Notes Computer Architecture (midterm)

Tricks

Multiplication avec shift et log

Quand on veut calculer $a \cdot b$, si on dispose du $\log_a$ on peut faire $b \ll \log_a$.

Charger un mot au milieu

Toutes les adresses doivent être alignées sur la taille des données qu'elles manipulent (donc quand on fait un `lw` ça doit être un multiple de 4). Mais RISC-V va modifier le code pour tomber sur le mot juste avant si on utilise un `lw` qui n'est pas un multiple de 4. `lw_ecrit = 4k + r`, donc `lw_corrige = 4k`.

Charger un immediate avec 32 bits

`li` supporte le chargement d'un immediate de 32 bits (tandis que toutes les autres opérations comme `addi`, la constante devant `lw`, etc.) ne supportent que des 12 bits.

donc si on veut faire un AND avec 12 bits :

```
# on ne doit pas faire ça :
andi t0, t0, 0xFFF # sera -1 !
# mais ça :
li t1, 0xFFF # sera 111..11 pendant 12 bits
and t0, t0, t1
```

car ce qui compte quand on veut faire un `andi`, `xori`, etc. c'est la valeur du 12ème bit pour savoir si on sign-extend ou pas.

donc `0x3` va donner `000000...011`.

Convention d'appel

Même si on sait que la fonction `b` que notre fonction `a` va appeler ne modifie pas les saved registers, la fonction appelante doit *TOUJOURS* utiliser et sauvegarder les saved registers :

```
# mettre à jour le stack pointer (on le fait remonter dans la mémoire)
addi sp, sp, -12
sw ra, 0(sp)
sw s1 4(sp)
sw s2, 8(sp) # 8... + 4 = 12 bytes libérés utilisés
```

```
# blabla

lw ra, 0(sp)
lw s1, 4(sp)
lw s2, 8(sp)
addi sp, sp, 12
```

Résumé interrupt handler

Attention, écrire avec le register `zero` ou l'immediate `0` dans le interrupt handler ne va jamais modifier les valeurs des CSR-s !

```
interrupt_handler:
# fait par le CPU automatiquement :
# MPIE = MIE
# MIE = 0

# nous on lit le MIP pour savoir qui est pending
# là on doit (nous) save les registers
# attention à bien sauvegarder le mepc et mstatus
# (en cas de nested interrupt, on sauvegarde le `mepc` dans `t0`, puis
`t0` dans le stack)

# avant d'appeler les service routines
# on doit pop les registers modifiés
# (entre le début de l'interrupt_handler) du stack

# là on doit (nous) gérer les service routines
# attention les services routines doivent parfois acknowledge les
interrupts!
# et/ou modifier le MIP

# avant on disable le MIE si on l'a remis
# là on doit (nous) restaurer les registers

# parfois on doit faire address + 4
# quand l'exception vient d'une faulty exception

mret
# fait par le CPU :
# MIE = MPIE
# PC = MEPC
```

Résumé service routine

- toujours sauver les registers si pas de handler précisé dans l'énoncé.

Résumé caches

- on veut à la fois pouvoir mettre des keys partout dans notre cache (**fully associative**), sans avoir une place assignée déjà mathématiquement via l'adresse → l'objectif est ainsi d'éviter les conflits.
- mais en même temps on veut avoir une place assignée déjà mathématiquement pour chaque key, car ça évite de devoir chercher où elle est (l'intérêt du **direct-mapped**).

→ on peut mélanger les deux (**k-way set associative**), avoir une place dédiée pour un set (le hash des 4 derniers bits par exemple) puis le remplir comme on veut (on minimise la recherche mais il y en a un peu au sein du bloc).

k-way n'a pas absolument besoin d'être une puissance de 2 mais si ce n'est pas le cas, la taille totale du cache ne sera pas une puissance de 2... donc c'est pas fou.

de la même façon, le nombre de lignes du cache sera toujours un power de 2, parce qu'on utilise une fonction de hachage qui part d'un nombre n de *bits*.

Où placer mon adresse ?

Dans l'énoncé :

- **block size**, ou **line size**, c'est combien d'adresses il y a dans chaque block/ligne
- **number of lines**, combien de lignes il y a dans chaque way/cache direct mapped
- **k-way**, combien de caches différents il y a en simultanés à côté (par exemple 4-way ça veut dire 4 caches direct mapped l'un à côté de l'autre, donc 4 emplacements possibles pour un même block). Quand on a un k-way cache et qu'on nous donne la taille d'une ligne dans chaquw way, la taille totale associée à une ligne est k fois la taille donnée.

```
Nombre de lignes = (nombre de bytes total du cache)/(block_size * k)
```

Par exemple on veut placer 20, on fait une division entière par $2^{\text{block\_size}}$, puis un modulo du nombre de lignes.

Pourquoi ? On stocke par *blocks*. Donc si on a 16, 17, 18, 19, on veut effacer les block_size nombre de bits (donc on divise par $2^{\text{block\_size}}$), puis on fait un mapping de ce qui reste vers le nombre de lignes.

Comme ça 16, 17, 18, 19 sont mappés au même block.

Endianness

Little endian : le moins significatif est stocké en premier. Exemple pour la valeur :
`0x12345678`
`0x100` stocke `0x78`
`0x101` stocke `0x56`
etc.

Paging

Chaque programme a plusieurs pages 0, 1, 2, 3, ... stockées un peu n'importe où dans la mémoire (RAM *et* disque). Une page table permet de nous dire où se trouve chaque page dans la mémoire.

Adresse virtuelle vers adresse physique ?

Si on veut convertir une adresse virtuelle en adresse physique, il faut trouver la page. Pour ça on fait :

```
numero_page = adresse_virtuelle / taille_page
```

(ça revient à enlever $\log\left(\text{taille\_page}\right)$ bits de l'adresse virtuelle).

Ensuite on va voir dans la **Page Table** l'adresse de la page :

```
physical_base_address = page_table_base_address + page_table_entry_size *
numero_page
```

Et on ajoute `adresse_virtuelle mod taille_page` (l'*offset*).

```
physical_address = physical_base_address + offset
```

Page table entry size, c'est la taille que prend le fait de stocker une virtual address to physical address dans la mémoire. Cela peut inclure des informations additionnelles.

Il y a donc plus de pages virtuelles que de pages physiques (ça compte le disque + la RAM).

Conversions

$2^{30}\ \text{bytes} = 1\ \text{GBytes}$
$2^{20}\ \text{bytes} = 1\ \text{MBytes}$
$2^{10}\ \text{bytes} = 1\ \text{KBytes}$

On a $2^{32} \cdot\ 4\ \text{bytes} = 2^{30} \cdot 4 \cdot 4\ \text{bytes} = 16\ \text{G bytes}$!