TLB hiérarchiques

Mirjana Stojilovic & Jean-Cédric Chappelier 2019

L'objectif de cette semaine et la semaine prochaine est d'implémenter un TLB hiérarchique sur deux niveaux, « *direct mapped* », tel que décrit dans le descriptif général :

1. deux TLBs de niveau 1 : un pour les instructions et un pour les données ;
2. un TLB commun (instructions et données) de niveau 2.

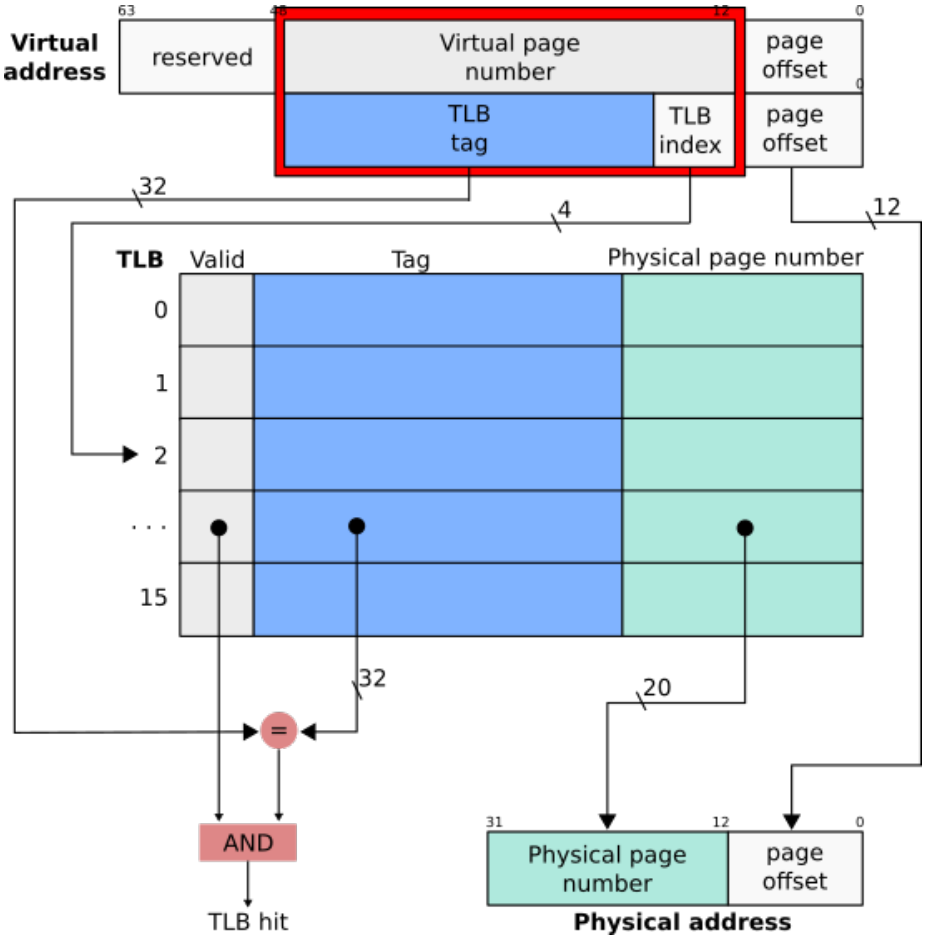
Ce que vous avez concrètement à faire pour cette semaine c'est de :

- définir les types `l1_itlb_entry_t`, `l1_dtlb_entry_t`, `l2_tlb_entry_t` et `tlb_t` dans `tlb_hrchy.h` ;
- créer et remplir le fichier `tlb_hrchy_mng.c`.

Cette partie étant plus conséquente, nous avons décidé de l'étaler sur deux semaines : ce sujet sert donc pour cette semaine et la suivante. De plus, l'état du projet à la fin de ce sujet (fin de semaine prochaine donc) correspond au *second rendu noté* de ce projet. Il sera à rendre (détails en fin de ce sujet) au plus tard le dimanche 12 mai.

I. Description générale (rappels du descriptif général)

Tous les TLBs considérés dans cette partie sont « *direct mapped* », c.-à-d. que chaque page virtuelle a sa propre entrée dans la TLB ; autrement dit, une partie des bits de l'adresse virtuelle sert également à indexer directement la TLB comme illustré sur cette figure :



Il n'y a donc pas besoin de politique de remplacement ici.

Notre hiérarchie de TLBs a deux niveaux :

- deux TLB de niveau 1 (« *Level-1 TLB* ») : un pour les instructions (ITLB) et un pour les données (DTLB), contenant 16 entrées chacun (pour simplifier le projet, nous avons décidé que le DTLB aurait la même implémentation que le ITLB; mais conceptuellement ce sont deux entités clairement différentes qui pourraient très bien être implémentées différemment) ;
- un seul TLB de niveau 2 (« *Level-2 TLB* ») utilisé à la fois pour les accès aux instructions et les accès aux données, contenant 64 entrées. Ce TLB partagé sera « *inclusive* » : en cas d'échec (« *miss* ») avec le ITLB (resp. la DTLB), l'information est alors recherchée dans le TLB niveau 2 ; si l'information s'y trouve, alors cette information est **recopiée** dans le ITLB (resp. DTLB) niveau 1 ; sinon, cette information est créée (mécanisme de traduction d'adresse virtuelle en adresse physique expliqué précédemment)

puis insérée **à la fois** dans le ITLB (resp. DTLB) niveau 1 **et** dans le TLB partagée niveau 2.

Pour résumer, nous aurons dans ce projet :

- un ITLB niveau 1, direct-mapped, à 16 entrées ;
- un DTLB niveau 1, direct-mapped, à 16 entrées ;
- un TLB niveau 2 partagé, inclusive, direct-mapped, à 64 entrées.

II. Définition des types

Dans le fichier `tlb_hrchy.h`, définissez les types `l1_itlb_entry_t`, `l1_dtlb_entry_t`, `l2_tlb_entry_t` et `tlb_t` à l'endroit indiqué.

Le type `l1_itlb_entry_t` est un bitfield ayant :

- 32 bits, nommés `tag` ; on lira la valeur de ces bits au travers d'un `uint32_t` ; (**note :** 32 c'est la taille d'un numéro de page virtuelle (36 bits) moins le nombre de bit pour représenter toutes les lignes (16 lignes, donc 4 bits)) ;
- `PHY_PAGE_NUM` bits, nommés `phy_page_num` ; on lira la valeur de ces bits au travers d'un `uint32_t` ;
- 1 bit de validation, nommé simplement `v` ; on lira la valeur de ce bit au travers d'un `uint8_t` ;

(revoyez aussi ce que vous avez fait pour le type `tlb_entry_t` la semaine passée).

Vu la description donnée en Section I, il est clair que dans ce projet `l1_dtlb_entry_t` est exactement la même chose que `l1_itlb_entry_t`, mais c'est uniquement un choix que nous avons fait pour simplifier le projet. Il est cependant important que le type `l1_dtlb_entry_t` soit explicitement défini (il peut très bien être un alias de `l1_itlb_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_tlb_entry_t` est un bitfield ayant :

- 30 bits, nommés `tag` ; on lira la valeur de ces bits au travers d'un `uint32_t` ; (**note :** 30 c'est la taille d'un numéro de page virtuelle (36 bits) moins le nombre de bit pour représenter toutes les lignes (64 lignes, donc 6 bits)) ;
- `PHY_PAGE_NUM` bits, nommés `phy_page_num` ; on lira la valeur de ces bits au travers d'un `uint32_t` ;
- 1 bit de validation, nommé simplement `v` ; on lira la valeur de ce bit au travers d'un `uint8_t`.

Pour finir, le type `tlb_t` permettra d'identifier les différents TLBs : c'est un type énuméré ayant comme valeur possibles `L1_ITLB`, `L1_DTLB` et `L2_TLB`.

III. Implémentation

Les types nécessaires étant définis, reste le plus gros du travail : l'implémentation des fonctionnalités de ce TLB hiérarchique. De façon similaire (mais différente dans leur réalisation), cela consiste en la définition, dans un fichier `tlb_hrchy_mng.c` à écrire, de fonctions permettant :

- d'initialiser une `tlb_entry` (`tlb_entry_init()`) ;
- d'initialiser un TLB (`tlb_flush()`) ;
- d'insérer une adresse dans le TLB en suivant sa politique de replacement (`tlb_insert()`) ;
- de vérifier si une adresse est déjà dans le TLB (`tlb_hit()`) ;
- (la fonction de plus haut niveau) de demander au TLB une traduction d'adresse virtuelle (`tlb_search()`).

Commencez par regarder le fichier `tlb_hrchy_mng.h` pour une description de ces fonctions, puis créez et complétez (et `git add`) un fichier `tlb_hrchy_mng.c` implémentant ces fonctions.

III.1 `tlb_entry_init()`

Comme pour le TLB simple de la semaine passée, cette fonction doit initialiser le `tlb_entry` fourni en troisième argument avec les deux informations fournies et du type de TLB demandé. Comme la semaine passée, le bit de validité de la `tlb_entry` doit être mis à 1.

Par contre, la différence par rapport au travail de la semaine passée est que le tag est cette fois réduit du nombre de bits correspondant au TLB visé (cf l'image ci-dessus):

- de `L1_ITLB_LINES_BITS` si on initialise l'entrée pour un `L1_ITLB` ;
- de `L1_DTLB_LINES_BITS` si on initialise l'entrée pour un `L1_DTLB` ;
- et de `L2_TLB_LINES_BITS` si on initialise l'entrée pour un `L2_TLB`.

III.2 `tlb_flush()`

Cette fonction met simplement à 0 tout le contenu du TLB reçu, que l'on suppose (sans vérification) être :

- pour un `L1_ITLB`, un tableau de `L1_ITLB_LINES` entrées de type `l1_itlb_entry_t` ;
- pour un `L1_DTLB`, un tableau de `L1_DTLB_LINES` entrées de type `l1_dtlb_entry_t` ;
- et pour un `L2_TLB`, un tableau de `L2_TLB_LINES` entrées de type `l2_tlb_entry_t`.

III.3 `tlb_insert()`

Cette fonction met simplement à jour l'entrée du TLB située à l'index `line_index` (si celui-ci est valide, c.-à-d. inférieur au nombre de lignes `L...TLB_LINES` correspondant) à partir des information stockées dans le l'entrée générique fournie en second argument.

Il est conseillé ici d'utiliser le « *casting* » et l'arithmétique de pointeurs.

III.4 `tlb_hit()`

Comme la semaine passée et contrairement à la majorité de nos fonctions, cette fonction ne retourne pas de code d'erreur, mais simplement 0 si l'adresse virtuelle n'est pas déjà dans le TLB demandé (indiqué par son type) et 1 si elle l'est. On retournera donc également 0 dès qu'il y a le moindre problème (puisqu'on aura alors pas trouvé l'adresse virtuelle recherchée).

Pour rechercher l'adresse virtuelle fournie, il faut bien sûr commencer par extraire, sur 64 bits, le « *virtual page number* » de l'adresse virtuelle fournie en premier argument (cf image ci-dessus). Mais la recherche est ensuite plus simple et plus directe que la semaine passée : l'index (= le numéro de ligne) dans le TLB concerné est simplement directement calculé à partir du « *virtual page number* » : c'est simplement son modulo avec le nombre de lignes correspondant (`L...TLB_LINES` ; revoir l'image donnée au début de ce sujet).

Si l'entrée située à cette ligne est valide (champ `v`) et que son « *tag* » correspond (cf image donnée au début de ce sujet), alors :

- on affecte l'adresse physique (second argument de la fonction) avec les valeurs correspondante ;
- et on retourne 1.

III.5 `tlb_search()`

Cette fonction est de loin la plus complexe. Elle gère de bout en bout la recherche d'une adresse physique à partir d'une adresse virtuelle dans toute la hiérarchie de TLB :

- après les vérifications d'usage, on commence par voir si l'adresse recherchée est dans (« *hit* », fonction précédente) le TLB niveau 1 correspondant (ITLB ou DTLB en fonction de l'accès demandé) ; si elle y est, c'est parfait, il n'y a rien de plus à faire qu'à mettre le paramètre `hit` à 1, tout le reste ayant déjà été par la fonction `tlb_hit()` ;
- si par contre, elle n'y est pas (« *L1 miss* »), alors on la recherche dans le TLB 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 : il ne reste plus qu'à mettre le paramètre `hit` à 1, mais aussi à reporter les bonnes information dans le TLB

niveau 1 correspondant (ITLB ou DTLB en fonction de l'accès demandé) ;
- si l'adresse n'est pas non plus dans le TLB niveau 2 (« *L2 miss* »), alors (c'est la partie compliquée) :
  * on met le paramètre `hit` à 0 ;
  * on appelle le « *page walker* » pour traduire l'adresse virtuelle ;
  * puis (s'il n'y a pas d'erreur) on initialise un `l2_tlb_entry_t` avec les valeurs obtenues ;
  * puis on l'insère dans le TLB niveau 2 à l'index correspondant (comme expliqué pour `tlb_hit()`) ;
  * et on fait de même (initialisation d'une entrée et insertion) pour le TLB niveau 1 correspondant (ITLB ou DTLB en fonction de l'accès demandé) ; **attention !** les indices pour la TLB de niveau 2 et ceux pour les TLB de niveau 1 ne sont pas les mêmes (pas la même taille) !!) ;
  * puis, enfin, on invalide l'entrée dans **l'autre** TLB niveau 1 si nécessaire : si on a inséré dans la `L1_ITLB` et qu'il y a une entrée valide correspondant à la même entrée du TLB niveau 2 dans la `L1_DTLB` (le vérifier et attention aux index !!), alors il faut invalider cette entrée de la `L1_DTLB` ; et de même si l'on a inséré dans la `L1_DTLB` (et qu'il y a une entrée valide correspondant à la même entrée du TLB niveau 2), il faut invalider l'entrée correspondant de la `L1_ITLB`.
  * Ouf !

VI. Tests

Comme la semaine passée, nous vous fournissons pour cette semaine :

- des tests de feedback sur les TLB ;
- 1 outil de test des TLB hiérarchique et 1 scénario de test associé (toujours le même, mais le résultat diffère).

L'outil de test en question est `test-tlb_hrchy.c`. A recopier, donc, depuis `provided/` dans `done/` puis à ajouter à votre `Makefile`. Il fonctionne exactement comme `test-tlb_simple` de la semaine passée, p.ex. :

```
./test-tlb_hrchy tests/files/commands02.txt tests/files/memory-dump-01.mem resultat.txt
```

Note : cet exemple est en fait exactement ce que fait `make check` (qui lance `tests/09.basic.sh` qui exécute exactement la commande ci-dessus et vérifie le résultat). Le résultat attendu est fourni dans `tests/files/output/tlb-hrchy-01-out.txt`.

Comme la semaine passée, vous pouvez bien sûr écrire vos propres scénarios d'accès à des adresses virtuelles (comme le fichier `tests/files/commands02.txt`) afin de vérifier le fonctionnement de votre TLB. 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 second rendu de la partie projet. Il est à rendre avant le **dimanche 12 mai 23:59**. Pour le rendre, le plus simple est de faire

```
make submit2
```

(**attention au chiffre 2 ici !**).

Avant la soumission, vérifiez 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`. Merci par contre de ne pas ajouter les fichiers `.o`, ni les exécutables.

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.

Ce qui sera considéré comme rendu sera ce que l'on trouvera dans (la branche `master` de) votre dépôt à la date indiquée ci-dessus et marqué d'un tag `projet01_NB`. C'est ce que fait la commande

```
make submit2
```

La raison pour laquelle nous étiquetons (`git tag`) votre contenu est pour vous permettre de continuer à travailler et prendre de l'avance : ainsi si votre dépôt contient à la date de rendu une version en avance sur le rendu et qui n'est pas fonctionnelle, ce n'est pas grave, nous ne prendrons que la dernière version pour laquelle vous aurez fait

```
make submit2
```

Ne faites donc pas de « `make submit2` » sur une version qui ne compile pas...