LI312 - Cours
Architecture logicielle et matérielle des ordinateurs

BARON Benjamin

Le 5 septembre 2013

D'après le cours de Alain Greiner

Chapitre 1

Architecture externe du processeur MIPS32

1.1 Introduction

Le processeur lit (fetch) le code d'instruction qu'il doit exécuter.
Il est alors capable de lire une instruction par cycle.

La valeur 32 bits fait référence à :
- La taille de l'adresse (adresse codée sur 32 bits) ;
- La capacité de stockage interne du processeur (ie. La capacité des registres).

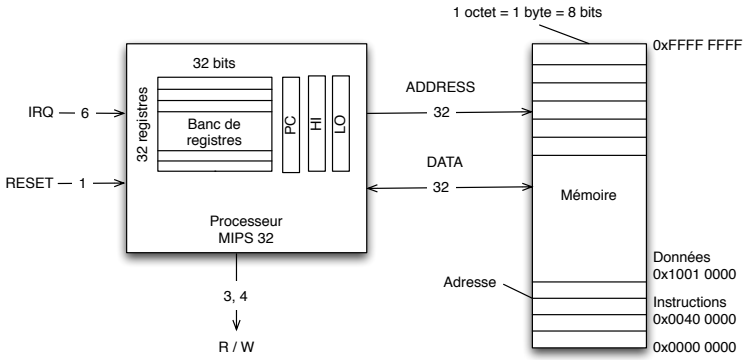


Figure 1.1 – Schéma du processeur MIPS 32

Remarque. Le MIPS est un processeur RISC (*Reduced Instruction Set Computer*) $\neq$ CISC (*Complex Instruction Set Computer*). Les instructions que le processeur manipule comportent 3 registres : 2 registres sources et un registre destination. De ce fait, il faut un même nombre de cycles pour exécuter chaque instruction du jeu d'instruction. Il est donc possible de débuter une instruction à chaque cycle d'horloge (*pipeline*).

Entrées / sorties du processeur :
- ADDRESS sur 32 bits en sortie ;
- DATA sur 32 bits en entrée / sortie ;
- Read / Write généralement sur 3 ou 4 bits, en sortie ;
- IRQ sur 6 bits en entrée – permet à un périphérique de déclencher une interruption ;
- RESET sur 1 bit en entrée – permet initialisation (ie. Mettre la machine dans un état connu).

Il existe différents types de registres.

Registres internes visibles du logiciel :
- 32 registres généraux sur 32 bits (notés $R(i)$ pour $i \in [0, 31]$) ;
- PC : Program Counter – sauvegarde la position de la dernière instruction exécutée ;
- HI et LO – utilisés pour les opérations arithmétiques + et /.

Registres protégés – non accessibles en mode utilisateur, seulement en mode superviseur :
- SR : Status Register sur 2 bits :
  - Processeur en mode user / root ;
  - Interruptions masquées ou non.
- CR : Cause Register :
  - Interruption matérielle = message d'un périphérique à un logiciel ;
  - Exception : programme utilisateur fait quelque chose d'interdit ⇒ programme tué, dénonciation de la cause de l'erreur ;
  - Appel système : appel à l'OS (eg. Ecriture disque).
- EPC : Exception Program Counter – pour les interruptions matérielles et les exceptions ;
- BAR : Bad Address Register – Erreur d'adressage (eg. segmentation fault) ;
- PROCID : Numéro de processeur : identifiant du processeur (utile pour les processeurs multicore) ;
- CYCLECOUNT : compteur de cycles.

Il y a deux modes de fonctionnement du processeur :
- mode utilisateur ;
- mode superviseur (mode *kernel*).

Remarque. Le matériel surveille en permanence le logiciel en train de s'exécuter. S'il tente d'accéder à une zone protégée, le matériel va lever une exception.

1.2 Communication avec la mémoire

La mémoire est un tableau d'octets (à l'image d'une commode et de ses tiroirs).

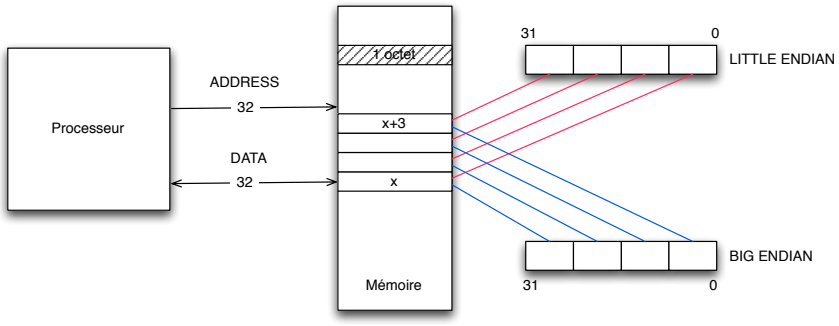


FIGURE 1.2 – Communication avec la mémoire

Remarque. Lorsque l'instruction exécutée par le processeur cherche à lire un mot (ie. un ensemble de 32 bits alignés), l'adresse **doit** être un multiple de 4 (ie. de la forme 0x——00).

Dans le cours nous utiliserons la disposition *little endian* : l'adresse la plus petite correspond à l'octet le plus petit (de poids faible).

L'adressage est divisé en deux zones :
- La zone utilisateur (*user*) ;
- La zone superviseur (*kernel*).

Remarque. Chaque fois que le logiciel demande au processseur d'exécuter une instruction qui accède au segment superviseur, le matériel vérifie que le bit de mode du registre STATUS REGISTER (SR) à la valeur *superviseur*.

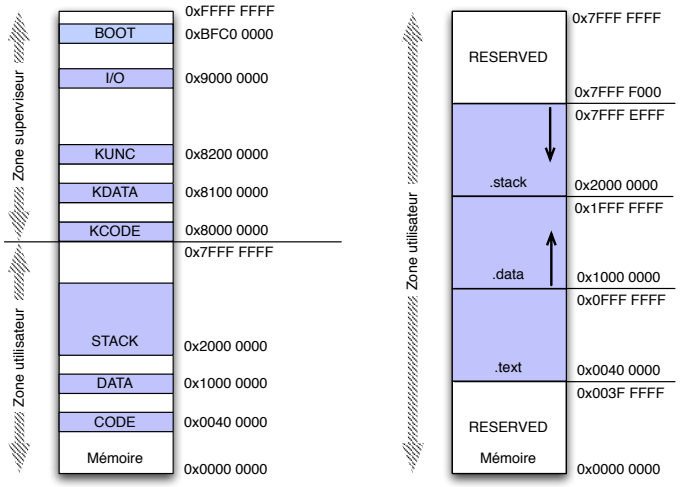


FIGURE 1.3 – Segmentation de la mémoire

Segments réservés de la partie superviseur :

- **Code de BOOT** : code qu'exécute la machine lors du démarrage / reset de la machine. Ce code est situé dans la mémoire ROM (ie. mémoire morte) et constitue le BIOS ;
- **KCODE** : code protégé du système d'exploitation. Le BIOS lit sur le disque le code du système d'exploitation et le range dans cet espace mémoire (lors du boot de la machine) ;
- **KDATA** : données globales privées de l'OS – tables contenant les contextes d'exécution des tâches en mode multiprocesseur notamment ;
- **UNC** : données non cachables (ie. les valeurs ne sont pas recopiées dans le cache). Ce sont des tampons de communication entre le processeur et les périphériques ;
- **I/O** : contrôleur des périphériques. Un périphérique peut être partagé par plusieurs utilisateurs différents. I/O est alors dans la partie superviseur de l'espace adressable.

Segments réservés de la partie utilisateur :

- **CODE** : code du programme utilisateur, ainsi que le code des fonctions permettant à un programme utilisateur d'accéder aux périphériques grâce aux appels systèmes ;
- **DATA** : données globales du programme utilisateur ;
- **STACK** : pile d'exécution du programme.

Remarque. Des espaces adressables sont câblés en dur dans le matériel :

- Le segment BOOT (0xBFC0 0000). Cette adresse câblée est commune à tous les processeurs ;
- La séparation mode superviseur (2 Go) / mode utilisateur (2 Go) pour un processeur sans mémoire virtuelle.

1.3 Format des instructions

Toutes les instructions (jeux d'instructions) sont codées sur 32 bits (alignés en mémoire).

Remarque. Vrai pour le MIPS R3000, mais par exemple pour les processeur x86 d'Intel, il y a la possibilité d'avoir des instructions codées sur 1, 2, 4, 6, 8 octets.

Il y a 3 sous-formats :

- Le **format R** est utilisé par les instructions nécessitant 2 registres sources (désignés par RS et RT) et un registre résultat désigné par RD (`add`, `or`...) ;

– Le **format I** est utilisé par les instructions de lecture/écriture mémoire (`lw` et `sw`), par les instructions utilisant un opérande immédiat (`addi`, `andi`...), ainsi que par les branchements courte distance (conditionnels) (`bne`, `beq`...) ;
– Le **format J** n'est utilisé que pour les branchements à longue distance (inconditionnels) (`jal`, `j`...).

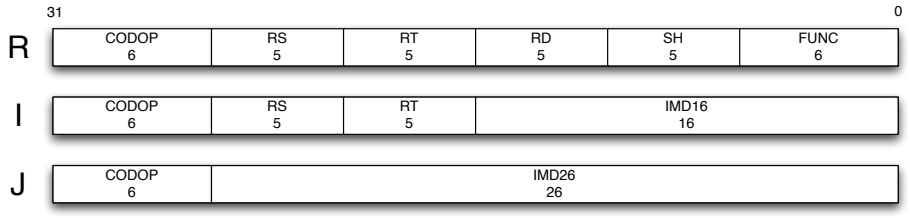


FIGURE 1.4 – Formats d'instructions du MIPS 32

Définition (Valeur immédiate)**.** Valeur que l'instruction va utiliser comme opérande sans avoir à la lire dans un registre ou en mémoire.

1.4 Conventions d'utilisation de la pile d'exécution

On souhaite utiliser les conventions utilisées par GCC.

Les 3 fonctions de la pile :

1. Transmission des arguments de la fonction appelante vers la fonction appelée ;
2. Sauvegarde de la valeur des « registres persistants » ;
3. Zone de travail : variables locales.

Remarque. Les deux instructions permettant d'appeler une fonction :
– jal PC ← IMD26
R(31) ← PC + 4
– bgezal PC ← IMD26
R(31) ← PC + 4

Par convention, on utilise le registre R(29) comme pointeur de pile (*Stack Pointer*). Le pointeur de pile pointe sur les dernières case occupées dans la pile.
La pile est remplie à partir de l'adresse 0x7FFF EFFF et utilise un système d'adressage *décroissant*.

On a alors :
– $na = \max(na_i, 4)$ où na_i représente ne nombre d'arguments des fonctions g_i appelées par f ;
– nv est le nombre de variables locales utilisées en C ;
– nr est le nombre de registres persistants utilisés + \$31.

Définition (Registres persistants et registres temporaires)**.** On distingue deux catégories de registres :
– Les registres **persistants** : sauvegarder leur état initial dans la pile avant l'exécution d'une fonction, puis restaurer l'état initial de ces registres à la fin de l'exécution de la fonction.
⇒ Registres \$16 à \$23 et \$28 à \$31 ;
– Les registres **temporaires** : à ne pas sauvegarder dans la pile. Une fonction peut les modifier sans en restaurer la valeur initiale.
⇒ Registres \$1 à \$15 et \$24, \$25 ;

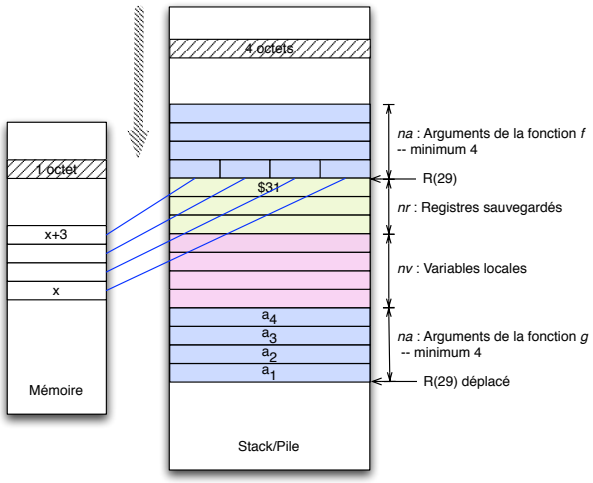
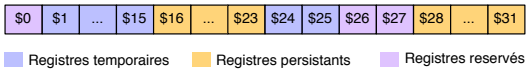


FIGURE 1.5 – Schéma de la pile d'exécution

– Les registres **réservés** au système d'exploitation : non utilisables par les applications utilisateur.
⇒ Registres \$0, \$26 et \$27.



Remarque. Utilité de réserver des places dans la pile pour les variables locales (nv) :
– Ca ne coûte pas cher en mémoire : on re-libère la mémoire à la fin de l'exécution de la fonction ;
– Si il y a des variables locales de tailles importantes (eg. des tableaux), on utilise la mémoire réservée dans la pile pour les stocker temporairement.

1.5 Optimisations de GCC

Le compilateur GCC opère au minimum deux optimisations.

La première. Distinguer les registres persistants (\$16 à \$23) des registres à vie courte. De ce fait, on va sauvegarder leur valeur dans la pile.

La seconde. Passage des arguments par registres. En effet, la transmission des arguments par la pile (mémoire) coûte cher en utilisation du processeur :
– Si les valeurs cherchées se trouvent dans le cache, c'est OK (environ 1 cycle) ;
– Si les valeur cherchées ne se trouvent pas dans le cache, il va falloir aller les chercher en mémoire (environ 400 cycles).

Objectif. Amélioration des performances.

Quand il y a moins de 5 arguments (de 1 à 4), GCC écrit la valeur des arguments dans les registres \$4, \$5, \$6 et \$7.

Il n'écrit pas les valeurs de ces 4 premiers arguments, mais réserve leur place dans la pile quand même, en plus de réserver une place dans la pile pour les autres arguments.

$$f(\underbrace{a_1, a_2, a_3, a_4}_{\$4,\ \$5,\ \$6,\ \$7}, \underbrace{a_5, \ldots, a_n}_{\text{pile}})$$

Remarque. Si une fonction f a au moins 4 arguments, elle en transmet 4 par registre et réserve leur place dans la pile. Si cette fonction appelle une autre fonction g. De ce fait, f va sauvegarder la valeur des registres \$4 à \$7 à leur place appropriée dans la pile pour continuer à utiliser les valeurs des arguments.

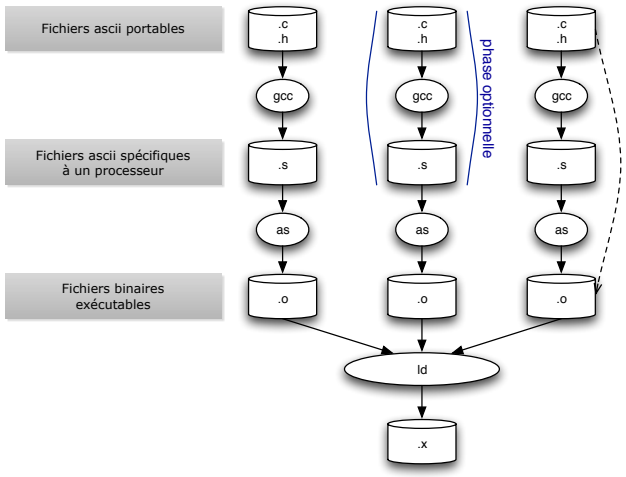
1.6 Organisation générale d'une chaîne de compilation – GCC

On utilise 4 outils :

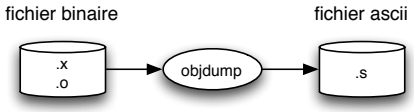
- assembleur *as* : `mipsel-li321-elf-as` ;
- compilateur *gcc* : `mipsel-li321-elf-gcc` ;
- linker *ld* : `mipsel-li321-elf-ld` ;
- désassembleur *objdump* : `mipsel-li321-elf-objdump`.

Remarque. Le nom `mipsel-li321-elf-XX` correspond à :

- `mpisel` : processeur MIPS, Little Endian ;
- `li321` : nom du système qui s'exécute ;
- `elf` : le code binaire généré respecte le format elf ;
- `XX` : outil utilisé.



(a) Chaîne de compilation



(b) Phase de décompilation — débogage

La compilation séparée est possible. Les fonctions d'un fichier peuvent faire référence (via des adresses mémoires) à des fonctions issues d'autres fichiers.

L'édition de liens (ld) résout les adresses : assigner des valeurs binaires aux valeurs symboliques des adresses (labels).
Les valeurs non résolues dans les fichiers binaires sont stockées dans des tables de symboles. On remplace alors les adresses non résolues à la fin de la chaine de compilation.

Les différents modules logiciels qui vont être exécutés n'entrent pas forcément dans la chaîne de compilation au niveau du fichier `*.c`. On a la possibilité de compiler à la fois des fichiers `*.h`/`*.c` et `*.s`.

Comment contrôler cette chaîne de compilation ?

Via l'édition de liens principalement : donner au programme d'éditeur de liens des consignes d'exécution :

- Chaque fichier objet (`*.o`) contient une ou plusieurs section (eg. section `.data`, `.text` dans un fichier assembleur).
  On peut guider le linker pour qu'il regroupe une ou plusieurs *section* dans un *segment* (KCODE, KDATA, DATA, CODE...) de l'espace adressable.
  Exemple. Regrouper le contenu de la section `.text` des différents fichiers objets (`*.o`) dans le segment CODE de l'espace adressable.
- On peut définir les adresses de base des différents segments cibles.

Chapitre 2

Bus système

2.1 Introduction

Le programme s'exécutant sur le processeur peut accéder à différents services via le bus système. Ce dernier permet la communication du processeur avec le monde extérieur.

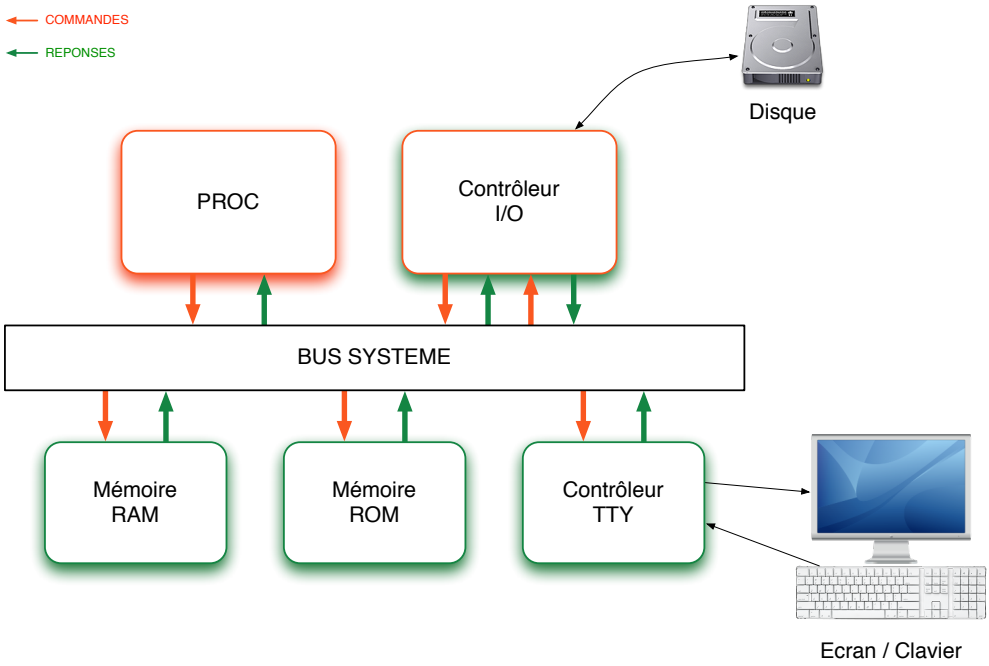


Figure 2.1 – Représentation de la machine dans son ensemble

Définition (Transaction)**.** Echange d'informations en utilisant le bus système. Il s'agit d'une écriture (`sw`), ou d'une lecture (`lw`) à une certaine adresse.

Toujours une paire :

- Une commande ;
- Une réponse.

Remarque. Il existe une réponse même dans le cas d'une commande écriture.

Le processeur peut commander une écriture vers autre chose qu'une mémoire. Il peut exécuter cet ordre sur un périphérique.

Définition (Maître / Initiateur)**.** Composant matériel capable de démarrer une transaction (ie. envoyer une commande).

Définition (Esclave / Cible)**.** Composant matériel capable de recevoir une commande et d'y répondre.

Remarque. Attention : un même composant matériel peut être à la fois maître et esclave.

Par exemple : le contrôleur de disque : Disque $\longleftrightarrow$ Mémoire RAM.
Il a besoin d'adresser la mémoire pour lire / écrire dans celle-ci.

2.2 Services offerts par le bus système

Hypothèse. Le bus système n'exécute qu'une seule transaction à la fois.

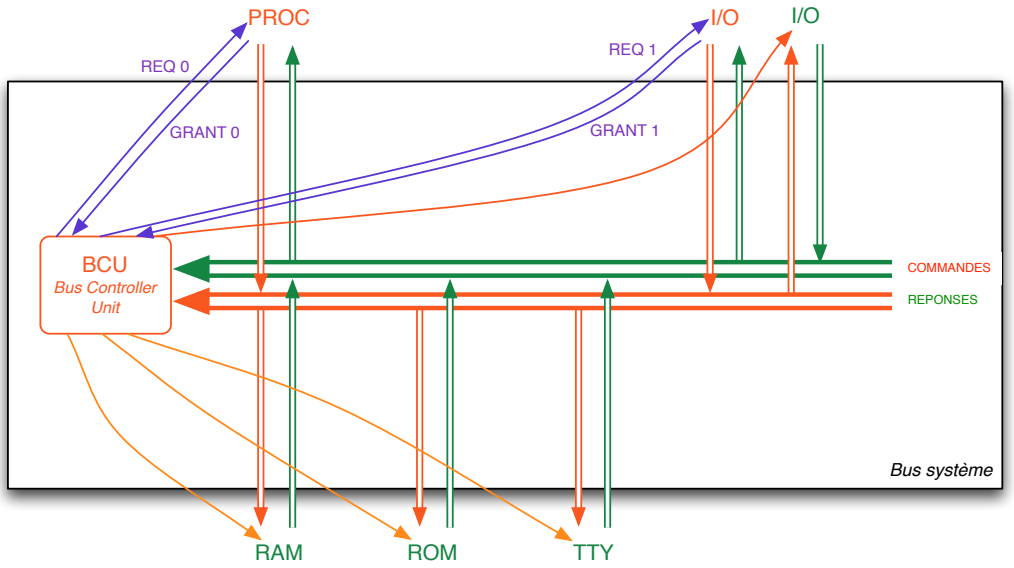


FIGURE 2.2 – Le bus système

Arbitrage entre les différents maîtres

Le bus système est une ressource partagée entre au moins deux maîtres.

Du fait du parallélisme et des requêtes asynchrones provenant de différents entités, des collisions apparaissent. Un arbitrage est nécessaire.

Deux maîtres veulent utiliser en même temps le bus système. Il faut donc assurer un mécanisme d'arbitrage pour accéder à la ressource partagée (ici : le bus d'adresse).

Déroulement d'une transaction :

- Request : avant d'émettre une transaction, l'initiateur demande l'autorisation d'émettre au BCU (*Bus Controller Unit*) ;
- Grant : l'initiateur attend ensuite la réponse du BCU.

Le rôle du BCU : choisir un initiateur. L'initiateur non élu devra attendre la fin de l'occupation de la ressource partagée par l'initiateur élu par le BCU.

Décodage de l'adresse émise par le maître

Un maître va transmettre l'adresse de la cible. Le bus système va devoir décoder l'adresse afin de bien acheminer la commande.

Signification de l'adresse. Le logiciel va demander aux maîtres d'effectuer des transactions. C'est-à-dire exécuter des commandes `lw` ou `sw` à une certaine adresse qui va désigner un composant matériel.

Segmentation de l'espace adressable (ensemble de 4 milliards d'adresses).
Un segment est composé d'une adresse de base et d'une longueur.

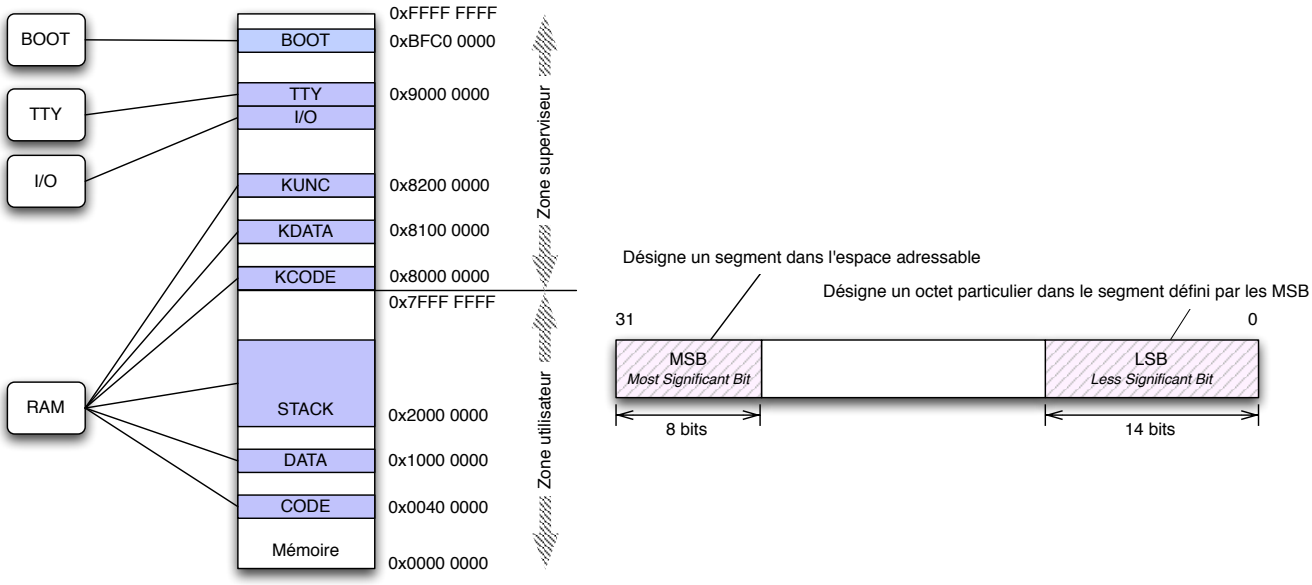


FIGURE 2.3 – La segmentation de l'espace adressable

Toutes les adresses sont alignées sur les frontières de mots de 32 bits. Ce sont donc des multiples de 4.
Dans le cas du segment de l'espace adressable associé au contrôleur TTY, l'encombrement est de 4×4 octets.

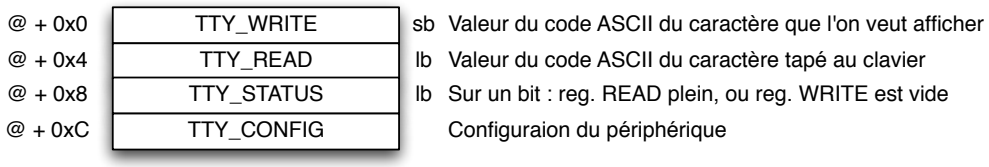


FIGURE 2.4 – Le segment associé au contrôleur TTY (0x9000 0000)

Remarque. Il y a autant de segments TTY dans l'espace adressable qu'il n'y a de périphériques TTY. Par exemple, s'i y a 3 périphériques TTY, il y aura alors 3 segments TTY correspondant à 12 registres utilisés (soit un encombrement de $3 \times 12 = 48$ octets).

Le programme utilisateur doit utiliser des appels systèmes pour accéder au TTY puisque le segment TTY se trouve dans la zone kernel (superviseur) de l'espace adressable. Puisque le contrôle des périphériques est exclusivement géré par l'OS, il faut impérativement passer par lui au travers d'un appel système (`syscall`)

Transmission de la réponse

Le bus système va se charger de transmettre la réponse du périphérique cible vers le périphérique maître qui a initié la transaction.

2.3 A propos des accès mémoire

A chaque exécution d'une instruction, le processeur doit lire en mémoire le code de l'instruction suivante. Par ailleurs, il y a différents types d'accès mémoire :

- Lecture de données (`lw`) ;
- Ecriture de données (`sw`).

Dans un programme, on trouve une moyenne de :

- 20% : lecture de données (`lw`) ;
- 10% : écriture de données en mémoire (`sw`) ;
- 40% : opérations entre registres (eg. `add`) ;
- 30% : branchements (eg. `jump`).

De ce fait, pour 100 instructions, on a :

- 120 lectures en mémoire ;
- 10 écritures en mémoire.

Remarque. De plus, une écriture est non bloquante pour un programme, alors que pour une lecture, le processeur est bloqué tant que la valeur n'a pas été lue.

Chapitre 3

Mémoires caches

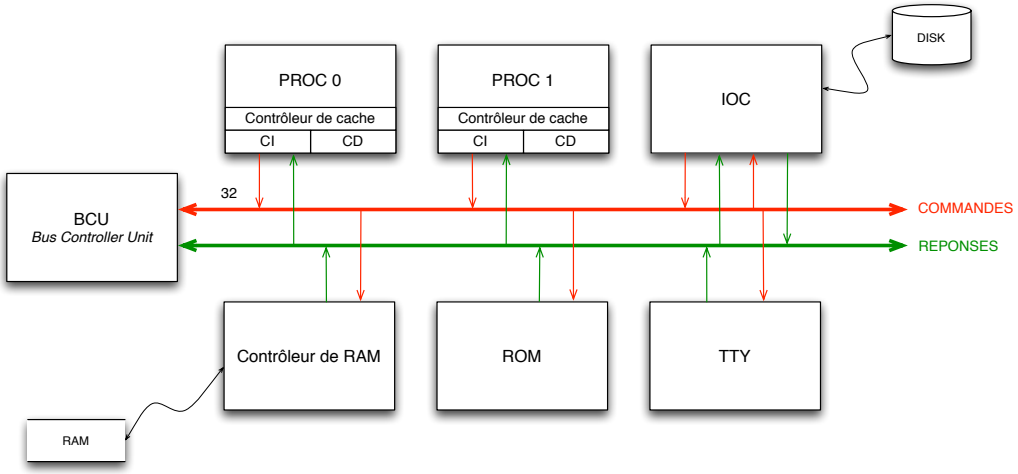


Figure 3.1 – Composition matérielle d'un ordinateur

3.1 Introduction

Définition (Cycle)**.** Période entre deux fronts montants du signal périodique envoyé (eg. un processeur 25 MHz aura un cycle de $1/25\,000\,000$ seconde).

Le signal est distribué à tous les composants matériels $\Rightarrow$ même référence de temps entre tous ces composants.

Objectif. Eviter au processeur de passer par le bus et le BCU pour accéder à la mémoire RAM.

Les mémoires caches doivent être très rapide pour répondre aux requêtes de lecture.

- Le temps de cycle du cache doit être égal au temps de cycle du processeur (ie. le cache doit répondre en 1 cycle) ;
- Cependant la capacité stockage du cache est très faible.

Caches de premier niveau :

- Cache d'**instruction** (CI) : mémoire spécifique contenant des instructions ;
- Cache de **données** (CD) : mémoire spécifique contenant des données.

Il existe donc deux types de mémoire :

- La **mémoire dynamique** – DRAM qui a une grande capacité de stockage
  L'accès à celle-ci est lent (au moins 100 fois plus lent que le cycle du processeur).

$$1 \text{ bit} \equiv 1 \text{ transistor}$$

– **Mémoire statique** – SDRAM : très rapide (calée sur le cycle du processeur).
Il y a cependant une faible capacité de stockage.

$$1 \text{ bit} \equiv 6 \text{ transistors}$$

Une mémoire cache contient des copies des informations présentes dans la mémoire principale :
- CI : copie d'extraits du segment `.code` de la mémoire DRAM ;
- CD : copie d'extraits du segment `.data` et `.stack` de la mémoire principale.

3.2 Comment faire en sorte de minimiser le taux de MISS ?

Définition (Taux de MISS)**.** Défini par la quotient :

$$\frac{\text{nombre de requêtes processeur qui sont des échecs}}{\text{nombre total de requêtes du processeur}}$$

Définition (Coût de MISS)**.** Nombre de cycles mémoire pour réalimenter le cache. C'est le nombre de cycles pendant lesquels le processeur est gelé.

La technique de cache est efficace grâce au principe de localité.

La localité spatiale

Si le processeur émet une requête à l'adresse x, les requêtes suivantes ont une forte probabilité de voir des adresses proches de x (eg. `PC`, `PC+4`, `PC+8`, ...).

Les éléments concernés sont :
- Les instructions sont rangées à des adresses voisines ;
- Les tableaux données rangées à des adresses consécutives.

La localité temporelle

Si le processeur émet une requête à l'adresse x au cycle n, il arrive très souvent qu'une autre requête soit effectuée à le même adresse x, au cours des cycles suivant le cycle n.

Les éléments concernés sont :
- Les programmes contiennent des boucles (cache d'instructions) ;
- Les programmes contiennent des compteurs (cache de données).

3.3 Comment exprimer le voisinage ?

Définition (Ligne de cache)**.** Découpage / tranche de l'espace adressable.

Attention. Une ligne de cache n'est pas une information actuellement stockée dans le cache.

Tous les octets d'une même ligne de cache sont voisins entre eux.

Définition (Voisinage)**.** Soit un octet rangé à l'adresse x.
Le voisinage de x est l'ensemble des octets qui appartiennent à la même ligne que x.

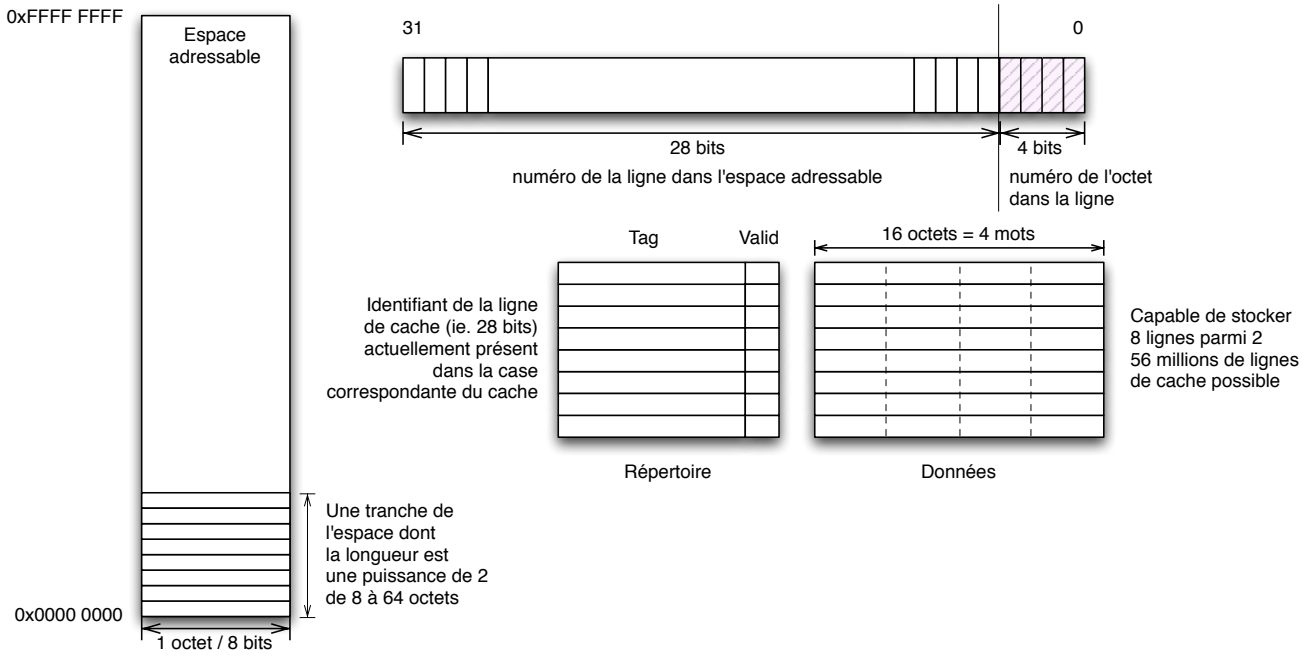


FIGURE 3.2 – Caches à correspondance directe

En cas de MISS, sur une requête à l'adresse x, le contrôleur de cache va ramener dans le cache une ligne complète contenant l'adresse x.
⇒ Utilisation de la localité **spatiale**.

Pour exploiter la localité **temporelle**, le contrôleur de cache conserve dans le cache le plus longtemps possible les informations récupérées dans la mémoire.

Stratégie 1 Cache associatif (*associative*)

⇒ N'importe quelle ligne peut être rangée dans n'importe quelle case
Les huit cases sont équivalentes entre elles. Le processeur cherche à savoir si la ligne est dans le cache → regarder dans chaque case si elle est présente (séquentiellement) : il y a `nb_cases` comparaisons.

Stratégie 2 Cache à correspondance directe (*direct mapping*)

⇒ Une ligne de cache ne peut être rangée que dans une seule case.
Les cases ne sont plus équivalentes entre elles. Toutes les lignes de cache d'une même famille sont en compétition pour la même case.
⇒ Retrouver une ligne de cache à partir de son adresse est beaucoup plus rapide car on accède directement à la seule case possible.

3.4 Comment faire cette partition ?

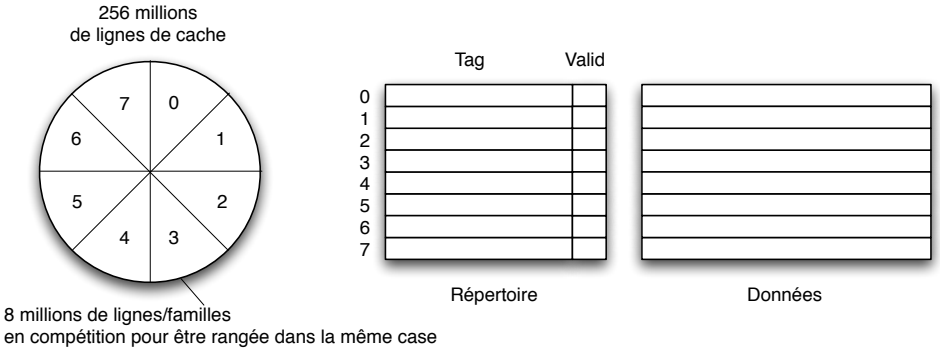


FIGURE 3.3 – Cache à correspondance directe

Si deux lignes voisines sont dans la même famille, elles seraient en compétition pour la même case de cache ⇒ Le principe de localité n'est pas respecté.

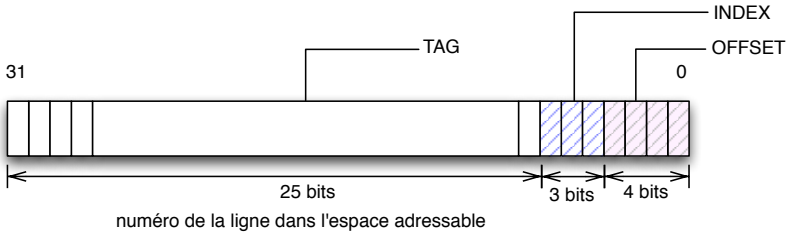


FIGURE 3.4 – Adresse dans le cache

Les champs de l'adresse :

- OFFSET ≡ numéro de mot dans la case du cache : $\log_2(\text{taille ligne})$;
- INDEX ≡ numéro de la famille ≡ numéro de case de cache : $\log_2(\text{taille cache}/\text{taille ligne})$;
- TAG ≡ numéro identifiant une ligne de cache dans la famille : $32 - (\text{OFFSET} + \text{INDEX})$.

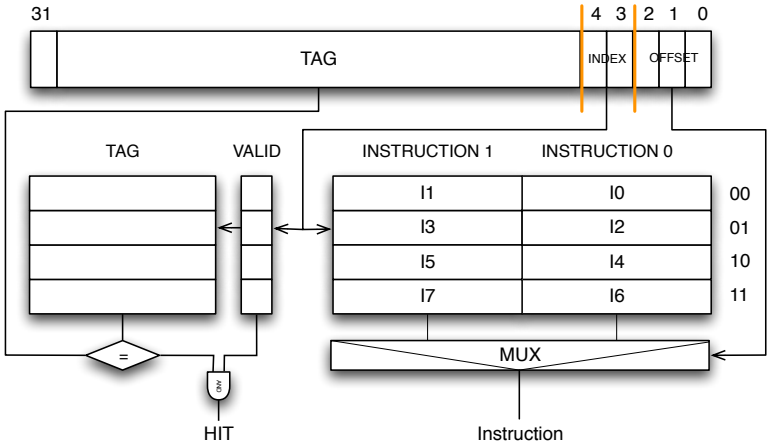


FIGURE 3.5 – Principe résumé d'un cache

3.5 Le problème des écritures

Il n'y a pas de symétries entre les lectures et les écritures.
But du cache : accélérer les lectures (`lw`).

Les caches contiennent des *copies* de la mémoire principale.
⇒ Problème de cohérence entre le cache et la mémoire

Synchronisation producteur / consommateur.

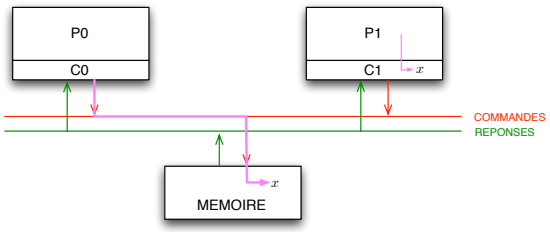


FIGURE 3.6 – Ecriture dans la mémoire

WRITE-THROUGH Une écriture est effectuée immédiatement vers la mémoire (avec mise à jour de la copie locale...si elle existe).

WRITE-BACK Une écriture est effectuée dans le cache local, et la mémoire n'est pas mise à jour plus tard, que en cas d'évincement.

Contradictions

- Quand le processeur fait un `sw`, cette instruction n'est pas bloquante ;
- Quand le contrôleur de cache fait une écriture, il attend un acquittement. Le contrôleur de cache contient un tampon d'écritures postées (file FIFO du cache de données *Write Buffer*)

Remarque. L'utilité de cette file d'attente matérielle est double :

- Le processeur n'est pas bloqué ;
- On peut espérer fabriquer des rafales d'écriture, et donc optimiser l'utilisation du bus.

3.6 Influence des caches sur les performances

Définition (CPI – *Cycle Per Instruction*)**.** Le CPI (*Clock cycles Per Instruction* ou *Clocks per Instruction*)est utilisé pour décrire la performance d'un processeur. Il correspond au nombre de cycles moyen pendant lesquels une instruction est exécutée.

L'objectif est de minimiser ce taux en le rapprochant de 1. Dans le cas où le CPI est égal à 1 pour un unique processeur, il y a un cycle mémoire parfait (ie. taux HIT = 100% et taux MISS = 0%).

Dans une architecture RISC pipeline, chaque instruction est découpée en 5 cycles :

1. *Instruction Fetch cycle* (IF) : accès au cache pour récupérer l'instruction ;
2. *Instruction decode / Register fetch cycle* (ID) : décodage de l'instruction ;
3. *Execution / Effective address cycle* (EX) : exécution de l'instruction ;
4. *Memory access* (MEM) : cherche en mémoire le contenu se trouvant à l'adresse donnée ;
5. *Write-back cycle* (WB) : transmission du résultat en mémoire dans un registre.

Chaque étape requière un cycle d'horloge et une instruction va suivre séquentiellement chaque étape.

Dans une architecture sans pipeline, une nouvelle instruction est récupérée (IF) dans la première étape seulement une fois que l'instruction précédente ait fini l'étape 5 (WB).

Dans une architecture avec pipeline, le CPI peut être amélioré en utilisant l'*instruction level parallelism* (ILP). Ainsi, une nouvelle instruction est récupérée (IF) à chaque cycle. Dans le cas où il y aurait 5 instructions dans 5 étapes de pipeline différentes (une instruction par étape), une instruction différente complèterait l'étape 5 (WB) pour chaque cycle d'horloge. De ce fait, le CPI est égal à 1 pour ce processeur.

Définition (IPC – *Instruction Per Cycle*)**.** Le CPI est l'inverse de l'IPC (*Instruction Per Cycle*) égal au nombre moyen d'instructions exécutées par cycle d'horloge.

Définition (IPS – *Instruction Per Second*)**.** Le nombre d'instructions par seconde IPS (*Instruction Per Second*) est égal à : $CPI \times clock\_speed$ où $clock\_speed$ est la fréquence du processeur considéré calculée en Hertz (Hz) ou en cycles par seconde.

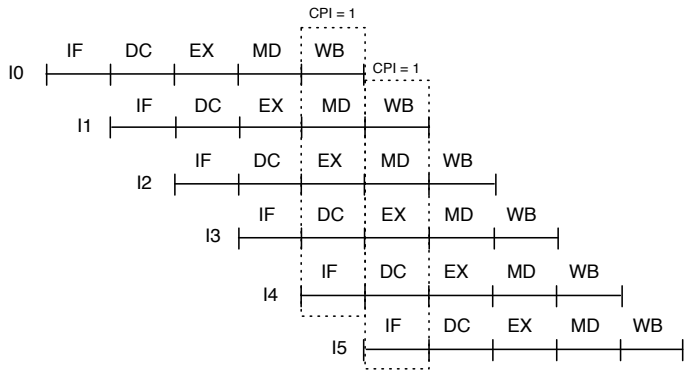


Figure 3.7 – Minimisation du CPI

Chapitre 4

Retour sur EXCEPTIONS / INTERRUPTIONS / TRAPPES

Plusieurs programmes peuvent s'exécuter simultanément sur une même machine. Le système d'exploitation gère alors les accès matériel.

Le système d'exploitation (GIET) est un logiciel qui est activé dans trois cas. Son **point d'entrée** se situe à l'adresse 0x8000 0180.

4.1 Appel système / Trappe

Exécution (par un programme utilisateur) de l'instruction `syscall` :
($2← code de l'appel / $4 à $7 ← arguments).

Ce sont des appels effectués par un programme utilisateur vers le système d'exploitation pour un service (eg. écriture sur un terminal , écriture sur un disque, ...).

Le gestionnaire d'appels systèmes du GIET doit effectuer :

- EPC ← PC
- PC ← 0x80000180
- SR ← mode kernel, IT masquées
- CR ← `system call handler address` (saut à cette adresse)
- Numéro de service demandé (dans le registre $2)
- Valeur des arguments (dans les registres $4 à $7)
- Appel de la fonction système requise
- EPC ← adresse de retour (instruction `eret`)

4.2 Exception

Exécution (par un programme utilisateur) d'une instruction illégale :

- ADEL – lecture illégale
- ADES – écriture illégale
- DBE – Bus erreur données
- IBE – Bus erreur instruction
- OVF – overflow arithmétique
- RI – codop indéfini !
- CPU – tentative d'exécution `mfc0` ou `mtc0`

Le gestionnaire d'exceptions doit déterminer le type de l'exception et afficher un message permettant au programmeur de localiser et de corriger l'erreur.

Remarque.

- Les codes du type d'exception sont écrits par le matériel dans le registre CR
- L'adresse de l'instruction fautive est écrite dans EPC
- BAR ← adresse fautive
- SR ← Mode Kernel (UM = 0) – IT (interruption) masquées (IE = 0)
- PC ← 0x80000180 : point d'entrée dans le système.

4.3 Interruptions / communication avec les périphériques

Deux entités communiquent :
- Le “périphérique” – l'utilisateur ;
- Le système d'exploitation (et non le programme qui s'exécute sur le processeur).

Les interruptions provenant de périphériques sont imprévisibles du point de vue du programme en cours d'exécution et peuvent donc survenir à tout instant.

Si elle n'est pas masquée, l'activation d'interruption déclenche l'exécution d'une routine spécifique (ISR) permettant au périphérique de “voler” des cycles au processeur. Le programme interrompu reprend ensuite son exécution normale.

Le gestionnaire d'interruptions doit déterminer la source de l'interruption pour activer la bonne ISR en interrogeant le composant ICU.

4.3.1 Communication avec les périphériques

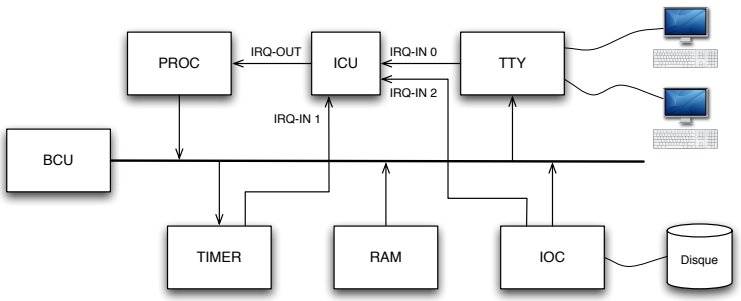


FIGURE 4.1 – Gestion des interruptions (IRQ)

Le système d'exploitation gère donc les mécanismes de contrôle. Il existe deux types de communications entre le système d'exploitation et les périphériques :
- **Accès en configuration** (opération d'écriture sur des registres du périphérique) ou **accès en consultation** (opération de lecture) ;
- Initiatives venant du périphérique pour signaler des évènements à l'OS (**interruptions**).

Communication entre deux processus : suivant le modèle producteur / consommateur.

Idée. Le périphérique et l'OS communiquent à travers un tampon mémoire partagé ou plusieurs. Dans le cas du TTY, il est nécessaire d'avoir deux tampons :
- Un tampon DATA (`TTY_WRITE`) qui contient le caractère tapé au clavier – 1 octet en mémoire correspondant à la taille mémoire d'un caractère ASCII ;
- Un tampon FULL (`TTY_STATUS`) contenant un booléen (1 bit) qui indique l'état du tampon DATA :
  - Producteur : FULL ← 1 si tampon DATA rempli ;
  - Consommateur : FULL ← 0 si tampon DATA lu et vidé.

Définition (IRQ – *Interrupt Request*)**.** Une interruption (IRQ) provient d'un contrôleur / périphérique et arrive dans l'ICU (*Interrupt Controllor Unit*). L'IRQ demande alors au processeur d'exécuter l'ISR (*Interrupt Service Routine*) correspondante.

Définition (ISR – *Interrupt Service Routine*)**.** Un périphérique (TTY, Timer, ...) souhaite effectuer une écriture en mémoire, active une interruption qui va forcer le processeur à exécuter quelques dizaines / centaines d'instructions.

Ce code s'appelle ISR (*Interrupt Service Routine*).

Activation : il y a une ISR différente pour chaque périphérique.

Instructions de l'ISR :

- Ecrit dans le registre de communication et de synchronisation ;
- Ecrit dans le registre du périphérique permettant d'acquitter la requête d'interruption.

Définition (ICU – *Interrupt Controller Unit*)**.** Ce contrôleur comporte (dans le cadre du cours) 32 lignes d'interruption en concurrence pour voler des cycles au processeur. Le composant ICU est un concentrateur d'interruptions permettant de **multiplexer** 32 interruptions IRQ-IN vers **une** interruption IRQ-OUT.

4.3.2 Comment le gestionnaire d'interruptions peut-il déterminer quel ISR exécuter ?

Organisation générale du GIET Lors d'un appel système, d'une interruption, d'une exception, il y a un branchement à l'adresse 0x80000180 – point d'entrée du GIET (Gestionnaire d'Interruptions, Exceptions et Trappes).

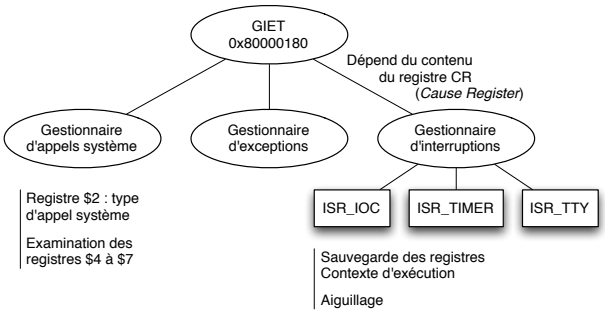


FIGURE 4.2 – Organisation du GIET

Les interruptions sont vectorisées.

Définition (Vecteur d'interruptions)**.** Un tableau en mémoire dont chaque entrée est une adresse d'une ISR. C'est donc un tableau d'adresse où chaque adresse correspond à un point d'entrée d'une fonction du GIET.

Remarque. Un tableau d'adresse ≡ table de sauts ≡ vecteur d'interruptions

Le gestionnaire d'interruptions détermine l'index (index compris entre 0 et 31) vers lequel il va se brancher. Pour obtenir l'index en question, le gestionnaire d'interruptions interroge le composant ICU en effectuant une lecture dans son registre INDEX.

Conflit : si plusieurs IRQ-IN sont actives en même temps, l'ICU renvoie l'index le plus petit.

Remarque. ICU (*Interrupt Controller Unit*) ≡ PIC (*Programmable Interrupt Controller*)

Registres de l'ICU :

- `ICU_MASK` : chaque bit représente la gestion d'une interruption des 32 interruptions possibles :
  - `ICU_MASK`$[i] = 0$: pas d'interruption sur l'entrée i (interruption masquée) ;
  - `ICU_MASK`$[i] = 1$: interruption autorisée sur l'entrée i.

  Les interruptions ont par défaut une priorité croissante (ie. l'interruption de l'entrée 1 sera traitée avant celle de l'entrée 2).
- `ICU_INDEX` : numéro de l'interruption la plus prioritaire active (registre codé sur 5 bit – $2^5 = 32$).

Résumé sur les services rendus par l'ICU :

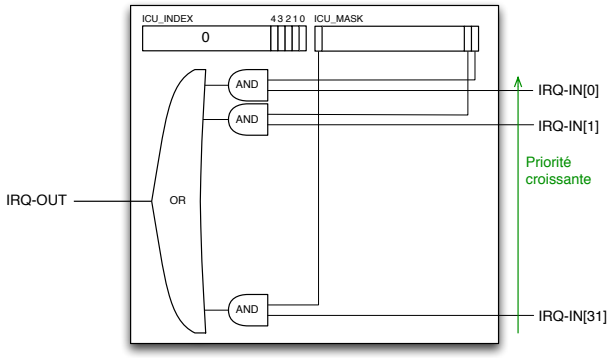


FIGURE 4.3 – ICU (*Interrupt Controller Unit*)

1. OU logique entre toutes les entrées IRQ-IN(i) ;
2. Encodeur de priorité ;
3. Masque sélectif M[i] des 32 IRQ-IN[i] (ce masque tient sur un mot de 32 bits).

Idée. On augmente la complexité du matériel en ayant plusieurs IRQ-OUT.

Si on réplique 8 fois le matériel de l'ICU, on aura les 32 mêmes entrées IRQ-IN, et 8 sorties IRQ-OUT.

On va avoir 8 IR-OUT[j] et on peut donc "router" chaque IRQ-IN[i] vers n'importe quelle sortie IRQ-OUT[j].

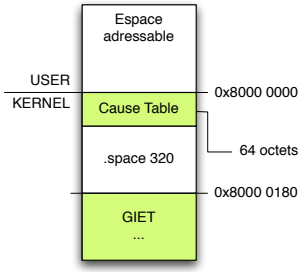
D'où PIC : ***Programmable*** *Interrupt Controller*

Remarque. L'OS doit configurer les registres de masque de l'ICU et doit initialiser le vecteur d'interruptions (généralement dans le code de BOOT).

Le code des ISR ne fait pas partie de l'OS, mais il s'exécute en général en mode superviseur.

4.4 A propos de l'appel au GIET

La cause de l'activation du GIET est située dans le registre CR (*Cause Register*). Le GIET parcourt alors la table *Cause table*, une table de sauts contenant les adresses des fonctions à appeler suivant la valeur du registre CR. Cette table est située à l'adresse 0x8000 0000 et contient 16 entrées (taille : $16 \times 4 = 64$ octets).



Lorsque le processeur se branche à la première instruction du GIET, les interruptions sont automatiquement masquées par le matériel :

- Si la cause du branchement est une exception, il y a alors un traitement prioritaire de l'exception et on ne souhaite pas être interrompu ;
- Si la cause du branchement est une interruption, il faut pouvoir exécuter le code du GIET sans reboucler sur cette adresse. En effet, à chaque cycle, le processeur examine le contenu du registre `IRQ_INDEX` ;
- Si la cause du branchement est un appel système, alors le GIET doit décider lui-même d'être interrompu ou non.

4.4.1 Le registre d'état SR (*Status Register*)

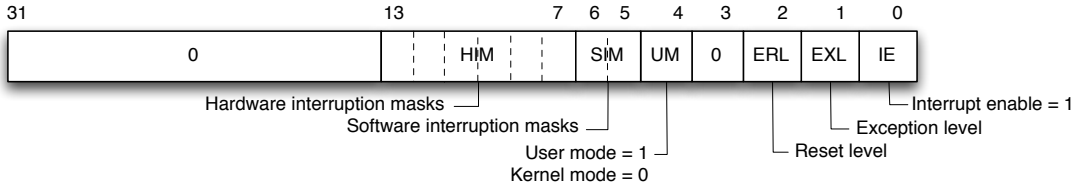


FIGURE 4.4 – SR (*Status Register*)

- Le processeur a le droit d'accéder aux ressources protégées (registres du CP0, et adresses mémoires supérieures à 0x7FFFFFFF) si et seulement si le bit UM vaut 0, ou si l'un des deux bits ERL et EXL vaut 1 ;
- Les interruptions sont autorisées si et seulement si le bit IE vaut 1, et si les deux bits ERL et EXL valent 00, et si le bit correspondant de IM vaut 1 ;
- Les trois types d'événements qui déclenchent le branchement au GIET (interruptions, exceptions et appels système) forcent le bit EXL à 1, ce qui masque les interruptions et autorise l'accès aux ressources protégées ;
- L'activation du signal RESET qui force le branchement au Boot-Loader force le bit ERL à 1, ce qui masque les interruptions et autorise l'accès aux ressources protégées ;
- L'instruction `eret` force le bit EXL à 0.

Lors de l'activation du RESET, SR contient donc la valeur 0x0004. Pour exécuter un programme utilisateur en mode protégé, avec interruptions activées, il doit contenir la valeur 0xFF11. Le code de boot doit écrire la valeur 0xFF13 dans SR et l'adresse du programme utilisateur dans EPC avant d'appeler l'instruction `eret`.

4.4.2 Le registre de cause CR (*Cause Register*)

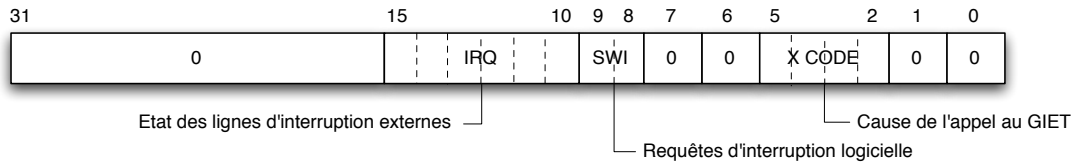


FIGURE 4.5 – CR (*Cause Register*)

Le registre CR contient trois champs. Les 4 bits du champs XCODE(3 :0) définissent la cause de l'appel au GIET. Les 6 bits du champs IRQ(5 :0) représentent l'état des lignes d'interruption externes au moment de l'appel au GIET. Les 2 bits SWI(1 :0) représentent les requêtes d'interruption logicielle.

Chapitre 5

Périphériques

Le pilote de périphérique contient deux parties :

- Un code de “configuration” : permet au logiciel (OS) de lire / écrire dans les registres internes au périphérique. A l'initiative de l'OS. Ce code se trouve dans le fichier `drivers.c` ;
- Le(s) code(s) de(s) ISR (*Interrupt Service Routine*) permettant au périphérique de communiquer avec l'OS. A l'initiative du périphérique. Ce code se trouve dans le fichier `isr.s`.

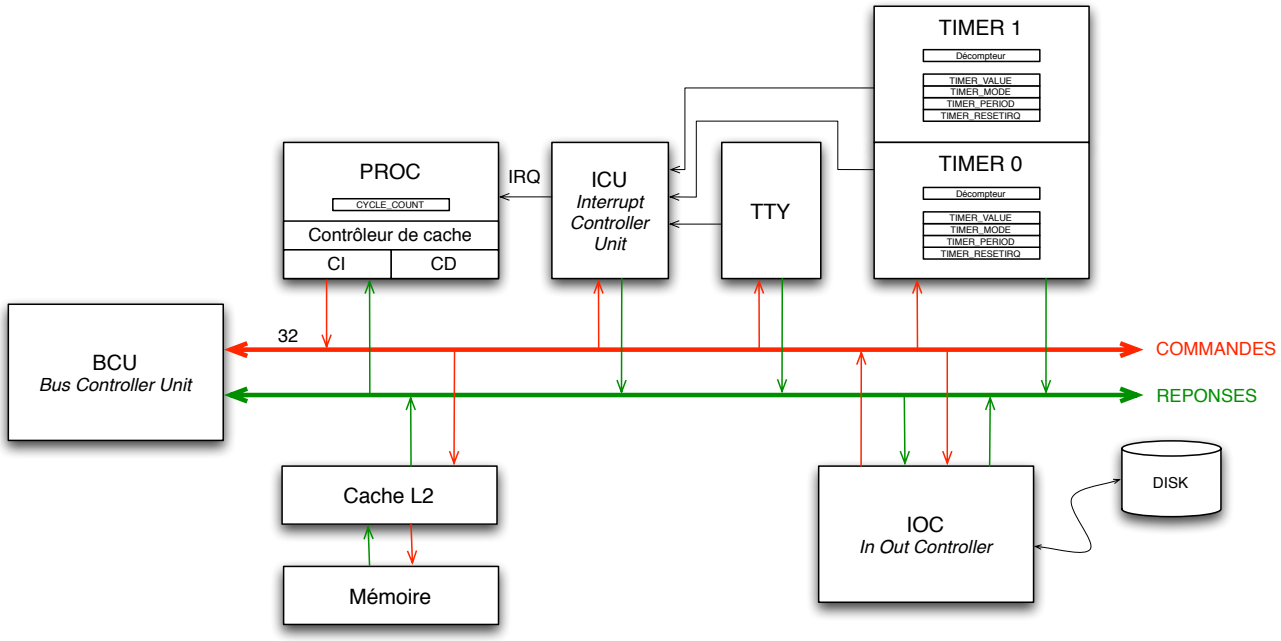


Figure 5.1 – Quelques périphériques

Les périphériques sont de deux sortes :

- Périphérique “simple” : pouvant recevoir des commandes de lecture / écritures, mais ne peuvent pas eux-mêmes lire ou écrire dans l'espace adressable (eg. TIMER, ICU, contrôleur TTY...) ;
- Périphériques DMA (*Direct Memory Access*) : a le droit de lire / écrire d'eux-même dans l'espace adressable.

5.1 Un périphérique simple : le TIMER

Il existe plusieurs types de TIMER :

- Le compteur de cycles (privé) dans chaque processeur – registre protégé `CYCLE_COUNT` ;
- Le périphérique externe utilisable par le logiciel (OS) pour générer des interruptions périodiques.

Les registres adressables du TIMER :

- Registre `TIMER_VALUE` (Base) : compteur de cycles ;
- Registre `TIMER_MODE` (Base + 4) : configuration – contient un CODOP écrit par le logiciel sur 2 bits :
  - — incrémenter (1 bit) ;
  - — masquer des interruptions (1 bit).
- Registre `TIMER_PERIOD` (Base + 8) : Valeur de la période où l'OS lève des interruptions (généralement égale à 100 000 ou 1 000 000) ;
- Registre `RESETIRQ` (Base + 12) : acquittement – fin du traitement de l'interruption.

Le registre caché `DECOMPTEUR` du TIMER n'est pas adressable. C'est un décompteur périodique se basant sur la période définie par l'OS dans le registre `TIMER_PERIOD` interne au TIMER.

Le périphérique TIMER peut contenir plusieurs timers : chaque timer individuel contient les 5 registres physiques et correspond à 4 registres adressables dans l'espace adressable.

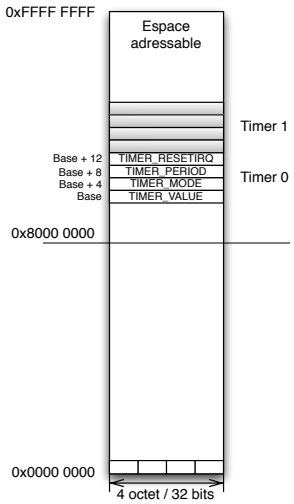


FIGURE 5.2 – Les registres adressables des timers

5.2 Un périphérique "BLOCS" : IOC (*In Out Controller*)

On parle de périphérique "orienté blocs" lorsque l'on doit désigner des objets dans un autre espace de stockage (généralement beaucoup plus grand...et beaucoup plus **lent**).

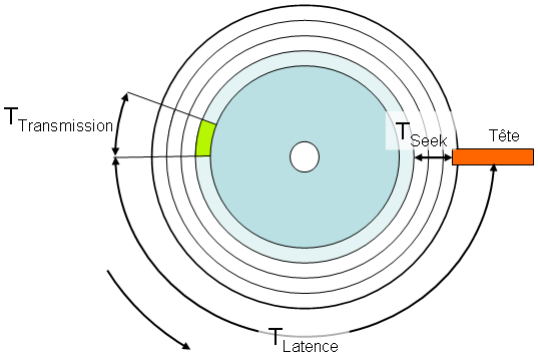


FIGURE 5.3 – Disque dur

Hiérarchie de stockage

Espace de stockage	Taille	Temps d'accès
32 registres généraux	$32 \times 4 = 128$ octets	< 1 cycle
Caches L1	4 Ko	1 cycle
Cache L2	4 Mo	20 cycles
Mémoire externe (RAM, ROM)	2 Go	400 cycles
Disque	200 Go	100 000 à 1 000 000 cycles

Définition (Bloc)**.** On appelle "bloc" un ensemble d'octets (varie entre 512 octets et 4096 octets) qui vont être rangés à des adresses consécutives en mémoire.

Définition (Secteur)**.** L'emplacement sur le disque permettant de stocker un bloc s'appelle un secteur. La taille typique d'un secteur est de 512 octet.

Pour désigner un bloc dans l'"espace disque", le logiciel utilise un "numéro de bloc".

Pour désigner un bloc à transférer, il faut deux instructions :
- Nom de fichier – chemin / nom ;
- Un numéro de bloc dans un fichier.

Transférer un (ou plusieurs) blocs depuis le disque vers la mémoire :
- Nom de fichier : appel système `ioc_read()` renvoie un descripteur de fichier (`FILE *`) ;
- Désigner le premier bloc et le nombre de blocs à transférer ;
- Désigner l'adresse de base du tampon mémoire cible.

Deux appels systèmes du GIET :
```
int ioc_read(size_t lba, void * buffer, size_t count);
int ioc_write(size_t lba, void * buffer, size_t count);
```

Signification des paramètres :
- `lba` : numéro du premier bloc sur le disque (*Logical Block Address*) ;
- `buffer` : adresse de base du tampon en mémoire ;
- `count` : nombre de blocs.

On utilise des appels systèmes pour accéder au disque :
- Le système doit vérifier que le programme utilisateur a le droit de lire / écrire des données sur le disque et donc de pouvoir faire l'accès en question ;
- Le système doit vérifier que l'adresse de destination appartient bien à l'espace de l'utilisateur (ie. inférieure à 0x8000 0000) ;
- Le disque dur est un système mécanique, donc est très lent. Le système va optimiser la position de la tête de lecture sur le disque en utilisant une file FIFO des requêtes de demande d'accès au disque ;
- Le système va "descheduler" (désordonnancer, mettre en attente) le programme qui demande un accès au disque. Au lieu de l'attente active, le système va mettre le processus en attente ;
- Si l'on veut faire un second transfert à la même adresse, il faut alors invalider les lignes de cache afin de pouvoir charger les données souhaitées avec la fonction `_dcache_buf_invalidate`.

Remarque. Les fonctions `ioc_read` et `ioc_write` ne sont pas bloquantes : elles rendent généralement la main au programme utilisateur bien avant que l'IOC ait terminé le transfert ⇒ parallélisme. Entretemps, l'IOC aura effectué le transfert demandé.

La fonction `ioc_read()` du segment CODE fait appel à la fonction `_ioc_read()` du segment KCODE. La fonction `_ioc_read()` vérifie que le périphérique n'est pas déjà utilisée en testant la valeur du registre `IOC_BUSY` dans une boucle de *polling* (scrutation) `ioc_completed()`.

Lorsque l'on sort de cette boucle (ie. `IOC_BUSY` = 0), on peut utiliser le périphérique (`IOC_BUSY` = 1). L'IOC active alors une interruption lorsque le transfert est terminé.

Remarque. On considèrera qu'il n'y a pas de système de fichiers : le disque ne contient q'un seul fichier qui est le fichier de la station de travail LINUX.

Définition (Driver de périphérique)**.** Logiciel permettant de contrôler un périphérique.

Le contrôleur IOC contient au moins 5 registres adressables :
- Adresse mémoire `IOC_BUFFER` ;
- Nombre de blocs à transférer `IOC_COUNT` ;
- Adresse logique (numéro) du premier bloc `IOC_LBA` ;

– Type du transfert à effectuer (read / write) `IOC_TYPE` ;
– Acquittement de l'interruption `IOC_BUSY` (non caché).

L'exécution de l'ISR associée au contrôleur IOC doit informer l'OS que le transfert est terminé. L'ISR écrit alors dans une case mémoire `IOC_BUSY` appartenant à l'OS dans le segment KUNC (données non cachables du kernel).

Pendant un transfert, il existe deux processus s'exécutant en parallèle ⇒ Besoin de synchronisation :
– Le processus IOC est propriétaire de la variable `IOC_BUSY` lorsqu'elle vaut 1. L'OS ne peut alors pas modifier cette variable et doit attendre s'il doit lancer un transfert.
– L'OS est propriétaire de cette variable lorsqu'elle vaut 0, et peut alors lancer un transfert.

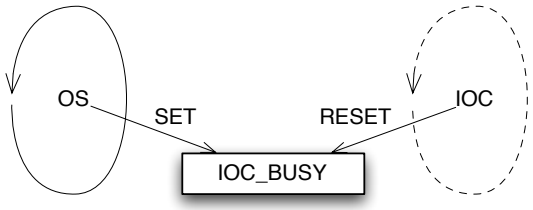


FIGURE 5.4 – Synchronisation entre IOC et OS

Remarque. Ce mécanisme est un mécanisme de synchronisation par variable SET / RESET. Ce mécanisme est identique à celui utilisé pour le TTY avec le registre `TTY_STATUS`.

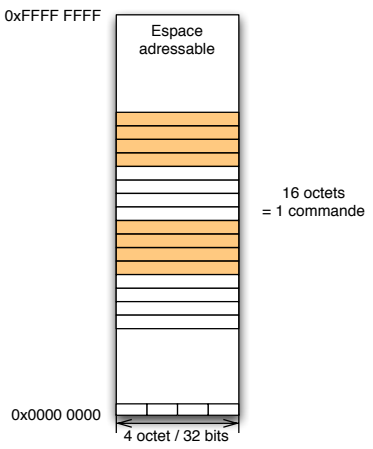
Le contrôleur de disque IOC présenté ici ne peut exécuter qu'un seul transfert entre la mémoire et le disque à la fois.

Pour éviter de rendre bloquant (ie. l'appel système entre dans une boucle d'attente active `ioc_completed` où l'attente peut être longue) les appels systèmes `ioc_read` et `ioc_write` , l'OS peut stocker les paramètres de la commande dans une file d'attente en mémoire.

Pour ce faire, il réserve des paquets de 16 octets dans l'espace adressable où chaque paquet représente une commande en attente.

Objectif. Eviter au processeur de gâcher des cycles en faisant de l'attente active.

Dans les contrôleurs IOC les plus récents, la file d'attente des commandes de transfert est une structure de données (file de type FIFO) partagée entre l'OS et l'IOC (variable de synchronisation nécessaire).



5.3 Architecture matérielle finale

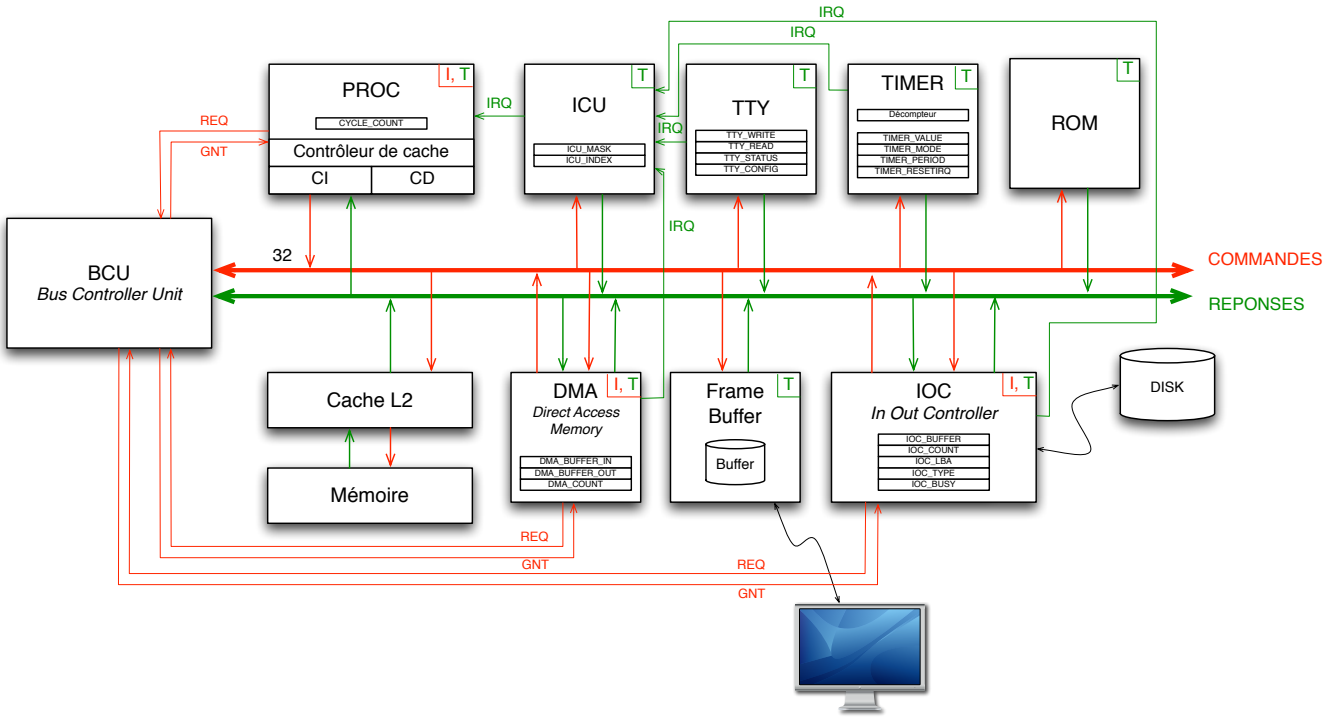


FIGURE 5.5 – Architecture matérielle ALMO5

5.3.1 DMA (*Direct Memory Access*)

Le composant « DMA » est utilisé pour déplacer des données, d'un tampon mémoire source vers un tampon mémoire destination.

C'est un maître sur le bus puisqu'il est capable de lire et d'écrire à n'importe quelle adresse dans l'espace adressable du système.

C'est également une cible, puisqu'il contient un (petit) nombre de registres adressables dans lesquels le système d'exploitation peut écrire, pour définir les caractéristiques du transfert à effectuer (adresse du tampon source, adresse du tampon destination, nombre d'octets à transférer). Une fois configuré, ce composant travaille en parallèle avec le processeur, et signale la fin du transfert en activant une ligne d'interruption spécifique.

Le composant DMA contient alors 3 registres adressables :

- Adresse tampon source `DMA_BUFFER_IN` ;
- Adresse tampon destination `DMA_BUFFER_OUT` ;
- Nombre d'octets à transférer `DMA_COUNT`.

5.3.2 Block I/O ou IOC (*In Out Controller*)

Le composant « Block I/O » est un contrôleur de disque. Il est utilisé pour transférer des données entre un tampon mémoire et un fichier situé sur un composant de stockage externe tel qu'un disque magnétique. Tous les transferts se font par blocs de 512 octets (appelés « secteurs » dans le cas des disques).

Le contrôleur de disque est aussi un maître sur le bus puisqu'il capable de lire et d'écrire en mémoire à n'importe quelle adresse dans l'espace adressable.

C'est également une cible, puisqu'il contient un (petit) nombre de registres adressables dans lesquels le système d'exploitation peut écrire, pour définir le transfert à effectuer (adresse du tampon en mémoire, désignation du premier bloc dans le fichier, nombre de blocs à transférer). Une fois configuré, ce composant travaille en parallèle avec le processeur, et signale la fin du transfert en activant sa propre ligne d'interruption.

5.3.3 Frame Buffer

Le composant « Frame Buffer » est un contrôleur vidéo permettant d'afficher des images sur un écran graphique. Ce composant contient une mémoire vidéo permettant de stocker une image de dimension prédéfinie. On s'intéressera à des images en « niveaux de gris ». Chaque pixel est codé sur un octet. Il y a donc 256 niveaux de gris, et la valeur 0 correspond à un pixel noir. Pour avoir des temps d'exécution raisonnables, on se limite à des images de 128 lignes de 128 pixels.

Dans un PC, la capacité de la mémoire vidéo atteint couramment 4 Moctets (1024 lignes de 1024 bits, chaque pixel étant codé sur 4 octets pour l'affichage couleur. Cette mémoire vidéo est parcourue à une fréquence fixe (typiquement 25 fois par seconde) de façon à générer le signal vidéo qui est envoyé vers le terminal graphique. Cette mémoire vidéo (ou « frame buffer ») est généralement implantée dans la partie de l'espace adressable réservée au système d'exploitation.

Le « Frame Buffer » est un composant cible sur le bus. Il ne possède pas de registres adressables.

```
/* Fonctions bloquantes */
int fb_sync_write(size_t offset, void* buffer, size_t length);
int fb_sync_read(size_t offset, void* buffer, size_t length);
```

Ces deux fonctions effectuent le transfert de façon purement logicielle : chaque octet du tampon source est chargé par le processeur dans un registre interne, puis écrit par le processeur dans le tampon destination.

Ces fonctions sont bloquantes et ne rendent la main que lorsque le transfert est terminé ⇒ transfert est synchrone. Cette technique est simple, mais elle est lente.

```
/* Fonctions non bloqantes */
int fb_write(size_t offset, void* buffer, size_t length);
int fb_read(size_t offset, void* buffer, size_t length);
int fb_completed();
```

Les deux fonctions `fb_write()` et `fb_read()` utilisent le composant matériel DMA pour accélérer le transfert. Le mécanisme est très similaire à celui utilisé par le contrôleur I/O. Ces deux fonctions sont non bloquantes.

Elles configurent le contrôleur DMA pour que celui-ci effectue le transfert et rendent immédiatement la main au programme utilisateur. Plus tard, le contrôleur DMA active une interruption pour signaler la fin du transfert.

Comme dans le cas du contrôleur I/O, la fonction `fb_completed()` est une fonction bloquante qui ne rend la main que lorsque le transfert est terminé.

On utilise des appels système car il faut invalider les lignes de cache avec la fonction `_dcache_buf_invalidate` – problème si l'on veut afficher une nouvelle image. De plus, il faut vérifier si le programme est en droit de faire l'affichage (ie. adresse $<$ 0x8000 0000).

Chapitre 6

Fonctionnement multitâche

6.1 Introduction

Remarque. Multitâche ≠ multiprocesseur :
- Multitâche : utilisation de la machine comme si plusieurs programmes s'exécutaient en même temps ;
- Multiprocesseur : (au niveau du matériel) une machine peut contenir plusieurs processeurs fonctionnant en parallèle.

Définition (Parallélisme)**.** On parle de parallélisme émulé lorsque l'on exécute plusieurs programmes utilisateurs simultanément sur un seul processeur.

On découpe le temps en tranches de durée fixe (quantum). Arbitrairement : 1 tanche = 10 ms = 10 millions de cycles (1 cycle = 10^{-9} s).

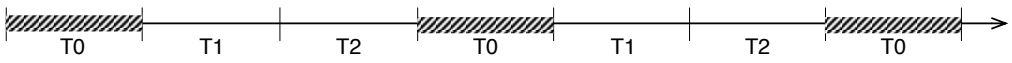


Figure 6.1 – Tranches de temps

Dans l'exemple ci-dessus, chaque tâche aura exécuté 33 tranches. Chaque programme va s'exécuter 3 fois moins vite que s'il avait été seul.

C'est du pseudo-parallélisme par multiplexage temporel.

Les évènements périodiques de changement de contexte (le processeur travaille pour la tâche T0 et passe à la tâche T1) sont générés par un TIMER TICK. Le terme TICK désigne au choix :
- L'interruption qui déclenche un changement de contexte ;
- La périodicité de l'évènement déclencheur d'un changement de contexte.

Définition (Contexte)**.** Ensemble des informations qui doivent être sauvegardées pour permettre à une tâche sortante de recommencer à s'exécuter lorsque ce sera son tour de rentrer (ie. devenir une tâche entrante).

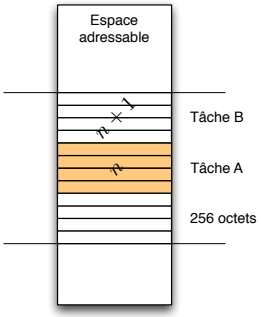
Sauvegarde des registres :
- Tous les registres généraux R(i) sauf R(0), R(26) et R(27) (registres réservés au système) ;
- EPC (PC sauvegardé dans EPC), `STATUS REGISTER` (SR), `CAUSE REGISTER` (CR), HI, LO.

Remarque. Ce système de sauvegarde des registres est similaire à celui de la sauvegarde du contexte d'exécution d'une fonction.

Chaque tâche ayant sa propre pile, il suffit de sauvegarder le pointeur de pile R(29).

L'OS a besoin de stocker en mémoire **les** contextes d'exécution des N tâches en cours d'exécution (1 contexte = 32-3+5 = 34 registres de 32 bits). L'OS possède donc un "tableau de contextes" où chaque "case" de ce tableau a une capacité de 256 octets.

L'OS a besoin d'une case mémoire permettant de stocker le numéro du programme utilisateur en cours d'exécution.



6.2 Mécanisme de changement de contexte

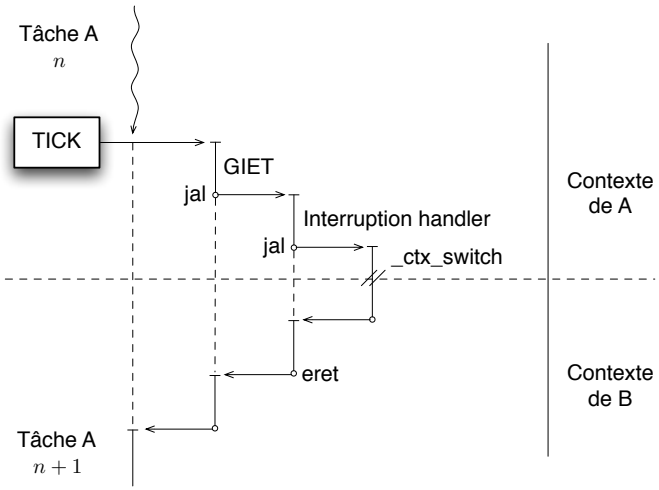


FIGURE 6.2 – Changement de contexte

Au moment du TICK (ie. le TIMER génère une IRQ) :

- Point d'entrée du GIET (0x8000 0180) ;
- Le registre CR contient le code d'une interruption : branchement à la fonction `_int_handler` ;
- Le registre `ICU_INDEX` contient le numéro de l'entrée d'interruption `IRQ_IN` associée au TIMER ;
- Exécution de l'ISR associée au TIMER `_isr_switch` ;
- Branchement à la fonction `_ctx_switch` ;

Remarque. Les tâches sont élues suivant l'algorithme *Round Robin* qui assigne des portions de temps égales pour chaque tâche et exécute toutes les tâches sans priorité, dans un ordre cyclique.

Remarque. Dans le GIET, on ne peut pas créer *dynamiquement* des tâches ; la création de tâches est réalisée *statiquement* dans la phase de BOOT.

Phase de BOOT :

- Initialisation des registres (pile, SR) ;
- Initialisation des périphériques ;
- Initialisation de la tâche principale ;
- Initialisation des tableaux de contexte des autres tâches qui démarrent au TICK.

Définition (Virtualisation)**.** Donner l'impression à l'utilisateur qu'il y a plus de ressources que réellement.

Le fonctionnement multitâche consiste à utiliser un seul processeur physique (matériel) comme n processeurs virtuels.

Pour permettre à plusieurs applications de s'exécuter en pseudo-parallélisme, il faut :

- Partager le processeur (ie. virtualiser le processeur) ;
- Partager la mémoire.

La **mémoire virtuelle** consiste à donner à chaque application l'impression qu'elle dispose de la totalité de l'**espace adressable**. Ce mécanisme permet de compiler chaque application comme si elle s'exécutait seule sur la machine.

Définition (Adresse logique vs. adresse physique)**.** Les adresses logiques (virtuelles) sont les adresses calculées par le compilateur (lors de l'édition de liens) que l'on retrouve dans le code binaire (fichier exécutable). Ce sont les adresses qui sont émises par le processeur. Une adresse physique permet de référencer, manipuler de la mémoire physique.

Les adresses logiques sont alors traduites en adresses physiques (réelles) au moment de sortir du processeur : entre le processeur et le cache ⇒ dans le MMU (*Memory Management Unit*).

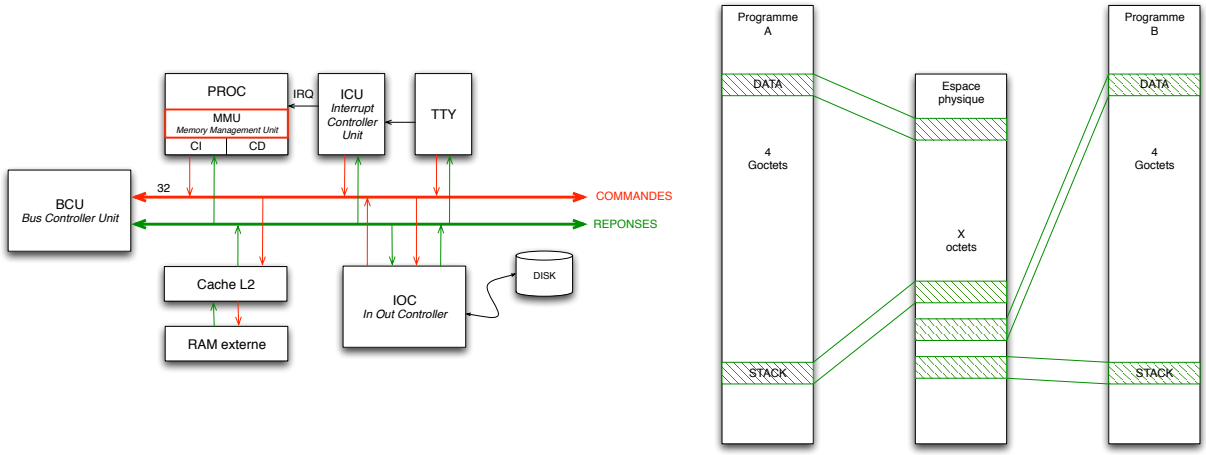


FIGURE 6.3 – MMU (*Memory Management Unit*)

Le composant matériel MMU est chargé de faire la traduction "à la volée" des **adresses logiques** vers les **adresses physiques** (ie. à chaque instruction où le processeur va lire dans le cache, il faut faire une traduction en 1 cycle de l'adresse d'instruction ou de donnée).

Ce composant matériel utilise **des** tables de traduction : une table de traduction différente pour chaque programme.

Les tables de traduction sont stockées en mémoire ; elles sont chargées lors de la phase de BOOT.

Il n'y a pas de collisions : lorsque l'OS reçoit la demande d'un programme pour accéder aux ressources de la machine, il détermine les segments physiques (STACK, DATA, CODE) sans collision.

Il faut alors :

- Un système de **tables** de traduction ou table de pages (en mémoire) ;
- Un composant matériel MMU capable d'utiliser ces tables.

La technique moderne de virtualisation de la mémoire est la **pagination**.

Définition (Page)**.** Une page est un bloc d'adresses consécutives dans l'espace adressable (logique ou physique) aligné et de taille fixe : 4Ko.

Remarque. Un bloc aligné : l'adresse la plus élevée est un multiple de l'adresse la plus petite.

Il y a défaut de page lorsqu'un VPN n'est pas associée à un PPN dans la table de pages de la MMU. Il y a alors une erreur de traduction d'adresse virtuelle. Il y a une levée d'exception et l'OS doit alors

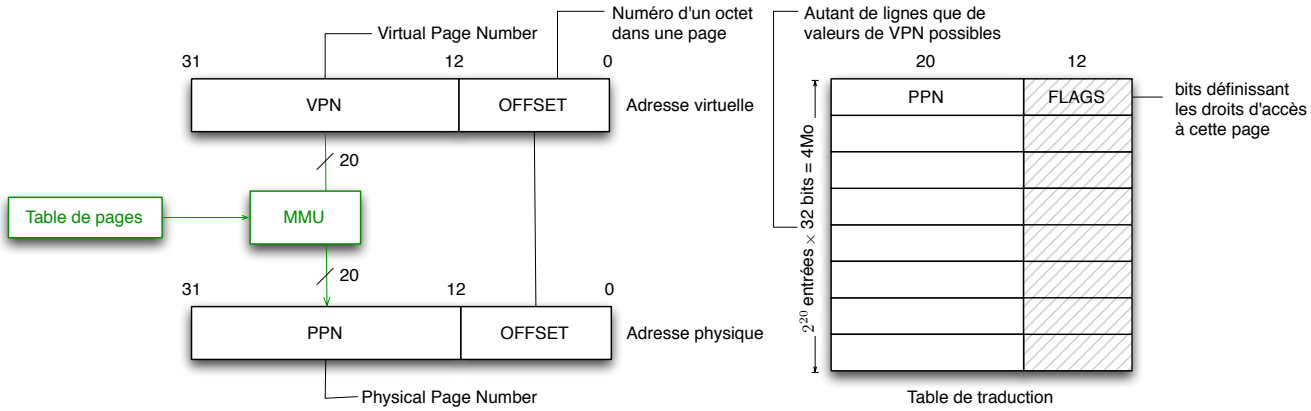


FIGURE 6.4 – Lien entre adresse physique et adresse virtuelle

effectuer un *mapping* pour associer un PPN au VPN recherché en trouvant une page de libre.
Attention. Un défaut de page est différent d'un MISS de cache.

La plupart des programmes utilisateurs n'utilisent d'une toute petite partie de leur espace adressable virtuel (ie. 2 Go). De ce fait, la plupart des "tables de pages" (ou "tables de traduction") sont creuses (ie. contiennent majoritairement des 0).

En général, l'OS utilise des pages à deux niveaux.

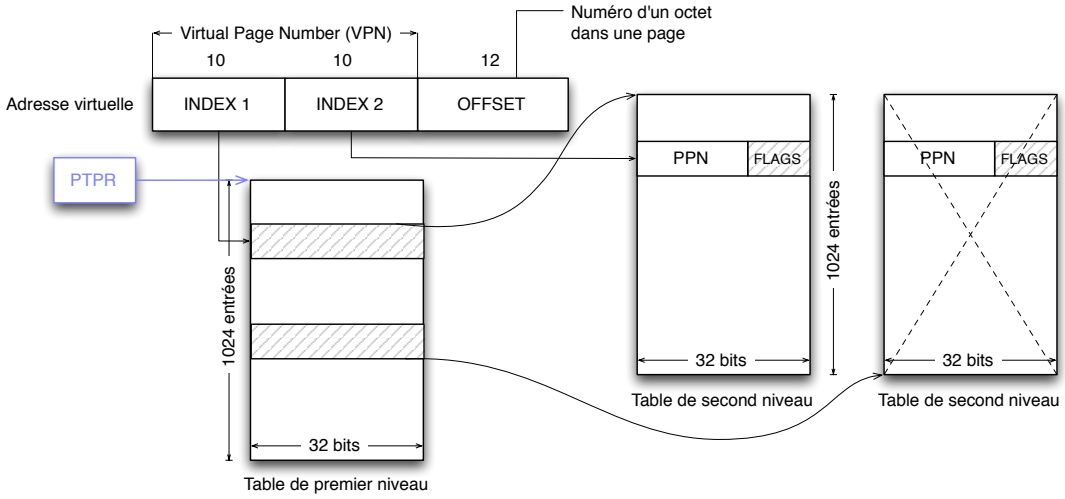


FIGURE 6.5 – Pagination à deux niveaux

L'OS ne crée que les tables de pages de 2^{e}niveau qui sont effectivement utilisées.
La traduction va alors nécessiter **deux** lectures mémoire au lieu d'une.

Remarque. Le registre PTPR (*Patch Table Pointer Register*) contenu dans le co-processeur contient l'adresse de base de la table de traduction de premier niveau.

On gagne de la mémoire et on ralenti le mécanisme de traduction. Une "vraie" traduction (effective) coûte au moins deux lectures (environ 30 cycles).

La solution est de placer dans la MMU un petit cache spécialisé : TLB (*Translation Look-aside Buffer*) qui utilise le principe de localité. En effet, puisque les changements de pages sont rares, on en profite pour conserver dans le TLB de la MMU les résultats des dernières traductions.

Définition (TLB – *Translation Look-aside Buffer*)**.** La TLB permet de conserver localement dans la MMU le résultat de quelques traductions VPN → PPN.

La TLB est un tableau associatif (ie. un extrait peut être rangé dans n'importe quelle case ≠ cache à correspondance directe : une ligne de cache ne peut être contenue que dans une seule et unique case).

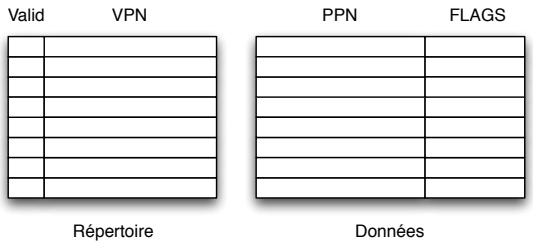


FIGURE 6.6 – TLB (*Translation Look-aside Buffer*)

Lors d'un MISS TLB, il faut récupérer la correspondance VPN ↔ PPN en effectuant 2 lectures des tables de traduction.
La TLB a un taux de MISS d'environ 10^{-3}. En cas de MISS sur la TLB, il faut aller consulter les tables de pages en mémoire.

Remarque. Lors d'un changement de contexte, il faut :
- Mettre à jour le registre PTPR ;
- Purger la TLB.

De même qu'il y a deux caches séparés pour les instructions et données (CI et CD), il y a deux TLB pour paralléliser les traductions.

En effet, s'il n'y a qu'un seul cache, pour une lecture de donnée, on perd deux cycles :
- Lecture de l'instruction ;
- Lecture de la donnée.

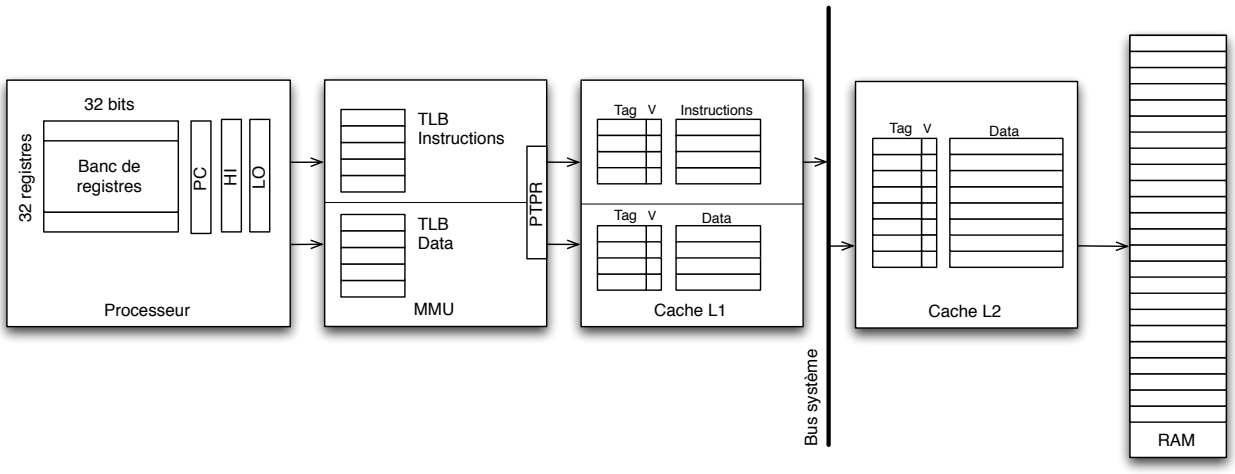


FIGURE 6.7 – Hiérarchie mémoire

Chapitre 7

A propos de l'architecture interne des processeurs

Comment le processeur peut-il exécuter une instruction à chaque cycle ?

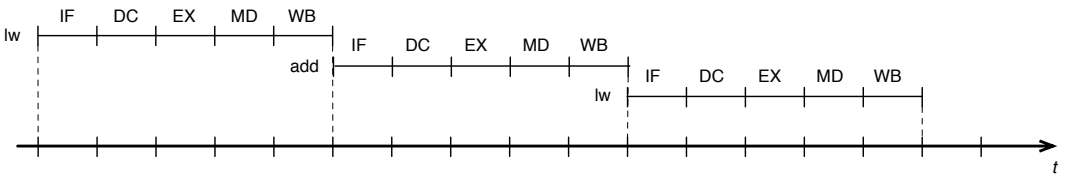
Exemple. Soit l'instruction `lw $d, $s(imd)# $d <- M[$s + IMD]`
Cette instruction utilise le processeur pour :

1. Lire l'instruction en mémoire (IF – *Instruction Fetch*) ;
2. Décoder le CODOP (DC – *DeCode*) ;
3. Calculer l'adresse `$s + IMD` (EX – *EXecute*) ;
4. Lecture de la donnée en mémoire (MD – *Memory Data*) ;
5. Rangement de la donnée dans `$d` (WB – *Write-back*) ;
6. PC $\leftarrow$ PC + 4.

La durée réelle d'exécution d'une instruction est de 5 cycles.

7.1 Réalisation "monoprogrammée"

Dans une réalisation "monoprogrammée", le processeur est considéré comme un automate, capable d'effectuer une seule instruction à la fois.



7.2 Réalisation "RISC pipe-line"

Dans une réalisation "RISC pipe-line", le processeur n'attend pas d'avoir terminé la dernière étape de l'instruction i pour commencer à exécuter l'instruction $i + 1$.