HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale

CONCEPTION DE SYSTÈME NUMÉRIQUE SUR FPGA
CSF

MÉMOIRE CACHE

João Miguel Domingues Pedrosa
Rick Wertenbroek

23 juin 2016

Table des matières

1 Introduction

Lors de ce laboratoire nous avons implémenté une mémoire cache. Mémoire qui sera directement implantée dans un design FPGA. Le but d'une mémoire cache étant de diminuer le temps des lectures et écritures en évitant de devoir accéder la mémoire à chaque fois. Notre implémentation de mémoire cache sera utilisée par un (et un seul) agent et interfacera une mémoire (par exemple RAM DDR, qui sera dans ce laboratoire simulée en VHDL).

2 Mémoire Cache

2.1 Mémoire

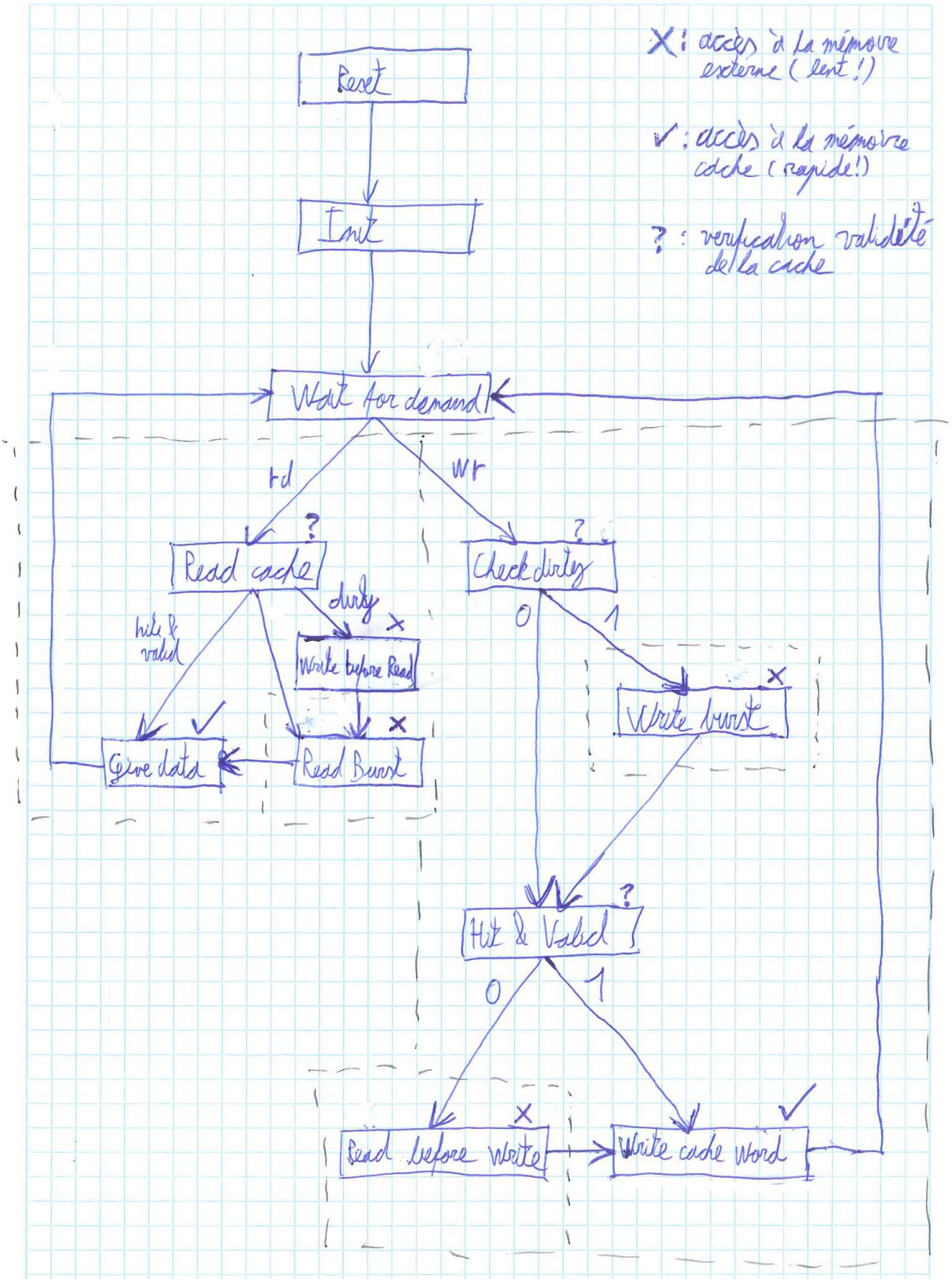
Pour ce laboiratoire nous avons implémenté la mémoire cache comme une cache à correspondance directe avec des lignes de 16 mots, ceci s'adapte en fonction des paramètres génériques (si l'adresse devient plus longue et la taille du tag et de l'index ne changent pas les lignes auront plus de mots. Nous avons aussi testé avec des tag et index plus grands qui nous donnaient des lignes de cache de 2 mots, cela fonctionnait aussi, mais c'était bien sur moins performant...).
La mémoire se sépare en plusieurs lignes qui ont les attributs suivants :

- Data (vecteur) : il s'agit des donnée (dans le tests bench 16 mots par ligne).
- Tag (vecteur) : la mémoire cache est plus petite que la RAM il y a alors collisions, le tag permet de savoir si l'adresse correspond à la ligne.
- Valid (bit) : indique si la ligne en cache est initialisée. (Si les données proviennent de la mémoire, car au reset les registres ne sont pas initialisés et la cache peut contenir n'importe quoi.
- Dirty (bit) : indique si il y a eu un accès en écriture dans la ligne. Ces lignes devront d'abord être écrites en mémoire avant de les remplacer.

Pour chaque attribut, nous avons un tableau de la taille du nombre de lignes de la cache que l'on accédera grâce à l'index bits après le tag dans l'adresse recherchée par l'agent.

2.2 Contrôleur

Pour le contrôleur de la mémoire cache nous avons créé une machine séquentielle synchrone.

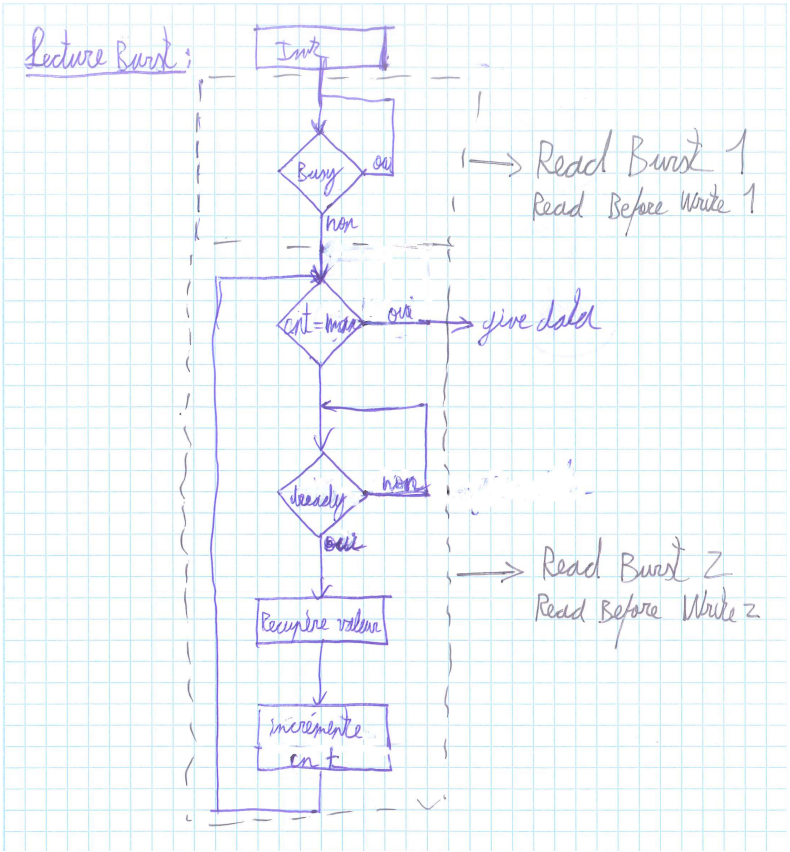


On commence tout d'abord par une initialisation des différents attributs de la cache. Il s'agit notamment de mettre les bits `valid` et `dirty` à 0 pour indiquer que la cache est vide (non initialisée en fait) et qu'elle doit donc être remplie avant lecture (son contenu peut être aléatoire).

Ensuite, dans `wait for demand`, on attend un signal (une demande) de lecture ou écriture (`rd` ou `wr`). On active le signal `busy` pour indiquer qu'on est occupé et on redirige dans la partie concernée. Il sera désactivé une fois l'action terminée.

2.2.1 Lecture

Pour la lecture, on vérifie d'abord s'il y a un hit. Pour cela, on regarde si le tag trouvé via l'adresse correspond à celui en cache et que la ligne est valide. Si la réponse est positive, on récupère directement la valeur dans la mémoire cache, sinon on doit chercher la ligne en mémoire d'abord. Avant de mettre à jour la ligne de la cache, on vérifie que la ligne actuelle n'est pas `dirty` si le tag est différent. Dans ce cas il faut d'abord l'écrire en mémoire avant de la remplacer par une nouvelle. L'accès en mémoire se fait à chaque fois en burst (plus rapide et plus simple) de la manière suivante :

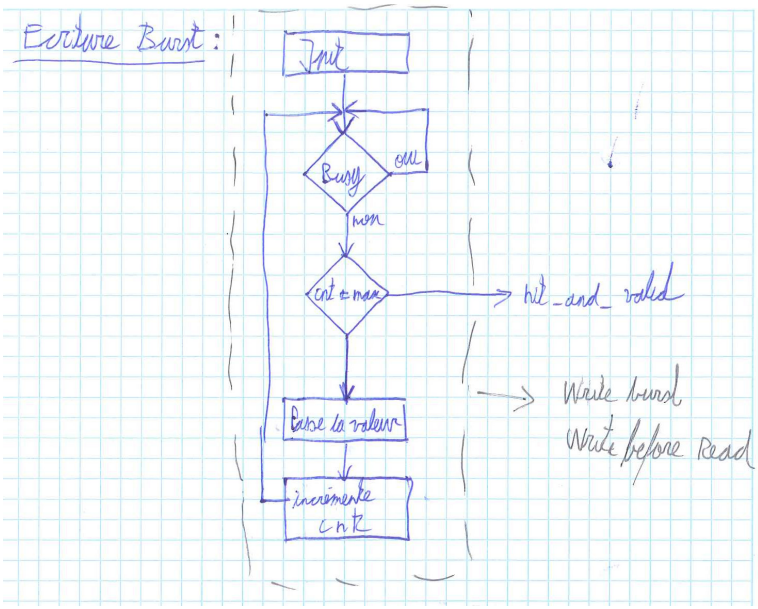


On attend tout d'abord sur le busy car il se peut que la mémoire soit occupé lorsqu'on veut l'accéder. Une fois la mémoire libre, on lui demande les données, on vérifie qu'elles soient prêtes via le `dready` et les enregistres. On regarde que la mémoire à finit le transfère via un compteur que l'on incrémente à chaque fois qu'une donnée a été récupéré avec succès.

2.2.2 Écriture

Lors de l'écriture il y a plusieurs choses à vérifier. Si il y a déjà une ligne de cache valide appartenant à une autre adresse et qu'elle a été modifiée (dirty) il faudra la stocker en mémoire avant de la remplacer. Ensuite il faudra récupérer en mémoire la ligne sur laquelle on veut écrire afin qu'elle soit cohérente (Read before write). Une fois que la ligne a été cherchée en mémoire elle est bien-sur valide et une fois qu'on y a écrit notre valeur on lui met à jour le bit dirty.

L'écriture se fait en burst et se fait de la manière suivante :



Par rapport à la lecture (Read before Write), on n'attend toujours sur le signal `dready` de la mémoire.

En résumé, pour l'écriture on a quatre cas :

1. La ligne de cache est vide : il faut aller chercher la ligne dans la mémoire (et donc mettre à jour le tag et le bit valid) puis on la modifie et met le bit dirty.
2. La ligne de cache correspond à l'adresse à écrire : il suffit de modifier la ligne et mettre le bit dirty.
3. La ligne de cache contient un autre tag non dirty : on traite comme le premier cas.
4. La ligne de cache contient un autre tag et est modifiée (dirty) : on doit stocker ces informations en mémoire puis on traite comme le premier cas.

3 Mémoire simulée

Pour la mémoire simulée nous avons créé une petite mémoire de 256*8bits qui peut être accédée de manière directe ou en burst, elle utilise des signaux busy et dready pour communiquer son état. La latence est simulée en attendant un nombre de coups de clocks qui est défini dans une constante. La latence aurait aussi pu être simulée ainsi : wait for xxx ns ; wait until rising_edge(clk_i) ; afin de pouvoir spécifier des latences en terme de temps minimaux (xxx ns) et rester quand même synchrone.
Le fichier correspondant est memory_emul_tb.vhd

4 Banc Test

Pour notre banc test nous avons vérifié le bon fonctionnement de la mémoire cache pour :

— Des cas de bord : Écrire après une écriture, on devrait voir que la seconde écriture se fait bien plus vite car on n'a pas besoin d'aller chercher la ligne en mémoire. On voit bien dans le chronogramme que le premier accès prends plus de temps (lecture burst de la ligne).
Écrire dans une adresse puis dans la suivante qui est dans la même ligne, on voit ici que la deuxième lecture est plus rapide, pour les mêmes raisons que ci-dessus. Lire la mémoire à une adresse pas encore en cache puis l'adresse suivante, on voit que la première lecture est lente puis la deuxième très rapide car la ligne est déjà cachée.
Écrire à deux adresses qui donnent le même index dans la cache (on a donc une collision) et voir que les données qui étaient présentes dans la cache se trouvent bien stockées en mémoire

avant que la cache soit mise à jour.

— Un parcours ascendant de toutes les adresses de la mémoire simulée : On parcourt ici toute la plage d'adresse en écriture puis en lecture, on va voir que la cache sera mis à jour en conséquence et qu'on ne perds aucune information, tout est stocké dans la cache ou la mémoire le cas échéant. Après qu'une ligne a du être mise à jour dans la cache tous les accès suivants à cette même ligne sont rapides. Ce test provoque aussi des collisions, faire toutes les écritures puis tout relire nous permet de nous assurer qu'aucune donnée n'a été perdue (car il y a forcément des flush de la cache vers la mémoire à cause des collisions).

— Un parcours descendant de toutes les adresses de la mémoire simulée : Même idée que le test ci-dessus mais cette fois ci on devra chercher les lignes en mémoire depuis l'adresse de poids fort et on peut voir que cela fonctionne tout aussi bien. Les accès aux adresses en dessous sont donc plus rapides, jusqu'à ce que l'on doive recharger une nouvelle ligne de la mémoire vers la cache. On voit aussi que toutes les données sont préservées.

— Des accès locaux successifs : On essaie de simuler ici un usage un peu normal de la cache, plusieurs écritures successives à des lignes proches les unes des autres, puis après toutes ces écritures on observe que les données sont bien correctes. On peut voir qu'une fois la ligne chargée en cache (16 mots et on accède 10 mots) qu'on ne travaille que dans la cache et qu'il n'y a pas d'accès à la mémoire ici. Tout est très rapide et on peut le voir dans le chronogramme.

La procédure control_read nous permet de vérifier la cohérence des données pour une adresse donnée en cache, il y aura une failure dans le cas contraire. (il aurait été bien de faire une collection des erreurs et indiquer à la fin du test bench le nombre d'erreurs et les adresses) mais pour l'instant nous avons simplement fait générer une failure.

5 Synthèse

Notre implémentation est synthétisable, la synthèse a été effectuée avec Quartus II. Il faut toutefois faire attention à définir les tailles des vecteurs dans les records du package cmf_pkg.vhd sans cela la synthèse ne passera pas.

6 Conclusion

Implémenter une mémoire cache n'était pas si simple que ça, il a fallu beaucoup penser au cas de bord comme les collisions par exemple afin de ne pas faire d'erreurs. Ce n'est pas évident de prévoir qu'une lecture demandera de stocker des informations en mémoire par exemple... Une fois tous les cas de bord étudiés, la machine d'état élaborée en conséquence, le test bench a pu nous faire découvrir les failles restantes dans notre design. Avec les latences choisies (env 10 cycles d'horloge par data de la ram) il est impressionnant de voir la différence de temps entre un cache miss et un hit. Lorsque le contrôleur doit aller chercher la ligne en mémoire il lui fait plus de 160 cycles (16 mots * 10cycles plus tous les cycles de contrôle) donc environ 200 cycles, et après une fois que la ligne est chargée, tous les accès (en lecture comme en écriture) se font en quelques cycles (<10). Sur le chronogramme on voit réellement la différence. Les attentes sur la mémoire sont bien lent ! D'où l'avantage énorme d'avoir une mémoire cache.
Ce laboratoire nous a permis de mieux comprendre le fonctionnement (et surtout l'implémentation) d'une mémoire cache et de bien se rendre compte encore une fois à quel point elle est utile et le gain énorme de temps qu'elle fournit. Le test bench nous donne confiance en notre design et je suis sur que si le laboratoire avait été à rendre sans faire de banc test que notre mémoire cache aurait bien des soucis dans de nombreux cas.