et donc réduit la fréquence possible de l'horloge.

Code Opération sur deux octets

Même un modeste Z80 (microprocesseur 8 bits du début des années 80, proche de notre microprocesseur imaginaire) est capable de reconnaître environ 560 instructions. Comment faire avec une RAM 8 bits, qui ne permet *a priori* que 256 Codes Opération ?
Il est possible de définir des Codes Opérations particuliers (par exemple CBh et EDh pour le Z80), dont le rôle est d'"ouvrir" d'autres jeux (ou assortiments) de Codes Opérations (les Codes Opération de 00h à FFh constituant le premier assortiment), dont l'identification se situe dans l'octet suivant en RAM. Voyons un exemple de fonctionnement :

- Si le Code Opération lu est 02h, l'instruction correspondante est LD Acc,D (fonctionnement classique que nous venons de voir).

- Si le Code Opération lu est CBh, le microprocesseur sait par convention qu'il doit lire l'adresse suivante de la RAM qui indiquera le Code opération du deuxième jeu d'instructions. Si celle-ci contient 02h, l'instruction à exécuter est (par exemple) JP S nn.
- Si le Code Opération lu est EDh, le microprocesseur sait par convention également qu'il doit lire l'adresse suivante de la RAM qui indiquera le Code opération du troisième jeu d'instructions. Si celle-ci contient 02h, l'instruction à exécuter est cette fois-ci (par exemple) ADC HL,BC.

Ainsi, la plupart des Codes Opérations exécutent des instructions différentes suivant qu'ils sont précédés (en RAM) ou non du préfixe CBh ou EDh.

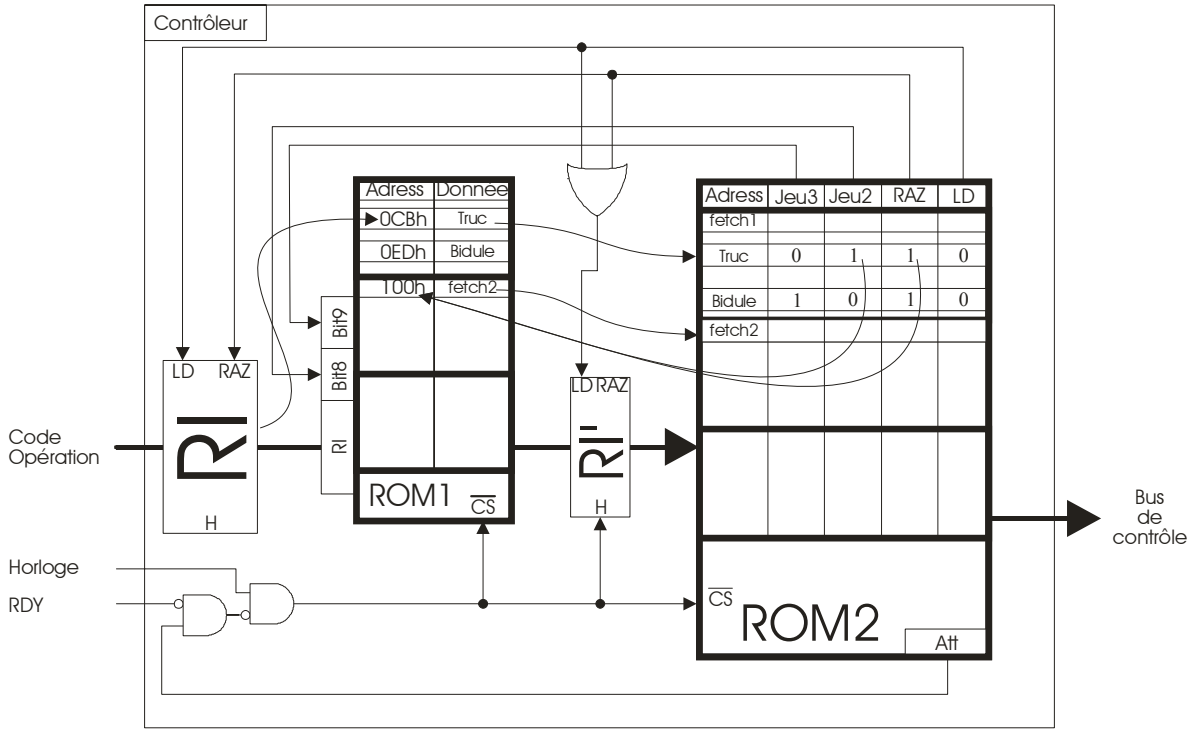
Comment pourrions-nous faire de même avec notre prototype ?
Nous supposerons que nous avons un système à deux ROM, ayant donc une première ROM de 256 adresses (une par Code Opération), l'adresse 0 exécutant le fetch. Notons que cette configuration n'est pas une nécessité. Quelles micro-instructions peuvent bien exécuter l'adresse CBh ?
Pour ouvrir un deuxième jeu de Codes Opérations, nous devons forcément doubler la capacité en adresses de notre ROM1 d'entrée, et augmenter celle de ROM2 selon la place des nouvelles micro-instructions. Ainsi, le jeu de base occupera les adresses de 0 (fetch) à 255d (11111111b) de ROM1, le deuxième jeu occupant les adresses de 256d (1 00000000b) à 511d (1 11111111b). Notre nouvelle ROM1 est donc adressée par 9 bits, que nous câblons ainsi :

- le bit 8 (de poids le plus fort) provoquant le choix entre les deux jeux.
- les 8 bits restants (0 à 7) étant normalement occupés par RI, et adressant la bonne instruction dans le jeu indiqué par le bit 8.

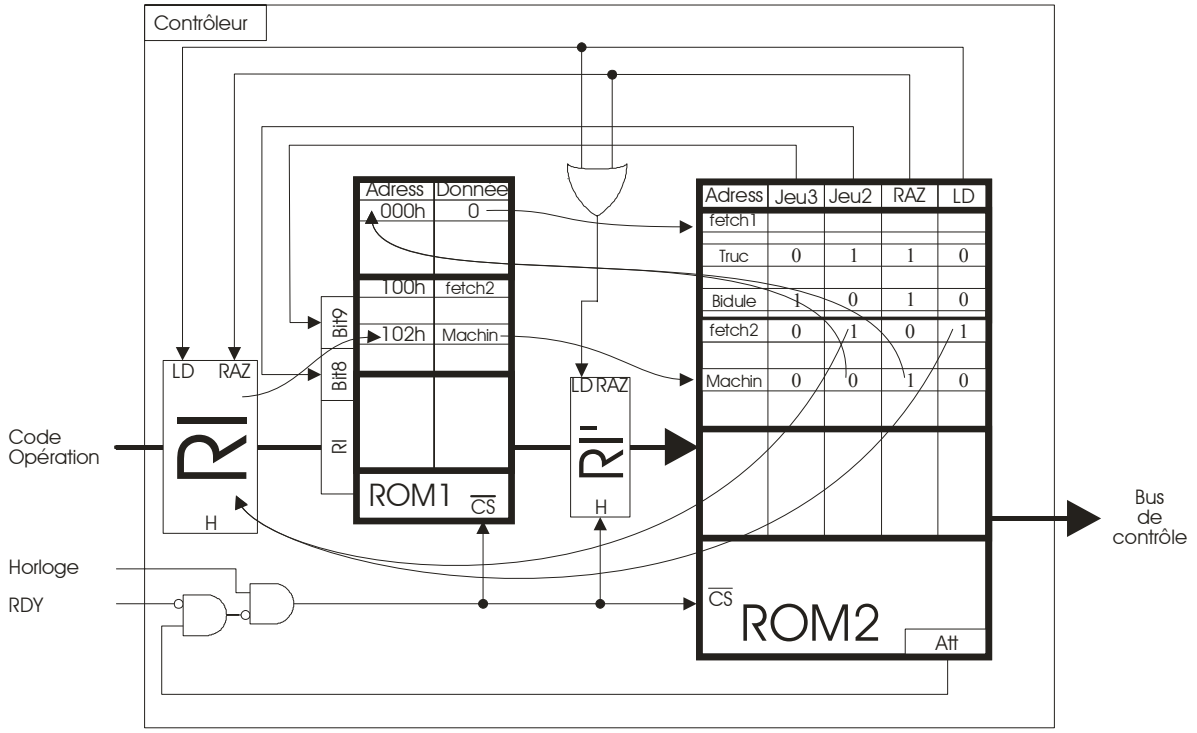
La fonction de CBh sera donc de forcer le bit 8 à 1 (à l'aide d'un bit supplémentaire dans les micro-instructions de ROM2, que nous appellerons Jeu2), et d'envoyer un signal de RAZ à RI. Ainsi, la prochaine adresse pointée par "le bit 8 + RI" sera l'adresse 0 du deuxième jeu, soit l'adresse 256d. A cette adresse, nous placerons un fetch normal, qui lira donc le Code Opération suivant dans la RAM pour le transférer dans RI. Nous mettrons alors les micro-instructions du deuxième jeu à l'adresse en ROM2 indiquée par la donnée disponible à l'adresse en ROM1 "bit 8 + Code Opération", soit 256+Code Opération. La dernière des micro-instructions de chaque instruction du deuxième jeu (et d'ailleurs du premier aussi, excepté CBh bien sûr) forcera Jeu2 à 0, avant d'envoyer le signal RAZ qui exécutera alors le fetch du premier jeu d'instructions.
RI' ne peut donc plus recevoir directement le même signal RAZ que RI : en effet, cela interdirait de lancer le fetch du deuxième jeu d'instructions (RI' pointant alors sur l'adresse 0, qui est le fetch du premier jeu d'instructions). L'entrée RAZ de RI' n'est donc plus utilisée (mais forcée à 0). Par contre, RI' doit toujours être mis à jour lorsque RI est mis à 0, ou est chargé d'une nouvelle valeur (quelle que soit la valeur de Jeu2). L'entrée LD de RI' sera donc activée pour tout signal LD ou RAZ sur RI, ce qui se traduit par une adaptation du signal LD de RI', via une porte OU.
La création du préfixe EDh est identique à celle de CBh, si ce n'est qu'il faut ajouter un jeu supplémentaire à la ROM, aux adresses allant de 512d (10 00000000b) à 767d (10 11111111b), un bit supplémentaire aux micro-instructions (Jeu3) et un "bit 9" à RI.

Le fonctionnement détaillé est le suivant :

- Lors du cycle fetch initial, CBh arrive dans RI. ROM1 active donc les micro-instructions correspondantes de la ROM2 : mise à 1 du bit8 de RI, mise à 0 du bit9 de RI, puis RAZ de RI.
- RI étant mis à 0, bit8 à 1 et bit9 à 0, cela amène à l'adresse 01 00000000b = 256d de la ROM1, qui mène à l'exécution du cycle fetch du deuxième jeu d'instructions de la ROM2 (car son adresse en ROM2 est indiquée par la donnée disponible à l'adresse 256d de la ROM1).



- Le fetch du deuxième jeu d'instructions va chercher en RAM le Code Opération à exécuter, comme un fetch normal, si ce n'est que celui-ci maintient en permanence la valeur 1 à Jeu2, ce qui a pour effet que le Code Opération chargé dans RI ne pointera pas une adresse du premier, mais du deuxième jeu d'instructions dans ROM1 (grâce au bit8 à 1), laquelle indique par construction une adresse en ROM2 appartenant également au deuxième jeu d'instructions. Notons qu'il n'y a pas en ROM2 de distinction physique entre les jeux d'instructions comme elle existe en ROM1 (la valeur des bit8 et bit9) : cette distinction, purement logique, découle simplement des valeurs inscrites en données des adresses de la ROM1.
- Cette dernière adresse est le début des micro-instructions "réelles" : celles qui réaliseront l'instruction voulue (JP S nn dans notre exemple). Sa dernière micro-instruction met Jeu2 et Jeu3 à 0, et envoie un signal de RAZ à RI. Cela a pour effet d'exécuter le fetch classique, celui du premier jeu d'instructions. La situation est redevenue "normale". Respirons…



Remarque : à avoir Jeu2 et Jeu3 utilisés seulement pour les combinaisons 00b, 01b et 10b, on peut très bien créer un quatrième jeu avec l'option Jeu2/Jeu3 = 11b.

Commandes intégrées aux Codes Opérations

Une méthode consiste à utiliser certains bits du Code Opération (et donc de RI) pour commander directement certains circuits, de la même manière que le font les bits de donnée de la ROM. Par exemple, on peut définir tous les codes opératoires du type 10xxxyyy (x et y = 0 ou 1) comme exécutant l'instruction ADD X,Y (X registre d'adresse 0xxx, Y registre d'adresse 0yyy). Les bits 3, 4 et 5 d'une part, et 0, 1 et 2 d'autre part, du Code Opération, donc de RI, étant reliés (via une logique combinatoire du type *porte 3 états* commandée, elle, par un bit de la ROM2 pour xxx, un autre pour yyy) au bus d'adresses de la mémoire statique du microprocesseur.
Ainsi, une seule instruction en ROM2 en remplacera-t-elle 64 (dans notre exemple), qui ne se seraient distinguées que par les adresses de la mémoire statique des registres.

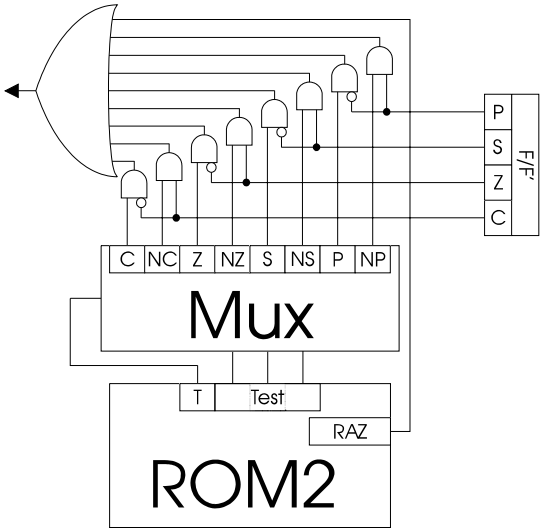
Utilisation des données

Multiplexeurs

A chaque étape, on ne peut pas s'empêcher de rajouter des bits aux micro-instructions : les mots obtenus risquent d'être très longs, et donc nécessiter des ROM de grande capacité. Un moyen pour limiter cette explosion de données est de recourir aux multiplexeurs. Par exemple, les bits dédiés aux tests des drapeaux (notamment pour les sauts conditionnels) ont cette propriété que soit aucun, soit un seul d'entre eux est à 1, les autres étant donc à 0. Plutôt que d'utiliser 8 bits pour dire ça, on peut en utiliser 4, alimentant un démultiplexeur :

- 1 bit pour dire s'il y a un test de drapeaux en cours : bit *T*
- 3 autres bits pour dire lequel (parmi les huit possibles) : bits *Test*

Avec ces informations provenant directement de la ROM, le démultiplexeur génère la valeur adéquate sur les huit signaux dédiés à chaque test.

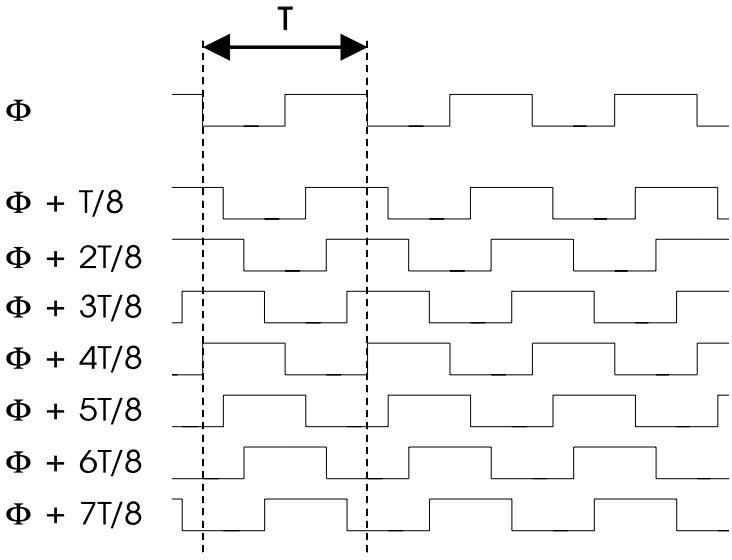


Notons que la définition en ROM des adresses des registres, telle que nous l'avons pratiquée jusqu'ici, utilise déjà parfaitement ce schéma, le démultiplexeur étant intégré à la mémoire statique des registres pour les activer un à un. On voit ici l'intérêt d'utiliser un adressage voisin entre les deux registres 8 bits (d'adresse 0xxx et 1xxx) constituant un registre virtuel 16 bits : il suffit de faire générer l'adresse xxx à la ROM, puis un câblage approprié ajoute le 1 et le 0 en bit3.

DE NOTRE PROTOTYPE A AUJOURD'HUI

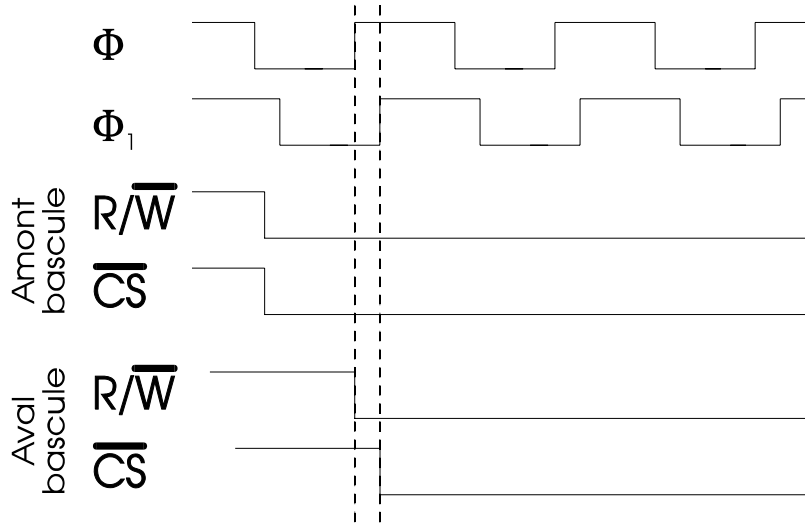
Plusieurs signaux d'horloge

Il est possible de créer une série de "sous-horloges", décalées chacune d'une fraction de cycle successive par rapport à l'horloge principale. Si on note Φ l'horloge principale, on génère Φ_1, Φ_2, Φ_3, etc… Par exemple, sur la base d'un décalage d'1/8ème de période T :



Cela permet de faire basculer plusieurs commandes lors d'une même micro-instruction, alors que nous avons dû répartir ces basculements sur plusieurs micro-instructions pour éviter leur simultanéité, qui est source de dysfonctionnements dus à une non-maîtrise des temps de propagation des signaux.
Par exemple, le bit R/~~W~~ (respectivement ~~CS~~) d'une mémoire ou de registres pourra basculer de 1 (respectivement 1) à 0 (respectivement 0) sur la même micro-instruction, si le bit ~~CS~~ est lié à une horloge légèrement en retard par rapport à celle liée à R/~~W~~. Il suffit pour cela de relier les bons bits des micro-instructions à des entrées D de bascules D, reliées en entrée T aux bons signaux d'horloge Φ_1, Φ_2, Φ_3, etc… Ainsi, la génération effective (c'est-

à-dire en aval des bascules) des bits des micro-instructions seront calés sur l'horloge associée. Par exemple, avec R/$\overline{W}$ et Φ en entrées d'une même bascule, et $\overline{CS}$ et Φ_1 sur une autre, on a :



Ce procédé économise des micro-instructions, et donc des cycles de l'horloge principale, pour une même instruction. On fait donc une pierre deux coups : la taille de la ROM2 se trouve nettement diminuée, et la rapidité du microprocesseur augmentée.

16 bits, 32 bits...

Nous avons utilisé une UAL 8 bits, en ce sens qu'elle effectue des opérations sur 8 bits. Nous avons vu qu'il était pourtant possible d'effectuer des opérations sur 16 bits, moyennant un "double traitement". Il est parfaitement possible de réaliser une UAL 16, voire 32 ou 64 bits, traitant directement des nombres de ces tailles-là. Cela engendre un redimensionnement d'autres composants, tels que les bus d'adresses, de données, les registres, etc…
Notons que contrairement aux autres caractéristiques des ordinateurs comme la capacité mémoire ou la rapidité, le nombre de bits d'un microprocesseur ne semble pas voué à une inflation galopante. En effet, celui-ci détermine principalement l'étendue des valeurs utilisables en une seule opération. Avec 8 bits, cette gamme allait de 0 à 256, ce qui est nettement insuffisant pour la plupart des applications. Par exemple, une application de comptabilité utilise des montants bien supérieurs à 256 Francs. Avec 16 bits, on couvre la gamme 0-65 535, ce qui est mieux mais reste juste pour de nombreuses applications (par exemple la même application comptable). Avec 32 bits, la gamme s'étend de 0 à 4 294 967 295, ce qui couvre la plupart des applications.
Il y a donc peu de pression actuellement pour passer à 64 bits, qui couvrirait alors la gamme 0-18 446 744 073 709 551 616, ce qui dépasse de loin toute application humaine.

Pipe-line

Décomposition d'une instruction

Le traitement de chaque instruction est décomposable en plusieurs sous-traitements, par exemple :

1. Chargement de l'instruction : correspond *grosso-modo* au cycle fetch
2. Analyse de l'instruction : correspond à la gestion du micro-programme (où sont en ROM les micro-instructions de l'instruction ?)
3. Calcul des adresses : adresses à comprendre au sens "modes d'adressage" (où sont les données à utiliser ?), que l'adresse indique une valeur numérique, un registre, une donnée en RAM, etc…
4. Chargement des données : à partir des "adresses" calculées à l'étape précédente
5. Exécution à proprement parler de l'instruction.

L'architecture générale de notre microprocesseur peut être enrichie, en séparant physiquement chacun de ces sous-traitements dans une unité fonctionnelle qui lui est propre (U1 pour le chargement de l'instruction à U5 pour son exécution), en prenant soin que chacune d'entre elles ait un fonctionnement indépendant de ses voisines.
Supposons que nous ayons une suite de 10 instructions en RAM (I1 à I10) à exécuter.

- U1 traite I1.
- Puis U2 traite I1. U1 est alors inactif. On utilise alors U1 à traiter I2.

- Puis U3 traite I1. U2 peut alors traiter I2. Alors U1 peut traiter I3.
- Et ainsi de suite…

		Cycles													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Tel que vu jusqu'ici	U1	I1					I2					I3			
	U2		I1					I2					I3		
	U3			I1					I2					I3	
	U4				I1					I2					I3
	U5					I1					I2				
Avec décomposition des traitements	U1	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10				
	U2		I1	I2	I3	I4	I5	I6	I7	I8	I9	I10			
	U3			I1	I2	I3	I4	I5	I6	I7	I8	I9	I10		
	U4				I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	
	U5					I1	I2	I3	I4	I5	I6	I7	I8	I9	I10

Dans notre microprocesseur, le traitement des 10 instructions aurait utilisé $10 \times 5 = 50$ cycles. Avec cette méthode, il utilise 14 cycles. Sur de grands traitements, le gain atteint le rapport 5, avec la décomposition que nous avons définie. Ce procédé s'appelle le *pipe-line* (en français : oléoduc).
Le pipe-line n'est en soi efficace que dans les phases où le programme en RAM est linéaire : chaque instruction à exécuter est situé juste après (dans la RAM) l'instruction précédente (dans le temps). Si un saut survient, des unités fonctionnelles auront travaillé sur des instructions qui ne seront finalement pas exécutées. Le pipe-line devra reprendre son fonctionnement à zéro à partir de la nouvelle adresse CP.
A moins d'anticiper ce genre de désagrément, comme nous le voyons dans le prochain paragraphe.

Prédiction de branchement et exécution spéculative

Ces barbarismes identifient des méthodes consistant à lire la RAM dans les adresses supérieures à CP, par pure anticipation. Ainsi, les branchement peuvent être statistiquement prévus, et le contenu des instructions aussi. Certains aspects des instructions sont alors déjà réalisés quand CP pointe dessus, d'où un gain de temps et une optimisation du pipe-line en cas de branchement.

Mémoire cache

La *mémoire cache* (ou *antémémoire*) est une mémoire construite sous la forme d'une RAM, si ce n'est qu'elle est constituée de cellules mémoire statiques. Elle permet des temps d'accès (en écriture comme en lecture) très rapides.
Mais elle est chère. Sa capacité est donc moindre que celle de la RAM. En conséquence, le microprocesseur s'applique alors à n'y stocker que les données dont il a souvent besoin. Ce processus porte sur une analyse statistique des échanges entre le microprocesseur et la RAM. Par exemple, on constate que le microprocesseur utilise généralement la RAM par blocs, un bloc étant une plage contiguë d'adresses (à commencer par le programme contenant les instructions). Ainsi, lorsqu'une opération en RAM concerne l'adresse X, les données d'adresses voisines de X vont être copiées en mémoire cache, et le microprocesseur travaillera dans la mémoire cache pour ces adresses.
La mémoire cache externe, ou mémoire cache de deuxième niveau, est contenue physiquement dans des composants séparés du microprocesseur (comme la RAM), alors que la mémoire cache interne, ou mémoire cache de premier niveau, qui joue en fait le rôle de "mémoire cache de la mémoire cache externe", fait partie intégrante de celui-ci (comme les registres).

Coprocesseur mathématique

Le coprocesseur mathématique est une unité spécialisée dans les calculs complexes, notamment les calculs en virgule flottante, que l'UAL ne réalise pas directement (il faut de nombreuses lignes de programme pour réaliser un calcul en virgule flottante avec les simples opérations en virgule fixe de l'UAL). Par le passé, cette unité était séparée physiquement du microprocesseur (boîtier séparé), mais est désormais intégrée à celui-ci.

Parallélisme

Certains microprocesseurs sont conçus pour pouvoir fonctionner à plusieurs en parallèle (de 2 à plusieurs milliers). Cette architecture est particulièrement bien adaptée aux applications réalisées sur des vecteurs. Par exemple, le vecteur A = [2 ; 5 ; -2 ; 8 ; 0] et le vecteur B = [0 ; -10 ; 56 ; 4 ; 7] peuvent être additionnés pour donner le vecteur A+B = [2 ; -5 ; 54 ; 12 ; 7].
La principale application est le calcul par éléments finis, de type résistance des matériaux ou météorologie. Nous ne sommes cependant plus dans le domaine de la micro-informatique, mais plutôt des supercalculateurs.

Instructions spécialisées

Enfin, la richesse du jeu d'instruction peut toujours être étendue. Ainsi, des instructions spécialisées dans les calculs de traitement d'images vidéo, de sons audio et d'images sont désormais disponibles dans les plus récents microprocesseurs. Ces calculs câblés dans le silicium sont bien plus rapides que les mêmes réalisés par une multitude d'instructions de base à la queue leu-leu dans la RAM.