

# Conception de systèmes embarqués temps réel

Dumitru Potop-Butucaru

[dumitru.potop@inria.fr](mailto:dumitru.potop@inria.fr)

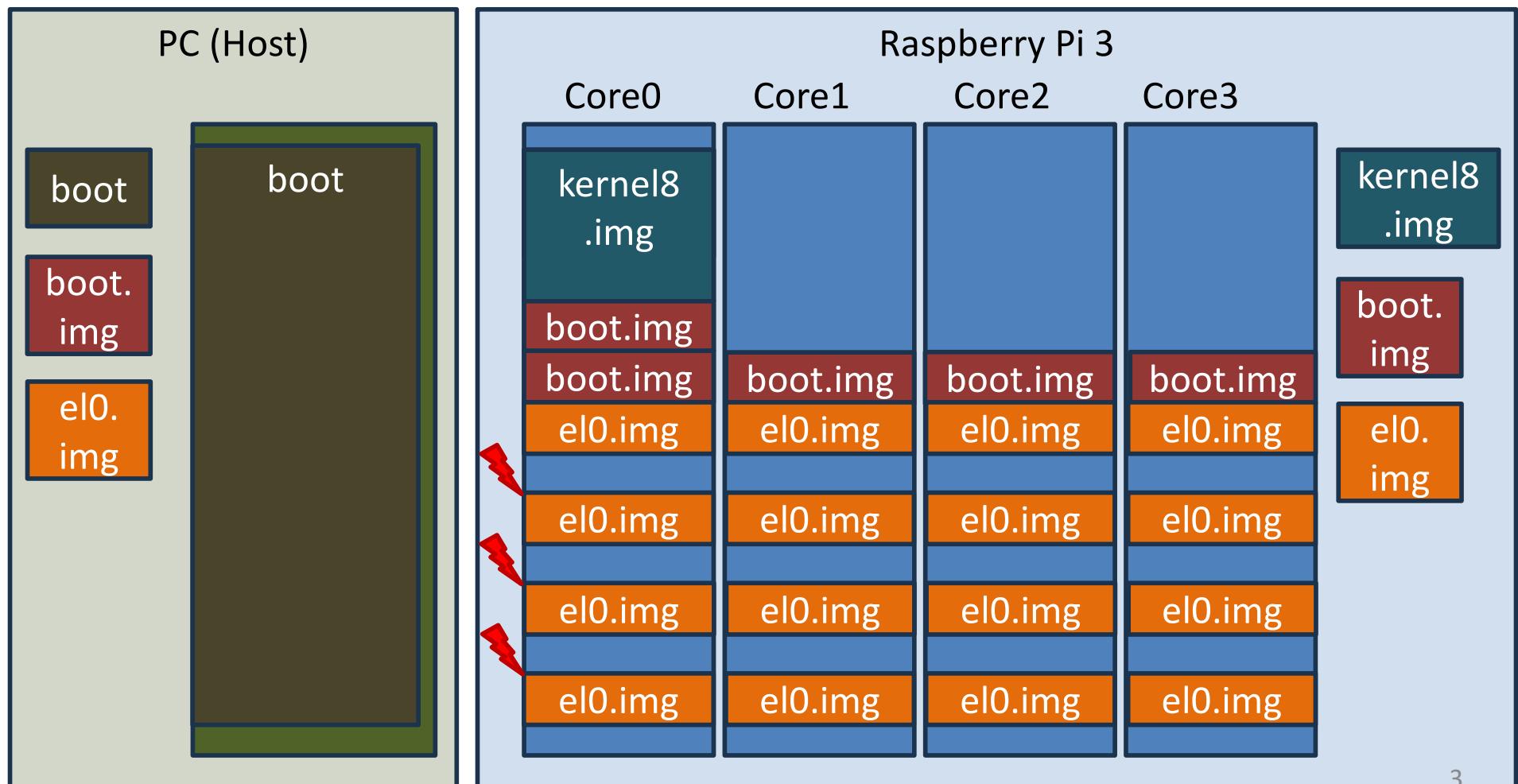
cours EIDD, 2025

# Contenu de ce cours

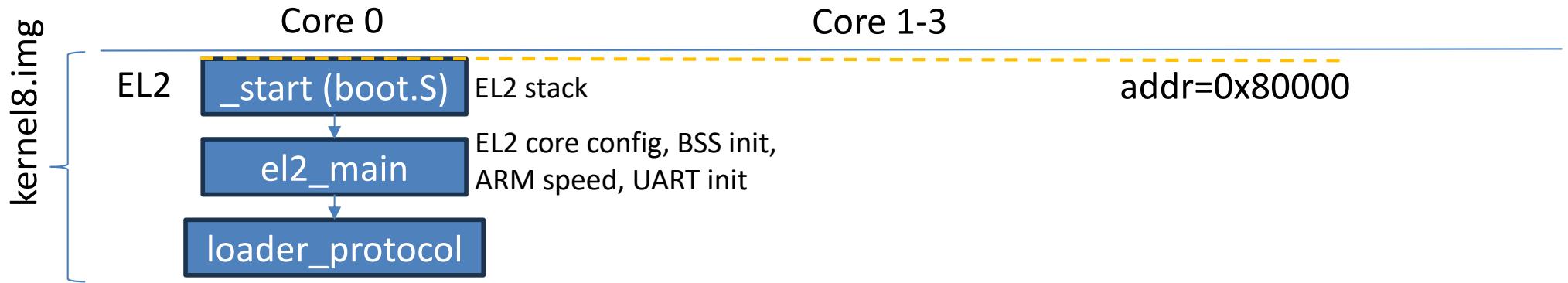
- Implantation multi-cœurs (2<sup>e</sup> partie)
  - Organisation d'un exécutif cyclique simple
  - Interruptions, niveaux de privilèges
  - Synchronisation
    - Avec le temps réel (timers)
    - Entre cœurs
      - Sémaphores
      - Accès mémoire à effet de barrière
- Préparation du TP

# Exécution multi-cœurs

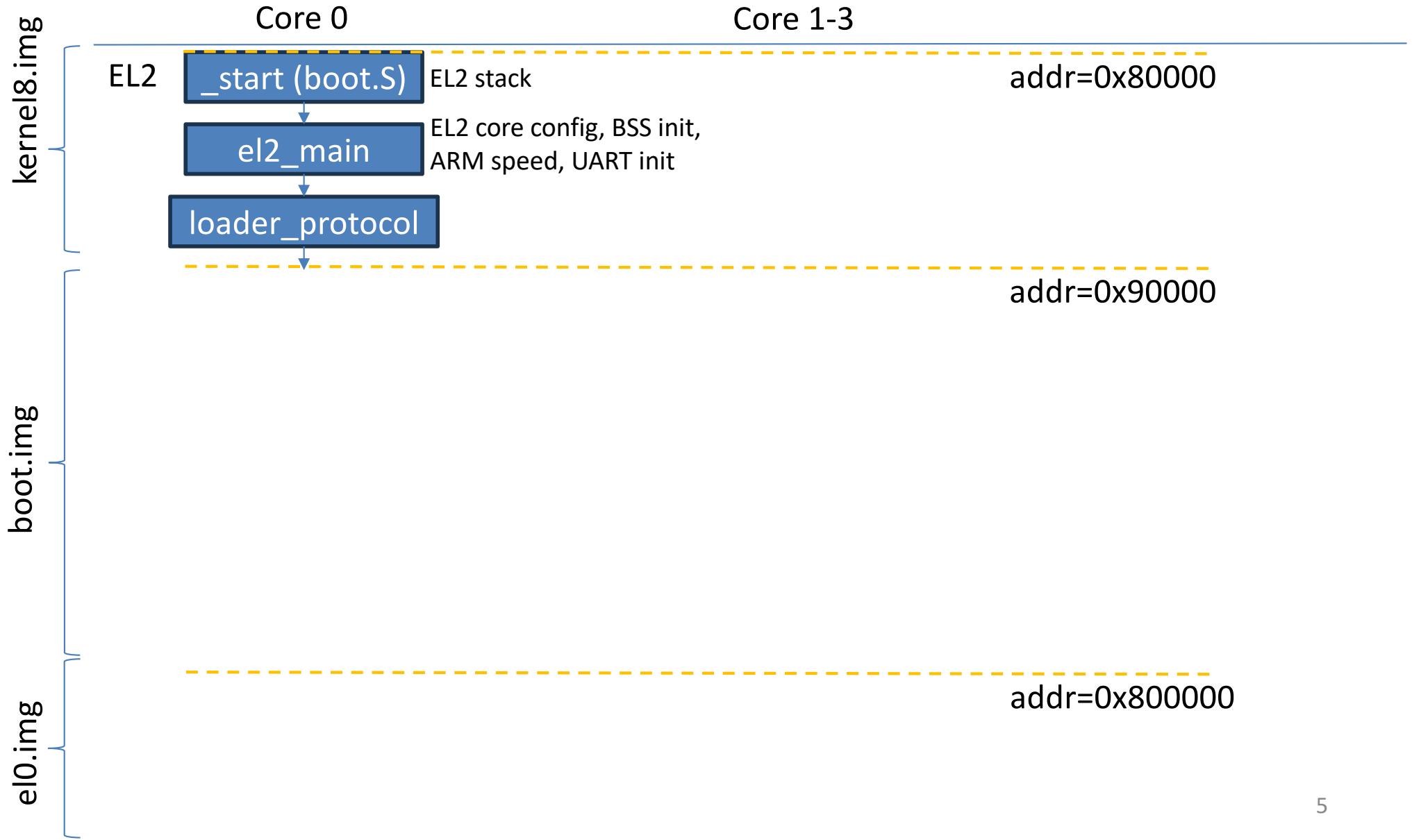
- Déclenchement timer (période = 1s)



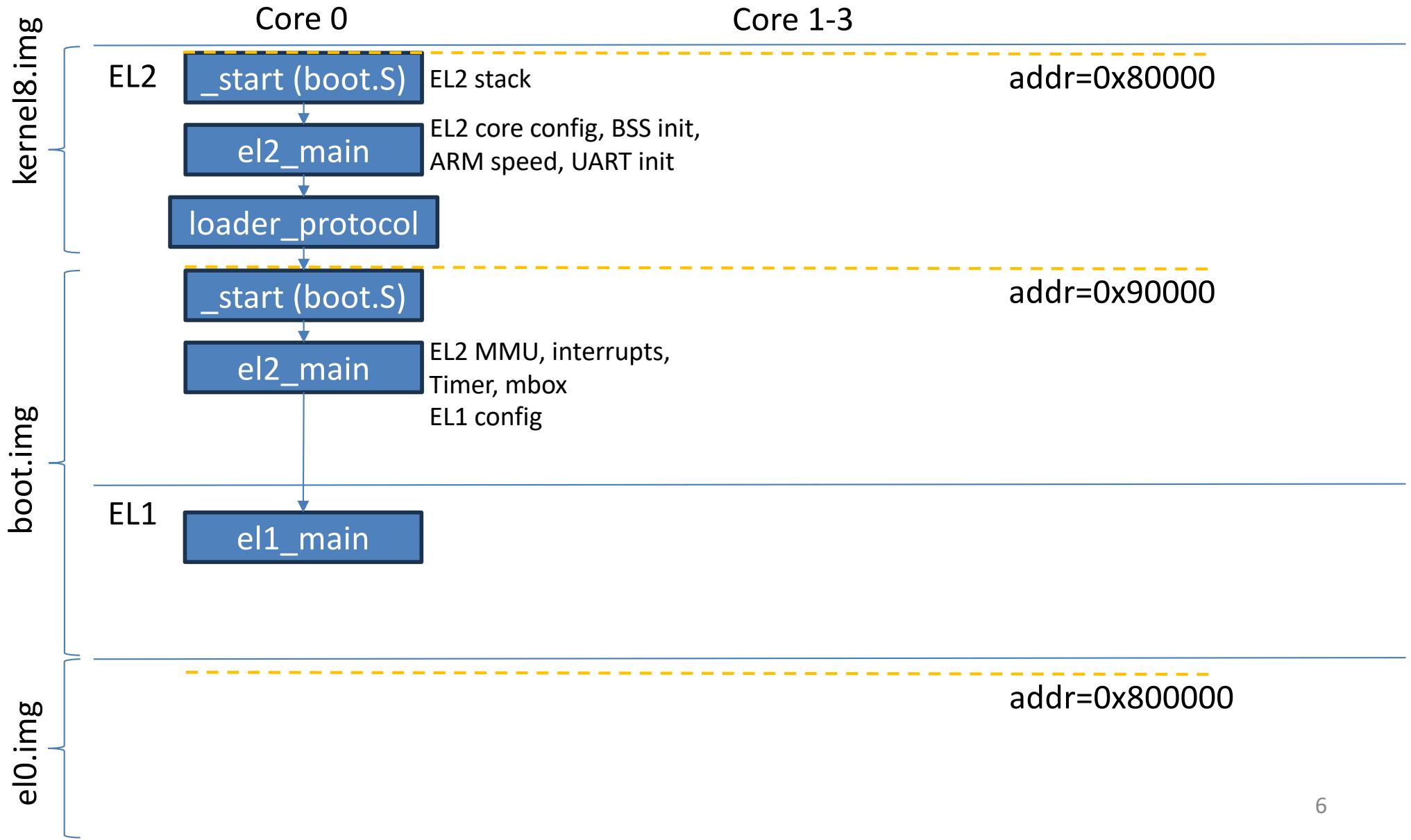
# Démarrage (détailé)



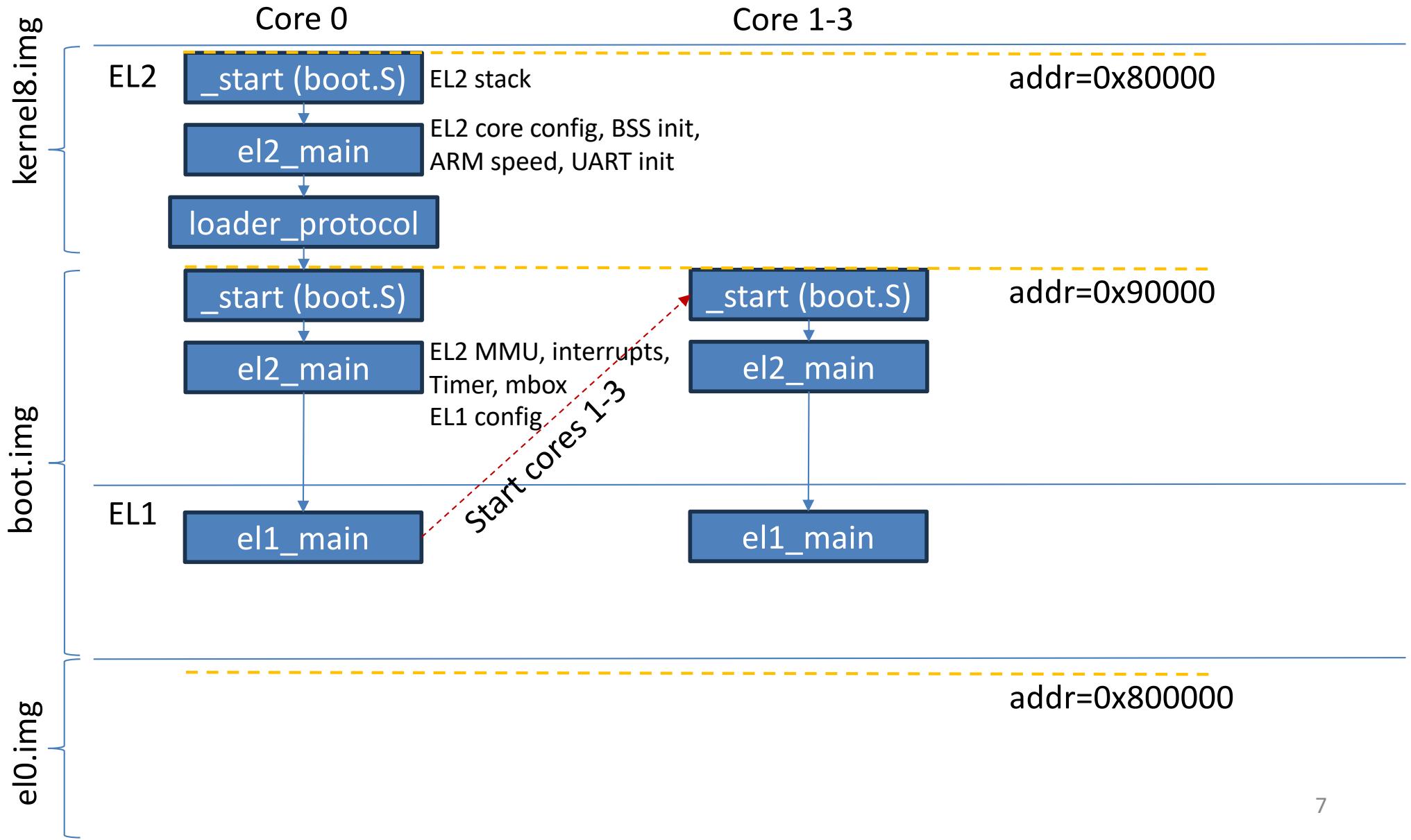
# Démarrage (détailé)



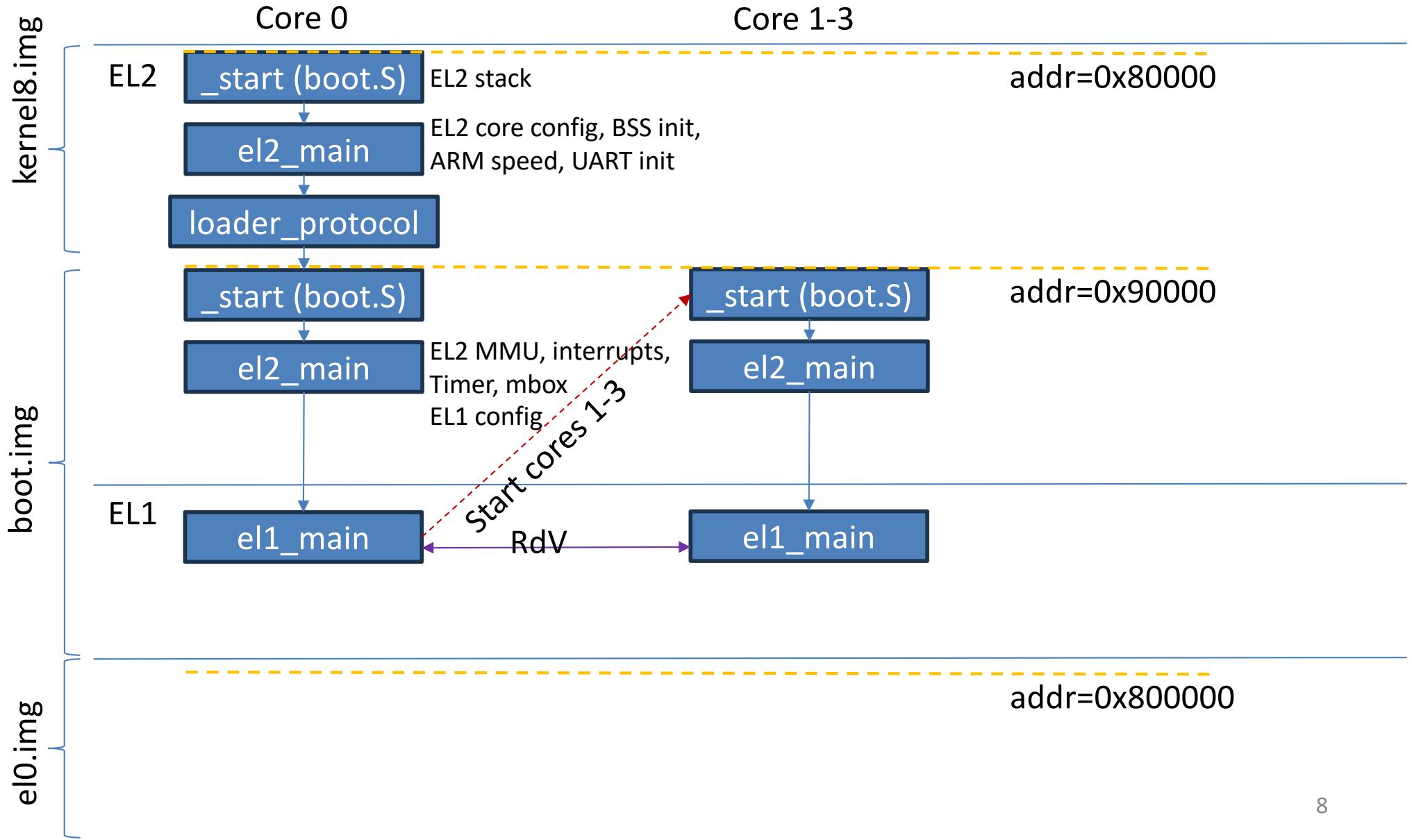
# Démarrage (détailé)



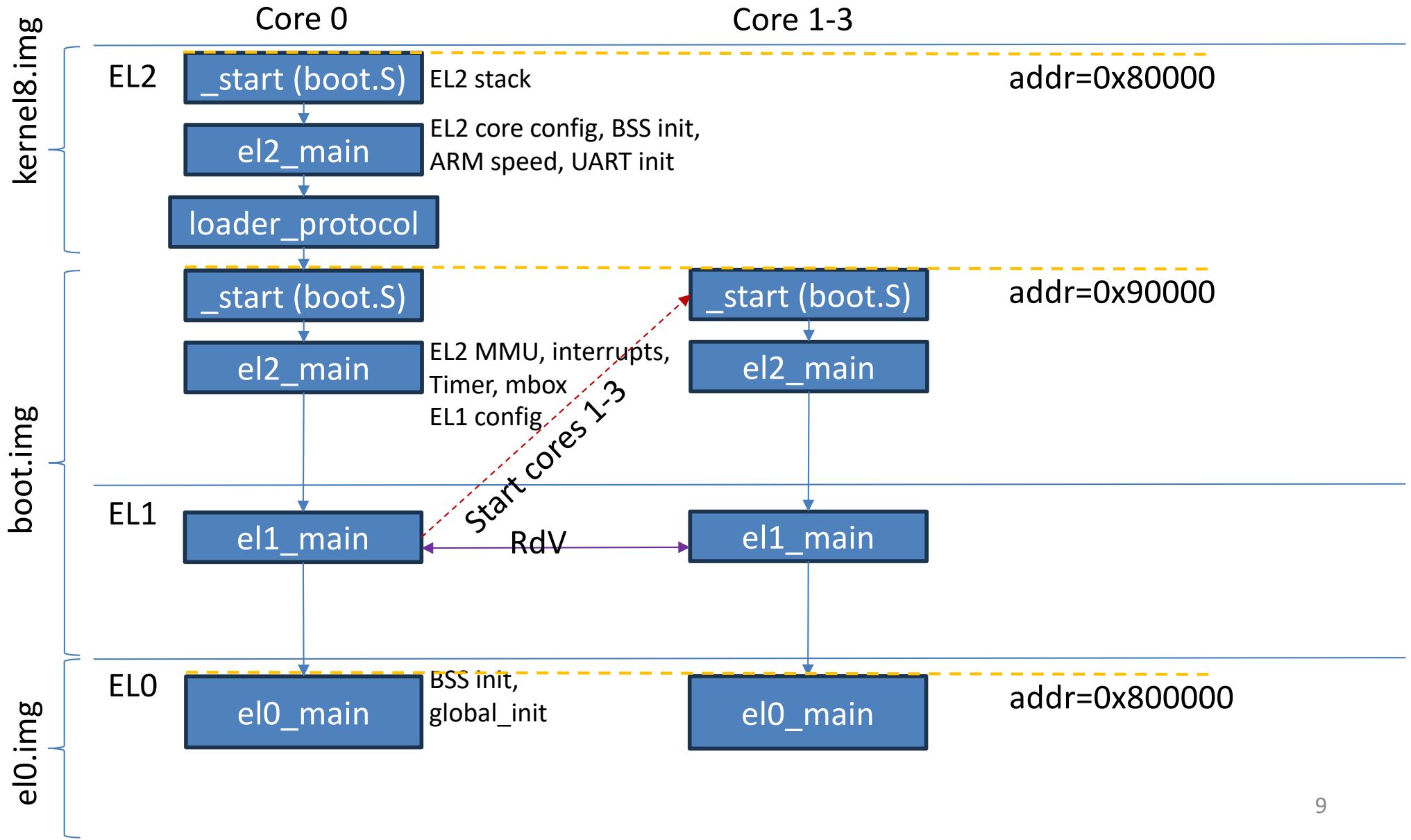
# Démarrage (détailé)



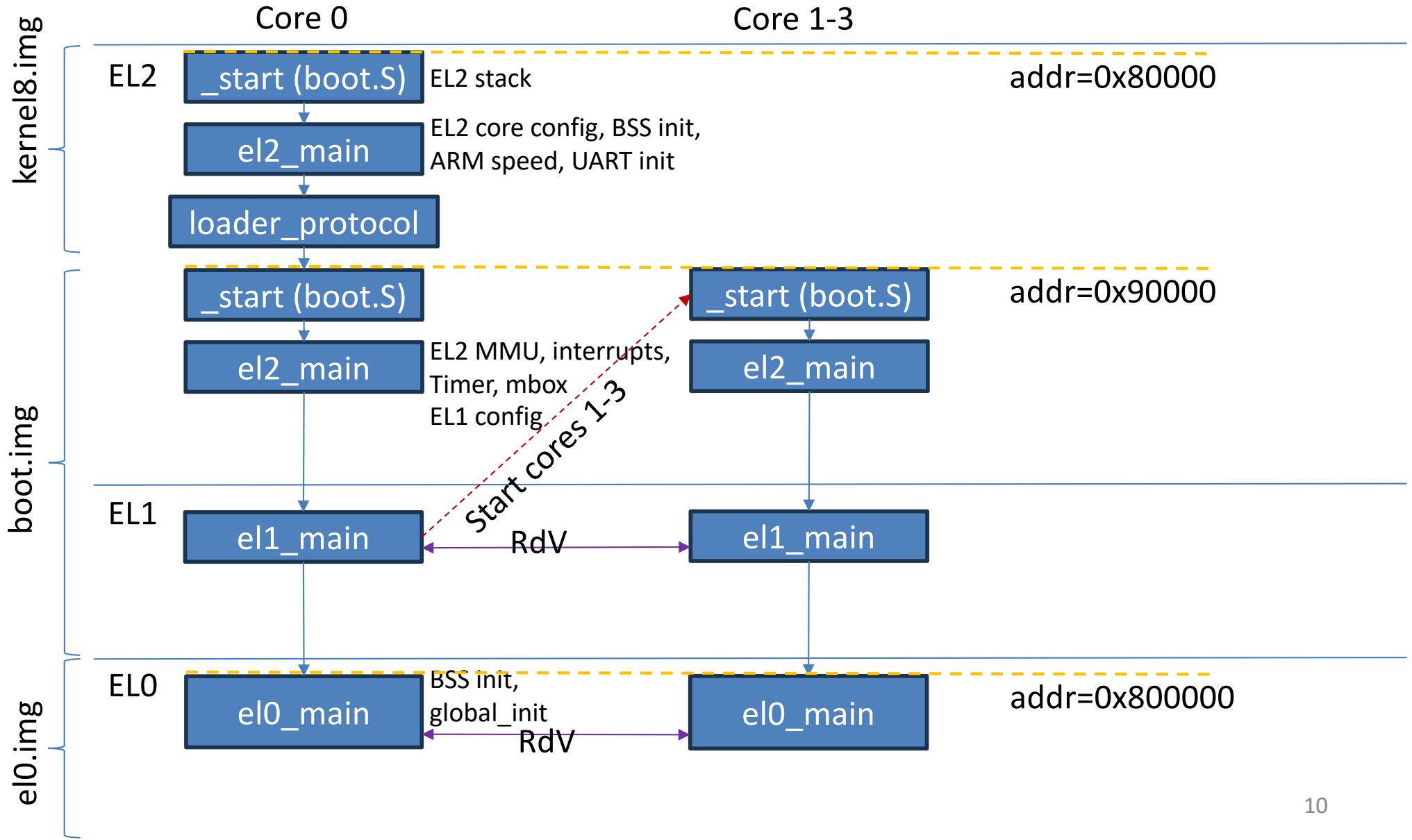
# Démarrage (détailé)



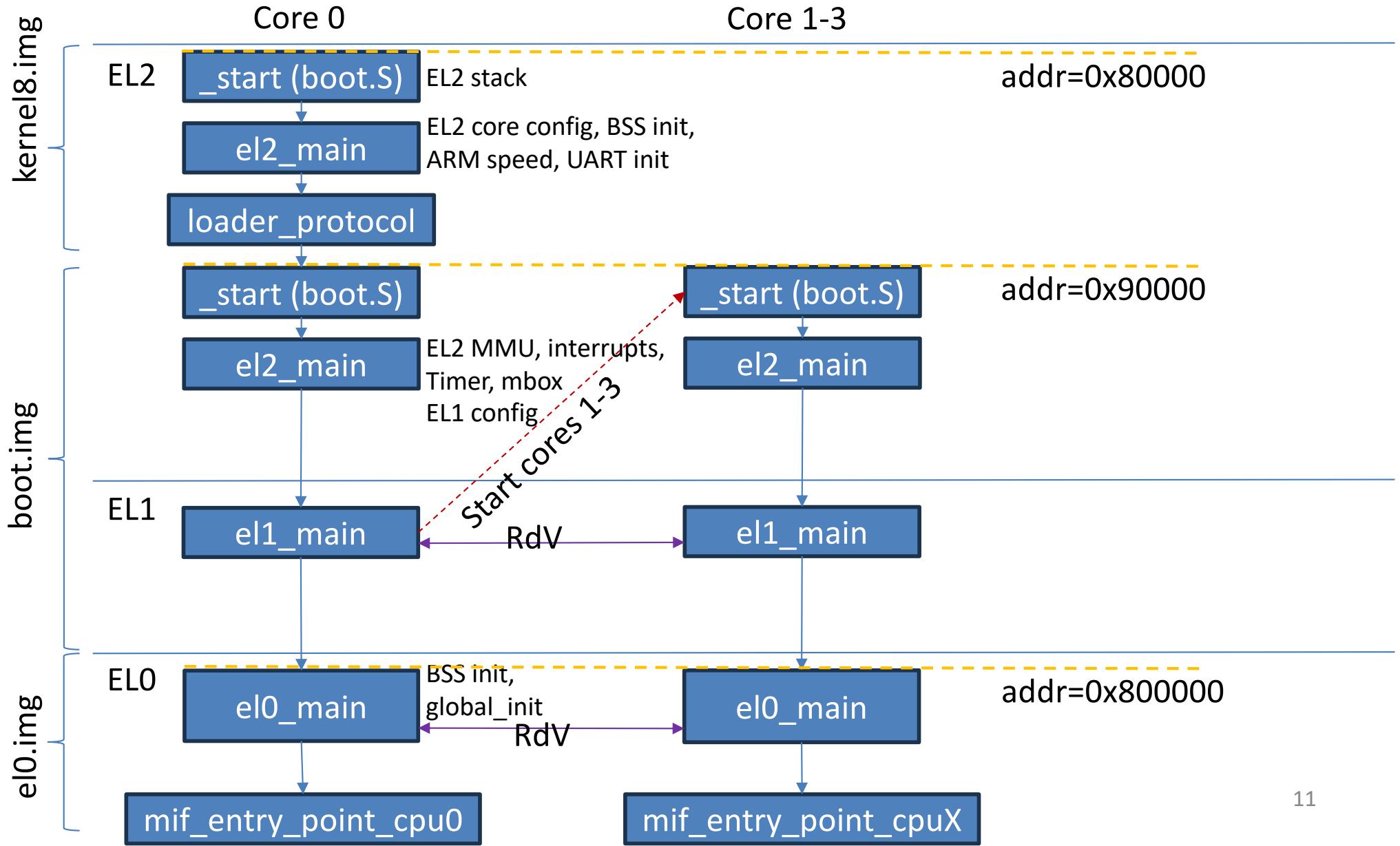
# Démarrage (détailé)



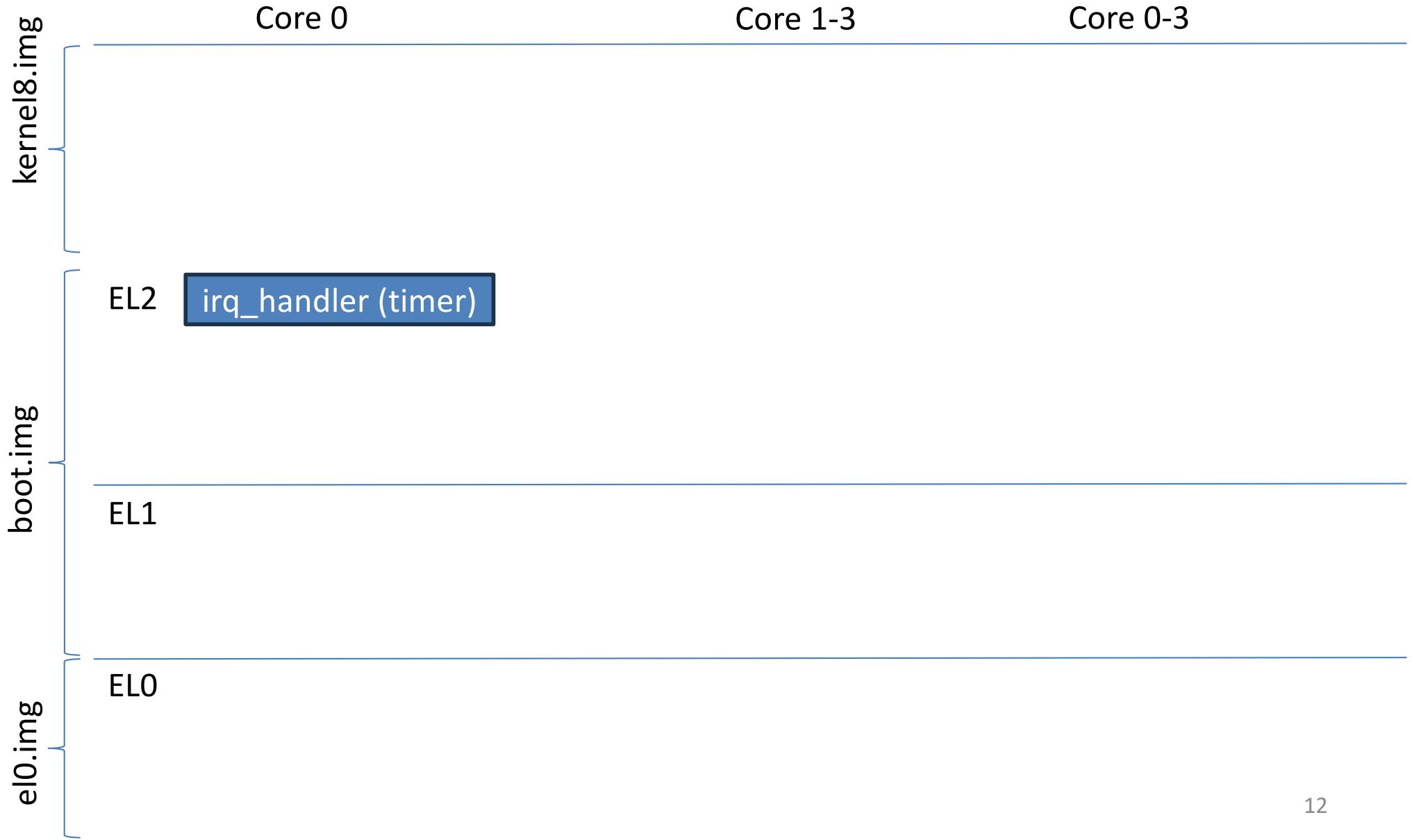
# Démarrage (détailé)



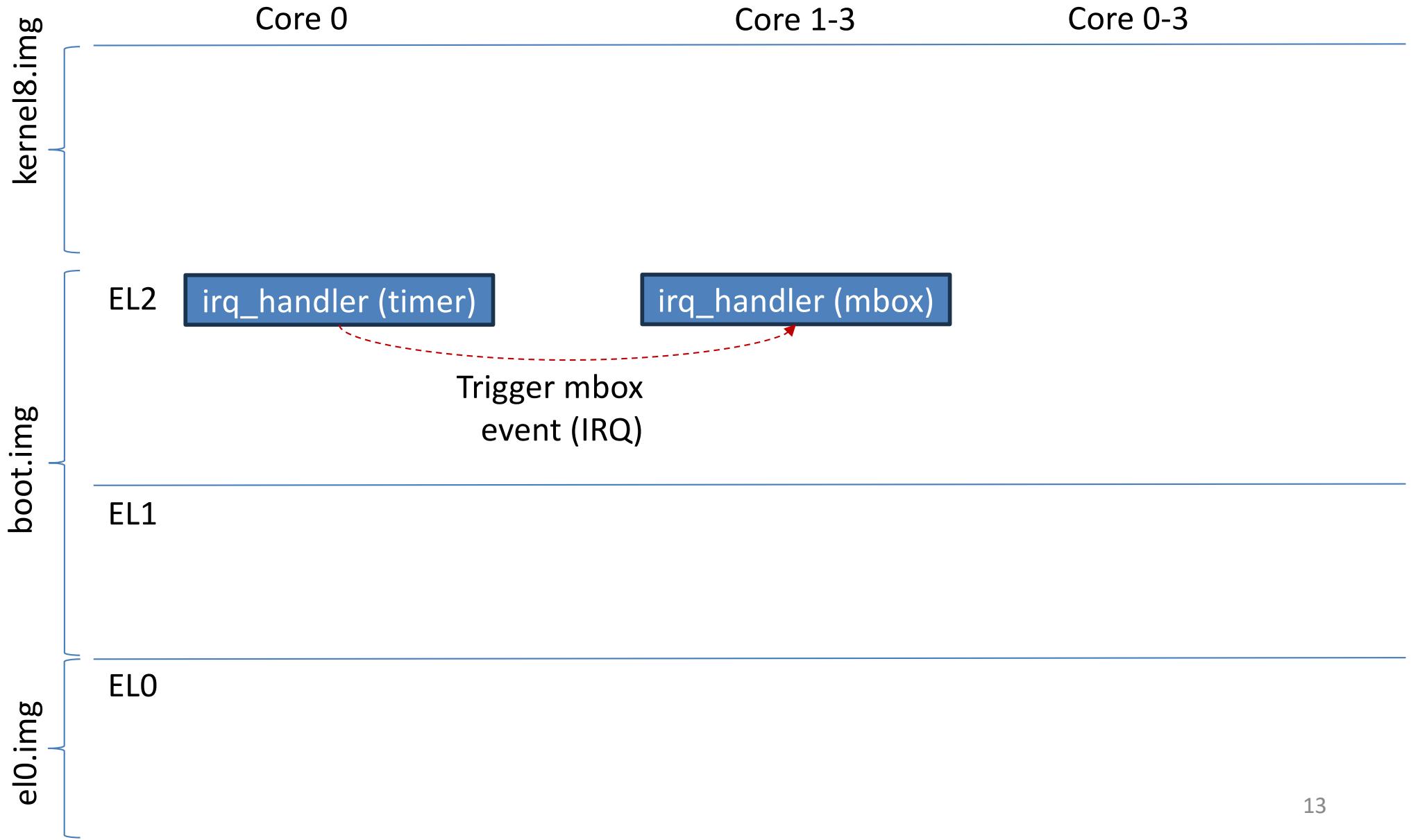
# Démarrage (détailé)



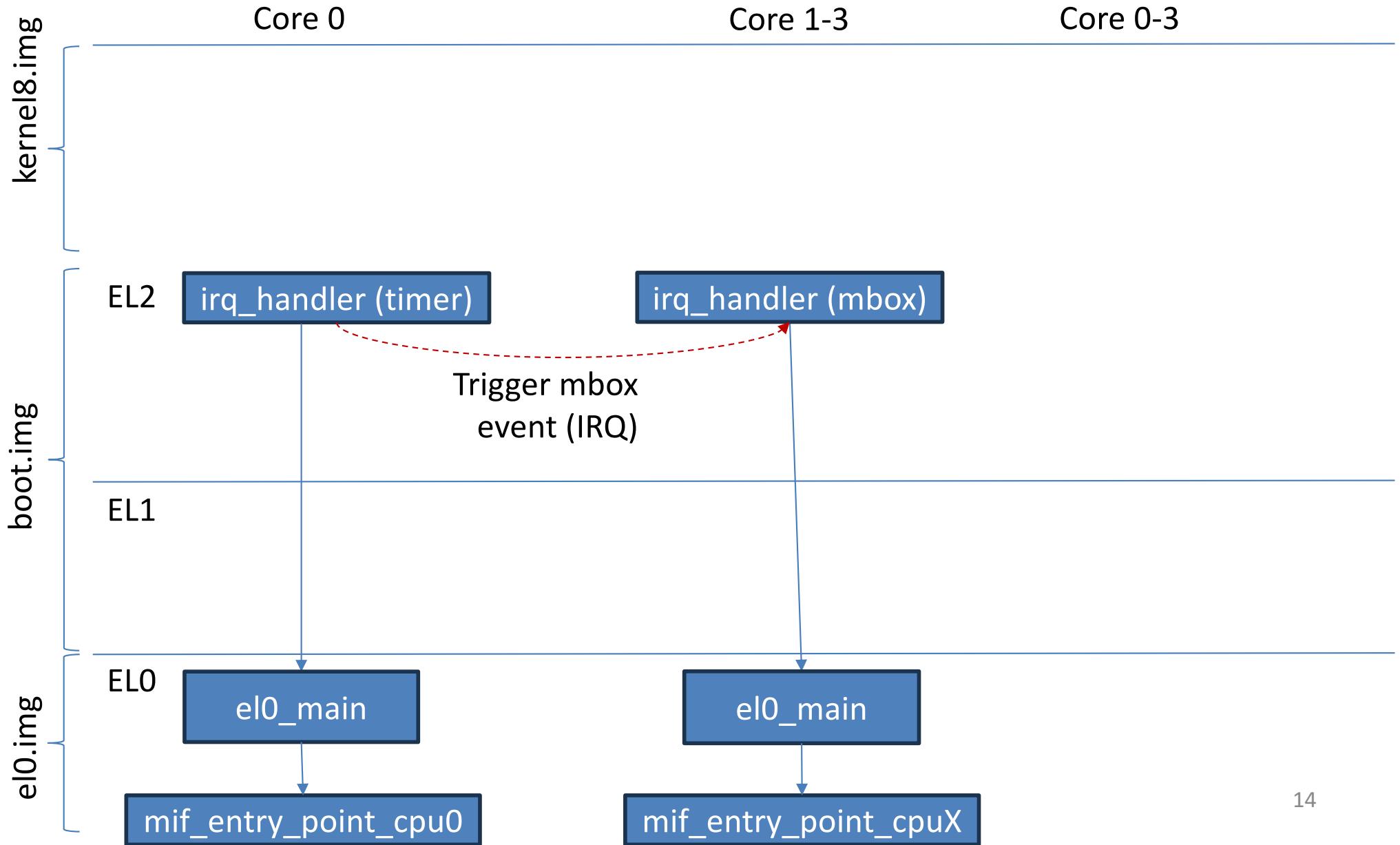
# Chaque cycle après démarrage



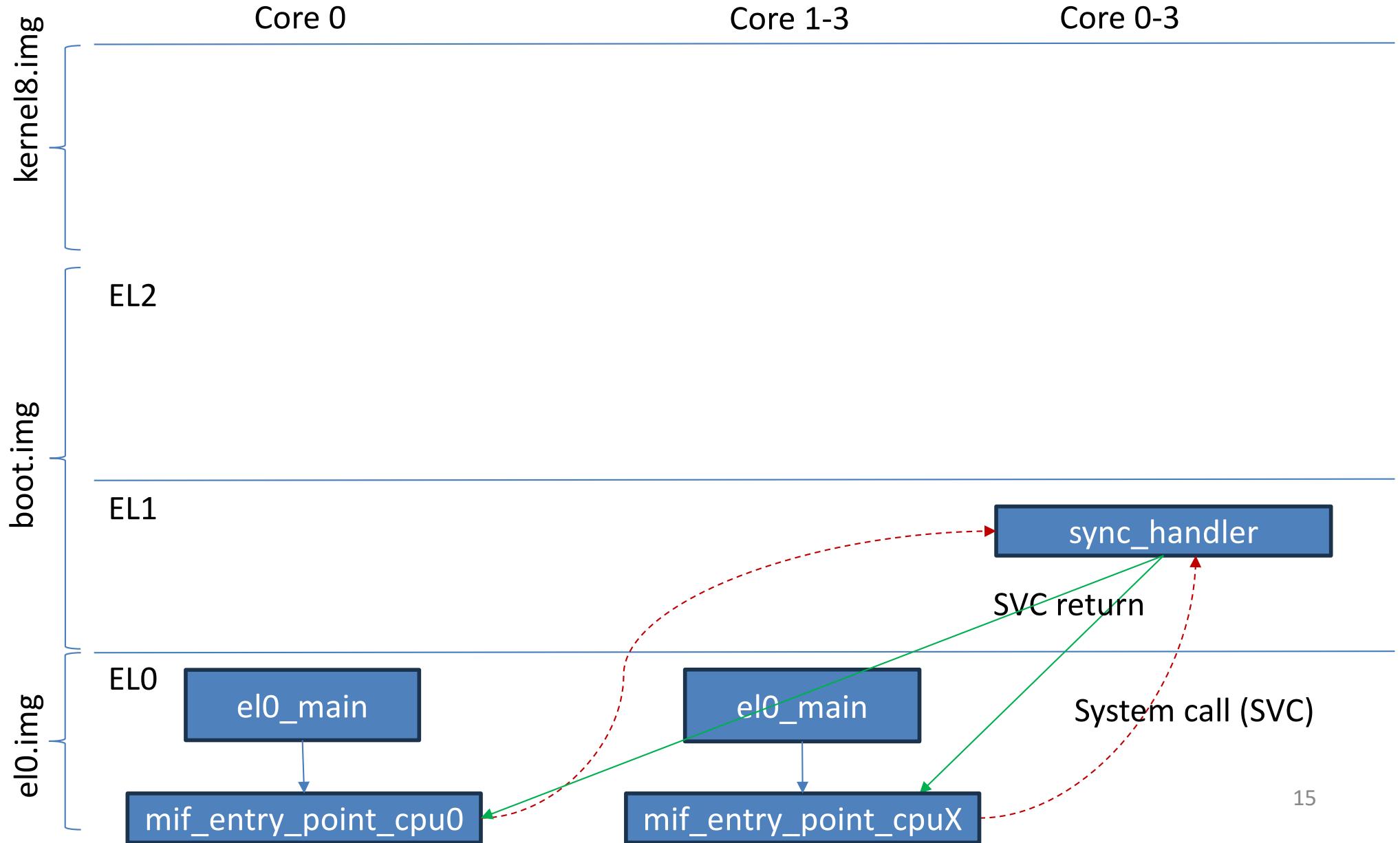
# Chaque cycle après démarrage



# Chaque cycle après démarrage



# Services système (à chaque cycle)



# Configuration matérielle

- MMU et caches activés
  - Cohérence de caches
  - Nécessaire pour performance et synchronisation
- Timers
  - IRQ sur cœur 0, EL2
- Mailboxes (synchronisation entre coeurs)
  - IRQ sur coeurs 1-3, EL2
  - Synchronisation très précise
    - Inversions entre f et g durant l'exécution
- SVC = services système (interruptions synchrones)

# Synchronisation

## 1. Avec le temps réel : timers

- Utilisation 1 : Compteurs qui peuvent être lus
  - E.g. fonction get\_cpuid (de svc.h)
- Utilisation 2 : Alarmes configurables
- Plusieurs timers disponibles : (SoC timer, multicore timer, local (ARM core) timers, GPU timer, USB timer...)
  - Le code fourni utilise le timer ARM (« local timer »)
    - Base à 38.4MHz (1s = 38400000 impulsions, cf. lignes 65-69 de arm-config.c)
    - Dérivé de la fréquence GPU et CPU, donc il faut fixer celles-ci (ajouts dans config.txt sur la carte SD)
    - <https://datasheets.raspberrypi.com/bcm2836/bcm2836-peripherals.pdf>
  - Routage vers Core 0 (ligne 74 de arm-config.c)
  - Configuration par arm\_timer\_config\_el2() (arm-config.c)

# Synchronisation

## 2. Entre coeurs

### a. Mailboxes

- Nécessaires en l'absence de cohérence de caches
- Plutôt utilisés pour communiquer entre processeurs
  - e.g. multi-core ARM <-> GPU
- Peu portable sur autres ISA
- Ici, synchronisation entre coeurs
- Configuration par `arm_timer_config_el2()` (`arm-config.c`)

### b. Mémoire partagée

# Synchronisation en mémoire partagée

- Vision matérielle – plus portable que les mbox
  - Cohérence des caches requise
  - Plusieurs protocoles matériels – instructions spécifiques
    - Load/Store-exclusive (e.g. ldxr/stxr)
    - Load-acquire/store-release (e.g. ldar/stlr)
    - Barrières (e.g. dmb)
- Vision logicielle – très portable
  - Sémaphores binaires (mutexes)
  - Horloges vectorielles

# Sémaphores binaires

- Construction fondamentale en algorithmique parallèle et concurrente (cf. Edsger Dijkstra)
  - `pthread_mutex`
  - Permet l'implantation de sections critiques
- Un type, deux opérations
  - type `semaphore_t` à deux valeurs (0=disponible, 1=pris)
  - `get(s)`
    - Si le sémafor « s » vaut 0, une des opérations « `get(s)` » active va obtenir le sémafor, le faisant passer à 1. Les autres « `get(s)` » en attente restent en attente. Une fois lancé, « `get(s)` » se termine quand il obtient « s ».
  - `release(s)`
    - Met « s » à 0. Si « s » vaut 0, l'opération n'a aucun effet.
  - Initialement, tout sémafor vaut 0

# Sémaphores binaires

- Implémentation sous Aarch64 (semaphore.S)

```
// void semaphore_get(semaphore_t s)
semaphore_get:
    mov    w2, 1      // registre w2 := 1
.lockloop:
    // Lire en w1 la location mémoire d'adresse x0, et y marquer l'accès exclusif.
    ldaxr  w1, [x0]
    cbnz   w1, .lockloop // si la valeur initiale de [x0] n'est pas 0
    // Si la location [x0] est toujours en accès exclusif, y stocker 1.
    // Si succès, w3 vaudra 0.
    stxr   w3, w2, [x0]
    cbnz   w3, .lockloop // si stxr a échoué (un autre accès sépare ldaxr et stxr)
    ret

// void semaphore_release(semaphore_t s)
semaphore_release:
    st1rb     wzr, [x0] // store release 0
    ret

– Détails: https://developer.arm.com/documentation/den0024/a/Multi-core-processors/Multi-processing-systems/Synchronization
```

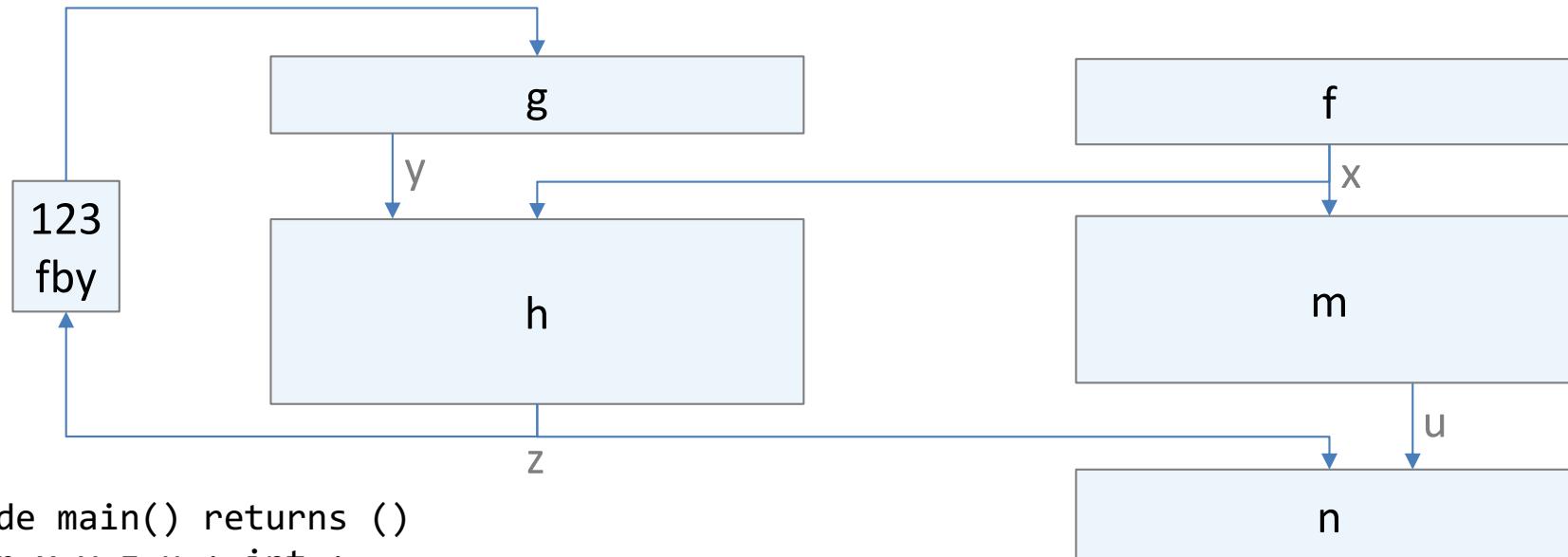
# Sémaphores binaires

- Permet l'implantation de sections critiques
  - Exemple: accès au terminal pour l'impression
    - Un sémaphore « console\_sem ». Pour imprimer un string « str », un cœur doit obtenir ce sémaphore

```
// svc.c, lignes 10, 18-39
semaphore_get(&console_sem);
minuart_puts(str) ;
semaphore_release(&console_sem) ;
```

# Horloges vectorielles

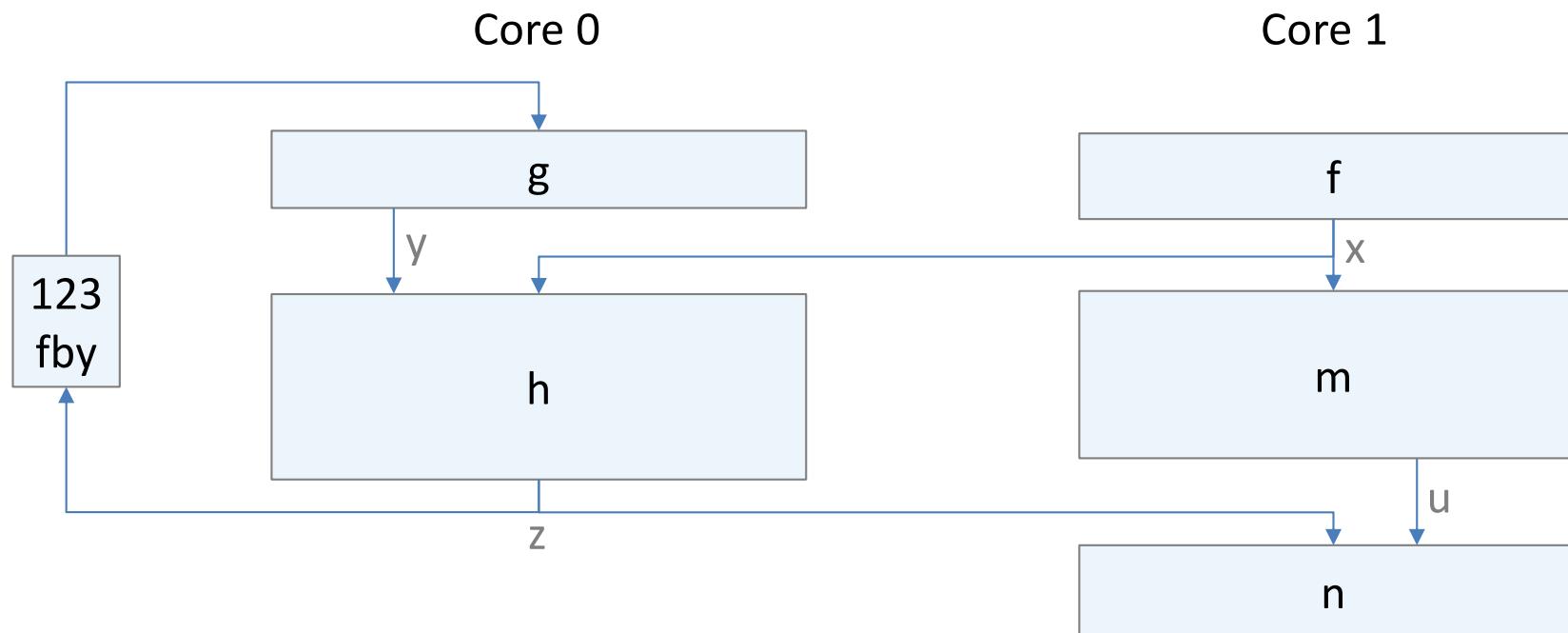
- Prenons un nouvel exemple de spécification fonctionnelle Heptagon



```
node main() returns ()  
var x,y,z,u : int ;  
let  
  y  = g(123 fby z) ;  
  x  = f() ;  
  z  = h(x,y) ;  
  () = n(z,u) ;  
tel
```

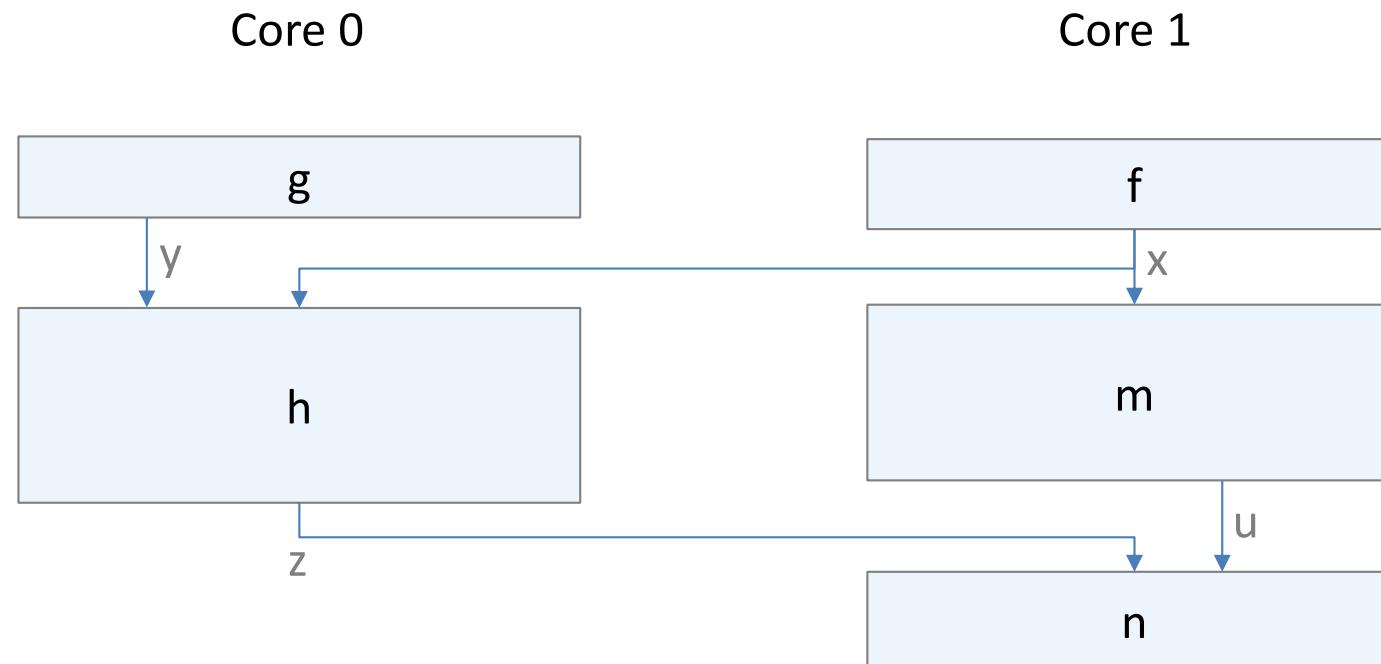
# Horloges vectorielles

- Allocation&ordonnancement sur 2 coeurs



# Horloges vectorielles

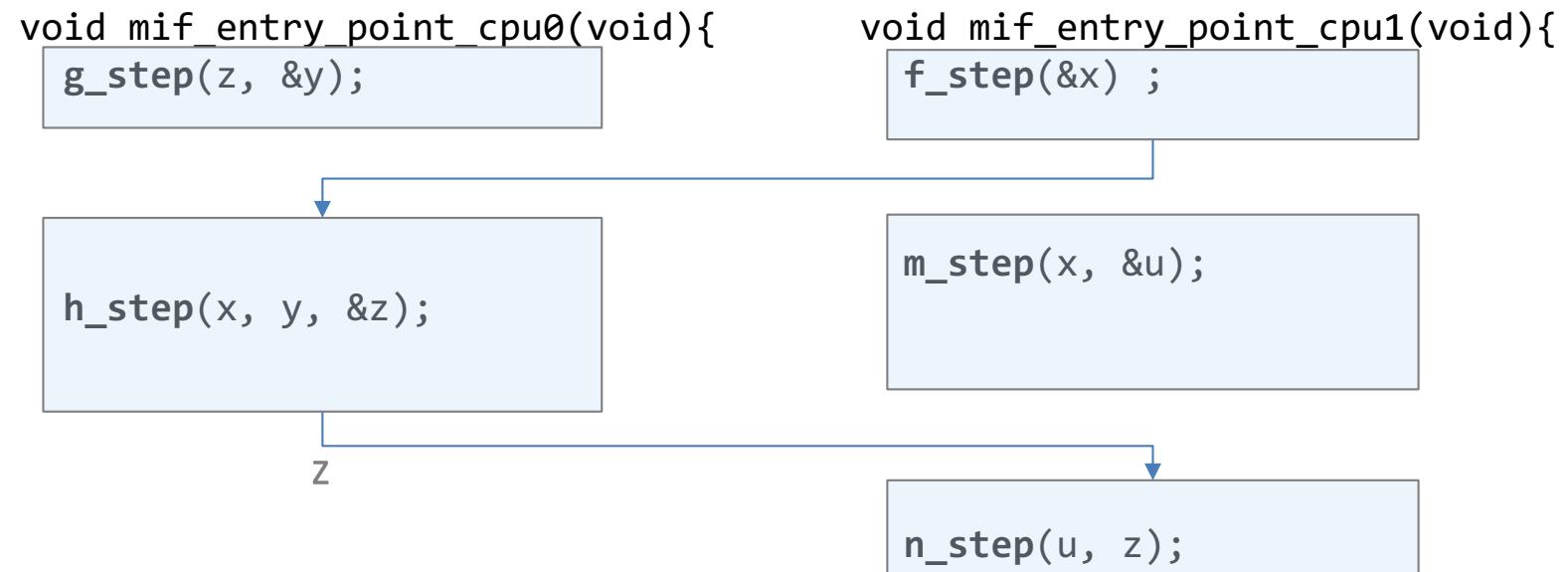
- Synthèse des variables de communication C



```
int x,y,u ;  
int z = 123;
```

# Horloges vectorielles

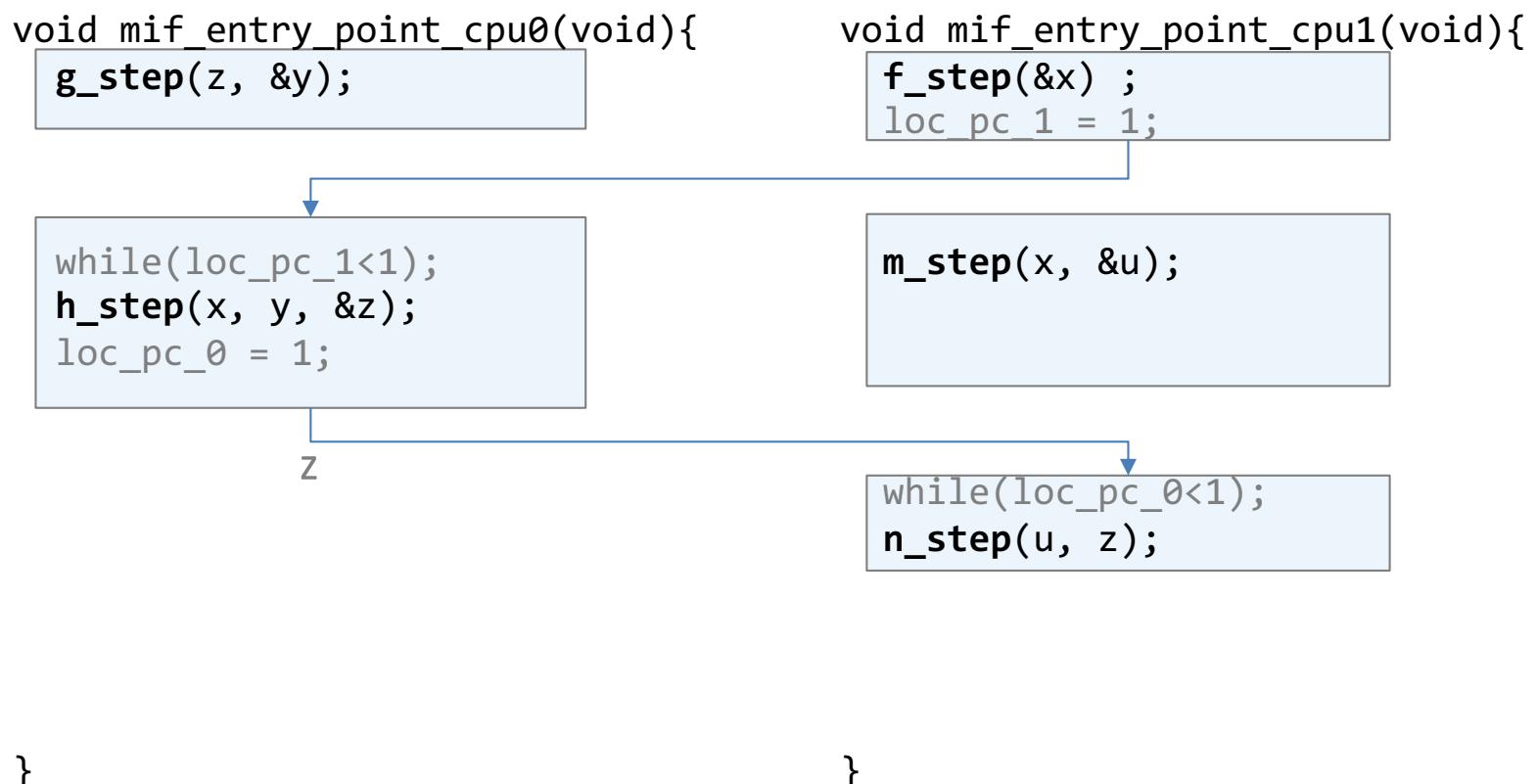
- Synthèse de code C (sauf synchronisation)



```
int x,y,u ; }  
int z = 123;
```

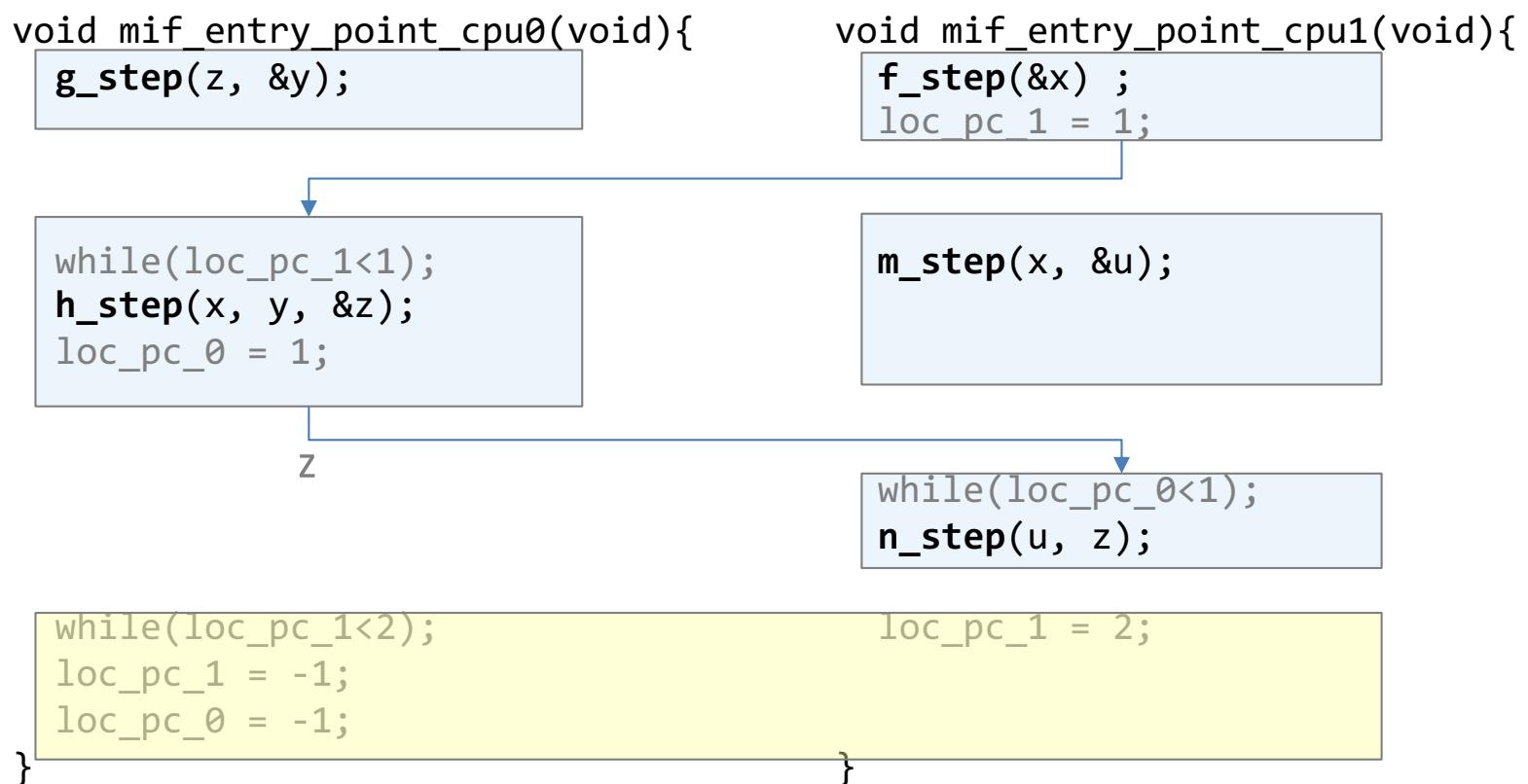
# Horloges vectorielles

- Synthèse de code C (synchro entre nœuds)
  - En début de cycle: loc\_pc\_X = -1



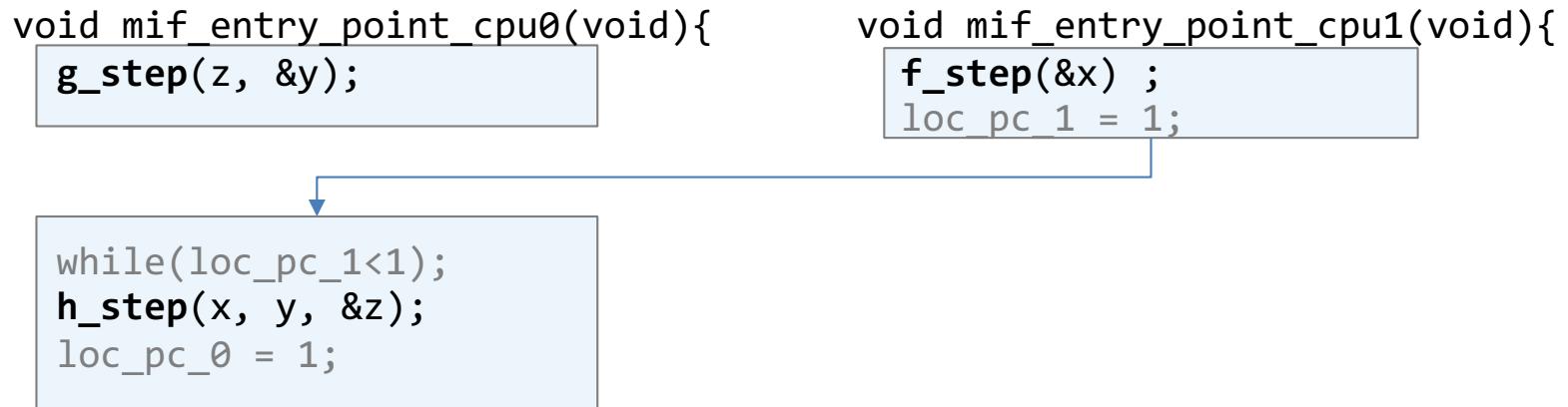
# Horloges vectorielles

- Synthèse de code C (remise à zéro)
  - En début de cycle: loc\_pc\_X = -1



# Horloges vectorielles

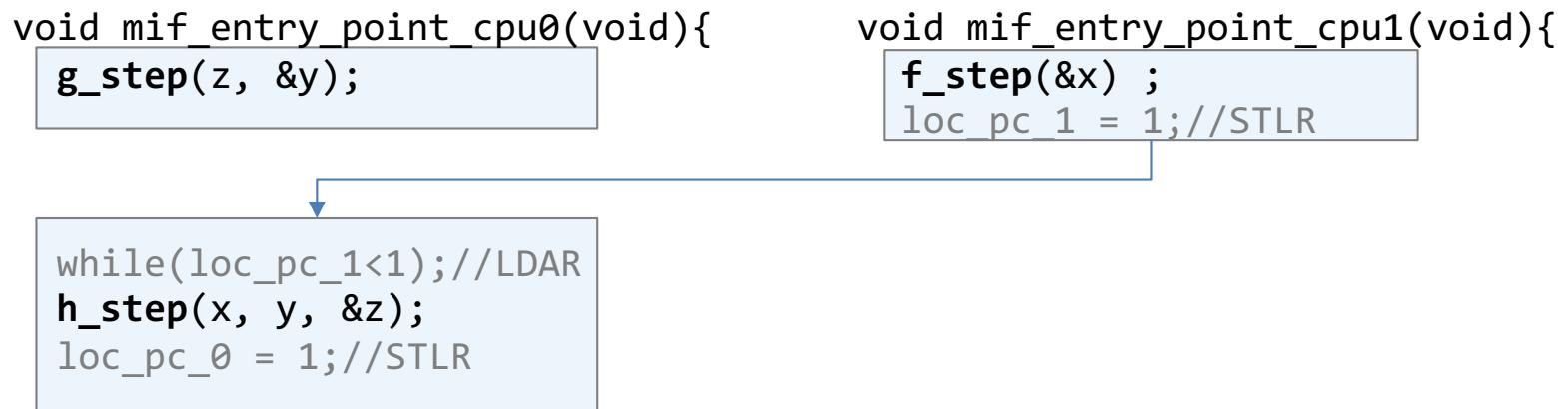
- Mais ce code est incorrect sur un CPU moderne



- Caches L1 séparés => `loc_pc_1` peut ne pas changer sur core 0 (si caches non-cohérents)
- Out-of-order execution => affectation de `loc_pc_1` exécutée avant `f` => `h` peut utiliser une ancienne valeur de `x`

# Horloges vectorielles

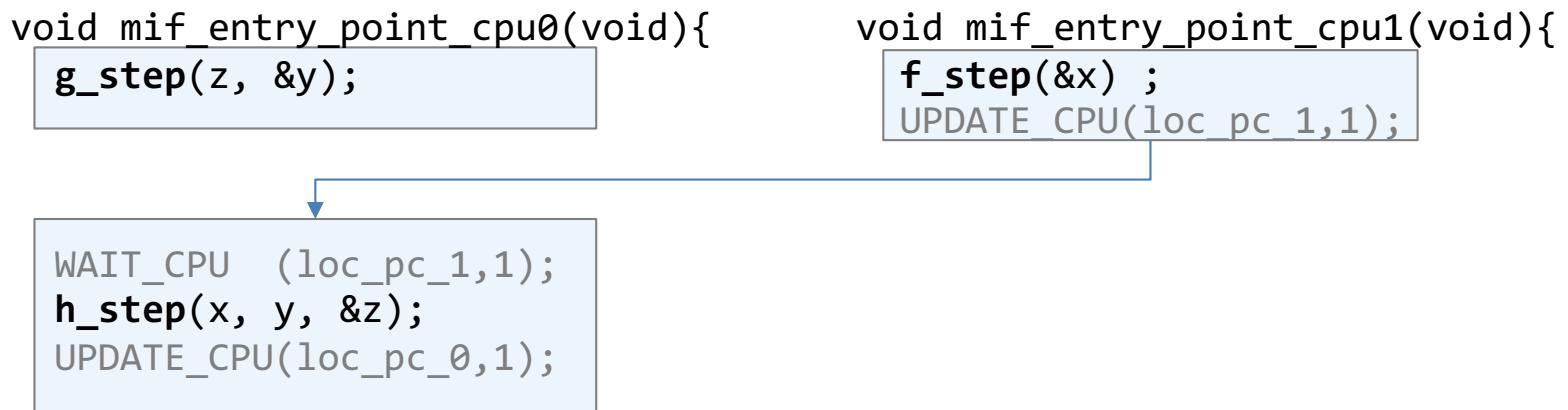
- AARCH64: accès mémoire à effet de barrière
  - Load-acquire (ldar), store-release (stlr)



- STLR: toutes les écritures le précédent dans le thread sont réalisées avant
- LDAR: toutes les lectures le suivant dans le thread sont réalisées après

# Horloges vectorielles

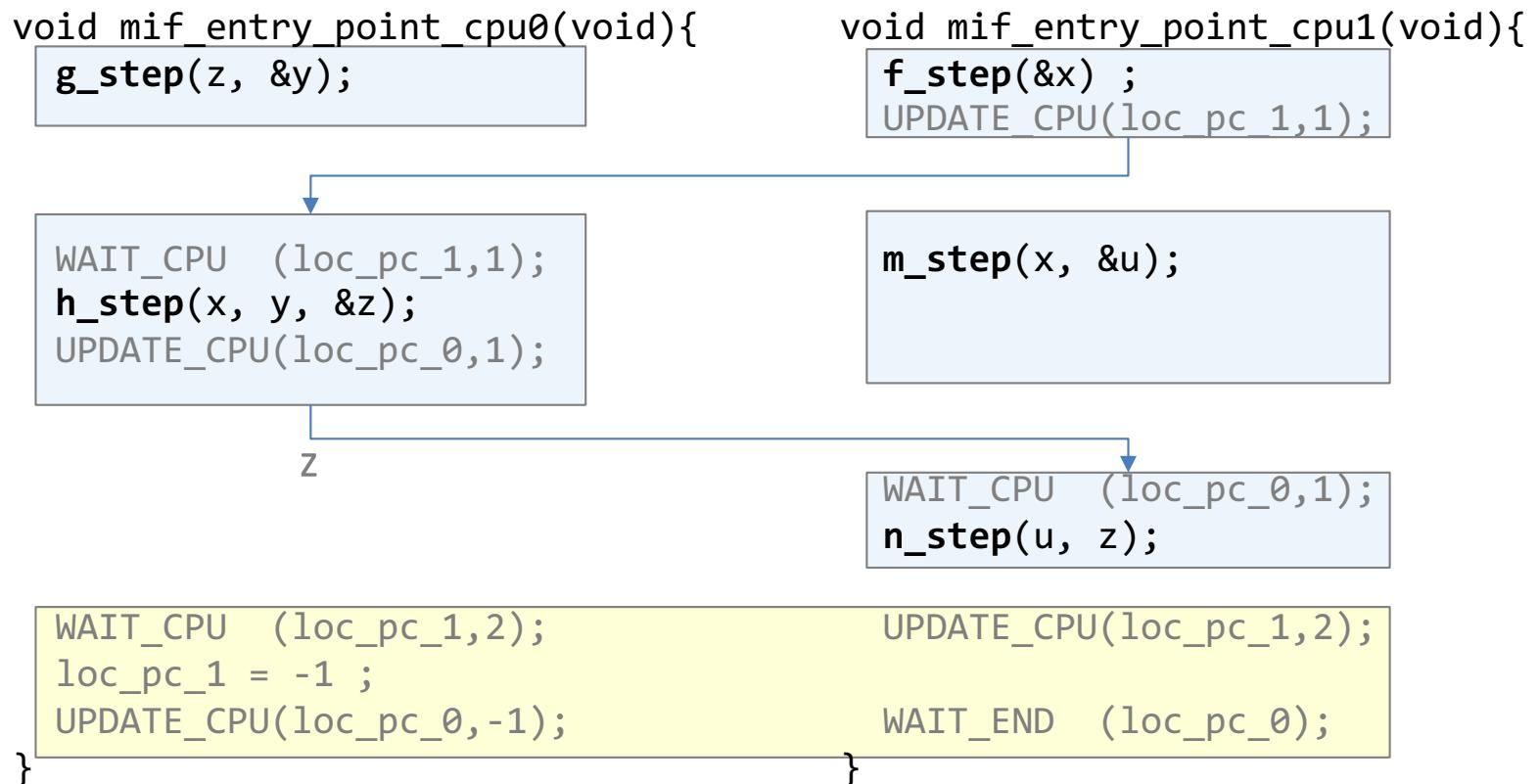
- AARCH64: accès mémoire à effet de barrière
  - Load-acquire (ldar), store-release (stlr)



- STLR: toutes les écritures le précédent dans le thread sont réalisées avant
- LDAR: toutes les lectures le suivant dans le thread sont réalisées après

# Horloges vectorielles

- Synthèse de code C (y compris remise à zéro)



```
#define WAIT_END(var) while(api_ldar(&var)!=-1)
```

# Préparation du TP

- Objectif 1: mise en œuvre de l'exemple du transparent 32
  - Partir de l'exemple de l'archive demo-tp5.tar.gz
  - Ajouter la définition des fonctions m,n
    - demo-tp5/aarch64-cortexA53-rpi3-runtime/application/nodes/nodes.c
    - Suivre le modèle de f,g,h (avec  $m(x) = x-17$ )
  - Ajouter la declaration de m,n
    - demo-tp5/aarch64-cortexA53-rpi3-runtime/application/gen-t1042/nodes.h
  - Modifier la définition des variables C de communication
    - demo-tp5/aarch64-cortexA53-rpi3-runtime/application/gen-t1042/variables.[ch]

# Préparation du TP

- Modifier les threads
  - demo-tp5/aarch64-cortexA53-rpi3-runtime/application/gen-t1042/threads/thread\_cpuX.c
  - Ne garder que les fonctions mif\_entry\_point\_cpuX et une fonction global\_init vide
- Modifier la définition des variables de synchronisation loc\_pc\_X
  - demo-tp5/aarch64-cortexA53-rpi3-runtime/application/gen-t1042/syncvars.[ch]
  - Initialiser les variables loc\_pc\_X à -1
- Compilation de boot.img, el0.img
  - Pas besoin de changer le fichier sur la carte SD
- Exécution sur carte Raspberry Pi