

Assembleur ARM

La pile

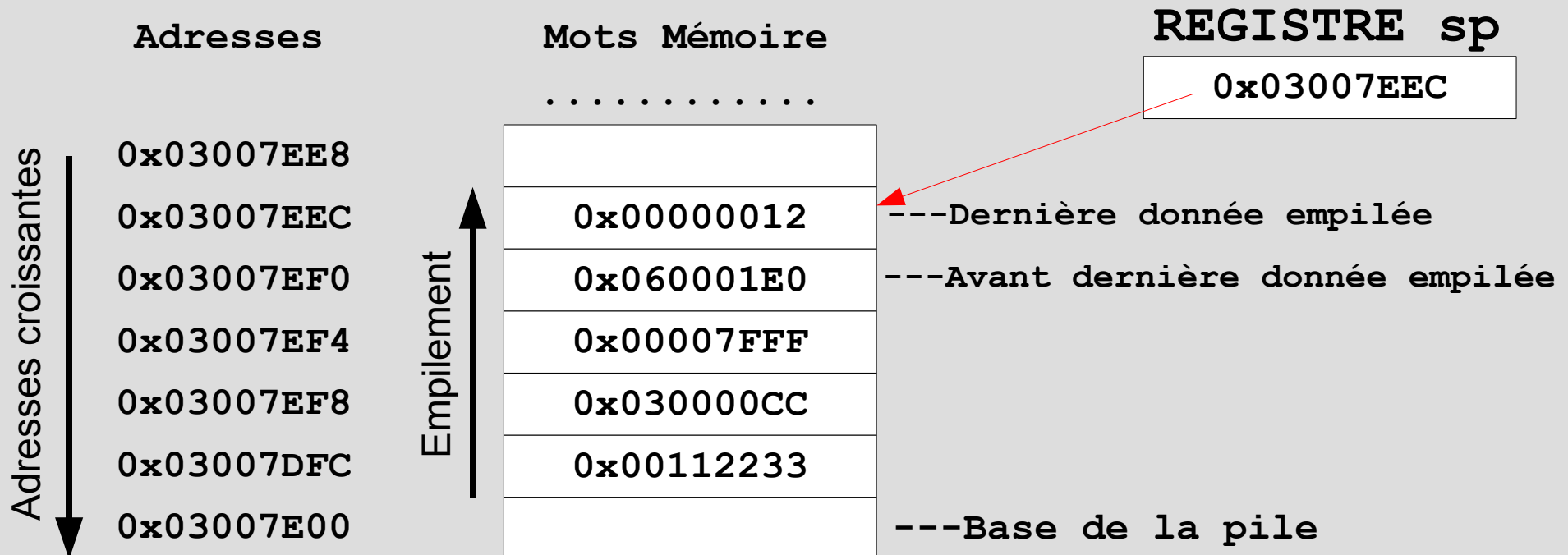
La pile



- Le registre r13 (alias sp) est un pointeur sur le sommet d'une pile de données dynamiques...
- Sur la plupart des systèmes programmables on dispose d'un tel registre nommé ***stack pointer***
- Cette pile sert pour
 - Sauver/Restaurer des valeurs de registres
 - Stocker des variables locales temporaires
 - Passer les paramètres des sous programmes
 - Gérer le retour des sous programmes (récursion...)

La pile

- Sur architecture ARM la pile est utilisée selon une convention *full descending*
- sp pointe sur la dernière donnée empilée et l'empilement se fait vers les adresses basses



La pile



- Il est possible d'utiliser la pile à l'intérieur d'un sous programme assembleur pour **sauver** (empiler) les valeurs de certains registres et **restaure**r ces valeurs plus tard (dépiler)

```
PUSH    {r2-r4,r7} @ Sauver valeurs
```

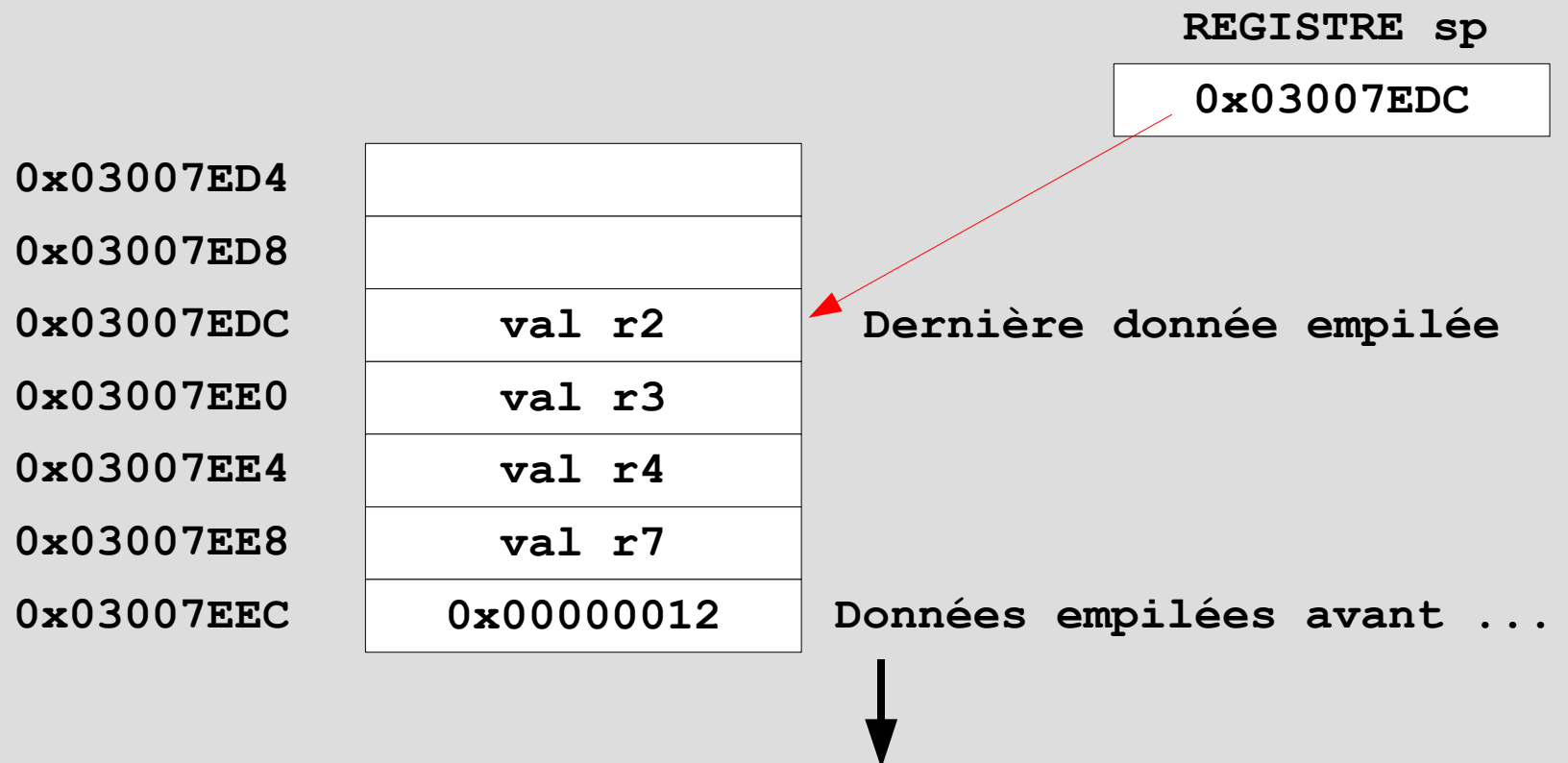
```
...      @ Modifier les valeurs
```

```
...      @      de r2,r3,r4,r7
```

```
POP     {r2-r4,r7} @ Restaurer
```

La pile

- **PUSH {r2-r4,r7} @ Empiler valeurs**



La pile

- L'ordre d'empilement des registres est automatique : les plus petits indices au dessus
- Il faut gérer la pile selon un principe LIFO : Last In First Out. En cas d'utilisations successives, les empilements et dépilements sont **imbriqués** (dépiler "en sens inverse").

```
PUSH    { r7 }      @ Empiler r7
PUSH    { r2-r4 }   @ Empiler r4, r3, r2
...
POP     { r2-r4 }   @ Dépiler r2, r3, r4
POP     { r7 }      @ Dépiler r7
```

La pile

- PUSH et POP sont des alias pour :
 STMFD **sp! , <reglist>**
 LDMFD **sp! , <reglist>**
- Il est possible d'accéder aux données qui ne sont pas au sommet de la pile sans dépiler

```
PUSH    { r2-r4 , r7 }
```

```
...
```

```
LDR    r5 , [sp , #8] @ r5 <- val. r4
```

```
...
```

```
POP    { r2-r4 , r7 }
```

La pile

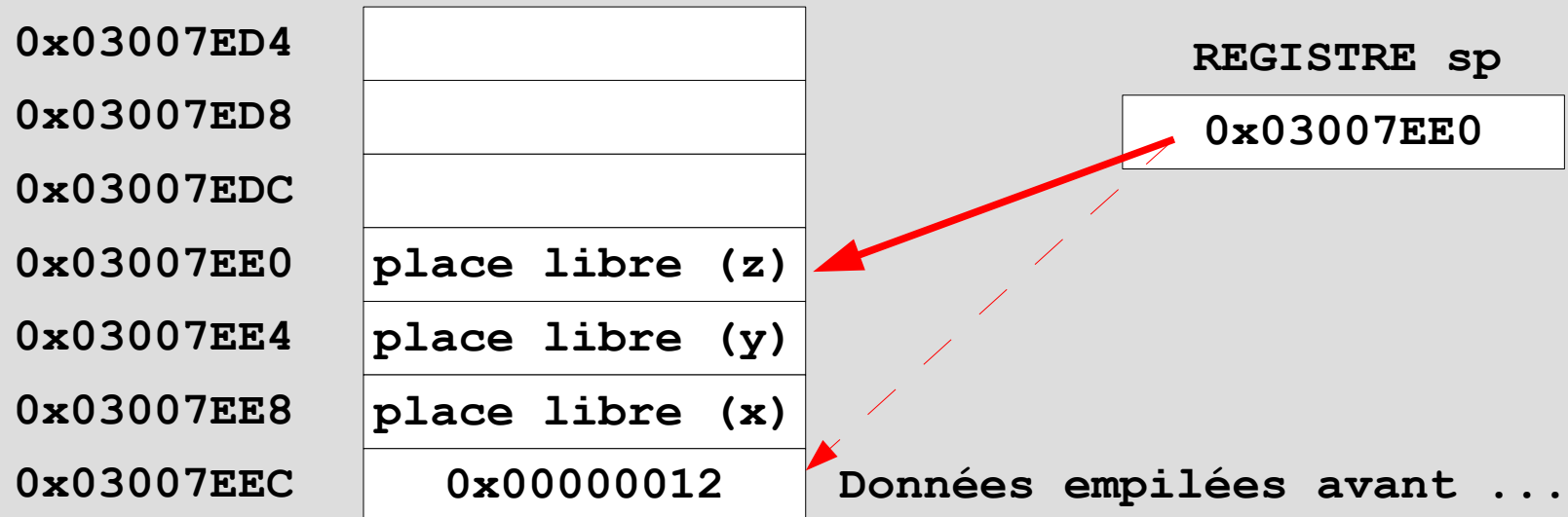


- Les **variables locales** d'un sous programme correspondent à des emplacements sur la pile, on parle de **variables automatiques**
- Pour réserver dynamiquement l'espace mémoire d'une variable locale il suffit de décrémenter `sp` de la taille de la variable
- Pour libérer l'espace à la fin du sous programme il suffit d'incrémenter `sp`
- Il faut libérer en "sens inverse" des réservations

La pile

```
void Procedure () {  
    int x,y,z; // variables automatiques  
    ...  
}
```

-> SUB sp, sp, #12



La pile

- Les variables locales automatiques sont accédées par adressage relatif à partir de `sp`
- Exemple écrire la valeur 0 dans `x`

```
MOV    r0, #0  
STR    r0, [sp, #8]
```

- Il faut connaître précisément l'organisation de la pile pour pouvoir l'utiliser ainsi

La pile

- Les variables globales d'un programme correspondent à des emplacements absolus en mémoire, on parle de **variables statiques**

en C { `int a,b,c; // 3 variables statiques`

`void Procedure () {`

`...`

`}`

en assembleur { `.WORD 0,0,0`

`@ 3 mots réservés une fois pour toutes`

`@ à un emplacement mémoire absolu`

La pile



- Ces différences au niveau assembleur/machine expliquent pourquoi par défaut :
 - les variables globales (statiques) sont a 0
 - les variables locales (auto) ne sont pas initialisées (on trouvera des valeurs résidus sur la pile)

```
// 3 variables statiques : val. initiales 0  
int a,b,c;
```

```
void Procedure() {  
    // 3 variables auto. : val. initiales ????  
    int x,y,z;  
    ...  
}
```

Assembleur ARM

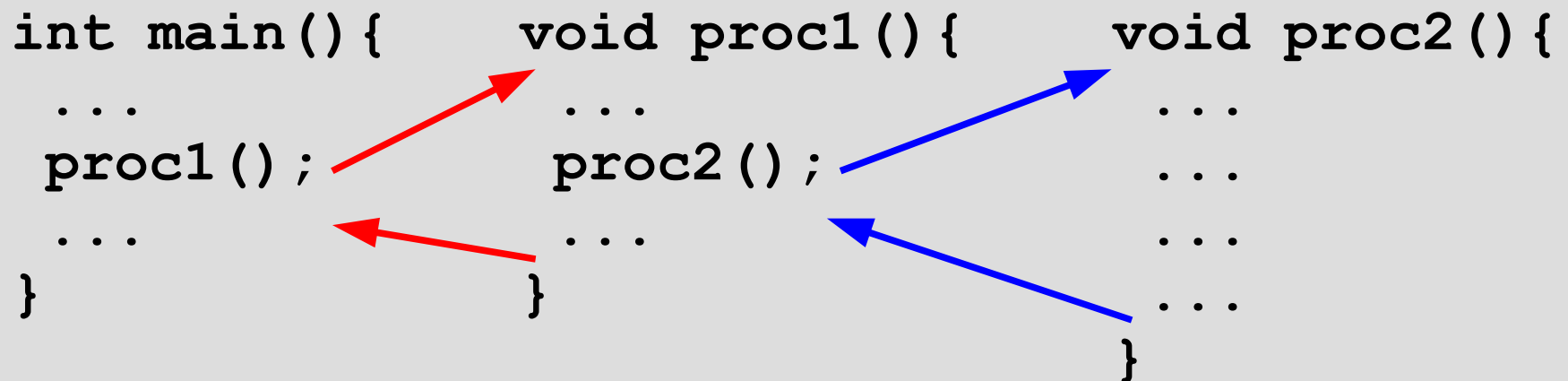
Mécanismes d'appels de sous programmes

Appels de sous programmes

- Un sous programme est un bloc d'instructions dont on peut lancer l'exécution depuis un autre sous programme (appel)
- A la fin de l'exécution du sous programme on reprend à l'instruction suivant l'appel

```
int main() {          void proc1() {          void proc2() {
  ...                ...
  proc1();           proc2();
  ...                ...
}                    }

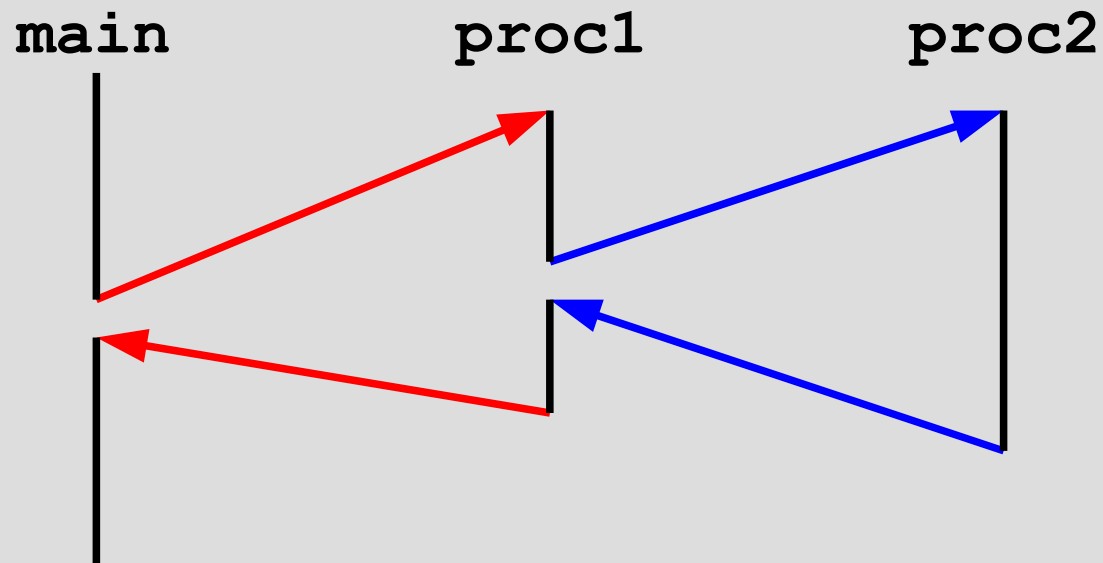
```



The diagram illustrates the flow of execution between three functions: `main`, `proc1`, and `proc2`. Red arrows indicate that `main` calls `proc1`, and `proc1` returns control back to `main`. Blue arrows indicate that `main` calls `proc2`, and `proc2` returns control back to `main`. The code snippets are arranged horizontally, with `main` on the left, `proc1` in the middle, and `proc2` on the right.

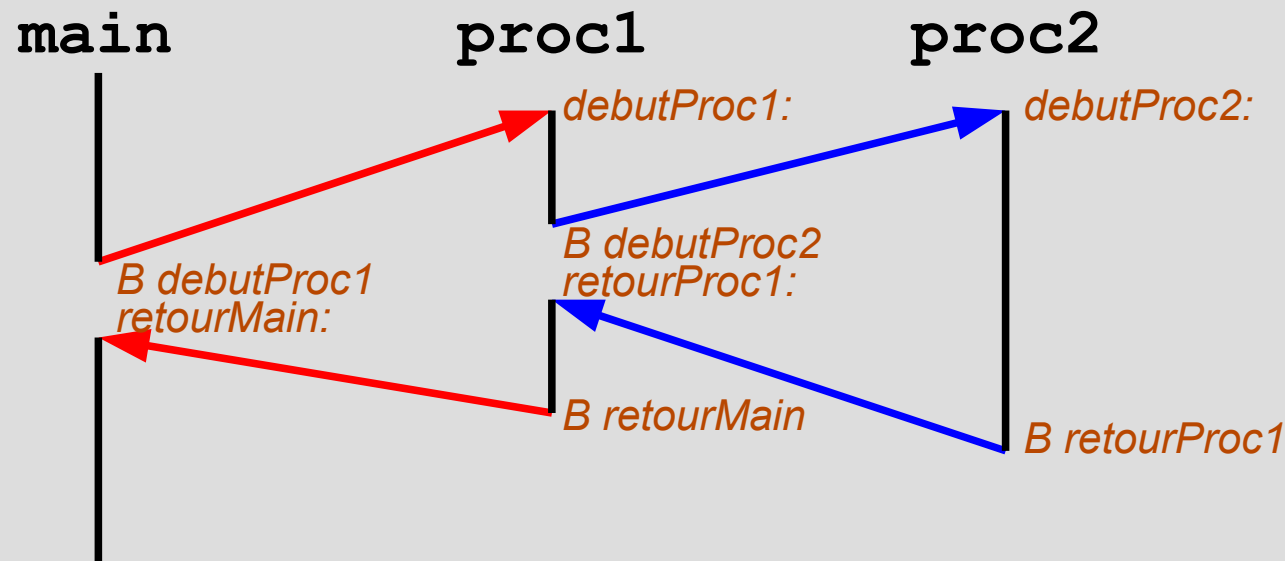
Appels de sous programmes

- On peut schématiser l'exemple précédent :



- Un tel schéma peut être implémenté au niveau des instructions ARM par des branchements...

Appels de sous programmes



- Idée d'implémentation avec des branchements
- Là ça marche **mais ça n'est pas satisfaisant ...**

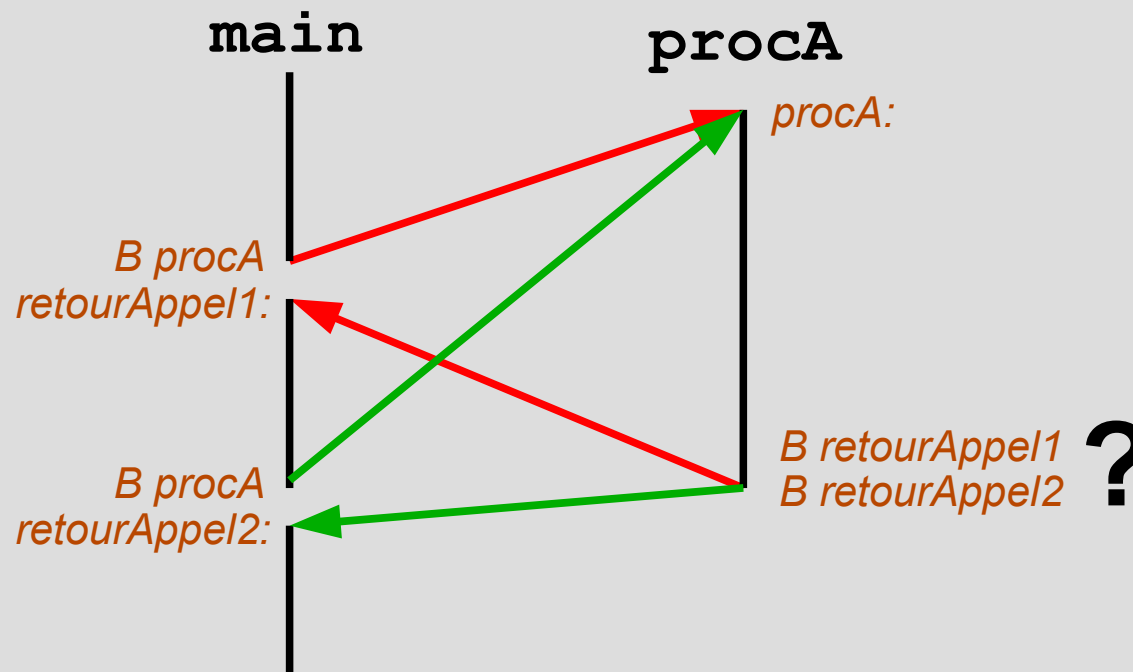
Appels de sous programmes

- L'intérêt d'un sous programme est de pouvoir être appelé depuis plusieurs emplacements

```
int main() {  
    ...  
    procA() ;  
    ...  
    procA() ;  
    ...  
}  
  
void procA() {  
    ...  
    ...  
    ...  
}
```

Appels de sous programmes

- On a alors le schéma :



- Un tel schéma ne peut pas être implémenté au niveau des retours par des branchements à des adresses fixes (correspondant à des étiquettes)

Appels de sous programmes



- Il faut un mécanisme pour communiquer au *sous-programme appelé* l'adresse de retour
- Sur architectures ARM cette information est contenue dans le registre de lien r14 (alias lr) : ***link register***
- Une instruction spéciale **BL** (***Branch with Link***) permet d'**appeler** un sous-programme en ayant **d'abord enregistré l'adresse de l'instruction suivante dans lr**

BL MaProc @ Appel de procedure

Appels de sous programmes

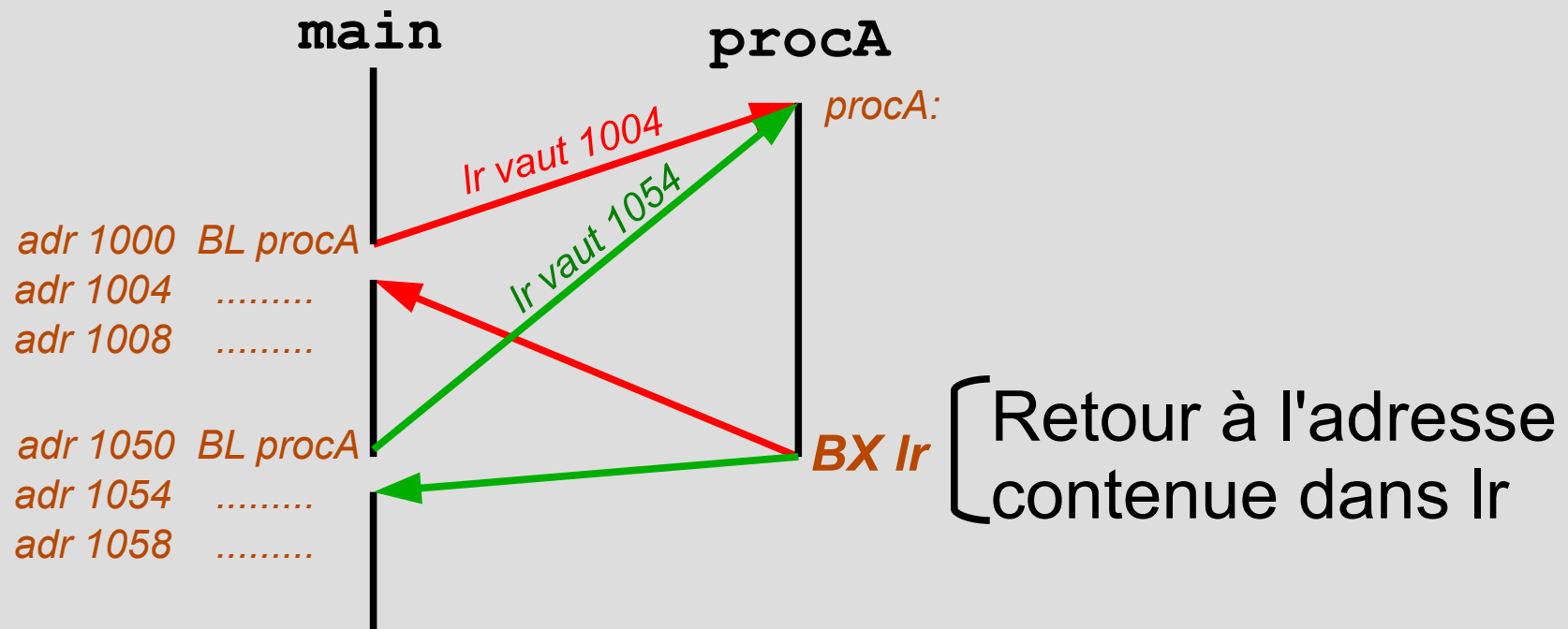


- Le retour au "bon endroit" se fait en mettant la valeur de $lr(r14)$ dans $pc(r15)$
- Une instruction spéciale **BX** permet de faire un branchement à une adresse contenue dans un registre.
- Le retour se fera donc avec cette instruction (dernière instruction d'un sous programme)

BX lr @ Retour à l'appelant

Appels de sous programmes

- On a alors le schéma :



- Cette fois ci on sait toujours d'où on vient + 4 (c'est enregistré dans `Ir`)
- Donc on sait où retourner !

Appels de sous programmes

- La traduction en instructions processeur d'un appel de sous programme est donc :

procédure appelante

```
...  
BL procA @ Appel à procédure procA  
...  
...
```

début procédure procA

```
procA: @ Etiquette de début de procA
```

```
...  
... } Corps de la procédure
```

```
BX lr @ Retour à l'appelant
```

fin de procédure procA

Appels de sous programmes



- Quand on écrit un sous-programme directement en assembleur on aura toujours

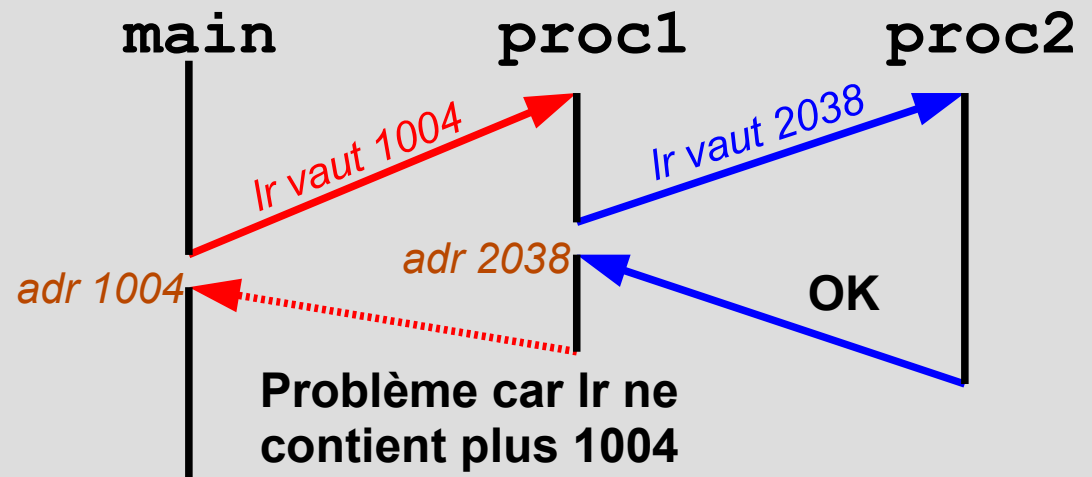
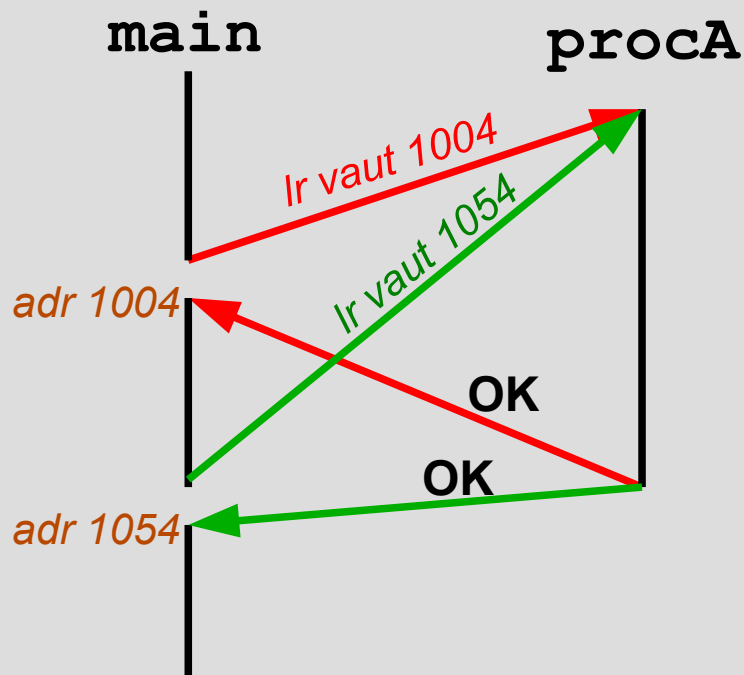
début procédure monSousProg

```
monSousProg:    @ Etiquette de début
                ...
                ...    @ Le corps du sous-prog.
                ...
                BX lr    @ Retour à l'appelant
```

fin de procédure procA

Appels de sous programmes

- Ce mécanisme ne permet pas à lui seul de gérer les appels imbriqués car le contenu de lr est écrasé par le deuxième appel



Gestion incorrecte des appels imbriqués ou récursifs

Gestion correcte des appels **successifs**

Appels de sous programmes

- La solution consiste à empiler le contenu de lr au début du sous programme proc1 et à dépiler avant de retourner au main

proc1 :

PUSH {lr} @ Sauver adr. retour

...

BL proc2 @ Ecrase l'adr. retour

...

POP {lr} @ Restaure adr. retour

BX lr @ Retour à l'appelant

Appels de sous programmes

- Sur la plupart des architectures c'est **la pile** qui est directement utilisée pour passer l'adresse de retour (au lieu d'un registre qu'il faut empiler)
- On parle alors de pile d'appels (***call stack***)
- Sur ces architecture (exemple Intel) la pile est également utilisée pour passer les paramètres...

Assembleur ARM

**Conventions d'appels APCS :
ARM Procedure Call Standard**

Conventions d'appels



- Plusieurs mécanismes sont envisageables pour communiquer des valeurs (**paramètres**) aux sous-programmes
 - Par variables globales (mémoire adresse absolue)
Pas très propre, incompatible avec la récursion
 - Par registres : on met les valeurs des paramètres dans des registres prédéterminés. Ceci n'arrange pas forcément la fonction appelante
 - Par la pile : on empile les valeurs des paramètres
C'est propre mais ça utilise des accès mémoire (plus lent que les registres)

Conventions d'appels



- La convention pour les architectures à base d'ARM est un hybride entre le passage par registres et par la pile :
 - Les **4 premiers paramètres** sont passés dans l'ordre par **r0, r1, r2, r3**
 - Si il y a moins de 4 paramètres les registres excédentaires ne sont pas utilisés
 - Si il y a plus de 4 paramètres, les paramètres supplémentaires sont empilés (le 5ème paramètre est au sommet de la pile, le 6ème en dessous ...)
 - Dans ce cas c'est à la fonction appelée de les dépiler avant de revenir à la fonction appelante

Conventions d'appels



- La **valeur de retour** si le sous-programme doit en donner une se fera par le registre **r0**
- Exemple : une fonction qui retourne la somme de ses deux paramètres

```
// Prototype pour le C :  
int asmAdd (int a, int b);
```

@ Le code assembleur :

```
asmAdd:
```

```
    ADD    r0, r0, r1
```

```
    BX    lr
```

Conventions d'appels

- Chaque sous-programme utilise les registres r0 à r12 pour stocker diverses informations selon une organisation qui lui est propre
- L'ensemble de ces valeurs lié à cette organisation s'appelle un **contexte**
- Lors de l'appel à un sous programme il y a **changement de contexte**
- En dehors des registres utilisés pour le passage de paramètres, des valeurs d'autres registres de la fonction appelante peuvent être modifiés par la fonction appelée

Conventions d'appels

- Deux approches possibles pour résoudre le problème des changements de contextes :
 - C'est la fonction appelante qui doit se débrouiller pour sauver et restaurer son contexte
Si elle ne connaît pas bien le fonctionnement de la fonction appelée elle risque de sauver plus que nécessaire (perte de temps)
 - C'est la fonction appelée qui doit sauver et restaurer les valeurs des registres qu'elle modifie pour que l'appelant retrouve son contexte au retour
- Dans les deux cas c'est **la pile** qui est utilisée pour sauver et restaurer des valeurs

Conventions d'appels

- La convention de sauvegarde de contexte pour les architectures à base d'ARM est un hybride entre ces deux approches
 - Les registres r0, r1, r2, r3, r12 peuvent être modifiés par la fonction appelée. Si la fonction appelante veut être sûre de retrouver les valeurs après l'appel elle doit les empiler, appeler, puis dépiler
 - Les autres registres doivent être rendus par la fonction appelée dans leur état d'origine : si la fonction appelée souhaite les modifier elle doit d'abord les empiler (début de fonction), puis dépiler les valeurs d'origine avant le **BX lr** de retour

Conventions d'appels

- Sécuriser le contexte au niveau de l'appelant

```
PUSH {r0-r3,r12} @ sauf r0 val. retour  
BL   proc  
POP  {r0-r3,r12} @ sauf r0 val. retour
```

- Sécuriser le contexte au niveau de l'appelé

```
proc:  
  PUSH {r4-r11,lr}  
  ...  @ Corps de la procédure  
  POP  {r4-r11,lr}  
  BX   lr
```

Conventions d'appels



- TDs TPs : retenez qu'un sous-programme sécurisé pour le contexte a la forme suivante

@ Code assembleur de monSousProg

monSousProg:

```
PUSH {r4-r11,lr}
```

```
...
```

```
... @ Corps de la procédure
```

```
...
```

```
POP {r4-r11,lr}
```

```
BX lr
```

Architecture GBA

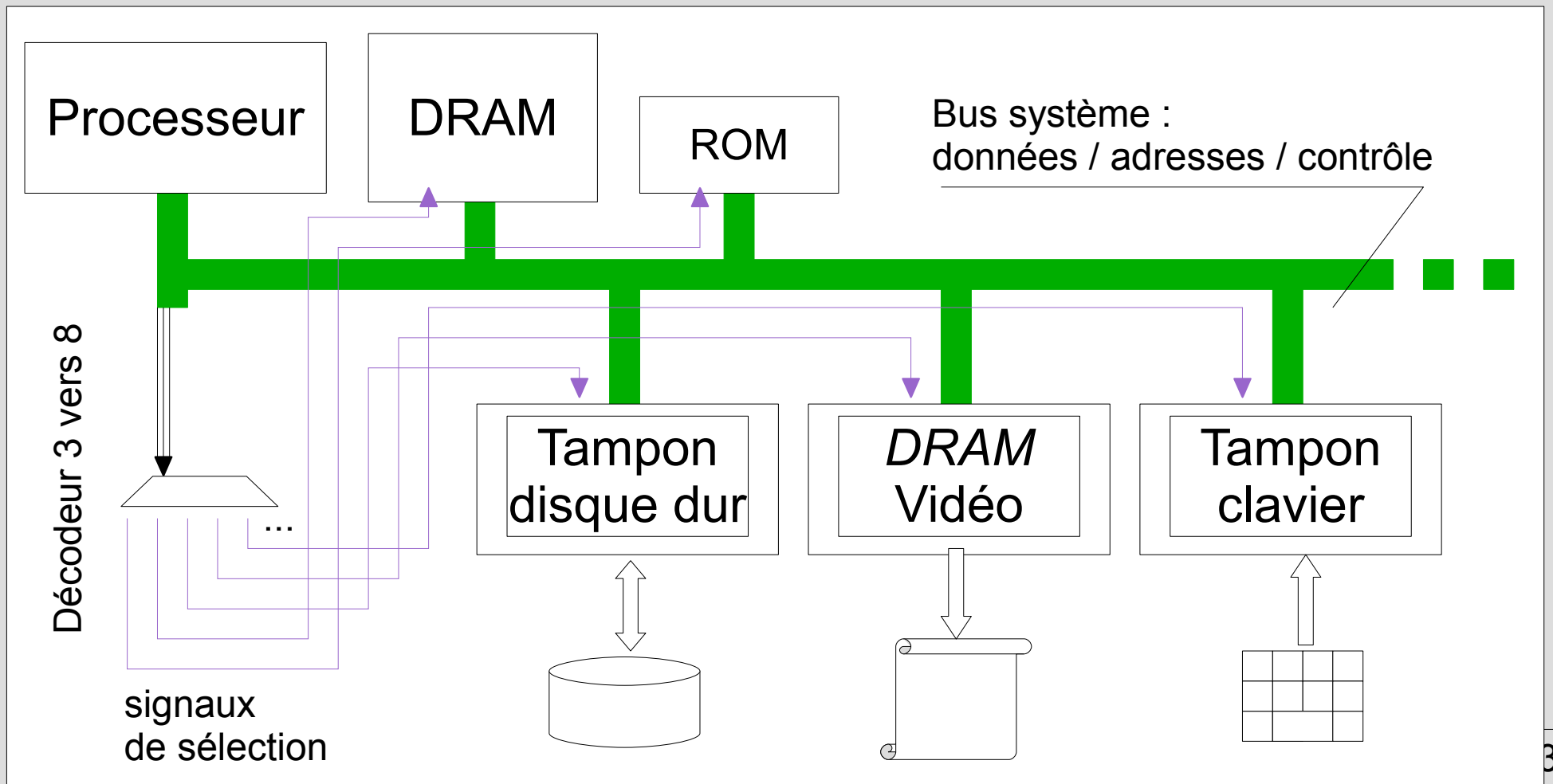
Organisation mémoire et Entrées/Sorties

Plages mémoire et E/S

- Dans une architecture à bus partagé, seule une partie des adresses active le contrôleur mémoire.
- Il reste des adresses vacantes qui permettent au processeur de désigner des périphériques et de gérer les **entrées/sorties**
- La mémoire peut correspondre à des blocs utilisant des technologies différentes :
 - ROM pour stocker le code de démarrage
 - DRAM pour stocker programmes et données chargées depuis une mémoire de masse

Plages mémoire et E/S

- Exemple de circuit auxiliaire pour sélectionner les composants par adresses



Plages mémoire et E/S

- Dans l'exemple précédent 3 bits de **poids fort** du bus d'adresse sélectionnent un module du système parmi 8 au maximum :

16 bits d'adresse

000xxxxxxxxxxxxx -> accès ROM (démarrage)

001xxxxxxxxxxxxx -> accès DRAM (mem. principale)

010xxxxxxxxxxxxx -> accès tampon disque dur

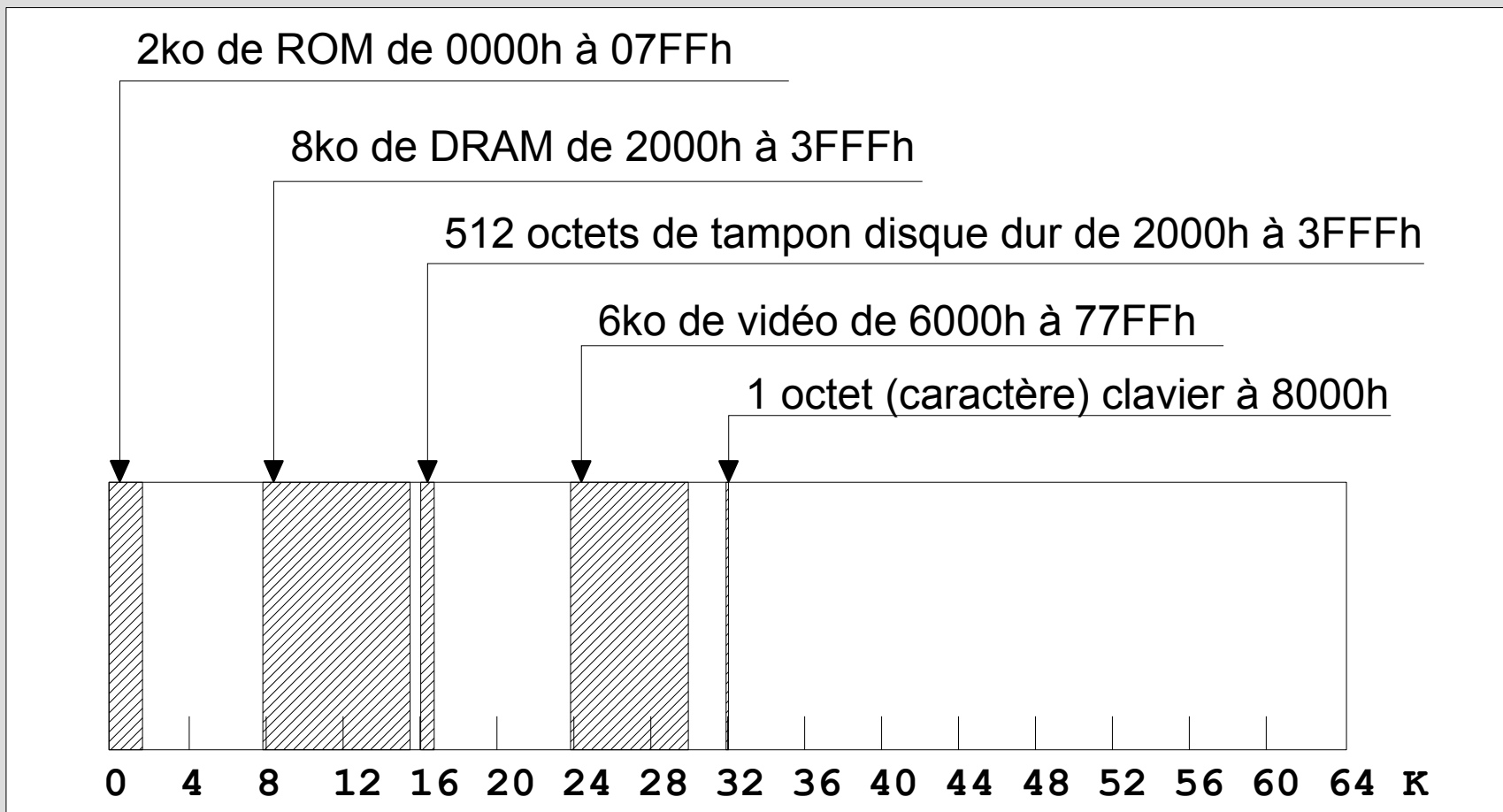
011xxxxxxxxxxxxx -> accès mémoire vidéo

100xxxxxxxxxxxxx -> accès tampon clavier

...

Plages mémoire et E/S

- Le mécanisme de sélection par **plages** définit un découpage de l'**espace d'adressage**



Exemple de plan d'espace d'adressage (système 16 bits)

Plages mémoire et E/S

- D'autres systèmes auxiliaires de sélection par adresses (circuits de logique combinatoire, comparateurs d'adresses) permettent d'organiser l'espace d'adressage de manière plus optimale

Plages mémoire et E/S



- Sur ce genre de système on dira que les Entrées/Sorties sont **mappées** en mémoire : on accède aux circuits périphériques en utilisant des adresses particulières
- Lire/Ecrire sur les périphériques se fait avec les mêmes instructions que pour la mémoire : LDR/STR
- Sur d'autres systèmes (Intel) les E/S se font avec des instructions spéciales IN/OUT

Plages mémoire et E/S GBA

PLAGE	DEBUT	FIN	TAILLE	ACCES	USAGE
System ROM	00000000	00003FFF	16ko	32bits	Mémoire BIOS, démarrage
EWRAM	02000000	0203FFFF	256ko	16bits	External Work RAM
IWRAM	03000000	03007FFF	32ko	32bits	Internal Work RAM
IO RAM	04000000	040103FF	1ko	16bits	Entrées/Sorties
PAL RAM	05000000	050003FF	1ko	16bits	Palettes
VRAM	06000000	06017FFF	96ko	16bits	Mémoire vidéo
OAM	07000000	070003FF	1ko	32bits	Gestion des objets
PAK ROM	08000000	var.	var.	16bits	Cartouche de jeu
Cart RAM	0E000000	var.	var.	8bits	Sauvegarde cartouche

Plages mémoire et E/S



- Sur la GBA les Entrées/Sorties sont mappées en mémoire : on accède aux circuits périphériques en utilisant des adresses particulières
- C'est le cas du dispositif d'affichage avec la mémoire VRAM (Video RAM) qui stocke les couleurs des pixels de l'image de l'écran

Plages mémoire et E/S

- La zone mémoire IO RAM correspond aux autres circuits périphériques, clavier, son ...

- Par exemple pour lire l'état des touches on accède au "registre"

REG_KEYINPUT : 0x04000130

- Pour accéder à la position verticale du balayage écran

REG_VCOUNT : 0x04000006

...

Plages mémoire et E/S



- La mémoire interne IWRAM est intégrée au bloc du processeur et les accès en 32 bits aux instructions se font en un seul cycle horloge
- C'est dans cette zone que les sous-programmes en assembleur ARM sont placés

(directive `.SECTION .iwram, "ax", %progbits`)

Plages mémoire et E/S



- Les variables déclarées en C sont également placées en IWRAM par défaut

- Ne pas déclarer de tableaux trop gros

```
int tab[10000]; // plante le système
```

Plages mémoire et E/S



- Une variable déclarée const va en EWRAM

```
const int tab[10000]={...}; //ok
```

- Des options de placement existent :

```
__attribute__((section(".ewram"))) int tab[10000];
```

Plages et jeux d'instructions



- Par défaut le code C compilé est placé en mémoire EWRAM
- La mémoire EWRAM est accédée par un bus 16 bits : un accès mot demande 2 cycles.
- En fait les instructions compilées depuis le C tiennent sur 16 bits, c'est le jeu d'instruction THUMB

Plages et jeux d'instructions



- Le processeur ARM7TDMI possède 2 jeux d'instruction :
 - Le jeu d'instruction ARM avec des instructions sur un format fixe 32 bits de large
 - Le jeu d'instruction THUMB avec des instructions sur un format fixe 16 bits de large
- Il y a deux modes d'exécution, selon le format

Plages et jeux d'instructions



- Le décodeur interprète les instruction en fonction du mode THUMB ou ARM
- L'unité de contrôle ne voit pas la différence : les instruction effectivement exécutées sont les mêmes (le THUMB est traduit en ARM)
- Le jeu d'instruction THUMB est moins expressif, accès direct seulement aux registres r0 à r7, pas de décalage en 2ème opérande, valeurs immédiates restreintes ...
- Implémente les instructions les plus utilisées et optimisées par les compilateurs

Plages et jeux d'instructions



- Le code THUMB est 65% plus compact que du code ARM
- Le code THUMB s'exécute 160% plus vite que du code ARM sur un système mémoire 16 bits
- Sur un système mémoire 32 bits l'ARM reste plus performant en terme de vitesse
- L'expressivité de l'ARM permet d'optimiser des routines assembleur "critiques" en 32 bits

Plages et jeux d'instructions



En résumé on trouve un compromis :

- Les sections exécutables correspondant à du code C compilé sont en THUMB, moins performant dans l'absolu mais non pénalisé sur les plages mémoires 16 bits (bus léger vers RAM plus volumineuse)
- Les section exécutables correspondant à de l'assembleur sont en ARM 32 bits, plus facile à coder et optimiser que THUMB pour un humain
- Sur système léger le volume accessible en 32 bits est limité : à réserver aux routines critiques

ARM/GBA

Interruptions (infos complémentaires : facultatif)

Interruptions

- Les interruptions sont un mécanisme implémenté au niveau matériel pour quitter provisoirement une tâche en cours, réaliser une autre tâche, et revenir à la tâche de départ
- Le mécanisme est proche de celui de l'appel de sous programme...
- Mais ici c'est un événement et non un appel explicite qui lance la **routine d'interruption**

Interruptions

- Les interruptions matérielles sont générées par un circuit périphérique qui signale au processeur un événement qui peut survenir à n'importe quel moment de l'exécution du programme
 - Une touche est enfoncée
 - Une information est disponible sur le port série
 - Un timer est arrivé à terme
 - Fin de balayage écran horizontal ou vertical
 - Une cartouche de jeu est enfichée
 - Fin d'un transfert DMA

Interruptions

- L'événement n'est pris en compte comme interruption que si l'indicateur *Interrupt Enable* correspondant a été activé par le programme :
REG_IE : 0x04000200
- Dans ce cas, quand l'événement arrive, le processeur sauve le contexte d'exécution actuel sur la pile, enregistre l'adresse de retour dans lr et branche l'exécution à l'adresse contenue à l'adresse **0x03007ffc**
- Il s'agit d'un **vecteur d'interruption**

Interruptions

- La routine de gestion d'interruption, dont l'adresse est en `0x03007ffc`, peut vérifier le type d'événement ayant généré l'interruption en lisant l'entrée **Interrupt Flags**
REG_IF : `0x04000202`
- Après avoir traité l'événement, la routine retourne à l'exécution interrompue avec BX Ir comme pour un appel de procédure usuel

Interruptions

- La routine d'interruption peut choisir de désactiver les interruptions dès son entrée
- Dans le cas contraire une nouvelle interruption peut venir interrompre le traitement de l'événement en cours ...
- Un système de priorité gère ces cas d'interruptions simultanées

Interruptions

- Les ***timers*** sont des compteurs indépendants de l'exécution du programme sur le processeur
- Ils permettent de prévoir le lancement d'une tâche dans un délais précis, ou la répétition d'une tâche à intervalles réguliers
- Ceci nécessite d'utiliser une interruption
- Sur GBA le programmeur dispose de 4 timers 16 bits évoluant à des rythmes réglables, de 60ns à 60µs pour incrémenter de 1

Interruptions

- Le **DMA** (Direct Memory Access) est un mécanisme de copie de zones mémoire par un dispositif auxiliaire sans passer par le processeur
- Sur la GBA le lancement d'une séquence DMA suspend l'exécution du programme par le processeur... ce mécanisme est moins intéressant que sur les plates-formes évoluées où DMA et exécution coexistent simultanément
- Cependant le taux de transfert est optimisé

ARM/GBA

Timers sur la GBA (infos complémentaires : facultatif)

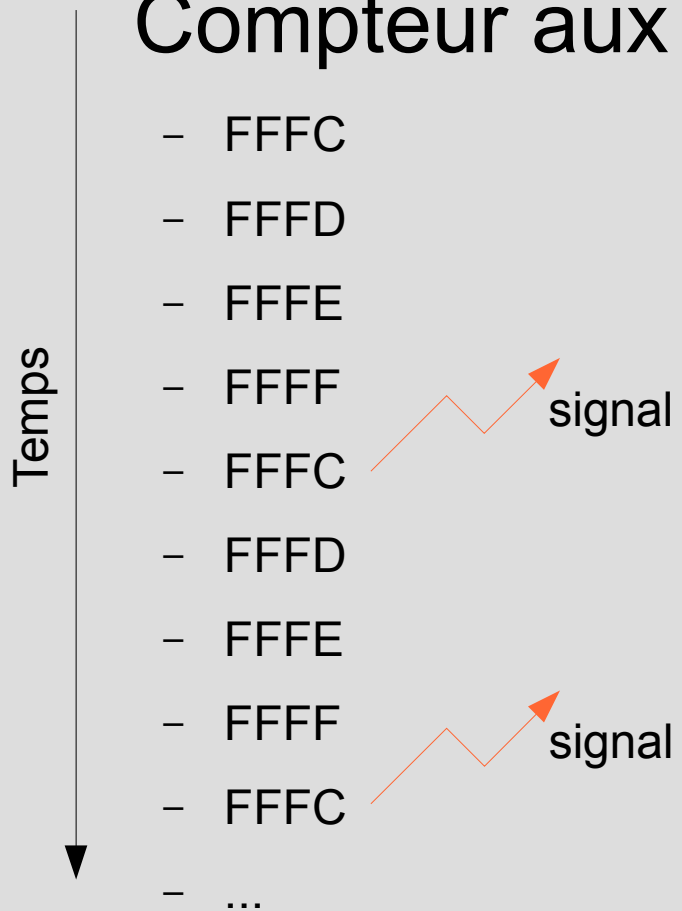
Timers

- Chaque timer GBA est constitué de 3 variables
 - Un compteur sur 16 bits incrémenté à chaque top
 - Une valeur de départ sur 16 bits
 - Un ensemble de 5 bits de contrôle
- Lorsque le compteur est actif, le compteur est incrémenté à chaque top présenté en entrée
- Lorsque le compteur arrive à 0xFFFF et qu'il est incrémenté une fois de plus il "déborde"
- Le débordement génère un signal, et le compteur est réinitialisé à la valeur de départ

Timers

- Exemple : si la valeur de départ est 0xFFFC

Compteur aux tops successifs :



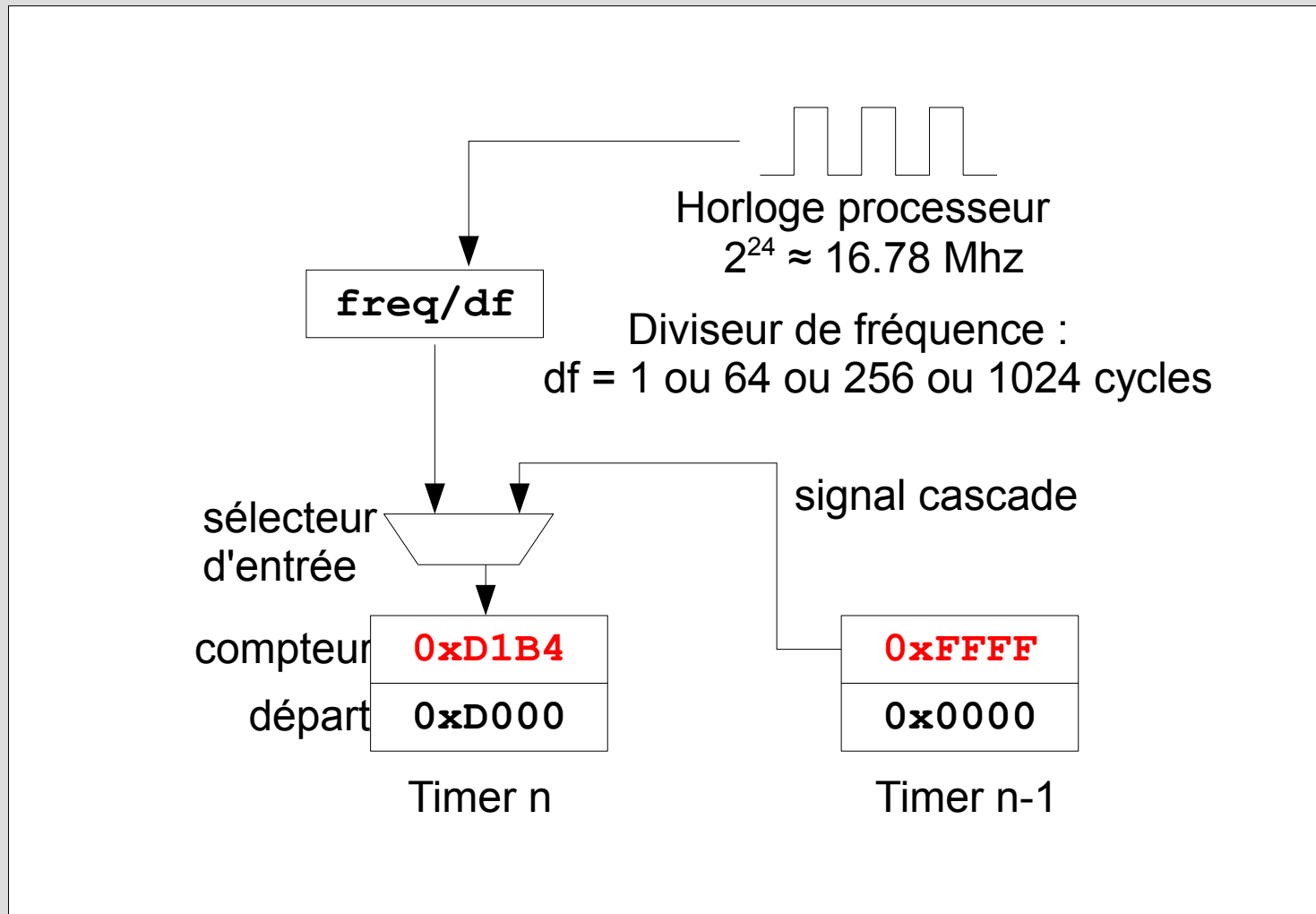
Un signal tous les **4** tops

A noter : 0xFFFC est la représentation sur 16 bits de la valeur signée **-4**

De manière générale pour un signal tous les **n** tops : valeur de départ = **-n**

Timers

- Les tops sont générés par horloge ou cascade



Timers

- 2 "registres" E/S par timer :

reg	fonction	adresse
• REG_TM0D	data	0x04000100
REG_TM0CNT	control	0x04000102
• REG_TM1D	data	0x04000104
REG_TM1CNT	control	0x04000106
• REG_TM2D	data	0x04000108
REG_TM2CNT	control	0x0400010A
• REG_TM3D	data	0x0400010C
REG_TM3CNT	control	0x0400010E

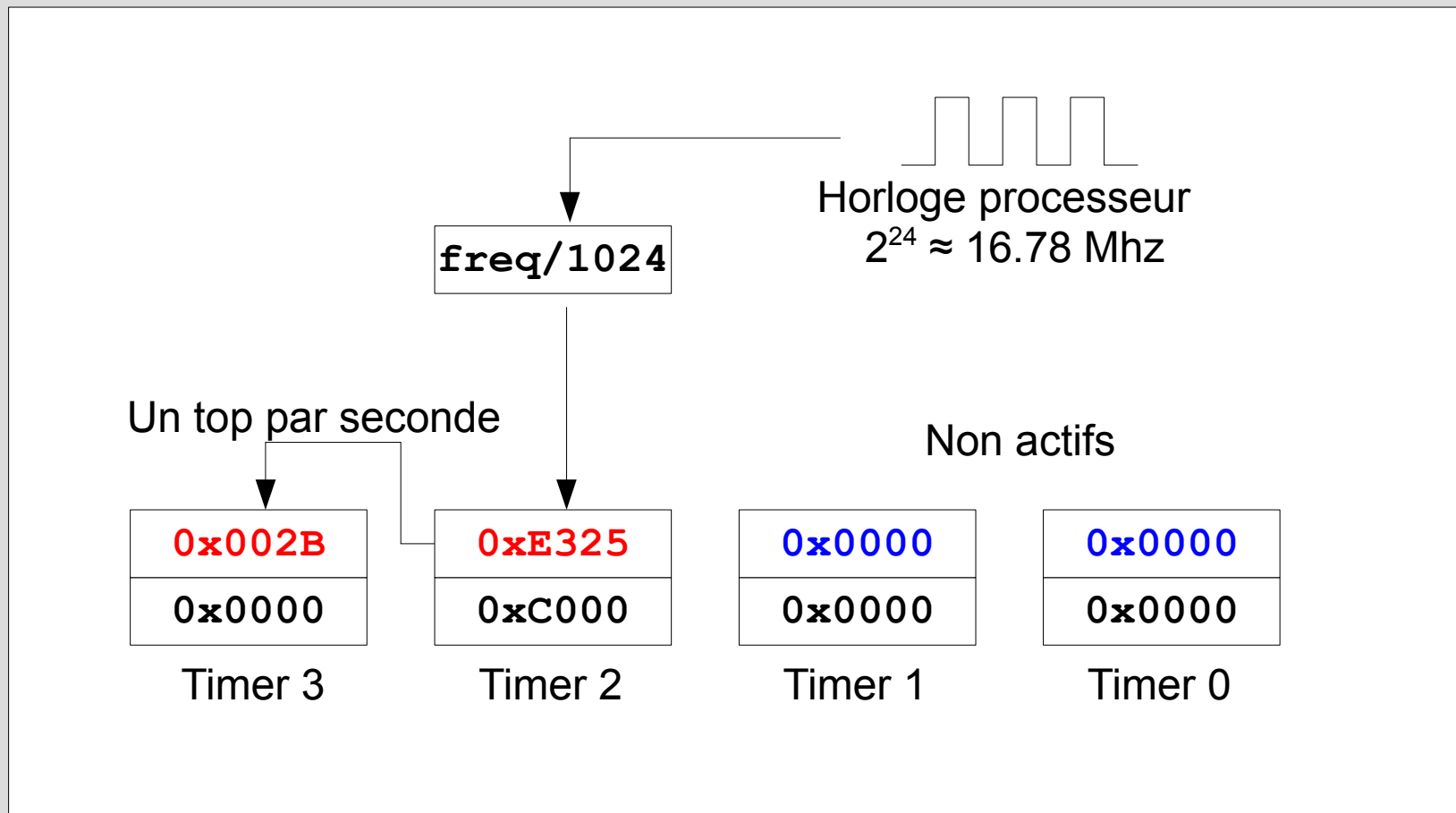
Timers

- Le registre data d'un timer fixe la valeur de départ quand il est accédé en écriture
- Le registre data d'un timer lit la valeur actuelle du compteur quand il est accédé en lecture
- Le registre control détermine le comportement
 - **En** : Enable, activation du timer
 - **I** : Interrupt, génère un signal d'interruption
 - **CM** : Cascade Mode
 - **FD** : Frequency Division (1, 64, 256 ou 1024)

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	n°bit
-	-	-	-	-	-	-	-	En	I	-	-	-	CM	FD		fonction

Timers

- Réalisation d'un compteur de secondes avec les timers 2 et 3 :



Timers

- $2^{24} / 1024 / 0x4000 = 1$
- Dans cette disposition le timer 2 émet un signal toute les seconde
- Le timer 3 compte automatiquement le nombre de secondes à partir de son activation

Timers

```
// Retourne le nombre de secondes écoulées depuis le 1er  
appel à la fonction
```

```
u16 timeSec(){  
    static int first=1;  
    if (first){  
        *(u16 *)0x04000108 = -0x4000;  
        *(u16 *)0x0400010A = 128 | 3;  
        *(u16 *)0x0400010E = 128 | 4;  
        first = 0;  
    }  
    return *(u16 *)0x0400010C;  
}
```

Timers

```
int main()
{
    int nbimage=0;

    setMode3();
    while(1)
    {
        vSync();
        animation();
        asmEffetZoom();
        if ( !((nbimage++)&15) ) {
            traceHex32(nbimage);
            traceHex32(timeSec());
            traceStr("\n");
        }
    }
    return 0;
}
```

Architecture ARM

Complément sur les accès mémoires (infos complémentaires : facultatif)

Compléments accès mémoires

- Une syntaxe permet d'accéder à une adresse à partir d'un registre de base + déplacement :

```
LDR    r0, [r1, #4]
```

- Le contenu mémoire de l'adresse $r1+4$ est copié dans $r0$
- C'est le mode d'adressage **pré-indexé**
- $r1$ n'est pas modifié

Compléments accès mémoires

- Une variante permet d'accéder à une adresse à partir d'un registre de base + déplacement et de modifier le registre

```
LDR    r0, [r1, #4] !
```

- Le contenu mémoire de l'adresse $r1+4$ est copié dans $r0$
- $r1$ est modifié : $r1 \leftarrow r1+4$
- C'est le mode d'adressage **auto-indexé**

Compléments accès mémoires

- Une variante permet d'accéder à une adresse à partir d'un registre de base et de déplacer l'adresse du registre après l'accès

```
LDR    r0, [r1], #4
```

- D'abord le contenu mémoire de l'adresse r1 est copié dans r0
- Puis r1 est modifié : $r1 \leftarrow r1 + 4$
- C'est le mode d'adressage **post-indexé**

Compléments accès mémoires

- L' adressage post-indexé est utile pour optimiser des transferts mémoire :

```
LDRH    r12, [r1]
STRH    r12, [r0]
ADD     r1, r1, #2
ADD     r0, r0, #2
```

Est équivalent à

```
LDRH    r12, [r1], #2
STRH    r12, [r0], #2
```

Compléments accès mémoires

- Il est possible de lire ou de stocker plusieurs registre en une seule instruction

```
LDMIA    r1 , { r4 , r5 , r6 , r7 }
```

Est équivalent à

```
LDR      r4 , [ r1 ]
```

```
LDR      r5 , [ r1 , #4 ]
```

```
LDR      r6 , [ r1 , #8 ]
```

```
LDR      r7 , [ r1 , #12 ]
```

- r1 n'est pas modifié

Compléments accès mémoires

- La variante auto-incrémentée des instructions de transferts multiples est la plus efficace pour copier des blocs de mémoire

```
LDMIA    r1! , {r4 , r5}
```

```
STMIA    r0! , {r4 , r5}
```

Est équivalent à

```
LDR      r4 , [r1] , #4
```

```
LDR      r5 , [r1] , #4
```

```
STR      r4 , [r0] , #4
```

```
STR      r5 , [r0] , #4
```