

## 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 RISCV 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 immédiat 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.

### 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

```
addi sp, sp, -12 # mettre à jour le stack pointer (on le fait remonter dans la
                   mémoire)
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
```

### Sign Extend

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

donc 0x3 va donner 000000...011

### Interrupt handler

Écrire avec le register `zero` ou l'immediate `0` dans le interrupt handler ne va jamais modifier les valeurs des CSRs.

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

    # là le CPU clear le MIP.
    # mais nous on lit le MIP.

    # là on doit (nous) save les registers
    # attention à bien sauvegarder le mepc et mstatus

    # 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

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

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

```

### Service routines

Toujours sauver les registers si pas de handler.

Quand on des nested interrupts, on sauvegarde le mepc dans t0, puis t0 dans le stack.

### Caches

On veut à la fois pouvoir mettre des keys partout dans notre cache (fully associative), sans avoir une place dédiée → objectif est d'éviter les conflits

mais en même temps on veut avoir une place dédiée pour chaque key → ça évite de devoir chercher où elle est (direct-mapped)

par contre 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 sera toujours un k de power of 2

pareil, la taille du cache sera toujours un power de 2, parce qu'on utilise une fonction de hashage qui part d'un nombre n de **bits**

### Trouver où placer

- block size, combien de words par tag
- k-way, combien de block par ligne
- une ligne est le résultat d'une fonction de hashage

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

20

### Endianess

#### Little endian

Le moins significatif est stocké en premier. Exemple pour la valeur : 0x12345678

0x100 stocke 0x78

0x101 stocke 0x56

etc.