heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

REDS

*Titre XI*

# *Architecture*

## *1ère partie*

Serge Boada
Janvier 2003
Version 2.1

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale

**Mise à jour de ce manuel**

Je remercie tous les utilisateurs de ce manuel de m'indiquer les erreurs qu'il comporte. De même, si des informations semblent manquer ou sont incomplètes, elles peuvent m'être transmises, cela permettra une mise à jour régulière de ce manuel.

**Modification**

v2 : recopie de l'original papier
v2.1 : correction d'erreur sur les images et numérotation

**Contact**

Auteur: Serge Boada
e-mail : serge.boada@heig-vd.ch

Coordonnées à la HEIG-VD :

Institut REDS
HEIG-VD
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud
Route de Cheseaux 1
CH-1400 Yverdon-les-Bains
Tél : +41 (0)24 / 55 76 330
Internet : http://www.heig-vd.ch

Institut REDS
Internet : http://www.reds.ch

Autres personnes à contacter:

| | | |
|---|---|---|
| M. Messerli Etienne | etienne.messerli@heig-vd.ch | Tél direct +41 (0)24/55 76 302 |
| M. Yoan Graf | yoan.graf@heig-vd.ch | Tél direct +41 (0)24/55 76 259 |
| M. Cédric Bardet | cedric.bardet@heig-vd.ch | Tél direct +41 (0)24/55 76 251 |

# Table des matières

## Chapitre XI

# *1ère partie: Architecture*

---

Dans ce chapitre, nous allons parler d'ordinateurs, de miniordinateur, de microprocesseur... et la première chose à faire est d'éclairsir ces termes. En fait, tous les ordinateurs, qu'ils soient maxi, mini, super-maxi, maxi-mini ou micro sont essentiellement semblables dans leur structure de base et leur but général. Si différence il y a, c'est au niveau de leur puissance de travail, de leurs dimensions physiques et de leur prix, mais ce sont là des critères qui évoluent très rapidement avec le temps.

Un ordinateur est un système de traitement de l'information, généralement digital et électronique, donc un système logique. Ce qui en fait plus spécifiquement un ordinateur est son **universalité**: il peut être programmé, de façon relativement simple, pour exécuter toutes sortes de tâches différentes mais relevant toujours du traitement de l'information (manipulation de données, calcul).

On peut imaginer des ordinateurs ayant toutes sortes de structures mais nous nous contenterons d'étudier celle définie en 1946 par John Von Neumann, alors éminent mathématicien à Princeton, car c'est celle qu'est de très loin la plus utilisée. Von Neumann avait imaginé de stocker le programme (suite opératoire) en mémoire, alors qu'auparavant seuls quelques résultats de calcul intermédiaires étaient stockés, comme dans une calculette de bas de gamme

actuelle. De plus, il proposa que la mémoire de l'ordinateur puisse être utilisée sans distinction pour le stockage des données et du programme.

En première approche, nous pouvons distinguer trois unités fonctionnelles comme nous le montre la figure 1.

En fait, nous pourrions parler de quatre unités fonctionnelles car les lignes de communication, représentées ici par de grosses flèches reliant les blocs, sont assez particulières et constituent un *bus* de communication. Nous reprendrons cette idée de bus plus tard.

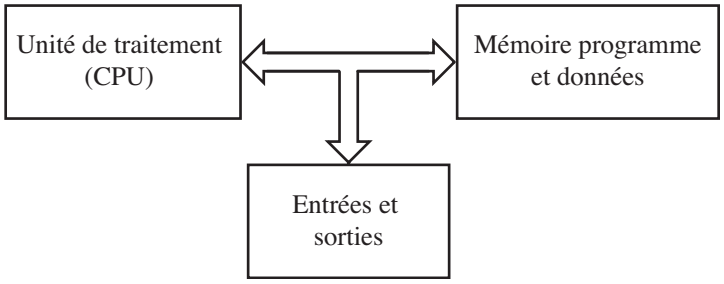


Figure XI- 1 : Unités fonctionnelles d'un CPU

L'unité de traitement, généralement connue sous son sigle anglais CPU pour *Central Processing Unit*, est une machine séquentielle complexe chargée de l'exécution des instructions constituant le programme stocké en mémoire, y compris leur séquencement. Lorsque le CPU est réalisé à l'aide d'un petit nombre de circuits intégrés à grande échelle (typiquement de 1 à 5 boîtiers), il est appelé **microprocesseur**. Du moins, un des boîtiers qui le composent l'est: celui qui effectue réellement le traitement de l'information, les autres, bien que souvent indispensables, s'occupant de tâche annexes ou partielles.

Notre tâche, dans ce chapitre, sera d'étudier de façon assez détaillée l'architecture interne des trois blocs de la figure 1, en commençant par le CPU. Le rôle, la composition et l'utilisation d'un bus de communication seront également abordés.

## XI-1 Structure d'un CPU

Le rôle d'un CPU est:

- d'aller chercher une instruction dans la mémoire (en fournissant l'adresse de la cellule concernée et un signal de lecture)
- de décoder cette instruction et de l'exécuter
- de provoquer les transferts de données de ou vers la mémoire ou les entrées/sorties s'il y a lieu
- de calculer l'adresse de la prochaine instruction et assurer ainsi le séquencement du programme.

C’est là un travail essentiellement séquentiel et notre CPU sera donc une machine séquentielle complexe. Comme telle, elle sera conçue en séparant l’unité de séquencement de l’unité d’exécution. Attention: nous entendons par là le séquencement du fonctionnement interne du CPU et non pas le séquencement des instructions stockées dans l’unité de mémoire, bien que celui-ci puisse également faire l’objet d’un module séparé à l’intérieur du CPU.

L’unité de séquencement interne du CPU peut être microprogrammée ou câblée. L’unité d’exécution est bien évidemment universelle.

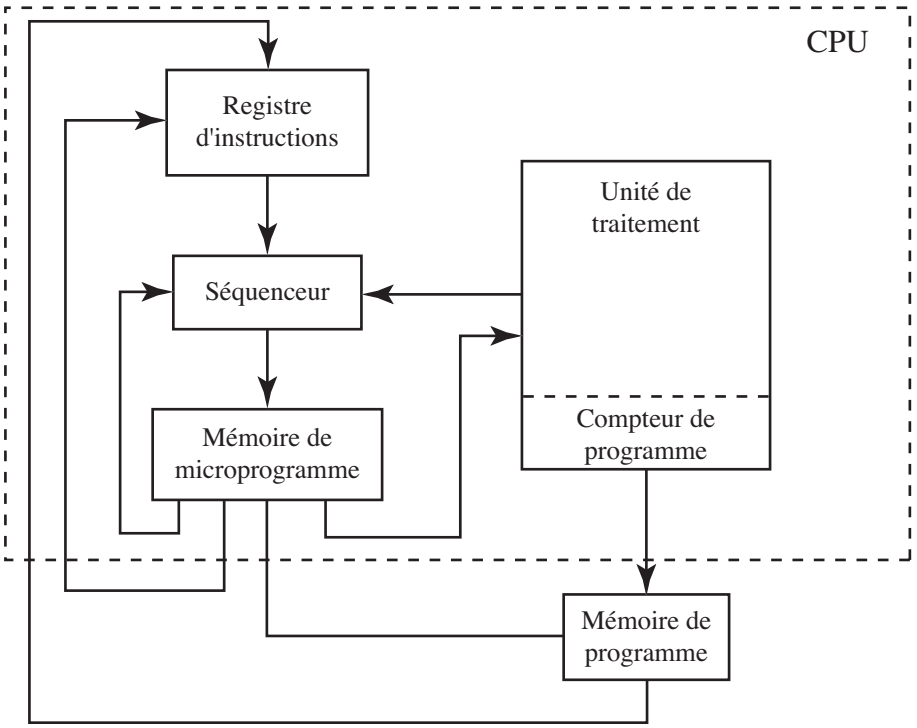


Figure XI- 2 : Structure d’un CPU

Dans le cas où le séquenceur interne du CPU est microprogrammé, nous avons deux niveaux de programmation: l’exécution d’une instruction externe se traduit par l’exécution de plusieurs microinstructions internes. Ces deux niveaux apparaissent dans la figure 2 qui représente le CPU et la mémoire contenant le programme.

## XI-2 La mémoire

La mémoire principale d’un ordinateur (Main Memory), celle qui est directement atteignable par le CPU à travers le bus, est une mémoire composée de circuits intégrés. Seule une réalisation avec des circuits à semiconducteurs permet en effet d’atteindre des temps d’accès compatibles avec les exigences d’un CPU.

Pour l'instant, nous ne nous occuperons pas de la mémoire de masse telle que les disques souples ou rigides, magnétiques ou optiques, car ce type de mémoire ne peut être connecté au bus d'un ordinateur qu'à travers une platine électomécanique commandée par un circuit fort complexe qui est souvent lui-même un système à processeur et à travers un circuit de couplage (interface) assez complexe.

Les mémoires à semiconducteurs se divisent en deux catégories: **les mémoires mortes** ou mémoires à la lecture seulement (ROM, Read Only Memory) et **les mémoires vives** ou à lecture et écriture (RWM, Read Write Memory).

Nous avons déjà décrit la structure des ROM dans le chapitre II. Nous y avons également mentionné brièvement les différents types existant, à savoir: ROM, PROM, EPROM, EEPROM.

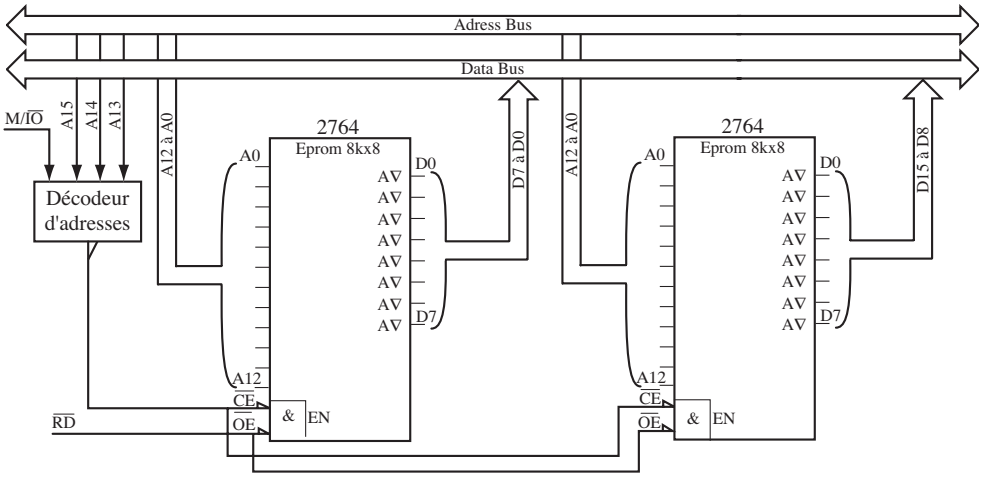


Figure XI- 3 : Rom de 8k x 16 bits connectée à un bus

Dans un ordinateur il y a toujours une partie de la mémoire principale qui est réalisée avec des ROM d'un des quatre types pré-cités. Les ROM sont en effet non volatiles (l'information reste mémorisée même si le circuit n'est pas sous tension) au contraire des RWM à semiconducteur: elles pourront donc contenir un bout de programme commençant à l'adresse de démarrage (dans notre CPU nous avons initialement mis le PC à 0) qui nous permettra d'assurer le contrôle du CPU dès l'enclenchement puis de transférer de la mémoire de masse vers la mémoire principale à lecture et écriture, le programme que l'on désire exécuter.

Nous ne reviendrons pas ici sur la façon d'assembler les circuits intégrés de ROM pour obtenir la largeur ou la profondeur de mémoire voulues (voir chapitre II, §II). Contentons-nous d'étudier, à l'aide de la figure 9, comment une mémoire ROM de 8kx 16 bits commençant à l'adresse 0, par exemple, peut être connectée au bus que nous avons défini.

Le décodeur d'adresse réalise la fonction d'activation CE (Chip Enable):

$$\overline{CE} = \overline{M/\overline{(IO)} \cdot A\overline{15} \cdot A\overline{14} \cdot \overline{A13}} \quad (1)$$

Ainsi notre mémoire répondra bien aux adresses allant de 000 hexa à 1FFF hexa. Mais elle ne forcera ses sorties sur le bus de données que lorsque l'activation du signal de lecture RD provoquera l'activation des sorties (Output Enable, OE).

Pour obtenir une largeur de 16 bits avec des circuits intégrés qui n'ont que 8 bits de largeur, il suffit de mettre deux boîtiers en parallèle.

Les mémoires à lecture seulement ne nous permettent pas de mémoriser des variables ni de charger facilement un nouveau programme à partir de la mémoire de masse. Nous aurons donc toujours besoin d'une partie de la mémoire réalisée avec des RWM. Le sigle le plus connu désignant les mémoires RWM à semiconducteurs est RAM pour Random Access Memory ou mémoire à accès aléatoire. Cette désignation les différenciait, à l'origine, des mémoires à accès séquentiel, dont une pile est un exemple particulier: pour atteindre une information dans une pile, il faut désempiler ce qui se trouve au-dessus, alors que dans une RAM toute information est accessible directement, il suffit d'avoir son adresse.

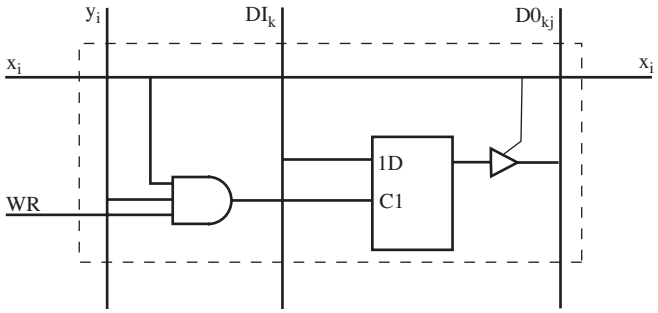


Figure XI- 4 : Structure d'une RAM statique

Nous devons encore distinguer deux types de RAM: les statiques et les dynamiques. Dans les RAM statiques, la cellule de mémorisation d'un bit est un latch, alors que les RAM dynamiques, la mémorisation se fait à l'aide d'un condensateur. Malheureusement, cette cellule mémoire basée sur un condensateur *perd la mémoire* assez rapidement: le condensateur se décharge. Il faut donc rafraîchir l'information périodiquement si on ne veut pas qu'elle disparaisse, et ceci demande une circuiterie supplémentaire. L'avantage des RAM dynamiques est que leurs cellules sont plus petites et consomment moins, donc on peut atteindre une intégration environ quatre fois supérieure à celle des mémoires statiques: pour de grandes mémoires (supérieures à 64 Kbytes) c'est là un avantage qui compense les inconvénients du rafraîchissement. Nous nous occuperons plus tard des mémoires dynamiques, nous contentant pour l'instant d'étudier les RAM statiques, plus simples à utiliser et plus intéressantes pour les petits systèmes à microprocesseur.

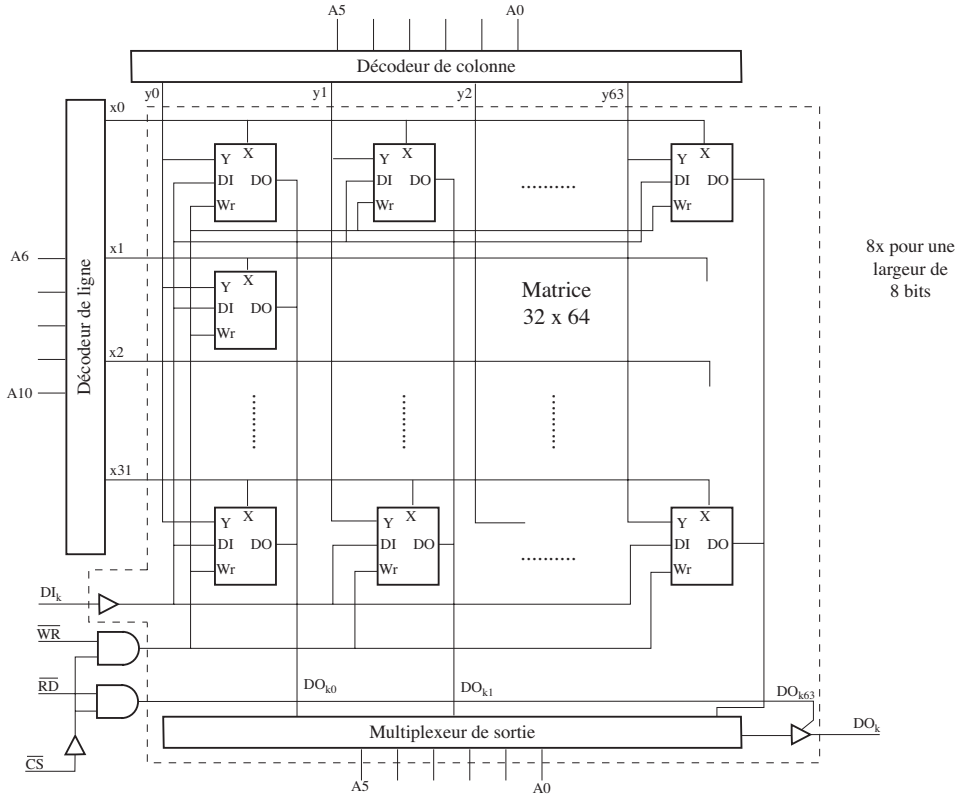


Figure XI- 5 : Une matrice de mémoire RAM

La figure 4 nous montre un schéma de principe d'une cellule RAM statique d'un bit. $x_i$ et $j_i$ sont deux bits de sélection de la cellule qui sont générés par les deux décodeurs d'adresse que nous voyons à la figure 5.

$DI_k$ et $DO_{ki}$ sont respectivement le bit d'entrée et le bit de sortie de rang k. Pour une mémoire de 8 bits de large (organisée en bytes), l'indice k varie donc entre 0 et 7.

WR est l'impulsion qui provoque le mémorisation en ouvrant le latch qui constitue l'élément mémorisant de la cellule.

La matrice de cellules mémoire pour un bit de rang k est montrée à la figure 5. Cette disposition matricielle est particulièrement avantageuse au niveau du circuit intégré: la structure est régulière, et le décodage de la position d'une cellule à partir d'un numéro de ligne et d'un numéro de colonne permet de diminuer très fortement le nombre de conducteurs à passer entre les cellules (les lignes de sélection sortant des décodeurs: 96 pour les décodeurs x-y d'une mémoire de 2 Kbytes au lieu de 2048 avec un décodeur linéaire).

| Type | capacité [Kbits] | configura-tion | cpmtaiblité p. à p. EPROM | NB. de pat-tes |
|---|---|---|---|---|
| i 2114 | 4 | 1k x4 | non | 18 |
| HM 6116 | 16 | 2k x 8 | oui | 24 |
| HM 6168 | 16 | 4k x 4 | non | 20 |
| HM 6167 | 16 | 16k x 1 | non | 20 |
| HM 6264 | 64 | 8k x 8 | oui | 28 |
| μPD 43256 | 256 | 32k x 8 | oui | 28 |

Tableau XI-1 : Exemple de RAM statiques du marché

En lecture, toute une ligne de cellules est activée pour un bit de sortie. La sélection de la colonne est faite par le multiplexeur de sortie.

Tout comme pour le CPU, il est intéressant d'économiser des interconnections entre la mémoire et le reste du système, en multiplexant l'entrée de données avec la sortie de données ($DI_k$ avec $DO_k$). Cela se fait simplement en passant $DO_k$ à travers une porte à trois états activée par le signal de lecture RD, puis en connectant à $DI_k$. Nous obtenons ainsi un passage de données bidirectionnel, directement compatible avec le bus que nous avons défini.

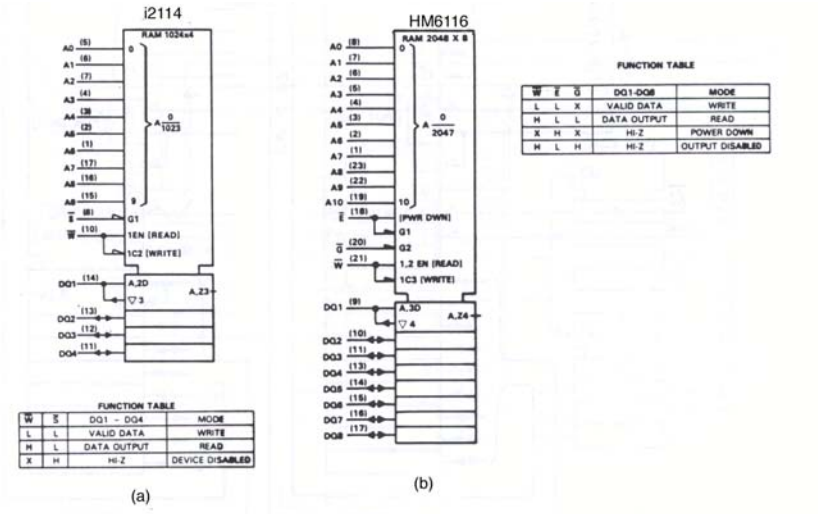


Figure XI- 6 : Exemple de symbole CEI de RAM statique

Il existe une grande variété de circuits intégrés de RAM statique et l'évolution est très rapide. Nous nous contenterons donc de mentionner un petit échantillonnage de circuits dans les diverses largeurs obtenables. Le tableau 1 liste ces quelques exemples de circuits qui portent les numéros Hitachi (HM), Intel (i) ou NEC (μPD), mais sont également produits par d'autre fabricants de semiconducteurs.

La figure 6 nous montre deux exemples de symboles CEI pour les RAM statiques. On voit apparaître la connection entre l'entrée et la sortie. Les doubles flèches sur les lignes de données indiquent qu'il s'agit de lignes bidirectionnelles.

Dans le symbole de la HM6116, la mention [PWR DWN] pour *Power Down* indique que la consommation du circuit est réduite lorsqu'il n'est pas activé (la patte 18 doit être à 0 pour que le circuit soit actif).

Nous ne reviendrons pas sur la façon d'obtenir une mémoire RAM plus large ou plus profonde, puisqu'il n'y a aucune différence avec les ROM, de ce point de vue-là. La connection au bus est également similaire à celle des ROM, sauf qu'il faut en plus connecter le signal d'écriture WR. La figure 7 nous montre la connection d'une mémoire RAM 8K x 16 à notre bus. Nous reviendrons plus en détail par la suite sur les problèmes de couplage de mémoires sur un bus, tant du point de vue temporel que du point de vue électrique.

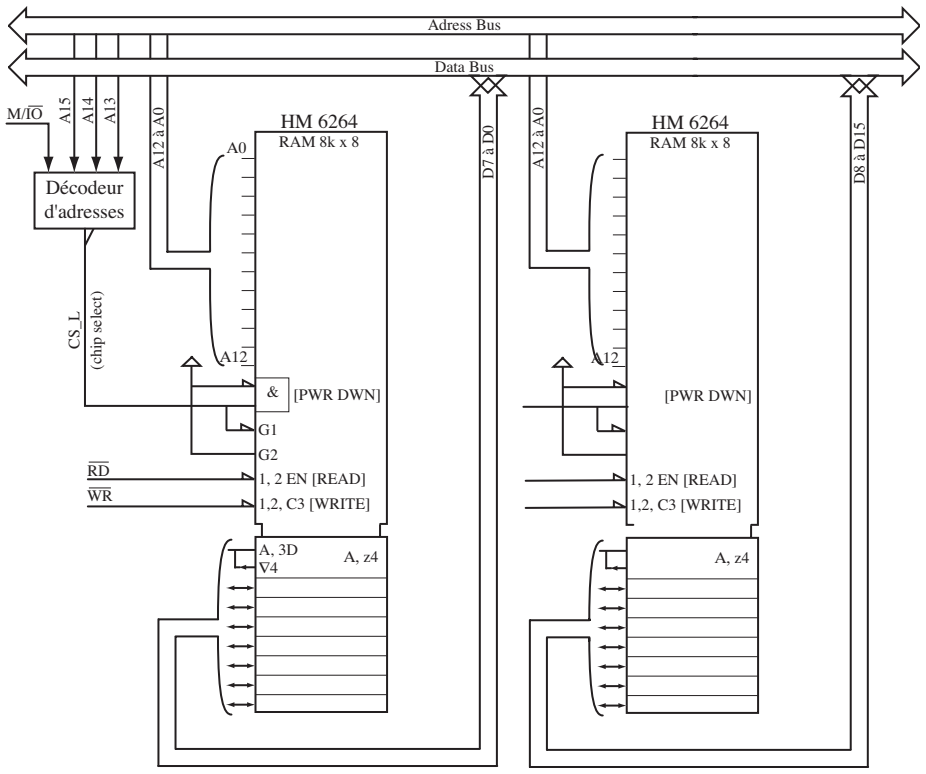


Figure XI- 7 : Connection d'une Ram à un bus

## XI-3 Les entrées/sorties

Dans un système à microprocesseur, à de rares exceptions près, les entrées/sorties se font à travers le bus, et sont donc multiplexées. Nous avons déjà utilisé des entrées/sorties multiplexées dans le microséquenceur MISEQv2, avec exactement les mêmes buts que maintenant: augmenter la flexibilité du systè-

me, diminuer la complexité de l'unité centrale et diminuer le nombre de lignes entrant ou sortant de celle-ci. Par contre, nous avons maintenant un bus de données bidirectionnel, à travers lequel devront passer aussi bien les entrées que les sorties.

Le schéma de la figure 8 nous montre le schéma le plus simple d'une entrée de 16 bits (la *largeur* de notre bus de données), dite à *zéro fil d'asservissement*.

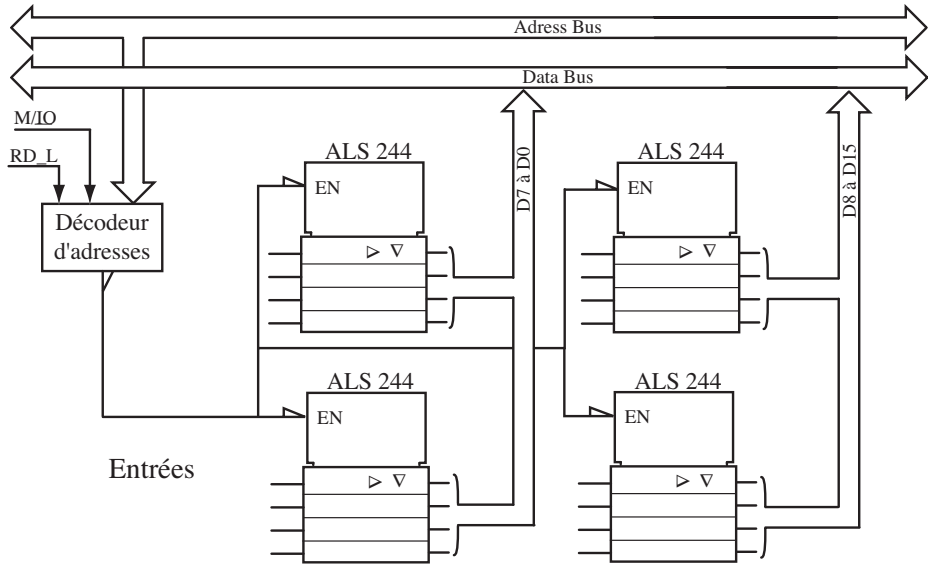


Figure XI- 8 : Entrée avec zéro fils d'asservissement

Les 74ALS244 sont des portes tampon (*buffers*) à sortie trois états. Elle permettent de connecter les 16 bits d'entrée sur le bus de données, au moment où l'adresse de cette porte d'entrée est sur le bus d'adresse et où les signaux de commande IO_L et RD_L sont actifs.

La figure 9 nous montre le schéma d'une sortie à zéro fil d'asservissement. Les latches 'ALS373 permettent de *cueillir au vol* les valeurs de sortie se trouvant sur le bus de données, au moment où l'adresse de cette porte d'entrée est sur le bus d'adresse et où les signaux de commande IO_L et WR_L sont actifs, et de les conserver jusqu'à ce que de nouvelles valeurs soient fournies par le CPU.

Dans les cas les plus simples, des portes à zéro fil de commande suffisent. Mais, bien souvent, il est nécessaire d'indiquer au périphérique (circuit connecté au bus à travers une entrée/sortie) ou au CPU, que les données se trouvent à la sortie, ou respectivement à l'entrée, sont valables. C'est le cas, par exemple, lorsque l'information à transmettre comporte plusieurs mots (de 16 bits dans notre cas) traités l'un après l'autre. Nous utiliserons alors une entrée ou une sortie à **un fil d'asservissement**.

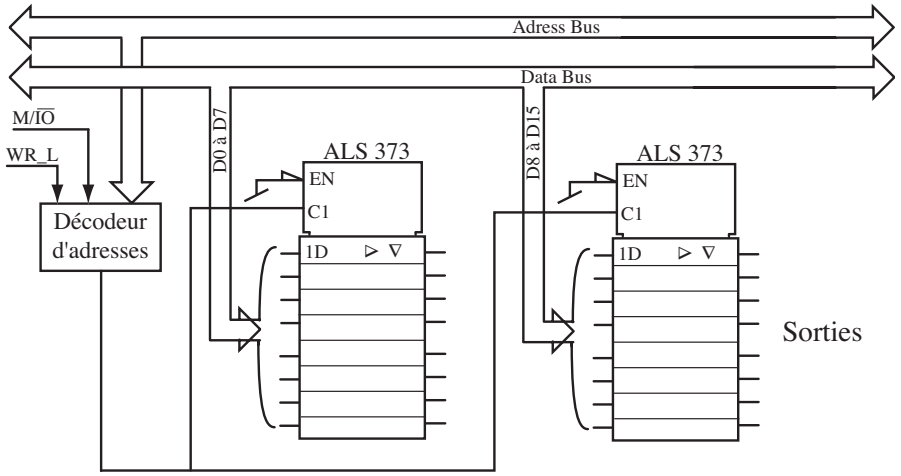


Figure XI- 9 : Sortie avec zéro fils d'asservissement

Un simple détecteur de flanc synchrone, partageant le même signal d'horloge que le CPU et travaillant sur le deuxième flanc du signal d'activation de la sortie (c'est la sortie du décodeur d'adresse dans le schéma de la figure 9) peut nous fournir une impulsion de commande signalant au périphérique que l'opération de sortie vient d'avoir lieu au niveau du CPU. Par exemple, le circuit de la figure 10 ajouté à celui de la figure 8 nous fournirait une sortie à un fil d'asservissement.

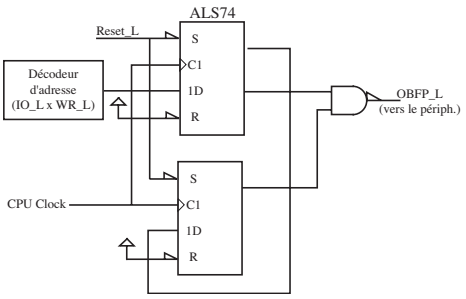


Figure XI- 10 : Circuit pour asservissement avec 1 fils

L'impulsion OBFP indique au périphérique que le registre tampon de sortie est plein (*Output Buffer Full Pulse*).

Dans le cas d'une entrée, le périphérique fournit un signal d'activation appelé généralement strobe qui permet de mémoriser les données dans un latch (si nécessaire) et qui est lui-même mémorisé dans un flip-flop. Ce flip-flop fournit au CPU le signal IBF (Input Buffer Full) qui signal au processeur qu'une nouvelle donnée peut être lue. Nous reviendrons sous peu sur l'utilisation qui peut être faite de ce signal. IBF est remis à zéro par la lecture. La figure 11 nous montre une entrée à un fil d'asservissement et à mémorisation des données.

Si nous avons signalé au récepteur qu'une nouvelle valeur d'entrée/sortie est présente, nous ne nous sommes pas occupés, pour l'instant, de savoir si le récepteur était prêt à accepter une nouvelle valeur. Pour cela, nous pouvons également concevoir des entrées/sorties à un fil de commande. Mais, s'il faut pouvoir signaler l'arrivée d'une nouvelle valeur, aussi bien que la disponibilité du récepteur, nous aboutissons à **une entrée/sotie à deux fils d'asservissement**.

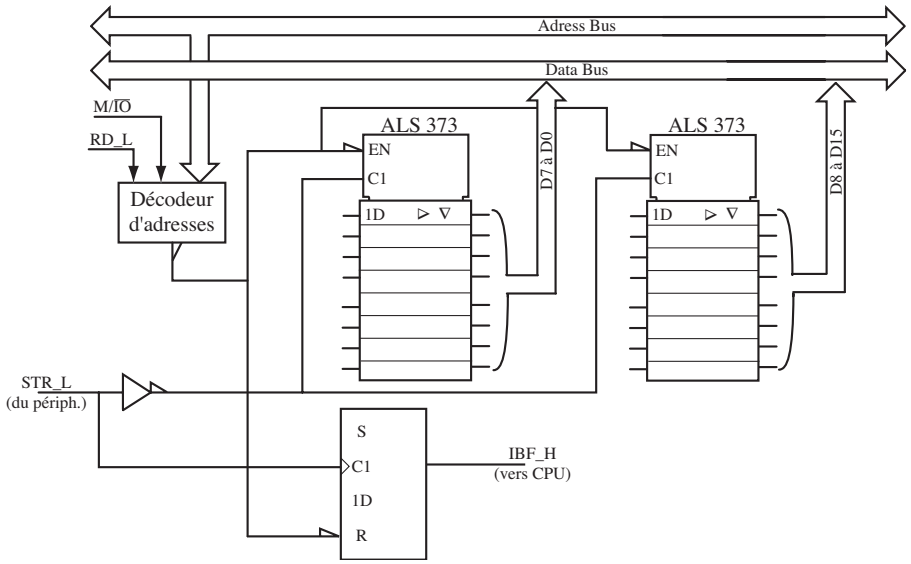


Figure XI- 11 : Entrée avec un fils d'asservisement

La figure 12 nous montre le schéma d'un circuit qui, combiné à celui de la figure 9, fournirait une implémentation possible d'une sortie à deux fils d'asservissement.

Le signal OSTR (*Output Strobe*) permet au périphérique d'échantillonner la nouvelle valeur de la sortie. OSTR n'est activé que si le signal REC_RDY (*Receiver Ready*) est actif, indiquant que le récepteur (le périphérique) est prêt à recevoir une nouvelle valeur.

Le signal OBF envoyé vers le CPU, permet à celui-ci de ne pas effectuer de nouvelle sortie avant que la valeur précédente n'ait été lue par le périphérique. Tout comme le signal IBF, nous reviendrons sur son utilisation dans le paragraphe suivant.

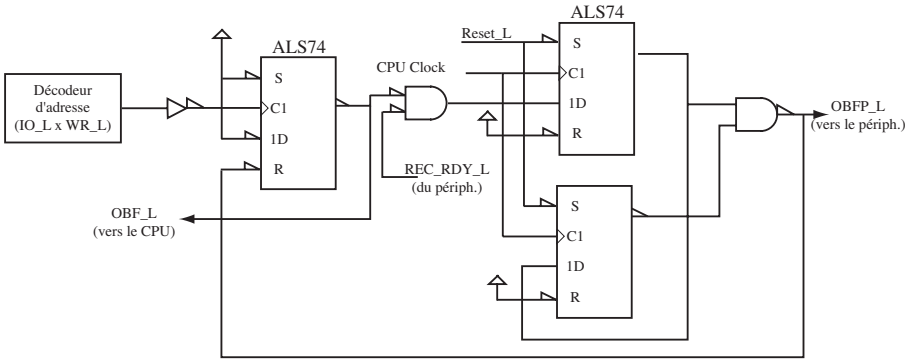


Figure XI- 12 : Circuit pour asservissement avec 2 fils

# XI-4 Autres entrées/sorties; méthode de gestion

Dans le paragraphe précédent, tous les exemples que nous avons donnés concernent des entrées/sorties digitales parallèles et unidirectionnelles. C'est dire qu'il existe également des entrées/sorties analogiques, séries et/ou bidirectionnelles. Mais, en fait, la connection au bus se fera toujours à travers des portes parallèles. Par exemple, une sortie analogique n'est rien d'autre qu'une sortie parallèle (sous-entendu *digital*) couplée à un convertisseur D/A, et une entrée série n'est rien d'autre qu'une entrée parallèle couplée à un convertisseur série/parallèle.

Nous nous occuperons des convertisseurs dans un prochain chapitre. Pour l'instant, contentons-nous de savoir qu'il n'est plus guère nécessaire de les concevoir, puisqu'ils sont obtenables sous forme intégrée et qu'ils incluent même souvent une porte parallèle, ce qui facilite leur couplage au bus.

En ce qui concerne les portes digitales parallèles bidirectionnelles, nous ne les étudierons pas en détail ici. Il nous suffit de savoir qu'elles englobent à la fois les caractéristiques d'une entrée et d'une sortie à deux fils d'asservissement (ce qui fait donc au total quatre fils), les lignes de données étant à l'état *haute impédance* sauf lorsqu'un transfert est effectué entre l'entrée/sortie et le périphérique. Concevoir un tel circuit est d'ailleurs sans objet car il est obtenable sous forme intégrée, ce qui est toujours plus économique.

Pour compléter notre bus et notre CPU, et finir ainsi notre premier tour de la structure d'un système à microprocesseur, il nous faut maintenant aborder les trois méthodes usuelles de gestion des entrées/sorties, soit la scutation, l'interruption et l'accès direct en mémoire, appliquées au transfert de données multimots.

La **scrutation** consiste, pour le CPU, à lire les signaux d'état tels que IBF dans la figure 11 et OBF dans la figure 12, pour déterminer s'il doit initier une opération de transfert de ou vers l'entrée/sortie. Pour lire ces signaux d'état,

le CPU va utiliser une porte d'entrée à zéro fil d'asservissement. Ainsi, pour une sortie, le CPU va exécuter un programme correspondant à l'organigramme de la figure 13.

Si le CPU n'a rien d'autre à faire que d'attendre que la sortie soit lue par le périphérique pour lui envoyer une nouvelle valeur, cela se passe très bien. S'il doit par contre s'occuper d'autre tâche en attendant que la sortie libre, il va falloir tester régulièrement l'état de la sortie, ce qui compliquera l'établissement du programme, augmentera sa longueur et, surtout, ralentira le temps moyen de réaction du CPU à la libération de la sortie: si la sortie se libère juste après avoir été testée, il peut s'écouler un temps important avant qu'elle ne soit de nouveau testée.

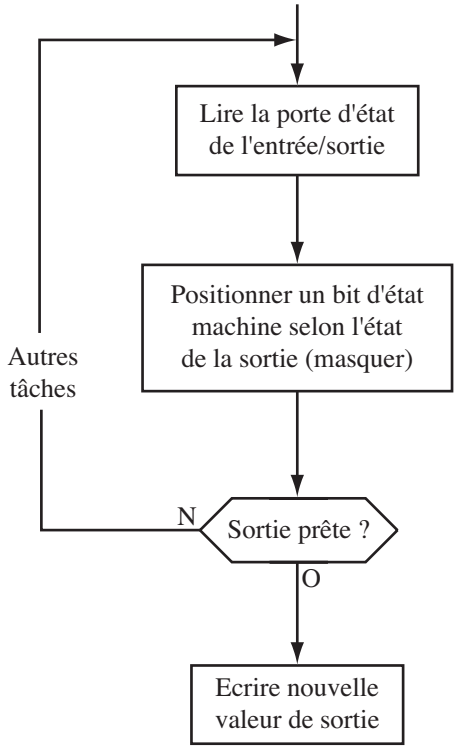


Figure XI- 13 : Organigramme du programme exécuté par un CPU

De même, s'il faut s'occuper quasi-simultanément de plusieurs entrées/sorties fonctionnant de façon aléatoire les une par rapport aux autres, le temps de réaction dépendra linéairement du nombre de ces entrées/sorties, puisqu'il faudra les tester les unes après les autres.

La gestion par interruption ne souffre pas de ces inconvénients. Elle consiste en effet à interrompre le déroulement du programme lorsqu'une entrée/sortie doit être servie, s'occuper alors de l'entrée/sortie à l'aide d'un sous-programme, puis continuer le programme interrompu. Quel que soit le moment où l'entrée/sortie devient libre et quel que soit le nombre d'entrées/sorties à ser-

vir, le temps de réaction de CPU reste pratiquement constant, du moins pour des interruptions dites vectorisées. Nous allons étudier les interruptions plus en détail dans le paragraphe suivant.

Une interruption implique un changement de contexte, avec la perte de temps que cela suppose. S'il faut effectuer des opérations d'entrées/sorties à une cadence élevée, le changement de contexte, voir même la simple lecture ou écriture en mémoire de la valeur transmise, peut représenter une perte de temps inacceptable. Il faut alors recourir à l'accès direct en mémoire ou DMA (*Direct Memory Access*).

L'accès direct en mémoire consiste à charger un circuit spécialisé assez complexe, appelé contrôleur de DMA, de faire le transfert entre non pas le CPU mais la mémoire et l'entrée/sortie. Pour ce faire, le contrôleur de DMA s'approprie le bus le temps de faire une lecture ou une écriture, temps pendant lequel le fonctionnement du CPU est suspendu (du moins en ce qui concerne ses accès au bus). Il n'y a pas lieu d'effectuer un changement de contexte, et le contrôleur de DMA gère le pointeur de mémoire avec du matériel (hardware), ce qui résulte en un gain de temps considérable. Malheureusement, la complexité d'un contrôleur de DMA en fait un circuit assez cher.

Dans ce chapitre, nous allons voir comment fonctionne le DMA au niveau du CPU et du bus, puis nous étudierons un contrôleur de DMA dans un prochain chapitre.

## XI-5 CPU et bus avec interruptions

Dans l'exercice VI.8.3, nous avons vu comment notre CPU peut prendre conscience d'une demande d'interruption qui se présente sous la forme d'un signal d'entrée IRQ_L: il suffit de terminer le microprogramme de chaque instruction machine par un saut conditionnel registre/pipeline (CJRP), la condition portant bien sûr sur l'entrée IRQ_L, après avoir pris soin d'initialiser le registre R du 2910 à l'adresse du microprogramme de traitement de la demande d'interruption. Il sera ainsi possible d'interrompre le CPU à chaque instruction machine, plus exactement: à la fin de l'exécution de chaque instruction machine.

Chaque programme en cours d'exécution devrait pouvoir décider s'il accepte d'être interrompu ou non. En effet, certains bouts de programme travaillant avec des contraintes de temps précises (par exemple, en régulation automatique) pourraient être totalement perturbés par une interruption intempestive. Il faut donc prévoir dans le CPU un flip-flop pouvant être mis à 1 et à 0 par deux instructions EI (*Enable Interrupt*) et DI (*Disable Interrupt*). Ce flip-flop, connu sous le nom de **flag d'interruption**, masque les demandes d'interruption lorsqu'il est à l'état 0 (*diasable*).

Il y a cependant des cas où il faut impérativement interrompre le CPU, quelles qu'en soient les conséquences. C'est le cas, par exemple, lorsque la tension d'alimentation chute: il faut alors prendre des mesures d'urgence avant que le système ne devienne incontrôlable à cause d'une tension d'alimentation insuffisante. Les microprocesseurs modernes ont donc tous une entrée de demande d'**interruption non masquable**. Nous en ajouterons également une à notre CPU.

Il est souvent nécessaire d'avoir plusieurs demandeurs d'interruption dans un système. Mais, pour des raisons techniques (il faut limiter le nombre de pattes du boîtier) et de souplesse d'emploi, un CPU ne peut pas avoir autant d'entrées de demande d'interruption qu'il y a de demandeurs. Pourtant, chaque demande doit donner lieu à l'exécution d'un programme spécifique au demandeur. Dans le cas où le CPU n'a qu'une seule entrée de demande d'interruption masquable (ce qui est presque toujours le cas), il faudra d'une part que cette ligne puisse être activée par tous les demandeurs, et d'autre part que le CPU puisse déterminer l'origine de la demande pour exécuter le programme correspondant.

Le fait qu'il y ait plusieurs demandeurs pose également le problème de la priorité: si plusieurs demandes sont détectées simultanément par le CPU, dans quel ordre doivent-elles être traitées?

Collecter toutes les demandes sur une même ligne est assez simple. Il s'agit de réaliser une fonction OU de toutes les demandes, soit à l'aide d'une porte OU, soit de façon décentralisée avec un OU câblé à l'aide de porte à collecteur ouvert.

La détermination de l'origine d'une demande peut être obtenue par deux méthodes: la scrutation ou la vectorisation.

La **scrutation** suppose que chaque demandeur dispose d'un bit d'indication de demande que le CPU peut aller lire. Lorsqu'une interruption est acceptée, le CPU va exécuter un programme débutant à une adresse fixe et qui a pour but de tester un à un les bits d'indication de demande puis de sauter au programme correspondant à la première demande active rencontrée. La scrutation est lente lorsque le nombre de demandeurs est grand. Par contre, le problème de priorité est directement résolu, puisque l'ordre dans lequel les demandeurs sont testés nous donne l'ordre de priorité décroissant.

La **vectorisation** est une méthode dans laquelle le CPU envoie un signal d'acceptation d'interruption INTA_L (*Interrupt Acknowledge*), à la suite de quoi un demandeur s'identifie en envoyant son vecteur spécifique. Ce vecteur peut être une instruction (généralement un appel à un sous-programme), ou une adresse, voir une partie d'une adresse seulement, et il est envoyé à travers le bus de données. En utilisant la vectorisation, le temps de réponse à une interruption ne dépend pas du nombre de demandeurs potentiels et reste toujours inférieur à celui obtenu par scrutation. Par contre, non seulement il faudra du matériel pour générer le vecteur, mais encore faudra-t-il résoudre le problème des priorités.

Le circuit nécessaire pour injecter un vecteur sur le bus de données nous est connu: il suffit d'un tampon à trois états activé par le signal de sélection de ce demandeur particulier (signal qui reste à créer). La valeur du vecteur peut être choisie à l'aide d'interrupteur ou préalablement chargée dans un latch du périphérique par le CPU. La figure 14 nous montre un circuit utilisant cette dernière possibilité avec un vecteur de 8 bits.

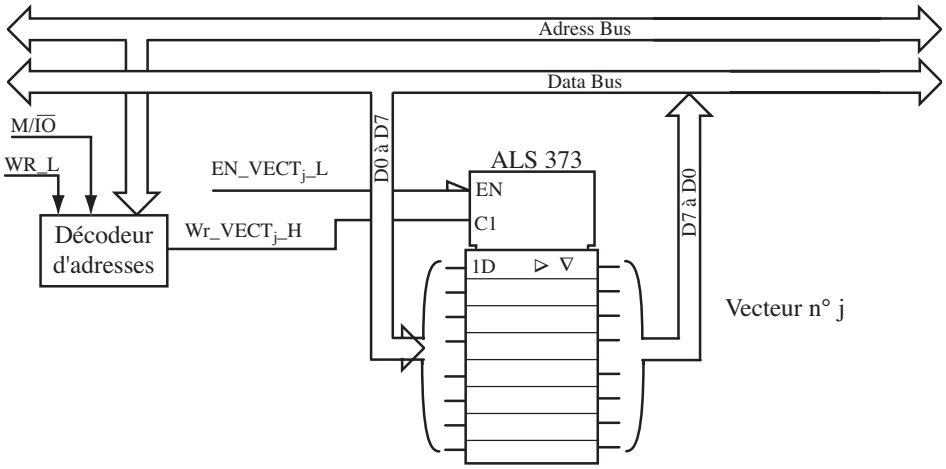


Figure XI- 14 : Circuit de vecteur

Selon que nous avons des interruptions à un seul niveau ou multiniveaux (*nested interrupts*), le problème de la prioritisation se résoudra plus ou moins simplement. Dans le cas d'interruptions à un seul niveau, un sous-programme de traitement d'une interruption x ne peut pas être interrompu à son tour par une demande d'interruption y, même si celle-ci a une plus haute priorité. Le CPU facilite un tel comportement en mettant automatiquement à l'état *disable* son *flag* d'interruption, aussitôt qu'il accepte une interruption. Il ne pourra donc y avoir de nouvelle interruption que lorsqu'une instruction EI (*Enable Interrupt*) aura été exécutée. Si cela n'est fait qu'à la fin du traitement d'une interruption, juste avant le retour au programme principal, nous obtenons un système à un seul niveau d'interruption. Dans un tel cas, le seul problème de priorité qui se pose est celui des demandes *simultanées*.

Dans un système acceptant des interruptions multiniveaux, il est possible qu'une demande d'interruption de niveau y soit acceptée alors que le traitement d'une interruption de niveau x est en cour, à condition que le niveau de priorité y soit supérieur au niveau x. Nous pouvons ainsi avoir une imbrication (*nesting*) de plusieurs interruptions de niveaux x,y,z croissants comme le montre la figure 15. Mais une demande de niveau w inférieur à x doit être bloquée par le circuit de prioritisation jusqu'à ce que le traitement de l'interruption de niveau x soit terminé.

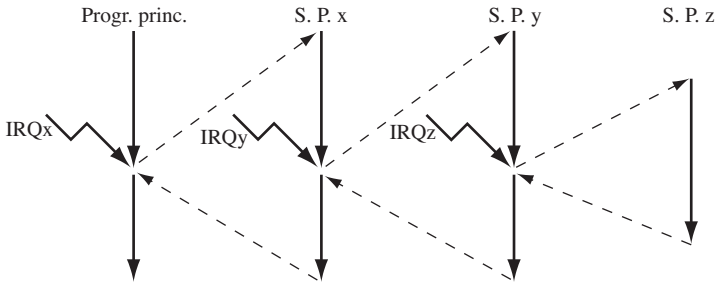


Figure XI- 15 : Cascade d'IRQ

Etudions tout d'abord des solutions adaptées aux systèmes à un seul niveau d'interruption. La première méthode qui vienne à l'esprit utilise un *encodeur de priorité* tel que celui que nous avons développé dans l'exercice II.15.2. La figure 16 nous montre un exemple d'un circuit de vectorisation pour 8 demandeurs d'interruption classées dans l'ordre de priorité croissant de I0 à I7.

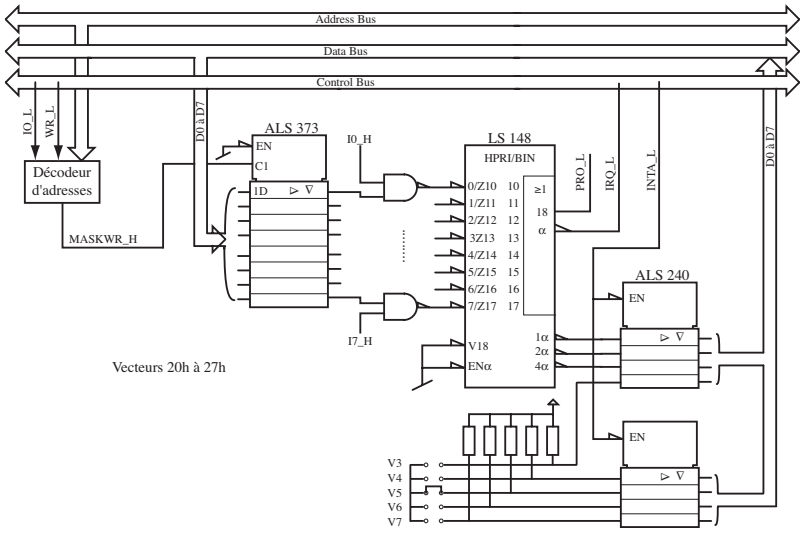


Figure XI- 16 : Circuit de vectorisation

L'encodage de priorité proprement dit est effectué par un circuit spécialisé 74LS148. La notation HPRI/BIN (*highest priority to binary*) spécifie qu'il s'agit d'un convertisseur *entrée prioritaire à binaire*, plus connu sous le nom d'encodeur de priorité. La relation v est une relation OU: ainsi, la sortie 18 résulte, non seulement du OU des signaux 10 à 17, mais également du OU avec l'entrée V18 (nous verrons l'utilité de cette entrée et de cette sortie sous peu).

Le tampon à trois états 74ALS240 permet d'envoyer sur le bus de données le vecteur de la demande d'interruption la plus prioritaire, ceci lorsque le signal d'acceptation de l'interruption INTA_L est actif. Le vecteur est composé de 8 bits: les trois bits de poids faible désignent le numéro du demandeur actif le

plus prioritaire parmi I0 à I7, alors que les cinq autres bits peuvent être choisis librement (en supposant que notre CPU accepte 256 vecteurs d'interruption).

Nous avons ajouté un latch de huit bits qui peut être chargé par le CPU. Les sorties de ce latch commandent des portes NAND qui permettent de masquer certaines demandes d'interruption et pas d'autre, selon les besoins du programme.

L'inconvénient d'un contrôleur d'interruption (nous utiliserons ce vocable désormais pour tout circuit s'occupant du masquage, de la prioritisation et de la vectorisation d'interruption) tel que celui de la figure 16 est que toutes les demandes d'interruption doivent être acheminées vers lui. Comme les demandes peuvent provenir de plusieurs cartes du système, le bus de liaison entre ces cartes devra comporter un nombre important de lignes de demande d'interruption, ce qui est coûteux.

Une **chaîne de priorité** (*daisy chain*) permet de décentraliser le contrôle des interruptions, donc d'améliorer la modularité et diminuer le nombre de liaisons inter-carte. Le principe est le suivant:

- Chaque circuit de contrôle d'un demandeur d'interruption dispose d'une entrée PRI (*priority in*) et d'une sortie PRD (*priority out*).
- Tous les contrôleurs sont connectés en chaîne, la sortie PRD de l'un étant reliée à l'entrée PRI du suivant, dans l'ordre de priorité décroissante.
- L'entrée PRI du demandeur le plus prioritaire (celui qui est au début de la chaîne) est mise à l'état actif par câblage.
- La sortie $PRO_j$ d'un maillon de rang j n'est active que si l'entrée $PRI_j$ est active et ce maillon ne demande pas d'interruption: $PRO_j\text{-}H = (PRI_j\text{-}H)(I_j\text{-}L)$; ainsi, si un maillon désire interrompre le processeur, il va automatiquement désactiver les entrées de priorité de tous les maillons de priorité inférieur.
- Lorsque le CPU accepte une interruption et génère le signal INTA, le seul demandeur actif (qui a placé une demande d'interruption) dont l'entrée PRI est active va mettre son vecteur sur le bus.

La figure 17 nous montre une implémentation d'un maillon d'une *daisy chain* dans un système à un seul niveau d'interruption.

Dans cet exemple, nous n'avons pas prévu de masquage mais il suffirait d'un flip-flop, plus exactement d'une sortie à un bit, et d'une porte ET pour ajouter cette possibilité. Par contre, nous avons introduit un détecteur de flanc suivi d'un flip-flop de synchronisation. Le détecteur de flanc permet au périphérique d'envoyer sa demande au contrôleur indifféremment sous forme d'un état ou d'un flanc actif (impulsion courte). La synchronisation permet d'empêcher qu'une demande de plus haute priorité ne vienne brouiller les cartes (modifier l'état de la chaîne) au cours d'un INTA, ce qui risquerait de perturber la lecture d'un code opératoire (*fetch*).

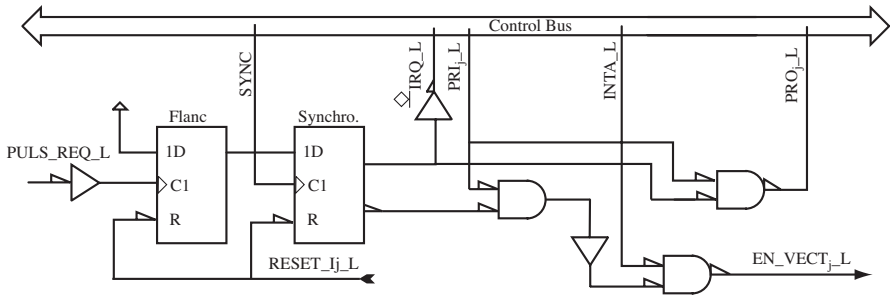


Figure XI- 17 : Circuit *Daisy Chain*

La génération du vecteur d'interruption n'est pas représentée dans la figure 17: il s'agit du circuit représenté dans la figure 14.

Les signaux PRI et PRO qui apparaissent dans l'encodeur de priorité 74ALS148 (voir figure 16) permettent bien sûr de réaliser une chaîne de priorité entre contrôleur d'interruption à encodeur. Pour obtenir un maillon à huit vecteurs à partir du contrôleur de la figure 16, il suffirait d'activer les tampons 74ALS240 avec le signal:

$$PRO_J\text{–}H \;=\; (PRI_J - H)\cdot(I_J - L) \qquad (2)$$

(au lieu de INTA_L), et de générer IRQ_L à partir de $\alpha_j$_L mais à travers un tampon non inverseur à collecteur ouvert, après avoir mis une résistance de *pull-up* sur $\alpha_j$_L, sans oublier de relier PRI et PRO aux maillons voisins.

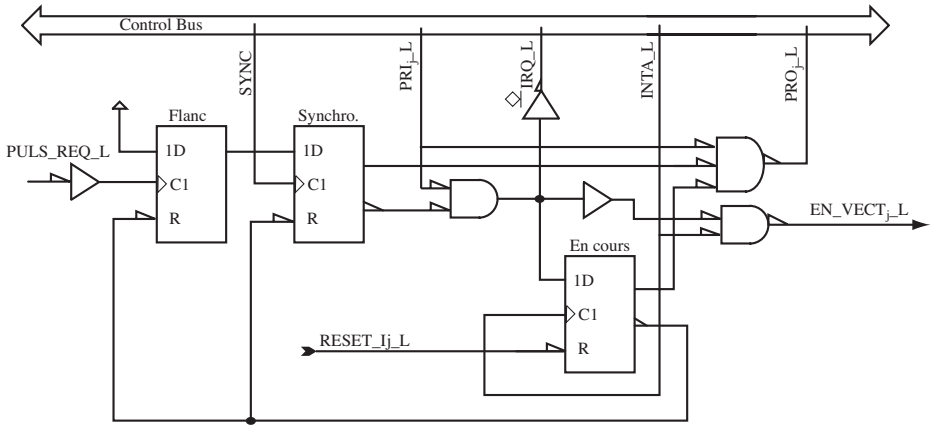


Figure XI- 18 : Circuit pour interruption multi-niveau

Dans le cas d'interruption imbriquée, ou multiniveau, le programme de traitement de chaque interruption doit mettre le *flag* d'interruption à *enable* dès le départ, de façon à laisser la voie libre à une interruption plus prioritaire. Par

contre, le contrôleur d'interruption doit inhiber toute demande de priorité inférieure.

La figure 18 nous montre une modification du circuit de la figure 17 en vue de l'adopter à des interruption multiniveaux.

La demande d'interruption IRQ est désactivé aussitôt que cette interruption est acceptée, ce qui est nécessaire pour mettre enable le flag d'interruption sans risquer d'interrompre de nouveau avec la même demande. Par contre, le flip-flop *en cours* mémorise le fait que la demande vient d'être acceptée et maintient désactivée la sortie PRO. Ce flip-flop, tout comme les flip-flops *flanc* et *synchro* de la figure 17 d'ailleurs, doit être remis à zéro à la fin du traitement de cette interruption, ce qui est généralement fait à travers une porte de sortie à un bit.

Nous n'allons pas essayer d'adapter le contrôleur à encodeur de la figure 16 au cas d'interruption multiniveau. Nous aboutirions en effet à un circuit très complexe, que l'on peut d'ailleurs obtenir sour forme intégrée (Am 2914, Intel 8259, etc), sans apprendre grand chose de nouveau quant aux principe.

Terminons ce tour d'horizon de l'aspect matériel des interruptions en récapitulant les nouveaux signaux de contrôle que nous avons été amenés à définir:

- IRQ : demande d'interruption masquable (entrée du CPU)
- NMI : demande d'interruption nom masquable et de plus haute priorité (entrée du CPU)
- INTA : impulsion de lecture du vecteur d'interruption, générée par le CPU lorsqu'il accepte une demande (sortie du CPU)
- SYNC : signal de synchronisation, pour le CPU que nous avons développé, le signal d'horloge suffit (sortie CPU)
- PRI, PRO : entrée et sortie de priorité dans un maillon d'une *daisy chain*; ces signaux (deux par maillon) n'ont rien à voir avec le CPU et ne sont pas *bussés* puisqu'ils sont spécifiques à un maillon

Pour compléter notre CPU, nous sommes donc obligés d'élargir d'un bit notre mémoire de micoprogramme pour générer INTA, à moins d'utiliser les trois bits de contrôle existant (RD, WR et M/IO) pour coder l'opération réalisée sur le bus: lecture mémoire, écriture mémoire, lecture entrée/sortie, écriture sortie, lecture code opératoire, lecture vecteur d'interruption,... et d'autres que nous verrons par la suite. Dans ce cas, un décodeur est utilisé en dehors du CPU de façon à décoder les signaux du bus de contrôle.

## XI-6 CPU et bus DMA

Jusqu'ici, nous avons considéré notre CPU comme le seul maître du bus, en ce sens qu'il décide souvrainement des transactions qui vont avoir lieu. En fait, ainsi que nous l'avons brièvement mentionné au § XI-4, il peut s'avérer nécessaire de gérer certaines entrées-sorties par accès direct à la mémoire

(DMA). L'accès direct à la mémoire se faisant à travers le bus, nous devons modifier notre CPU de façon à ce qu'il soit possible de lui retirer la maîtrise du bus au profit du contrôleur DMA.

Sans vouloir être exhaustif, nous étudierons la solution généralement implémentée dans les systèmes à microprocesseur comportant un ou plusieurs contrôleurs de DMA. Normalement, c'est le CPU qui a la maîtrise du bus, c'est-à-dire qu'il impose ses signaux de contrôle et d'adresse, voir même les signaux de données s'il s'agit d'une opération d'écriture. Lorsqu'un contrôleur de DMA veut s'approprier le bus, il doit en faire la demande au CPU à l'aide d'un signal appelé généralement HOLD ou BUSREQ (bus request) ou DMA-REQ (DMA request). Ce signal est testé par le CPU de façon similaire à une demande d'interruption, à la fin de chaque micoprogramme (instruction machine). Mais au lieu d'aller lire un vecteur d'interruption, le CPU va simplement mettre ses sorties à l'état haute impédance, sauf une sortie de quittance appelée HOLDA ou BUSACK (bus acknoledge) ou DMAACK, et attendre passivement que la ligne HOLS soit désactivée avant de reprendre son travail comme si de rien n'était. La ligne de quittance HOLDA est nécessaire, bien qu'une demande de bus ne soit généralement pas masquable, car il faut lasser au CPU le temps de réagir: il doit terminer l'instruction en cours avant de s'apercevoir de la demande et de passer à l'état haute impédance.

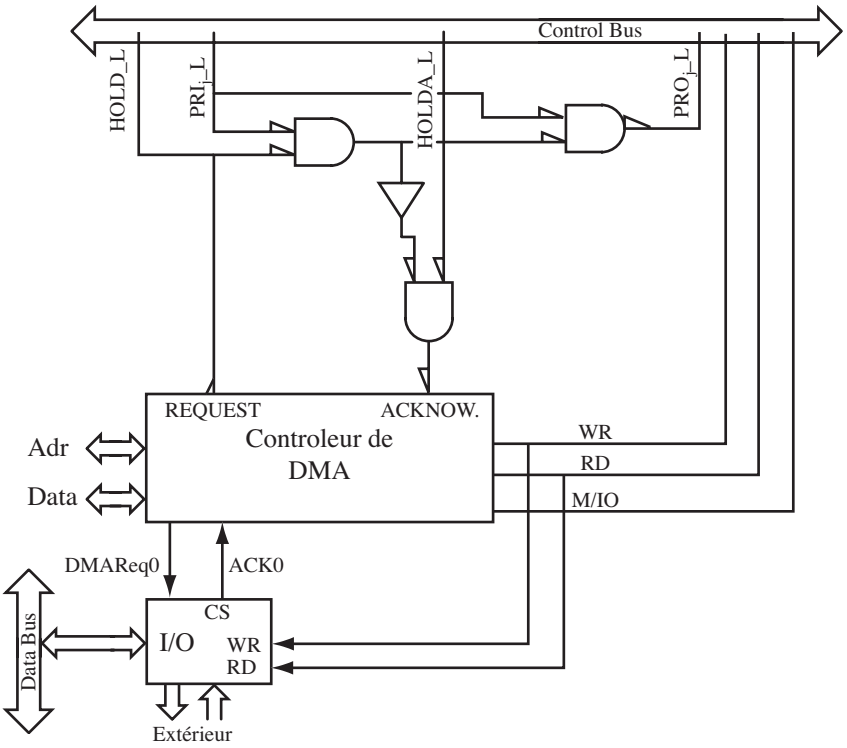


Figure XI- 19 : Connection d'un contrôleur DMA sur un bus

Nous étudierons le contrôleur de DMA dans le chapitre sur l'interfaçage. Pour l'instant, nous nous contenterons de savoir que la connection d'un contrôleur

de DMA au bus pour lequel il a été prévu, ne demande pas de circuiterie supplémentaire. S'il y a plusieurs entrées/sorties en DMA dans un système, la priorité est généralement réglée à l'aide d'une chaîne de priorité (daisy chain) similaire à celle utilisée pour les interruptions à un seul niveau. La figure 19 nous en montre un maillon.

Pour qu'il permette le DMA, notre CPU devra donc comprendre une entrée supplémentaire HOLD (ou BUSREQ ou DMAREQ) pour recevoir une demande, et une sortie supplémentaire HOLDA (ou BUSACK ou DMAACK) pour signifier son acceptation. Comme pour le INTA, le HOLDA peut être obtenu par une combinaison des trois bits de μprogramme déjà existant à savoir RD, WR et M/IO, puisque nous avons affaire à des situations mutuellement exclusives: le CPU ne peut à la fois accepter une demande de DMA ou une interruption et lire un code opératoire, par exemple.

## XI-7 Bus synchrones, semi-synchrones et asynchrones

Dans les instructions que nous avons microprogrammées pour notre CPU, nous avons toujours ménagé un temps correspondant à deux périodes d'horloge entre l'envoi d'une adresse et la conclusion d'une lecture ou d'une écriture. Ceci tient compte du fait que le temps d'accès de la mémoire est généralement supérieur au temps de cycle du CPU. En procédent de cette manière nous obtenons des transferts dits **synchrones** : c'est le CPU qui donne la cadence, et le temps alloué pour un transfert ne peut être varié qu'en variant la fréquence de l'horloge ou en modifiant la microprogrammation. Cette rigidité fait que, la microprogrammation étant normalement inaccessible dans les microprocesseurs, la fréquence d'horloge doit être choisie en fonction du module de mémoire ou d'entrée/sortie le plus lent faisant partie du système, limitant ainsi les performances de l'ensemble.

Lorsqu'un CPU doit communiquer avec des mémoires et des entrées/sorties ayant des temps d'accès très différents les uns des autres, il est intéressant d'utiliser un protocole de transfert qui adapte les temps de transfert aux possibilités des différents circuits. C'est le cas des protocoles asynchrones totalement ou partiellement entrelacés, dont les figures 20 (a) et (b) nous montre le principe.

Dans un protocole asynchrone, l'initiateur d'un transfert (il s'agit généralement du CPU) active un signale de commande READ ou WRITE et attend la réponse de son interlocuteur sour la forme de l'activation d'un signal de quittance, souvent appelé DATACK (data acknowledge). Lorsque le CPU (le maître du bus) reçoit la quittance, il termine le transfert et désactive le READ ou le WRITE. Dans un protocole asynchrone partiellement entrelacé, la conversation s'arrête là, et la désactivation de DATACK par l'interlocuteur (esclave) a lieu après un temps indépendant du CPU. Par contre, dans un protocole totalement entrelacé, la désactivation du READ ou du WRITE par

le maître indique à l'esclave que le transfert est terminé et qu'il peut alors désactiver DATACK; ce n'est que lorsque DATACK sera inactif que le CPU pourra initier un autre cycle READ ou WRITE.

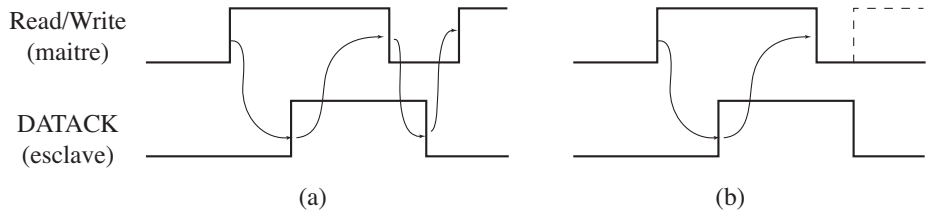


Figure XI- 20 : Chronogramme d'un protocole asynchrone

Un protocole asynchrone partiellement entrelacé (figure 20 (b)) impose de spécifier une durée maximum à ne pas dépasser pour l'impulsion DATACK, sous peine de voir un nouveau cycle READ ou WRITE commencer alors que l'impulsion de quittance du précédent n'est pas terminée (voir traitillé dans la figure 20 (b)). Le protocole asynchrone totalement entrelacé ne souffre pas d'un tel inconvénient. Par contre, son plus grand nombre d'échanges conversationnels le rend plus lent et plus gourmand en circuiterie.

Le protocole de loin le plus utilisé dans les microprocesseurs est de type **semi-synchrone**. Dans ce protocole, au lieu de conclure un transfert dans un nombre de cycles d'horloge fixe comme dans un protocole synchrone, le CPU va tester un signal appelé READY (prêt) ou BUSY (occupé) ou WAIT (attend) avant de terminer son impulsion READ ou WRITE. Si ce signal, généré par l'interlocuteur interpelé, indique qu'il faut attendre (READY_H = 0 ou BUSY_H = 1 OU WAIT_L = 0), le CPU attend un cycle d'horloge complet puis reteste ce signal et ainsi de suite. Ainsi donc, si une mémoire ou une entrée/sortie ne sont pas assez rapides pour effectuer des échanges avec le CPU selon le protocole synchrone, elles peuvent lui demander d'attendre un certain nombre (entier) de périodes d'horloge.

Le protocole semi-synchrone permet donc de relier par un bus des éléments ayant des temps de réponse très divers. Comme seuls les éléments trop lent pour travailler en synchrone doivent générer le signal WAIT (ou READY ou BUSY), le protocole semi-synchrone est plus économique que les protocoles asynchrones, même le partiellement entrelacé. Par contre, vu que la demande d'attendre doit arriver au CPU dans certain délais (avant qu'il teste la ligne WAIT), le bus sera limité en longueur (le temps de propagation d'un signal le long d'une ligne est proportionnel à la longueur de la ligne). De plus, l'attente durant obligatoirement un nombre entier de cycles d'horloge, elle sera souvent supérieur au strict minimum.

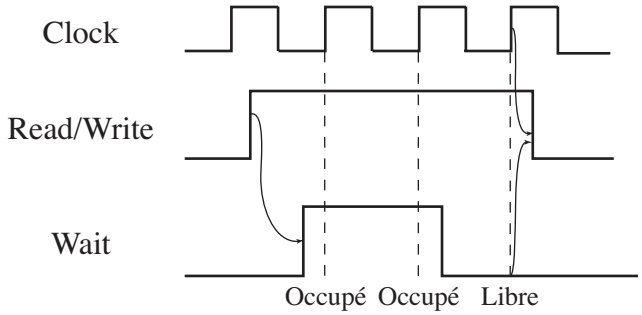


Figure XI- 21 : Chronogramme d'un protocole semi-synchrone

La figure 21 nous montre un échange semi-synchrone avec deux cycles d'attente.

Le signal WAIT est généré de façon synchrone, par exemple avec le circuit de la figure 22, plutôt qu'avec un monostable.

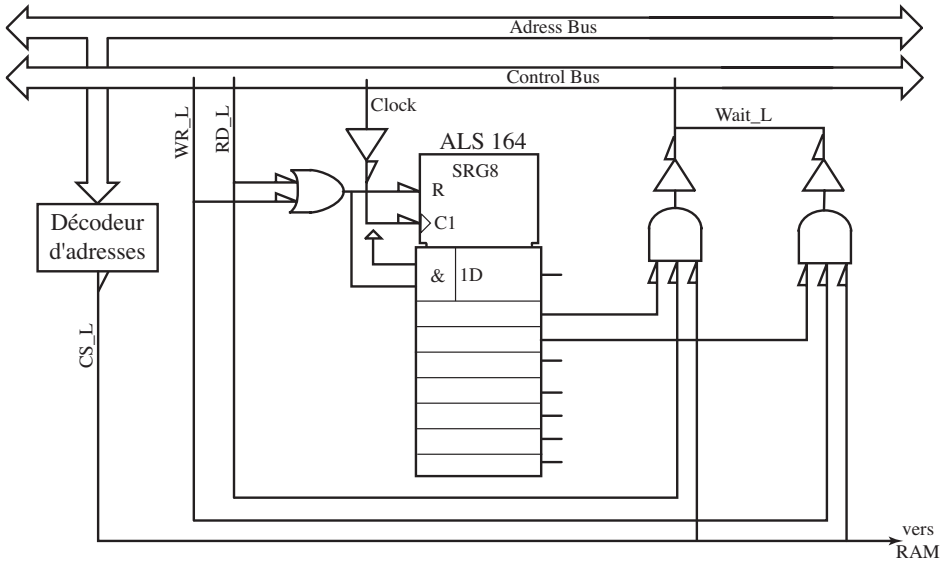


Figure XI- 22 : Circuit permettant la génération d'un signal Wait synchrone

Ce schéma nous montre comment générer une période d'attente à la lecture et deux à l'écriture, par exemple pour une RAM. Lorsque ni READ ni WRITE ne sont actifs, le registre à décalages est remis à zéro. Aussitôt que la RAM est activée (CS_L devient actif) en lecture ou en écriture, le signal WAIT devient actif. Il faut noter qu'il est généré par une sortie à collecteur ouvert pour pouvoir effectuer un OU câblé avec les signaux correspondants générés par les autres éléments lents connectés sur le bus. Alors que le signal WAIT est testé sur le flanc montant de l'horloge, il reste actif, dans ce montage, jusqu'au flanc descendant du deuxième (en lecture) ou du troisième (en écriture) coups d'horloge qui suit l'apparition du READ ou du WRITE.

Pour que notre CPU effectue ses transferts sur le bus selon ce protocole semi-synchrone, il suffit de le doter d'une entrée supplémentaire, WAIT_L, tester cette entrée dans les microprogrammes avant chaque transfert, et attendre qu'elle soit inactive. Souvent, ce travail supplémentaire ne demandera pas de microinstruction supplémentaire.

## XI-8 Bus multiplexés

En intégrant un CPU en un seul boîtier, on se heurte à un problème d'encapsulage: le nombre de connections que nécessite le CPU, donc le nombre de pattes du boîtier, dépasse facilement la cinquantaine, ce qui rend le boîtier coûteux. Pour diminuer le nombre de connections, il est possible de coder les signaux de commande. Nous avons déjà préconisé cette technique pour économiser de la mémoire de microprogramme. Les signaux de commande sortant du CPU étant codés, il faudra alors utiliser un décodeur externe.

Une autre technique permettant de diminuer considérablement le nombre de connections du CPU est le multiplexage des adresses et/ou des données: les mêmes lignes permettront de transmettre tantôt des adresses, tantôt des données, l'orchestration étant faite bien sûr par le CPU à l'aide de signaux de commande.

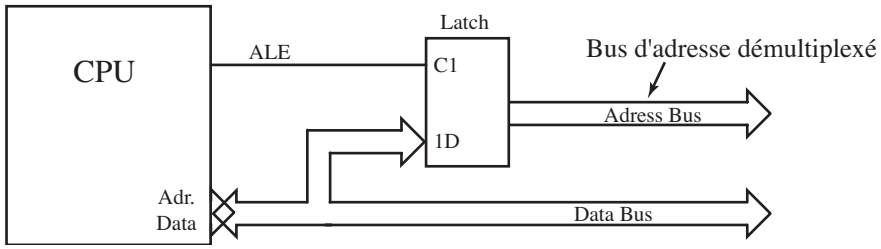


Figure XI- 23 : Exemple de bus multiplexé

Dans le CPU que nous avons développé, multiplexons les 16 bits de données, par exemple. Il suffit pour cela de commander de façon appropriée les *output enable* des registres MAR et DOR, ce qui n'introduira que des changements minimes des nos microprogrammes. Mais il faudra bien pouvoir indiquer si c'est une adresse ou une donnée qui est sur le bus, et pour cela un signal de commande supplémentaire est nécessaire. Appelons-le ALE pour *Address Latch Enable*, et activons ce signal lorsque l'adresse est présente: il nous permettra alors de mémoriser l'adresse dans un latch et d'effectuer ainsi simplement le démultiplexage, comme nous le montre la figure 23.