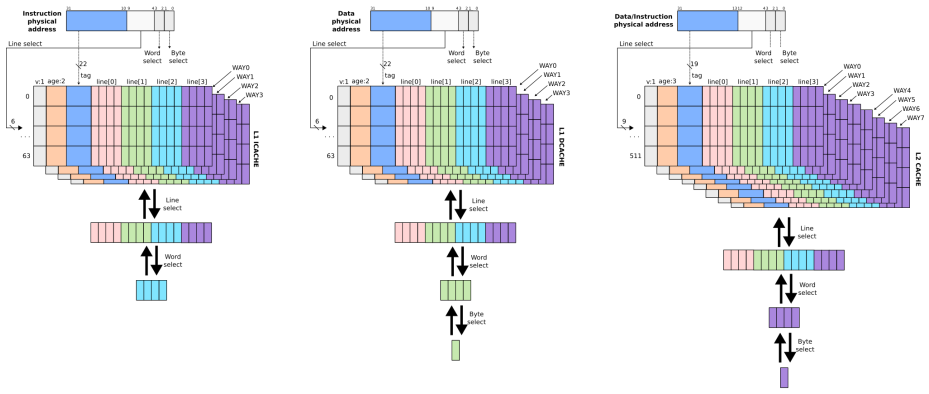
Mémoire caches hiérarchiques

Mirjana Stojilovic & Jean-Cédric Chappelier 2019

L'objectif de ces *trois* dernières semaines est l'aboutissement de tout le projet : l'implémentation d'une mémoire cache hiérarchique sur deux niveaux telle que décrite dans le descriptif général :

- deux mémoires cache niveau 1 séparées : une pour les instructions (ICache) et une pour les données (DCache) ; chacune de ces deux mémoires cache sera « *4-way set associative* », avec 64 lignes par « *way* », de 4 mots par ligne chacune ;
- une mémoire cache niveau 2, partagée (à la fois pour les instructions et pour les données) ; ce sera une mémoire « *8-way set associative* », avec 512 lignes par « *way* », de 4 mots par ligne chacune ;
  contrairement aux TLBs des semaines passées, cette mémoire cache partagée sera **exclusive** : elle ne contient que des données/instructions qui ne sont pas présentes dans les mémoires cache de niveau 1 et sera utilisée comme une « **victim cache** » (expliqué ci-dessous) ;
- la politique de remplacement utilisée sera « *least recently used* » (LRU).

Voici une illustration *complémentaire* de celle donnée dans le descriptif général qui devrait vous permettre de mieux comprendre la donnée :



Ce que vous avez concrètement à faire pour cette semaine c'est de :

- définir les types `l1_icache_entry_t`, `l1_dcache_entry_t`, `l2_cache_entry_t` et `cache_t` dans `cache.h` ;
- compléter le fichier `cache_mng.c` ;
- créer et compléter le fichier `lru.h` (mais il n'a pas de `.c`) ;
- éventuellement créer vos tests-unitaires ;
- (et, bien sûr, mettre à jour votre `Makefile`).

Cette partie étant la plus conséquente de toutes, nous avons décidé de l'étaler sur trois (voire quatre) semaines : ce sujet est donc le tout dernier que vous aurez. Votre projet à la fin de ce sujet (fin du semestre, donc) correspondra au *rendu complet* attendu pour ce cours. Il sera à rendre (détails en fin de ce sujet) au plus tard le lundi 03 juin.

I. Définition des types

Dans le fichier `cache.h`, définissez les types `l1_icache_entry_t`, `l1_dcache_entry_t`, `l2_cache_entry_t` et `cache_t` à l'endroit indiqué. Revoir pour cela le descriptif général et l'illustration qui s'y trouve, ainsi que l'illustration ci-dessus. Par rapport aux TLBs des semaines passées, une grosse différence au niveau des structures de données est que les caches considérées ici ont des *tableaux* de lignes. Ces tableaux sont appelés « *ways* ».

Le type `l1_icache_entry_t` est une structure représentant une entrée dans la L1_ICACHE ayant comme champs (voir les macros fournies en fin de `cache.h`) :

- 1 bit de validation, nommé simplement `v` ; on lira la valeur de ce bit au travers d'un `uint8_t` ;
- 2 bits, nommés simplement `age`, pour le calcul de « l'âge » d'une entrée de la cache : c'est le compteur utilisé par la politique d'éviction LRU pour savoir quelle est l'entrée « *least recently used* » ; 2 bits suffisent puisque cet « âge » peut au maximum être égal au nombre de « ways » (moins 1, puisqu'on part de 0) ;
  on lira leur valeur au travers d'un `uint8_t` ;
- 22 bits, nommés `tag` ; on lira la valeur de ces bits au travers d'un `uint32_t` ; (**note :** 22 c'est la taille d'une adresse physique (32 bits) moins le nombre de bits pour indexer une informations dans la cache : 6 bits pour la ligne (puisqu'il y a 64 lignes) + 2 bits pour le mot (il y a 4 mots par lignes) et 2 bits pour l'octet (puisqu'on adresse par octet et qu'un mot à 4 octets)) ;
- et enfin un tableau, nommé `line`, de `L1_ICACHE_WORDS_PER_LINE` mots (`word_t` dans `addr.h`, pour rappel).

La L1_ICACHE (que vous n'aurez pas à coder explicitement à moins que vous souhaitiez le faire dans vos propres tests) sera simplement un tableau de `L1_ICACHE_LINES * L1_ICACHE_WAYS l1_icache_entry_t`.

Pour simplifier dans ce projet, `l1_dcache_entry_t` est exactement la même

chose que `l1_icache_entry_t`, mais c'est uniquement un choix que nous avons fait pour simplifier le projet. Il est cependant important que le type `l1_dcache_entry_t` soit explicitement défini (il peut très bien être un alias de `l1_icache_entry_t`) et utilisé à bon escient à tous les endroits où il doit l'être en tant que tel (c.-à-d., bien qu'ils soient identiques au niveau implémentation par choix de simplification, ne les confondez pas au niveau conceptuel dans votre code).

De façon similaire, le type `l2_cache_entry_t` est une structure représentant une entrée dans la L2_CACHE ayant comme champs :

- 1 bit de validation, nommé simplement `v` ; on lira la valeur de ce bit au travers d'un `uint8_t` ;
- 3 bits, nommés `age`, pour le calcul de « l'âge » d'une entrée de la cache ; 3 bits sont nécessaires ici puisque l'on a huit « ways » ;
  on lira leur valeur au travers d'un `uint8_t` ;
- 19 bits, nommés `tag` ; on lira la valeur de ces bits au travers d'un `uint32_t` ;
  (**note :** 19, c'est 32 moins : 9 bits pour la ligne (puisqu'il y a 512 lignes) + 2 bits pour le mot (il y a 4 mots par lignes) et 2 bits pour l'octet ;
- et enfin un tableau, nommé `line`, de `L2_CACHE_WORDS_PER_LINE` mots.

Pour finir, le type `cache_t` permettra d'identifier les différentes caches : c'est un type énuméré ayant comme valeur possibles `L1_ICACHE`, `L1_DCACHE` et `L2_CACHE`.

II. Implémentation

Les types nécessaires étant définis, reste le plus gros du travail : l'implémentation des fonctionnalités de ces mémoires cache hiérarchiques. De façon similaire aux TLBs (mais très différents dans leur gestion), cela consiste en la définition, dans un fichier `cache_mng.c` à écrire, de fonctions permettant :

- d'initialiser une `cache_entry` (`cache_entry_init()`) ;
- d'initialiser une mémoire cache (`cache_flush()`) ;
- d'insérer une entrée dans la cache (`cache_insert()`) ;
- de vérifier si une donnée ou instruction est déjà dans la cache (`cache_hit()`) ;
- de demander à la cache un mot de donnée ou d'instruction en lecture (`cache_read()`) ;
- de demander à la cache un octet de donnée ou d'instruction en lecture (`cache_read_byte()`) ;
- de demander à la cache un mot de donnée en écriture (`cache_write()`) ;
  note : on n'écrit jamais d'instruction ;
- et enfin de demander à la cache un octet de donnée en écriture (`cache_write_byte()`).

Commencez par regarder le fichier `cache_mng.h` pour une description de ces fonctions, puis complétez le fichier `cache_mng.c` fourni.

II.1 `cache_entry_init()`

Cette fonction doit initialiser le `cache_entry` générique fourni en troisième argument à partir de l'adresse physique (et de l'adresse mémoire de départ) fournie(s) et du type de mémoire cache demandé. Comme avec les TLBs, le bit de validité de la `cache_entry` doit être mis à 1 ; et l'âge doit (bien sûr) être mis à 0.

Par contre, le tag est cette fois calculé à partir de l'adresse physique, en la décalant du nombre de bits correspondants à la mémoire cache visée :

- de `L1_ICACHE_TAG_REMAINING_BITS` si on initialise l'entrée pour un `L1_ICACHE` ;
- de `L1_DCACHE_TAG_REMAINING_BITS` si on initialise l'entrée pour un `L1_DCACHE` ;
- et de `L2_CACHE_TAG_REMAINING_BITS` si on initialise l'entrée pour un `L2_CACHE`.

Il faudra bien sûr au préalable convertir la `phy_addr_t` en une adresse sur 32 bit. Nous vous conseillons pour cela de faire une fonction ou une macro.

Enfin, il faut initialiser la ligne à partir des 4 mots mémoire situés à l'adresse physique donnée. Attention, à nouveau, au fait que l'adresse physique adresse des octets et que l'on cherche ici des mots en mémoire (revoir peut être à ce sujet ce que vous aviez fait pour `read_page_entry()` du « *page walker* » et le texte de la section II.3 du sujet de la semaine 6). De plus, la partie de l'adresse physique utilisée pour rechercher un **bloc** de mots doit (bien sûr) être alignée sur la taille de la ligne (= taille du bloc recherché) c.-à-.d multiple de la taille de la ligne (p.ex., multiples de `L1_ICACHE_LINE`).

Par exemple, l'adresse physique `0xA005` donnera l'index de bloc de mots `0x2800` parce que le multiple de 16 (un bloc de mots à mettre en ligne utilise 16 octets, car il y a 16 octets par ligne) correspondant à `0xA005` est `0xA000` et que `0xA000` octets correspondent à `0x2800` mots.

II.2 `cache_flush()`

Cette fonction met simplement à 0 tout le contenu de la mémoire cache reçue, que l'on suppose (sans vérification) être :

- pour un `L1_ICACHE`, un tableau (unidimensionnel) de `L1_ICACHE_LINES` fois `L1_ICACHE_WAYS` entrées de type `l1_icache_entry_t` ;
- pour un `L1_DCACHE`, un tableau (unidimensionnel) de `L1_DCACHE_LINES` fois `L1_DCACHE_WAYS` entrées de type `l1_dcache_entry_t` ;
- et pour un `L2_CACHE`, un tableau (unidimensionnel) de `L2_CACHE_LINES` fois `L2_CACHE_WAYS` entrées de type `l2_cache_entry_t`.

II.3 `cache_insert()`

Cette fonction met simplement à jour l'entrée de la mémoire cache située à l'index `cache_line_index` (si celui-ci est valide, c.-à-d. inférieur au nombre de lignes `L...CACHE_LINES` correspondant) dans la « table » (« *way* ») `cache_way` (si celui-ci est valide, c.-à-d. inférieur au nombre de « tables » `L...CACHE_WAYS` correspondant) à partir des information stockées dans le l'entrée générique fournie en troisième argument.

Il s'agit donc ici simplement d'assigner au bon endroit le `l...cache_entry_t` reçu.

Notez bien que cette fonction suppose travailler sur une `cache_line_in` et une `cache` de même type (invérifiable) indiqué par `cache_type`. Elle n'est donc pas faite pour insérer une ligne de L2 dans une cache L1 (pour cela, voir plus bas, vous devrez écrire votre propre code, différent).

II.4 `cache_hit()`

Cette fonction vérifie si l'instruction ou les données indiquée par l'adresse physique se trouvent dans la mémoire cache passée en argument, et si oui effectue les mises à jour nécessaires (expliquées ci-dessous).

Pour rechercher l'adresse physique fournie, il faut bien sûr commencer (après les vérifications d'usage) par la convertir sur 32 bits comme déjà effectué dans `cache_entry_init()`. L'index de ligne dans la mémoire cache concerné est ensuite calculé à partir de cette adresse en la divisant par 16 (il y a 16 octets par ligne) puis en prenant le modulo avec le nombre de lignes correspondant (`L...CACHE_LINES`).

On calcule ensuite le tag correspondant (comme expliqué dans `cache_entry_init()`).

On recherche ensuite **dans l'ordre des « *ways* »** (vous pouvez utiliser pour cela la macro fournie `foreach_way()`) :

- soit une entrée invalide (on appelle cela « *cold start* » et l'on mémorisera ce fait pour la suite) ;
- soit une entrée valide dont le tag correspond au tag recherché ; auquel cas on met à jour les informations `hit_way` par le numéro de « *way* » trouvé, `hit_index` par l'index de ligne et `p_line` par le contenu mémorisé correspondant (voir la macro `cache_line()` fournie).

On l'arrêtera la recherche dès que l'une de ces deux situations est atteinte (= la première place vide ou qui correspond).

Si rien n'est trouvé (ou que l'on a un « *cold start* »), il faut alors mettre `hit_way` à `HIT_WAY_MISS` et `hit_index` à `HIT_INDEX_MISS`. Autrement dit, le seul cas de « *hit* » est si l'on trouve une entrée valide dont le tag correspond au tag recherché ; tous les autres cas sont des « *miss* ».

Si l'on a trouvé une place (« *cold start* » ou « *hit* »), alors il faut encore mettre à jour les informations « d'âge » pour la politique de remplacement :

- si l'on a eu un « *cold start* », il faut simplement augmenter l'âge de toutes les autres entrées (cf `LRU_age_increase()` dans la section suivante) ;
- si l'on a eu un « *hit* », il faut mettre à jour tous les âges inférieurs (cf `LRU_age_update()` dans la section suivante).

II.5 `lru.h`

Dans le fichier `lru.h` (à inclure simplement dans `cache_mng.c`), nous vous demandons simplement de définir deux macros :

- `LRU_age_increase(TYPE, WAYS, WAY_INDEX, LINE_INDEX)` qui augmente de 1 l'âge de l'entrée de chacune des tables (« *way* ») situées à la ligne `LINE_INDEX` et met à 0 l'âge de celle située (à la ligne `LINE_INDEX`) dans la table `WAY_INDEX` ; on veillera cependant à ne pas incrémenter les âges qui sont déjà au maximum (`WAYS - 1`) ;
- `LRU_age_update(TYPE, WAYS, WAY_INDEX, LINE_INDEX)` qui augmente de 1 l'âge de l'entrée de chacune des tables (« *way* ») situées à la ligne `LINE_INDEX` et dont l'age est strictement inférieur à l'âge de l'entrée située ligne `LINE_INDEX` de la table `WAY_INDEX` ; puis met à 0 l'âge de celle-ci (celle située à la ligne `LINE_INDEX` dans la table `WAY_INDEX`).

II.6 `cache_read()`

Cette fonction est avec `cache_write()` une des deux plus complexes de cette partie (mémoires-caches). Elle gère de bout en bout la recherche d'une information (donnée ou instruction) lue à partir d'une adresse physique dans toute la hiérarchie de mémoires-caches. Mais si vous modularisez bien ce qu'elle doit faire, ce n'est pas si difficile :

- après les vérifications d'usage (qui doivent inclure la vérification de l'adresse fournie est bien alignée sur des adresses de mots), on commence par voir si l'information recherchée est dans (« *hit* », fonction précédente) la mémoire niveau 1 correspondante (ICACHE ou DCACHE en fonction de l'accès demandé) ; si elle y est (pas de `HIT_WAY_MISS`), c'est parfait : pratiquement tout ayant déjà été par la fonction `cache_hit()`, il n'y a rien de plus à faire que ce qu'il faut de toutes façons faire au final dans tous les cas : affecter `word` à la valeur du mot trouvé ;
- si par contre, l'information recherchée n'est pas dans la cache niveau 1 (« *L1 miss* »), alors on la recherche dans la cache niveau 2 (que ce soit un accès pour instruction ou pour donnée n'importe pas ici puisque le niveau 2 est commun) :
  - si on la trouve au niveau 2 (pas de `HIT_WAY_MISS`) : alors il faut transférer cette information dans la mémoire cache niveau 1 correspondante (ICACHE ou DCACHE en fonction de l'accès demandé)

et l'*invalider* dans la cache niveau 2 (détails ci-dessous); puis, comme pour tous les autres cas, affecter `word` à la valeur du mot trouvé ;
  - si l'information recherchée n'est pas non plus dans la mémoire cache niveau 2 (« *L2 miss* »), alors il faut aller la chercher en mémoire principale et la mettre dans la cache niveau_1 correspondante (ICACHE ou DCACHE en fonction de l'accès demandé) : le principe est le même que dans le cas précédent sauf que l'information vient de la mémoire (`cache_entry_init()`).

Pour le déplacement de l'information de la cache niveau 2 vers la cache niveau 1, voici comment procéder :

- assigner les informations de ligne de la `l2_cache_entry_t` vers la `l1_...cache_entry_t` ;
- calculer le tag de la nouvelle `l1_...cache_entry_t`
- invalider l'entrée correspondante dans la cache niveau 2 ;
- vérifier s'il y a une place dans la cache niveau 1 ;
- si *oui* :
  - insérer à cette place ;
  - puis mettre à jour (pour la cache niveau 1) les informations « d'âge » pour la politique de remplacement, exactement comme cela a été fait pour `cache_hit()` ;
- si *non* :
  - supprimer(=remplacer, mais la mémoriser pour ci-dessous) l'information qui doit être remplacée dans la cache niveau 1 (suivant la politique de remplacement) ;
  - insérer à cette place ;
  - puis mettre à jour (pour la cache niveau 1) les informations « d'âge » ;
  - rechercher s'il y a de la place dans la cache niveau 2 ;
  - si *oui* :
    - insérer dans la cache niveau 2, l'information éjectée de la cache niveau 1 ;
    - puis mettre à jour (pour la cache niveau 2) les informations « d'âge » ;
  - si *non* :
    - supprimer l'information qui doit être remplacée dans la cache niveau 2 (suivant la politique de remplacement) ;
    - insérer dans la cache niveau 2, l'information éjectée de la cache niveau 1 ;
    - puis mettre à jour (pour la cache niveau 2) les informations « d'âge ».

Conseil : il pourrait être utile de définir des fonctions ou des macros pour :

- mettre à jour les informations « d'âge » pour la politique de remplacement ;
- trouver une place vide dans une « table » (« *way* ») ;
- sortir/supprimer (« *evict* ») une entrée d'une mémoire cache, suivant la politique de remplacement ;

- insérer une information dans la cache niveau 1 (et déplacer vers le cache niveau 2 l'information supprimée en cas d'éviction) ;
- déplacer une information de la cache niveau 2 vers une cache niveau 1 (algorithme décrit ci-dessus) ;
- et bien sûr toute autre tâche que vous trouver répétitive.

II.7 `cache_read_byte()`

Cette fonction a pour but de lire un seul octet de la mémoire. Elle est simple à écrire en ce sens qu'il suffit de lire le mot correspondant, puis de ne « renvoyer » (= affecter le contenu pointé par `p_byte`) que l'octet correspondant (via décalage et masquage). Le seul travail à faire ici consiste à :

- calculer l'adresse du mot dans lequel l'octet se trouve (on dit : l'adresse « alignée sur les mots », il suffit simplement que l'offset soit un multiple de la taille des mots) ;
- d'affecter le bon octet à `p_byte`.

II.8 `cache_write()`

Cette fonction est assez similaire à `cache_read()` sauf qu'il s'agit ici d'écrire une information (« donnée » forcément, on n'écrit pas d'instruction) vers la mémoire ; ce qui rend les choses encore un peu plus compliquées.

Au niveau des vérifications d'usage, il ne faudra pas oublier, comme pour `cache_read()` de vérifier que l'adresse reçue est bien « alignée sur les mots ».

Si l'information se trouve dans la `L1_DCACHE` (pas de `HIT_WAY_MISS` sur `cache_hit()`), alors :

- lire la ligne correspondante ;
- modifier le mot visé ;
- ré-insérer la ligne (modifiée) ;
- mettre à jour (pour la `L1_DCACHE`) les informations « d'âge » ;
- écrire l'information (toute la ligne) dans la mémoire centrale (« *write-through cache* »).

La cache `L2_CACHE` n'est pas modifiée.

Si l'information ne se trouve pas dans la `L1_DCACHE` (`HIT_WAY_MISS` sur `cache_hit()`), alors on la recherche dans la cache niveau 2. Si on l'y trouve :

- lire la ligne correspondante (dans `L2_CACHE`, donc) ;
- modifier le mot visé ;
- ré-insérer la ligne (modifiée) dans la cache `L2_CACHE` ;
- mettre à jour (pour la `L2_CACHE`) les informations « d'âge » ;
- appliquer le même déplacement de l'information de la cache niveau 2 vers la cache niveau 1 que pour `cache_read()` ;
- écrire l'information (toute la ligne) dans la mémoire centrale.

Enfin, si l'information n'est trouvée ni dans la `L1_DCACHE` ni dans `L2_CACHE`, alors :

- lire (toute) la ligne correspondante dans la mémoire centrale (revoyez ce que vous avez fait dans `cache_entry_init()` ;
- modifier le mot visé ;
- écrire l'information (toute la ligne) dans la mémoire centrale ;
- mettre à jour la `L1_DCACHE` à partir de la mémoire (si possible en s'appuyant sur des fonctionnalités déjà écrites).

II.9 `cache_write_byte()`

Cette fonction a pour but d'offrir l'interface pour écrire un seul octet de la mémoire. Bien sûr en réalité, c'est toute une ligne de mots qui est finalement écrite, mais c'est justement le rôle de cette fonction que de trouver le bon emplacement correspondant à un octet donné. Comme pour `cache_read_byte()`, il faudra donc commencer par calculer l'adresse du mot correspondant à l'octet désiré.

Ensuite on ira lire le mot correspondant ; puis l'on y modifiera l'octet indiqué et l'on terminera en écrivant (`cache_write()`) le mot ainsi modifié.

III. Tests

Comme toujours, nous vous fournissons pour cette dernière étape :

- des tests de feedback sur les mémoires caches ;
- un outil de test de ces mémoire et 1 scénario de test associé (toujours le même, mais le résultat diffère).

L'outil de test en question est `test-cache.c`. A recopier, donc, depuis `provided/` dans `done/`, puis à ajouter à votre `Makefile`. Il fonctionne de façon la façons suivante :

```
./test-cache (dump|desc) mem_filename command_filename
```

p.ex. :

```
./test-cache dump tests/files/memory-dump-01.mem tests/files/commands01.txt resultat.txt
```

Note : cet exemple est en fait exactement ce que fait `make check` (qui lance `tests/11.basic.sh` qui exécute exactement la commande ci-dessus et vérifie le résultat). Le résultat attendu est fourni dans `tests/files/output/cache-01-out.txt`.

Comme la semaine passée, vous pouvez bien sûr écrire vos propres scénarios d'accès à des adresses virtuelles (comme les fichiers `tests/files/commands01.txt` ou `tests/files/commands02.txt`) afin de vérifier le fonctionnement de vos mémoires caches. Nous vous encourageons même à partager de tels scénarios de tests, par exemple dans un fil de discussion dédié sur le forum Moodle, en fournissant le fichier de commandes et son fichier résultat attendu.

IV. Rendu

Le code à ce stade (c.-à-d. tout le travail depuis la semaine 4) constitue le **rendu final** de la partie projet de ce cours. Il est à rendre avant le **lundi 3 juin 23:59**. Pour le rendre, il n'y aura rien à faire de plus que d'avoir bien ajouté (`git add`), validé (`git commit`) et transmis (`git push`) toutes vos dernières versions de tous vos fichiers sources `.c` et `.h`, ainsi que le `Makefile`. Ce sera en effet la version se trouvant dans votre branche principale (`master`) le mardi 4 juin 00:00 qui sera considérée comme votre rendu final.

Merci de ne pas ajouter les fichiers `.o`, ni les exécutables à votre rendu. De plus, avant de soumettre, veuillez également retirer (ou commenter) tous les appels à `printf()` superflus que vous auriez pu ajouter. Nous vous conseillons d'ailleurs d'utiliser plutôt le flux d'erreur `stderr` (`fprintf(stderr,`) car nous ne testons pas son contenu.