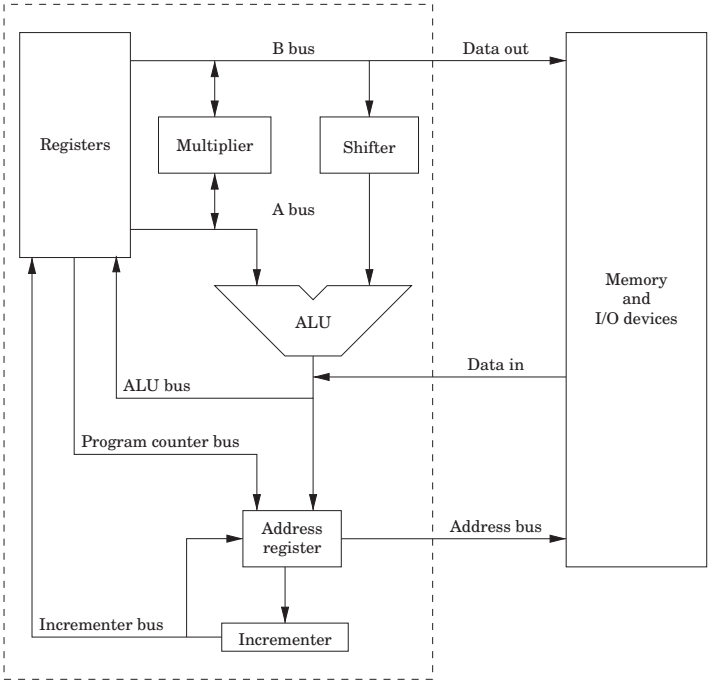
Rappels Programmation ARM
Utilisation OpenOCD/GDB

Architecture du processeur ARM : composants logiques et chemins des données



- ☐ **deux registres** peuvent servir de **source** pour une instruction en passant par les bus A et B ;
- ☐ les données sur le bus B passent par un «*shifter*» : on peut décaler la seconde opérande avant qu'elle atteigne l'ALU ;
- ☐ les bus A et B peuvent fournir des **opérandes** pour le «*multiplier*» et le «*multiplier*» peut fournir des données pour les bs A et B ;
- ☐ les données en **lecture** depuis la mémoire ou des I/Os peuvent aller directement dans l'ALU puis dans un registre.
- ☐ les données en **écriture** vers la mémoire ou dans les I/Os sont prises directement dans le bus B, qui peuvent provenir de regsitres, mais ces données ne peuvent être modifiées sur le chemin.

☐ le **registre d'adresse** est un registre temporaire utilisé à chaque opération de lecture/écriture mémoire/I/Os. Peut être chargé :

- ◇ depuis le «*program counter*» pour **chercher**, «*fetch*», la **prochaine instruction** ;
- ◇ depuis l'ALU pour permettre des **modes d'adressages** où un registre est utilisé comme **adresse de base** et un **décalage** est calculé à la volée.
  Après l'accès, **l'adresse de base** peut être **incrémentée** et cette valeur **stockée** dans un registre ;
  $\Longrightarrow$utilisé pour **incrémenter** le «*program counter*» à chaque instruction ;
  $\Longrightarrow$utilisé pour certains mode d'adressage où un pointeur est **incrémenté** à chaque accès mémoire.

L'architecture ARM : les registres du processeur

Le processeur dispose de 16 registres

- **R0** à **R12** : 13 registres à usage générique utilisable comme on le veut ;
- **R13** : le registre de pile, «*stack*» ;
- **R14** : le registre de lien, «*link*» ; ❶
- **R15** : le «*program counter*» ou registre ordinal.

❶ *Ces registres sont utilisés dans le cas des appels de fonction :*
Le «link register» *contient l'adresse de retour après exécution de la fonction.*
Le «stack register» *contient l'adresse du sommet de la pile, où on empilera les valeurs courantes des registres, et pour créer les variables locales de la fonction appelée.*

- le **CPSR**, «*Current Program Status Register*» : registre d'état contient des bits d'information sur la dernière instruction utilisée. *Utilisé notamment pour les conditions et branchements.*

Exécution des instructions

Effet Pipeline

Chaque instruction est exécutée en **trois cycles d'horloge** :

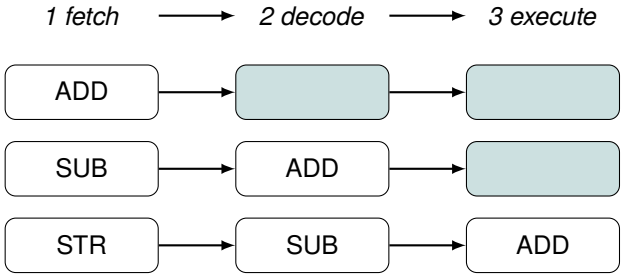
- ▷ un cycle pour le **chargement de l'instruction** depuis la mémoire : *premier étage du pipeline* ;
- ▷ un cycle pour **décoder l'instruction** : *second étage du pipeline* ;
- ▷ un cycle pour **l'exécution** : *troisième étage du pipeline* ;

Lorsqu'une instruction quitte le premier étage du pipeline, une nouvelle instruction peut y entrer :

$\Longrightarrow$ une instruction **sort à chaque cycle** du pipeline

$\Longrightarrow$ une séquence d'instruction est exécutée **en un cycle chacune** !

Processeur ARM : le pipeline d'exécution



En ARMv7 le pipeline a 3 étapes :

- **fetch** : charge une instruction depuis la mémoire ;
- **decode** : identifie l'instruction à exécuter et établit les chemins passant par les bus d'échanges ;
- **execute** : exécute l'instruction et écrit le résultat dans un registre.

Pipeline et «*program counter*»

```
0x8000 LDR pc,[pc,#0]
0x8004 NOP
0x8008 NOP
0x800C DCD JumpAddress
```

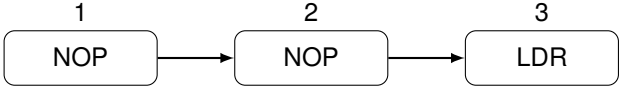
La notation assembleur `LDR pc, [pc,#0]` se traduit par :

- ▷ charge dans PC, «*program counter*» ❶ ;
- ▷ le contenu de PC, «*program counter*», + 0 ❷.

⟹ce qui permet un adressage par décalage.

Mais que vaut PC ?

⟹cela dépend du pipeline d'instruction :



Le PC évolue de la manière suivante :

1. étape 1 du pipeline : `PC = 0x8000` pour l'instruction `LDR` ;
2. étape 2 du pipeline : `PC = 0x8004` pour l'instruction `NOP` ;
3. étape 3 du pipeline : `PC = 0x8008` pour l'instruction `NOP` ;

⟹losrque l'instruction `LDR` est exécutée dans l'étape 3 du pipeline, `PC` vaut `0x8008` !

L'architecture ARM : le format des instructions

31-28	27-25	24-21	20	19-16	15-12	11-0
Condition	Operand type	OpCode	Set Condition Codes	Operand register	Destination register	Immediate Operand

- **Condition** : autorise l'exécution de l'instruction suivant les bits dans le registre CPSR ;
- **Operand type** :spécifie le format de l'opérande des bist 19-0 :
  - par exemple deux registres suivis d'une opérande immédiate ;
- **Opcode** : quelle instruction on veut réaliser comme `Add` ou `MUL` ;
- **Set condition code** : un seul bit indiquant si l'instruction doit mettre à jour le registre CPSR, valeur 0, ou non, valeur 1 ;
- **Operand register** : un registre à utiliser comme entrée ;
- **Destination register** : un registre à utiliser comme sortie ;
- **Immediate operand** : une donnée de petite taille que l'on peut donner directement dans l'instruction.
  - exemple : si on veut ajouter 1 à un registre, on peut mettre la donnée à 1 ce qui évite de mettre 1 dans un autre registre et de faire la somme de ces deux registres.

Architecture ARM : le processeur

Architecture 32 bits et 64 bits

- En **32bits** :
  - les **adresses mémoires** sont sur 32 bits ;
  - les **registres du processeur** sont sur 32 bits ;
- En **64bits** :
  - les **adresses mémoires** sont sur 64 bits ;
  - les **registres du processeur** sont sur 64 bits ;
- En **32bits** comme en **64bits**, les **instructions sont sur 32bits** :
  - *Comment peut-on charger une variable depuis la mémoire dasn un regsitre donné avec une instruction sur 32bits ?*
    - l'instruction fait 32bits ;
    - 4bits sont utilisés pour un opcode ;
    - 4bits pour une instruction conditionnelle ;
    - 3bits sont utilisés pour indiquer le type de l'opérande ;
    - 1bit pour indiquer si l'opération affecte le CPSR ;
    - 4bits sont nécessaires pour indiquer le registre ;

    $\Longrightarrow$ **il reste 16bits pour indiquer l'adresse mémoire !**

    $\Longrightarrow$ **il reste 12bits pour indiquer l'adresse mémoire si on a besoin d'indiquer 2 registres !**
- Comment faire ?
  - on peut utiliser un registre pour indiquer l'adresse mémoire : **accés mémoire indirect**
  - Mais comment charger l'adresse mémoire initialement dans le registre en une seule instruction ?
  - charger dans deux registres séparés l'adresse désirée, puis décaler et combiner les deux registres en un seul registre $\Longrightarrow$4 instructions pour arriver au résultat ce qui est excessif !
  - Et alors ? On peut utiliser le PC, «*Program Counter*» pour charger une **mémoire pas trop éloignée**, à 12bits de décalage par rapport au PC, soient **4096 octets accessibles** et beaucoup plus **par décalage de ces bits**.

Utilisation des registres

Définition de contenu

```
label: .byte 74, 0112, 0b00101010, 0x4A, 0X4a, 'J', 'H' + 2
       .word 0x1234ABCD, -1434
       .ascii "Hello World\n"
```

Charger un registre

L'instruction `LDR` peut servir à charger une addresse dans le registe ou la donnée pointée par cette addresse.

Il est également possible d'indexer la mémoire en utilisant la valeur d'un registre comme valeur d'index.

- ▷ addressage relatif au PC, «*Program Counter*» ;
- ▷ charger depuis la mémoire ;
- ▷ indexer à travers la mémoire.

Adressage relatif au PC

Si les données ne sont pas loin de l'instruction utilisant leur adresse, on peut utiliser cet adressage :

```
LDR R1,=helloworld
```

ce qui donne après assemblage :

```
LDR R1, [pc, #20]
```

Ici, la valeur du décalage est de `0x20` en hexa, soit 32.

Ce décalage est :

- ▷ déterminé lors de l'assemblage ;
- ▷ sur 12 bits dans l'instruction elle-même :
  - ◇ ce qui donne de 0 à 4095 ;
  - ◇ un bit est utilisé dans l'instruction pour indiquer dans quel sens :
    ⟹le décalage peut être de $\pm$4095 *words* au maximum.
  - ◇ le décalage peut être par multiple de 1 à 2 octets, comme indiqué dans la table ⟹

`LDR{type} Rt, =label`

Type	Meaning
B	Unsigned byte
SB	signed byte
H	Unsigned halfword (16 bits)
SH	signed halfword (16 bits)
–	omitted for word

Assembleur ARM : appel de fonction

Utilisation de la pile

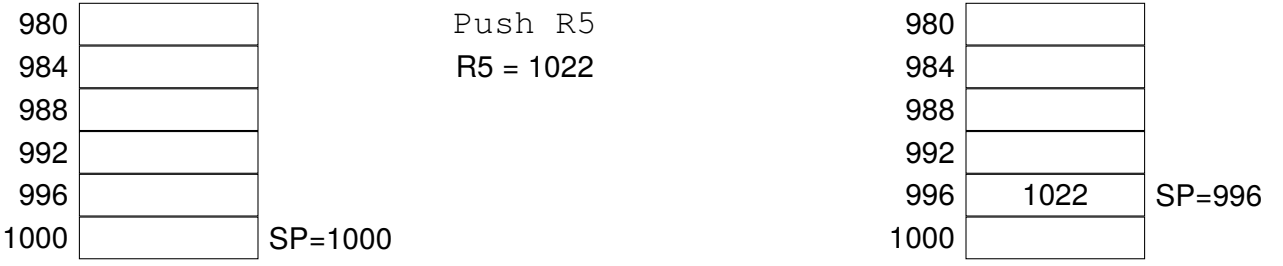
Deux opérations possibles :

- ▷ **push** : ajouter un élément ;
- ▷ **pop** : enlever et retourner l'élément le plus récemment ajouté.

Gestion dans le processeur ARM :

- ☐ **registre R13** aussi appelé **SP**, «*Stack Pointer*» : il pointe sur l'emplacement de la pile en mémoire ;
- ☐ deux instructions du jeu d'instruction ARM32 : **LDM**, «*Load Multiple*» et **STM**, «*Store Multiple*» ;
- ☐ ces instructions sont adaptables :
  - ◇ choix du sens d'augmentation de la pile par incrémentation ou décrémentation des adresses ;
  - ◇ le registre pointe soit sur la fin de la pile, soit sur le prochain emplacement libre ;

  $\Longrightarrow$ *le système est adaptable aux besoins de différents systèmes d'exploitation*.
- ☐ l'assembleur GNU offre des pseudo-instructions qui s'appuient sur les instructions LDM et STM :
  - ◇ PUSH liste de registre : `PUSH {R0, R5-R12};`
  - ◇ POP liste de registres : `POP {R0-R4, R6, R9-R12};`



Assembleur ARM : appel de fonction

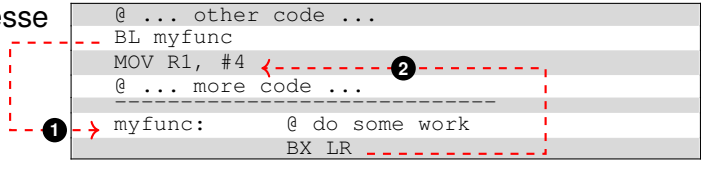
Branchement avec lien de retour

Après l'appel de la fonction, il faut retourner à l'exécution des instructions qui suivent :

- le registre R14, «*link register*» : sert à stocker l'adresse de retour ;
- l'instruction `BL`, «*Branch with Link*» : réalise un branchement aprés avoir stocké l'adresse de la prochaine instruction dans LR ;
- l'instruction `BX` : réalise le retour de fonction en sautant à l'adresse présente dans le registre LR ;

▷ `BL` saute à l'adresse de ❶ et sauvegarde l'adresse de retour dans `LR` ;

▷ `BX` saute à l'adresse stockée dans `LR` ❷



Et comment cela se passe si la fonction appelle une autre fonction ?

On utilise de nouveau dans la fonction `myfunc`, l'instruction `BL` :

⟹`BL` **copie** l'adresse de la prochaine instruction dans le registre `LR`

⟹*ce qui* ***écrase*** *l'ancienne valeur contenue dans `LR`*

⟹la fonction `myfunc` **ne pourra plus retourner** !

⟹il faut :

▷ **sauvegarder** la valeur du registre `LR` dans la pile❸ avant le `BL`

▷ **restaurer** la valeur du regsitre `LR`❹ avant de retourner avec `BX`

```
   @ ... other code ...
   BL myfunc
   MOV R1, #4
   @ ... more code ...
-----------------------------
myfunc: PUSH {LR}❸
   @ do some work ...
   BL myfunc2
   @ do some more work...
   POP {LR}❹
   BX LR
myfunc2: @ do some work
   BX LR
```

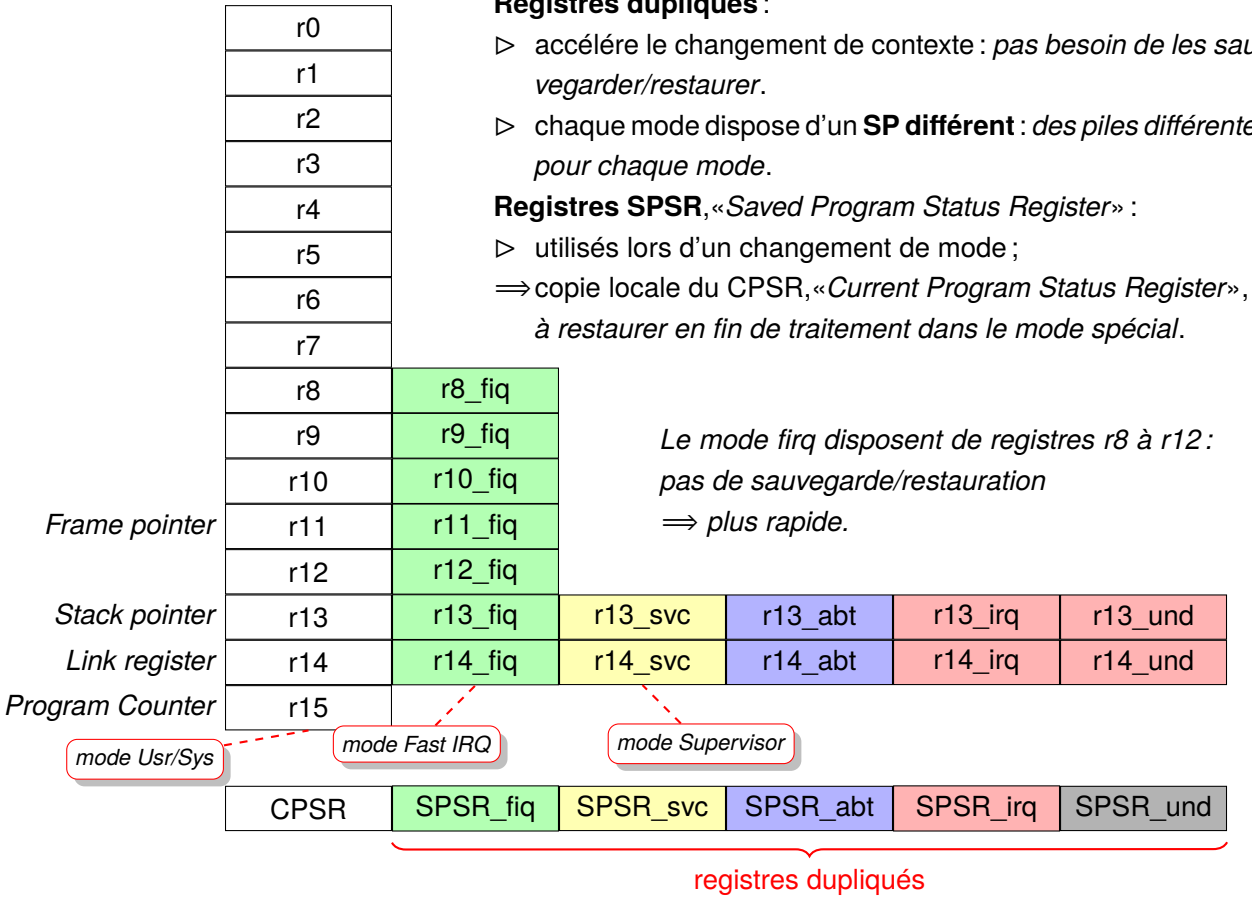
Différents modes d'exécution

- 1 mode utilisateur,«*usr*», non privilégié ;
- 5 modes pour des exceptions + 1 pour le système, priviégiés :

Mode	Sigle	Description
User	usr	mode normal d'exécution non privilégié
Interrupt	_irq	activé lors d'une exception causée par une interruption matérielle
Fast Interrupt	_fiq	activé lors d'une exception causée par une interruption matérielle à gérer rapidement
Abort	_abt	activé lors d'une exception d'accès à la mémoire
Undefined	_und	activé lorsqu'une instruction indéfinie est exécuté (une valeur sur 32bits qui ne correspond à aucune instruction existante)
Supervisor	_svc	Software Interrupt: activé lors du reset ou lors d'un appel système
System	sys	mode privilégié pour le travail de l'OS.

- le passage d'un mode à l'autre peut être fait **manuellement** en modifiant des bits du registre CPSR ;
- les **modes privilégiés** sont activés pour traiter des **interruptions** ou des **exceptions** ;
- le **mode système**, «*sys*», est un **mode spécial** pour accéder à des **ressources protégées** (comme la MMU par exemple si disponible) ;
- les modes traitant les **interruptions** disposent de **registres dupliqués**, remplaçant les registres qu'ils dupliquent durant l'exécution du traitement, ce qui évitent les collisions et corruptions.
- les modes «*usr*» et «*sys*» **partagent le même ensemble** de registres.

Registres et mode d'exécution : 37 registres au total pour 7 modes d'opérations



Registres dupliqués :

- ▷ accélére le changement de contexte : *pas besoin de les sauvegarder/restaurer*.
- ▷ chaque mode dispose d'un **SP différent** : *des piles différentes pour chaque mode*.

Registres SPSR,«*Saved Program Status Register*» :

- ▷ utilisés lors d'un changement de mode ;

⟹copie locale du CPSR,«*Current Program Status Register*», *à restaurer en fin de traitement dans le mode spécial*.

Le mode firq disposent de registres r8 à r12 : pas de sauvegarde/restauration ⟹ plus rapide.

Mais ça marche comment les interruptions ?

Interruptions : gestion matérielle et logicielle

Assigner les interruptions

C'est le **designer du SoC** qui décide quel matériel peut produire quelle interruption : il s'agit d'interconnecter des circuits avec le processeur.

- ▷ SWI, «*SoftWare Interrupt*» ou svc, utilisée pour accéder à des fonctions privilégiées de l'OS ;
- ▷ IRQ, assignées à des interruptions génériques comme des timers périodiques ;
- ▷ FIQ, réservée pour une seule source qui nécessite un faible temps de réponse.

Les latences dans le traitement d'une interruption

La **latence** est l'intervalle de temps entre :

- ▷ l'instant où le signal externe change d'état ;
- ▷ la première instruction du traitement est chargé dans le processeur (fetch) ;

Le système essaie d'atteindre deux buts :

- ☐ gérer **plusieurs interruptions** simultanément ;
- ☐ **minimiser** la latence.

Deux métodes pour l'atteindre :

- ▷ autoriser la **gestion imbriquée**, «*nested*», des interruptions ;
- ▷ donner des **priorités** aux différentes sources d'interruption.

Activer/désactiver les interruptions

Les instructions dédiées :

MRS	lire le CPSR
MSR	stocker dans le CPSR
BIC	effacer un bit
ORR	opération OR

Activer une IRQ / FIRQ :

```
MRS r1, cpsr
BIC r1, r1, #0x80/0x40
MSR cpsr_c,r1
```

Désactiver une IRQ / FIRQ :

```
MRS r1, cpsr
ORR r1, r1, #0x80 / 0x40
MSR cpsr_c,r1
```

Interruptions : gestion de la pile du handler d'interruption

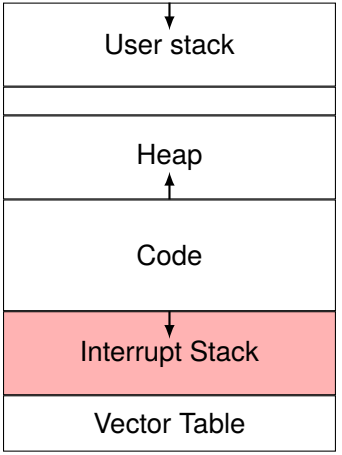
Pourquoi une pile ?

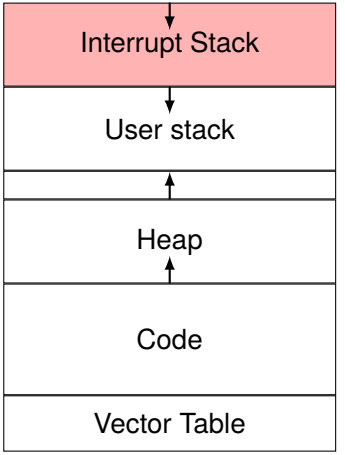
La pile est nécessaire pour le changement de context :

- ▷ empiler les valeurs des registres pour les sauvegarder ;
- ▷ dépiler les valeurs pour restaurer les registres ;
- ▷ variables locales nécessaires pour la gestion des interruptions.

Il faut choisir :

- □ la taille de la pile ;
- □ sa position dans la mémoire.





Dans ce cas là, la table des vecteurs d'interruption ne peut être écrasée lors d'un dépassement de pile.

Table des vecteurs d'interruption

La **table des vecteurs d'interruption** est une table d'adresses :

- ▷ le processeur ARM **saute** à l'adresse associée à l'exception qui s'est déclenchée ;
- ▷ à cette adresse se trouve une **instruction de saut** vers le code de traitement de cette exception.

Dans le tableau, on trouve une instruction de branchement :

```
ldr pc, [pc,#_IRQ_HANDLER_OFFSET]
```

Où `#_IRQ_HANDLER_OFFSET` est le décalage pour atteindre le code de gestion de l'interruption.

Adresse	Exception	Mode en entrée
0x00000000	Reset	Superviseur
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIRQ	FIRQ

Cette table doit être remplie par l'OS lors du démarrage de la machine.

Exceptions : priorités et gestion de l'adresse de retour

Priorités

indique si le handler de l'interruption peut être ou non interrompu

définie quelle exception est la plus importante parmi celles déclenchées

Exception	Priorité	bit I	bit F
Reset	1	1	1
Data Abort	2	1	-
FIQ	3	1	1
IRQ	4	1	-
Prefetch Abort	5	1	
SWI	6	1	-
Undefined Instruction	6	1	-

déclenchée lors de l'étape «*execute*» du pipeline

Décalage du registre PC sauvegardé dans le registre LR

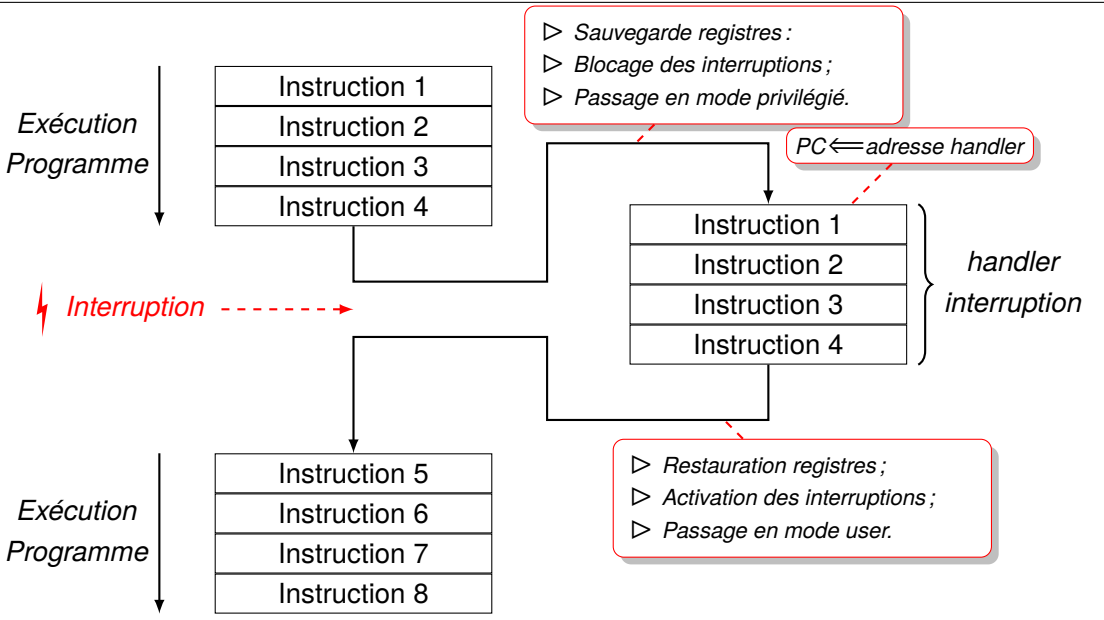
Exception	Adresse de retour
Reset	aucune
Data Abort❷	LR - 8
FIQ, IRQ, prefetch Abort❷	LR - 4
SWI, Undefined Instruction	LR

Dans certain cas, le registre PC a avancé au delà de l'instruction qui a causé l'exception :

- ▷ en ❶, il faut revenir en arrière de deux instructions par rapport à la valeur du PC sauvegardée dans le registre LR ;
- ▷ en ❷, il faut revenir en arrière d'une seule instruction.

Ceci est du au pipeline d'exécution du processeur ARM.

Interruption : prise en compte dans le logiciel



Entrée dans le handler d'exception :

⇒sauver l'adresse de l'instruction suivante dans le registre LR aproprié ;

⇒copier le CPSR dans le SPSR du nouveau mode ;

⇒changer le mode en modifiant les bits du CPSR ;

⇒aller chercher l'instruction suivante dans la table des vecteurs d'interruption.

Sortie du handler d'exception :

⇒charger le registre LR dans le PC (avec le bon décalage) ;

⇒restaurer le SPSR dans le CPSR, ce qui remettra les bits de mode comme avant ;

⇒effacer le bit de blocage des interruptions s'il avait été positionné.

Et sur un exemple d’interruption logicielle
ça donne quoi ?

Programmation assembleur ARM

On va utiliser l'assembleur `as` pour assembler l'exécutable :

```
@
@ Assembler program to print "Hello World!"
@ to stdout.
@
@ R0-R2 - parameters to linux function services
@ R7 - linux function number
@

.global _start @ Provide program starting
@ address to linker
@ Set up the parameters to print hello world
@ and then call Linux to do it.
_start: mov R0, #1 @ 1 = StdOut
        ldr R1, =helloworld @ string to print
        mov R2, #13
        mov R7, #4
        svc 0 @ Call linux to print

@ Set up the parameters to exit the program
@ and then call Linux to do it.
@ length of our string
@ linux write system call
        mov R0, #0   @ Use 0 return code
        mov R7, #1   @ Service command code 1
                     @ terminates this program
        svc 0        @ Call linux to terminate
.data
helloworld: .ascii "Hello World!\n"
```

l'étiquette `_start` indique au «linker» le point d'entrée du programme

On utilise un appel système pour faire l'affichage

On utilise un appel système pour terminer le programme

`.ascii` indique à l'assembleur de mettre la chaîne dans le segment data, et d'utiliser l'étiquette pour avoir son adresse

Assemblage et exécution :

```
xterm
$ as -o HelloWorld.o HelloWorld.s
$ ld -o HelloWorld HelloWorld.o
$ ./HelloWorld
Hello World!
$
```

- l'instruction `MOV` déplace des données dans un registre.
- le `#4` indique une opérande directe, soit la valeur 4.
- `LDR R1, =helloworld` charge le registre avec l'adresse de la chaîne.
- `svc 0` réalise une interruption logicielle pour donner le contrôle au noyau Linux.

Assembleur ARM : appel à Linux

Comment utiliser une fonctionnalité de l'OS ?

```
_start: mov R0, #1 @ 1 = StdOut
        ldr R1, =helloworld @ string to print
        mov R2, #13
        mov R7, #4
        svc 0 @ Call linux to print
```

Les **registres R0 à R4** vont être utilisés pour passer les paramètres à l'appel système.
Lors du retour de l'appel système, la **valeur de retour** sera donnée dans le registre R0.

Ici, on fait appel à l'appel système `print` :

- ▷ on indique dans le registre R0 la destination `stdout` ;
- ▷ l'adresse de la chaîne à afficher est mise dans le registre R1 ;
- ▷ le numéro de l'appel système est donné dans le registre R7, ici c'est la valeur 4 pour indiquer `print`.

On réalise ensuite une **interruption logicielle** avec l'instruction `svc 0` :

- ▷ le contrôle, le CPU, est transmis au code du traitement de cette interruption ;
- ▷ le **numéro** de l'appel système permet de savoir quel code doit être appelé pour réaliser l'appel système ;
- ▷ le code de traitement de l'interruption et de l'appel système est dans le noyau Linux ;
- ▷ grâce au **mécanisme d'interruption** :
  - ◇ le programme **ne sait pas où se trouve** ce code de traitement : *il n'y a même pas accès pour cause de protection d'accès mémoire* ;
  - ◇ le code de traitement est **exécuté dans un mode protégé** du processeur : *il accède à toutes les ressources de la machine, comme l'écran pour y afficher du texte*.
  - ◇ si le code de traitement de l'appel système est **mis à jour**, il n'y a **pas de problème** : *le programme utilise simplement un numéro pour l'identifier*.

Utilisation de la commande `objdump`

On peut obtenir le désassemblage du programme

```
$ objdump -s -d HelloWorld.o

HelloWorld.o:     file format elf32-littlearm

Contents of section .text:
 0000 0100a0e3 14109fe5 0d20a0e3 0470a0e3  ......... ...p..
 0010 000000ef 0000a0e3 0170a0e3 000000ef  .........p......
 0020 00000000                             ....
Contents of section .data:
 0000 48656c6c 6f20576f 726c6421 0a         Hello World!.
Contents of section .ARM.attributes:
 0000 41110000 00616561 62690001 07000000  A....aeabi......
 0010 0801                                 ..

Disassembly of section .text:

00000000 <_start>:
   0:   e3a00001        mov     r0, #1
   4:   e59f1014        ldr     r1, [pc, #20]   ; 20 <_start+0x20>
   8:   e3a0200d        mov     r2, #13
   c:   e3a07004        mov     r7, #4
  10:   ef000000        svc     0x00000000
  14:   e3a00000        mov     r0, #0
  18:   e3a07001        mov     r7, #1
  1c:   ef000000        svc     0x00000000
  20:   00000000        .word   0x00000000
```

instructions en mode LittleEndian

les différents segments du programme

Le segment de code

mnémonique

valeur réelle de l'instruction

L'intérêt du mode LittleEndian ?
pour convertir un entier sur 4 octets en 1 octet, il suffit de lire que le premier octet.

Ici, on remarque que :

- il n'y a plus d'étiquettes comme dans le source assembleur : elles sont remplacées par des adresses ;
- les **instructions** sont indiquées avec **leur valeur réelle**, indiquée en **hexadécimal** en plus de leur notation sous forme de **mnémonique**, comme `e3a00001` pour `mov r0, #1`.

Analyse du désassemblage

Décomposition d'une instruction

```
00000000 <_start>:
   0:   e3a00001        mov     r0, #1
   4:   e59f1014        ldr     r1, [pc, #20]   ; 20 <_start+0x20>
   8:   e3a0200d        mov     r2, #13
   c:   e3a07004        mov     r7, #4
  10:   ef000000        svc     0x00000000
```

Soit l'instruction `e3a00001 movr0, #1`:

Hex Digit	e	3	a	0	0	0	0	1
Binary	1110	0011	1100	0000	0000	0000	0000	0001

- Chaque instruction du code commence par un digit hexa à `e` :
  - champs sur 4bits indiquant une exécution conditionnelle suivant les valeurs du registre CSPR ;
  - *ici, la valeur est `e` qui indique que l'instruction doit être réalisée de manière inconditionnelle*
- les 3bits suivants `001` indique le **type des opérandes** :
  - *ici, un registre et une valeur immédiate* ;
- les 4bits suivants `1110` est l'**opcode** pour l'instruction `MOV` ;
- le bit suivant `0` indique le **type du paramètre** pour le mode immédiat, qui ici ne sert pas ;
- les 4bits suivants `0000` indiquent le **registre** R0 ;
- les 4bits suivants `0000` indiquent un **autre registre** dans le cas où le `MOV` travaillerait sur deux registres *ce qui n'est pas le cas ici* ;
- les 12bits restants : la **valeur immédiate**, *ici* `1`.

Assembleur ARM

Le source :

```
_start: mov R0, #1 @ 1 = StdOut
        ldr R1, =helloworld @ string to print
        mov R2, #13
        mov R7, #4
        svc 0 @ Call linux to print
```

est devenu :

```
00000000 <_start>:
   0:   e3a00001        mov     r0, #1
   4:   e59f1014        ldr     r1, [pc, #20]   ; 20 <_start+0x20>
   8:   e3a0200d        mov     r2, #13
   c:   e3a07004        mov     r7, #4
  10:   ef000000        svc     0x00000000
```

`ldr R1, =helloworld` est devenu `ldr r1, [pc, #20]; 20 <_start+0x20>`

L'assembleur :

- ▷ l'étiquette `=helloworld` a été traduite en un décalage, «*offset*», depuis le «*program counter*» ;
- ▷ la chaîne de caractères a été placée 20*4 octets plus loin que l'origine du programme indiqué par l'étiquette `_start`, c-à-d :

  ```
  1c:   ef000000        svc     0x00000000
  20:   00000000        .word   0x00000000
  ```

  juste après le code.

- ▷ l'accès à cette mémoire est fait depuis un registre contenant déjà une adresse, car une seule instruction ARM ne permet pas de charger une **adresse entière sur 32bits** directement dans un registre.

Utilisation de gdb

```
$ gdb helloword
GNU gdb (Raspbian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
Reading symbols from helloword...
(gdb) list
1
2   @
3   @ Assembler program to print "Hello World!"
4   @ to stdout.
5   @
6   @ R0-R2 - parameters to linux function services
7   @ R7 - linux function number
8   @
9
10   .global _start @ Provide program starting
(gdb) l
11   @ address to linker
12   @ Set up the parameters to print hello world
13   @ and then call Linux to do it.
14   _start: mov R0, #1 @ 1 = StdOut
15      ldr R1, =helloworld @ string to print
16      mov R2, #13
17      mov R7, #4
18      svc 0 @ Call linux to print
19
20   @ Set up the parameters to exit the program
(gdb) l
21   @ and then call Linux to do it.
22   @ length of our string
23   @ linux write system call
24      mov R0, #0  @ Use 0 return code
25      mov R7, #1  @ Service command code 1
26               @ terminates this program
27      svc 0       @ Call linux to terminate
28   .data
29   helloworld: .ascii "Hello World!\n"
(gdb)
```

GDB	
Command (short form)	*Description*
list (l)	*List the program*
break (b) line	*Set breakpoint at line*
run (r)	*Run the program*
step (s)	*Single-step program*
continue (c)	*Continue running the program*
quit (q or control-d)	*Exit gdb*
control-c	*Interrupt the running program*
info registers (i r)	*Print out the registers*
info break	*Print out the breakpoints*
delete n	*Delete breakpoint n*
x /nuf expression	*Show contents of memory*

Utilisation de gdb : désassemblage et pose de point d'arrêt, «*breakpoints*»

xterm

```
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x00010074 <+0>:    mov    r0, #1
   0x00010078 <+4>:    ldr    r1, [pc, #20] ;
                      0x10094 <_start+32>
   0x0001007c <+8>:    mov    r2, #13
   0x00010080 <+12>:    mov    r7, #4
   0x00010084 <+16>:    svc    0x00000000
   0x00010088 <+20>:    mov    r0, #0
   0x0001008c <+24>:    mov    r7, #1
   0x00010090 <+28>:    svc    0x00000000
   0x00010094 <+32>:    muleq   r2, r8, r0
End of assembler dump.
(gdb)
```

décalage

xterm

```
(gdb) b _start
Breakpoint 1 at 0x10074: file HelloWorld.s, line 14.
(gdb) r
Starting program:
/home/pi/ASMRASPI/hello_world_asm/helloword
Breakpoint 1, _start () at HelloWorld.s:14
14    _start: mov R0, #1 @ 1 = StdOut
(gdb) s
15      ldr R1, =helloworld @ string to print
(gdb) info registers
r0             0x1                 1
r1             0x0                 0
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff620          0x7efff620
lr             0x0                 0
pc             0x10078             0x10078 <_start+4>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00010074
HelloWorld.s:14
   breakpoint already hit 1 time
```

décalage

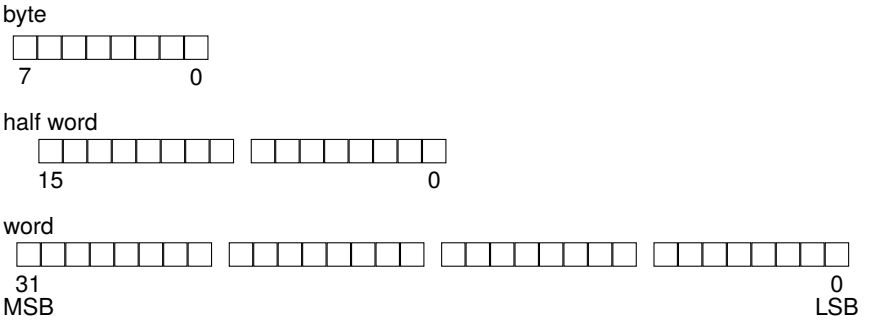
Utilisation de gdb : exécution, affichage registres, affichage mémoire

xterm

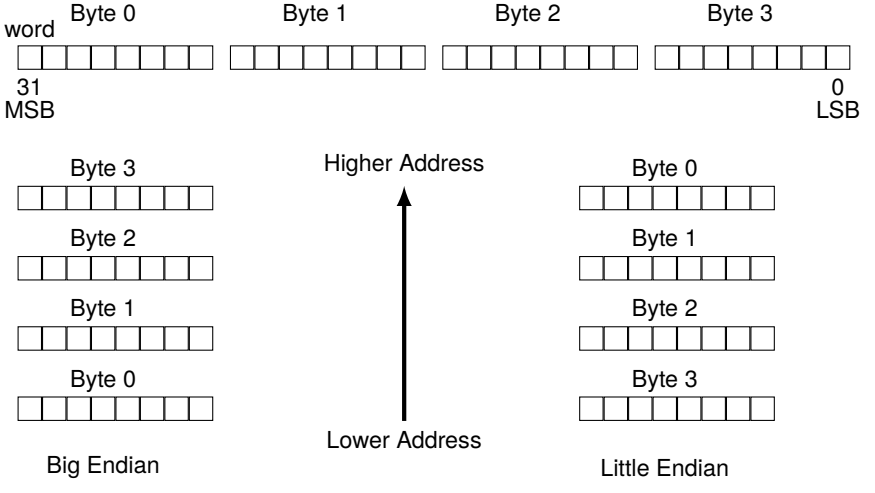
```
(gdb) r
Starting program: /home/pi/ASMRASPI/hello_world_asm/helloword

Breakpoint 1, _start () at HelloWorld.s:14
14   _start: mov R0, #1 @ 1 = StdOut
(gdb) s
15      ldr R1, =helloworld @ string to print
(gdb) info registers
r0             0x1                 1
r1             0x0                 0
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff620          0x7efff620
lr             0x0                 0
pc             0x10078             0x10078 <_start+4>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00010074 HelloWorld.s:14
   breakpoint already hit 1 time
(gdb) x /4ubft _start
0x10074 <_start>:   00000001   00000000   10100000   11100011
(gdb) x /4ubfi _start
   0x10074 <_start>:   mov   r0, #1
=> 0x10078 <_start+4>:   ldr   r1, [pc, #20]   ; 0x10094 <_start+32>
   0x1007c <_start+8>:   mov   r2, #13
   0x10080 <_start+12>:   mov   r7, #4
(gdb) x /4ubfx _start
0x10074 <_start>:   0x01   0x00   0xa0   0xe3
(gdb) x /4ubfd _start
0x10074 <_start>:   1   0   -96   -29
```

Architecture ARM et organisation mémoire

ARM family	ARM architecture
ARM7	ARM v4
ARM9	ARM v5
ARM11	ARM v6
Cortex-A	ARM v7-A
Cortex-R	ARM v7-R
Cortex-M	ARM v7-M



Big Endian vs Little Endian



Instructions et accès mémoire :

ldr	Load Word
ldrh	Load unsigned Half Word
ldrsh	Load signed Half Word
ldrb	Load unsigned Byte
ldrsb	Load signed Bytes
str	Store Word
strh	Store unsigned Half Word
strsh	Store signed Half Word
strb	Store unsigned Byte
strsb	Store signed Byte

Retour sur les registres

Number	Alias	Purpose
R0	–	General purpose
R1	–	General purpose
R2	–	General purpose
R3	–	General purpose
R4	–	General purpose
R5	–	General purpose
R6	–	General purpose
R7	–	Holds Syscall Number
R8	–	General purpose
R9	–	General purpose
R10	–	General purpose
R11	FP	Frame Pointer
Special Purpose Registers		
R12	IP	Intra Procedural Call
R13	SP	Stack Pointer
R14	LR	Link Register
R15	PC	Program Counter
CPSR	–	Current Program Status Register

Le registre d'état ou CPSR, «*Current Program Status Register*»

31	30	29	28	27	-	24	-	19 - 16	-	9	8	7	6	5	4 - 0
N	Z	C	V	Q		J		GE		E	A	I	F	T	M

- **N**egative : N vaut 1 si la valeur est négative, et 0 si la valeur est positive ;
- **Z**ero : Z vaut 1 si le résultat est zéro (par exemple lors d'une comparaison), et 1 si le résultat n'est pas zéro ;
- **C**arry : indique pour
  - une opération d'addition s'il y a un bit de retenu comme résultat du calcul, c-à-d un dépassement, «*overflow*», de capacité ;
  - une opération de soustraction s'il y a un bit de retenu, c-à-d un dépassement, «*underflow*», de capacité ;
  - une opération de décalage, le bit qui a été décalé vers l'extérieur ;
- o**V**erflow : pour l'addition et la soustraction, indique un «*overflow*».
  Pour certaines autres instructions, ce bit peut être utilisé pour indiquer une erreur.
- les bits liés aux **interruptions** :
  - **I** : lorsqu'il est à 1 : désactive les IRQ, «*interrupt request*» ;
  - **F** : lorsqu'il est à 1 : désactive les FIQ, «*Fast interrupt request*» (par exemple : traiter les paquets réseau, les mouvements de la souris) ;
  - **A** : lorsqu'il est à 1 : désactive les abandons ;
- les bits liés aux **instructions** :
  - **Thumb** : lorsqu'il est à 1 : indique des instructions compactes sur 16 bits ;
  - **Jazelle** : lorsqu'il est à 1 : mode obsolète pour l'exécution directe de bytecode Java ;
- les **autres** bits :
  - **Q** : lorsqu'il est à 1 : indique un «*underflow*» ;
  - **GE** : contrôle le «*Greater than Equal*» dans le traitement des données SIMD ;
  - **E** : contrôle l'«*endianness*» pour le traitement des données.
- **M** : indique si le processeur est en mode «***user***» ou «***supervisor***».

Branchement sur condition

On consulte le registre CPSR pour déterminer la condition :

`B{condition} label`

Condition Code	Meaning (for cmp or subs)	Status of Flags
EQ	Equal	Z==1
NE	Not Equal	Z==0
GT	Signed Greater Than	(Z==0) && (N==V)
LT	Signed Less Than	N!=V
GE	Signed Greater Than or Equal	N==V
LE	Signed Less Than or Equal	(Z==1) || (N!=V)
CS or HS	Unsigned Higher or Same (or Carry Set)	C==1
CC or LO	Unsigned Lower (or Carry Clear)	C==0
MI	Negative (or Minus)	N==1
PL	Positive (or Plus)	N==0
AL	Always executed	–
NV	Never executed	–
VS	Signed Overflow	V==1
VC	No signed Overflow	V==0
HI	Unsigned Higher	(C==1) && (Z==0)
LS	Unsigned Lower or same	(C==0) || (Z==0)

Le jeu d'instruction ARM

Format d'une instruction

```
MNEMONIC{S}{condition} {Rd}, Operand1, Operand2
```

`MNEMONIC`	Short name (mnemonic) of the instruction
`{S}`	An optional suffix. If S is specified the **condition flags** are **updated** on the result of the operation
`{condition}`	**Condition** that is needed to be met in order for the **instruction** to be **executed** according to bit(s) of the **CPSR**
`{Rd}`	Register (destination) for storing the result of the instruction
`Operand1`	First operand: either a register or an immediate value
`Operand2`	Second (flexible) operand: can be an immediate value (number) or a register with an optional shift

La seconde opérande est flexible, elle peut être sous différentes formes : une valeur immédiate, un registre ou un registre avec un décalage :

`#123`	Immediate value (with limited set of values)
`Rx`	Register x (like R1, R2, R3 ...)
`Rx, ASR n`	Register x with arithmetic shift right by n bits (1 <= n <= 32)
`Rx, LSL n`	Register x with logical shift left by n bits (0 <= n <= 31)
`Rx, LSR n`	Register x with logical shift right by n bits (1 <= n <= 32)
`Rx, ROR n`	Register x with rotate right by n bits (1 <= n <= 31)
`Rx, RRX`	Register x with rotate right by one bit, with extend

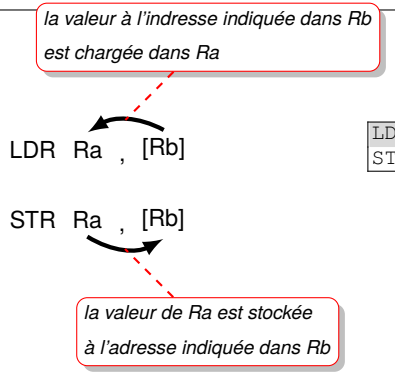
Exemples d'instructions :

`ADD R0, R1, R2`	Adds contents of R1 (Operand1) and R2 (Operand2 in a form of register) and stores the result into R0 (Rd)
`ADD R0, R1, #2`	Adds contents of R1 (Operand1) and the value 2 (Operand2 in a form of an immediate value) and stores the result into R0 (Rd)
`MOVLE R0, #5`	Moves number 5 (Operand2, because the compiler treats it as MOVLE R0, R0, #5) to R0 (Rd) ONLY if the condition LE (Less Than or Equal) is satisfied ⟹ CPSR[N], *negative*, bit at 1 (result of a previous instruction `CMP`)
`MOV R0, R1, LSL #1`	Moves the contents of R1 (Operand2 in a form of register with logical shift left) shifted left by one bit to R0 (Rd). So if R1 had value 2, it gets shifted left by one bit and becomes 4. 4 is then moved to R0.

Les instructions les plus utilisées

Instruction	Description	Instruction	Description
MOV	Move data	EOR	Bitwise XOR
MVN	Move and negate	LDR	Load
ADD	Addition	STR	Store
SUB	Subtraction	LDM	Load Multiple
MUL	Multiplication	STM	Store Multiple
LSL	Logical Shift Left	PUSH	Push on Stack
LSR	Logical Shift Right	POP	Pop off Stack
ASR	Arithmetic Shift Right	B	Branch
ROR	Rotate Right	BL	Branch with Link
CMP	Compare	BX	Branch and eXchange
AND	Bitwise AND	BLX	Branch with Link and eXchange
ORR	Bitwise OR	SWI/SVC	System Call

Instructions Load/Store




```
LDR R2, [R0]    @ [R0] - origin address is the value found in R0.
STR R2, [R1]    @ [R1] - destination address is the value found in R1.
```

Exemple de code

```
.data           /* the .data section is dynamically created and its addresses is not easily known */
var1: .word 3   /* variable 1 in memory */
var2: .word 4   /* variable 2 in memory */

.text           /* start of the text (code) section */
.global _start

_start:
    ldr r0, adr_var1  @ load the memory address of var1 via label adr_var1 into R0
    ldr r1, adr_var2  @ load the memory address of var2 via label adr_var2 into R1
    ldr r2, [r0]      @ load the value (0x03) at memory address found in R0 to register R2
    str r2, [r1]      @ store the value found in R2 (0x03) to the memory address found in R1
    bkpt

adr_var1: .word var1  /* address to var1 stored here */
adr_var2: .word var2  /* address to var2 stored here */
```

Une instruction Load/Store peut utiliser un «*offset*» :

- ▷ avec une valeur immédiate ;
- ▷ la valeur d'un registre (comme sur l'exemple plus haut) ;
- ▷ la valeur d'un registre avec un décalage ;

Assemblage de l'exemple et analyse du code produit

xterm

```
$ arm-none-eabi-as exemple.S
$ ll
total 16K
drwxrwxr-x  2 pef pef 4.0K Nov  6 20:18 ./
drwxr-x--- 85 pef pef 4.0K Nov  6 20:18 ../
-rw-rw-r--  1 pef pef  772 Nov  6 20:18 a.out
-rw-rw-r--  1 pef pef  741 Nov  6 20:17 exemple.S
$ arm-none-eabi-objdump -d a.out

a.out:     file format elf32-littlearm

Disassembly of section .text:

00000000 <_start>:
   0:   e59f000c        ldr     r0, [pc, #12]   @ 14 <adr_var1>
   4:   e59f100c        ldr     r1, [pc, #12]   @ 18 <adr_var2>
   8:   e5902000        ldr     r2, [r0]
   c:   e5812000        str     r2, [r1]
  10:   e1200070        bkpt    0x0000

00000014 <adr_var1>:
  14:   00000000        .word   0x00000000

00000018 <adr_var2>:
  18:   00000004        .word   0x00000004
```

adressage relatif par rapport au registre `pc`

Pour calculer la valeur `12` de l'accès par offset de l'instruction à l'adresse `0`, on sait que :

- ▷ lors de l'exécution de l'instruction à l'adresse `0`, le `pc` est déjà, deux instructions en avant ;
- ▷ chaque instruction est sur 32bits ou 4 octects ;
- ▷ la donnée «*addr_var1*» est à l'adresse 20, «*14 en hexa*», soit un décalage de $20 - (2 * 4) = 12$ par rapport à la valeur courante du `pc`

Pour l'instruction à l'adresse `4`, c'est la même chose avec une donnée «*addr_var2*» en adresse 24, «*18 en hexa*».

Autre exemple

Utilisation d'un registre et d'un offset avec mise à jour du registre

```
.data
var1: .word 3
var2: .word 4

.text
.global _start

_start:
    ldr r0, adr_var1  @ load the memory address of var1 via label adr_var1 into R0
    ldr r1, adr_var2  @ load the memory address of var2 via label adr_var2 into R1
    ldr r2, [r0]      @ load the value (0x03) at memory address found in R0 to register R2
    str r2, [r1, #2]  @ address mode: offset. Store the value found in R2 (0x03) to the memory ad
dress found in R1 plus 2. Base register (R1) unmodified.
    str r2, [r1, #4]! @ address mode: pre-indexed. Store the value found in R2 (0x03) to the memory
address found in R1 plus 4. Base register (R1) modified: R1 = R1+4
    ldr r3, [r1], #4  @ address mode: post-indexed. Load the value at memory address found in R1 to
register R3. Base register (R1) modified: R1 = R1+4
    bkpt

adr_var1: .word var1
adr_var2: .word var2
```

Différents accès

On peut utiliser la valeur d'un registre pour servir d'offset :

```
str r2, [r1, r2]  @ address mode: offset. Store the value found in R2 (0x03) to the memory address
found in R1 with the offset R2 (0x03). Base register unmodified.
```

On peut modifier l'adresse de base **avant** de l'utiliser :

```
str r2, [r1, r2]! @ address mode: pre-indexed. Store value found in R2 (0x03) to the memory address
found in R1 with the offset R2 (0x03). Base register modified: R1 = R1+R2.
```

On peut modifier l'adresse de base **après** de l'utiliser :

```
ldr r3, [r1], r2  @ address mode: post-indexed. Load value at memory address found in R1 to register
R3. Then modify base register: R1 = R1+R2.
```

1 GDB : une meilleure interface avec `dashboard`

L'interface **dashboard** est récupérable à `https://github.com/cyrus-and/gdb-dashboard`

Cette extension de `gdb` est écrite en Python et ajoute les fonctionnalités suivantes :

- la commande `dashboard` affiche un écran composé de :
  -

```
 Assembly
 0x000006e0  ? isb      sy
 0x000006e4  ? cpsid    i
 0x000006e6  ? bx       lr
 0x000006e8  ? ldr      r3, [pc, #52]   ; (0x720)
 0x000006ea  ? ldr      r2, [pc, #56]   ; (0x724)
 0x000006ec  ? ldr      r1, [pc, #56]   ; (0x728)
 0x000006ee  ? mov      r0, sp
 0x000006f0  ? cmp      r0, r2
 0x000006f2  ? bhi.n    0x6fc
 0x000006f4  ? cmp      r0, r3
 Breakpoints
 Expressions
 History
 Memory
 $sp
0x200001b0  00 00 00 00 50 08 00 20 06 00 00 00 8d 06 00 00  ····P·····.....
0x200001c0  00 00 00 00 fd ff ff ff 01 00 00 00 08 08 00 20  ·····.....·····
0x200001d0  00 00 00 10 00 ed 00 e0 08 00 00 00 d3 01 00 00  ·····.....·····
0x200001e0  7c 06 00 00 00 00 00 41 95 01 00 00 00 00 00 00  |·····A·····.....
 Registers
    r0 0x20000828      r4 0x20000808      r8 0x00000008     r12 0x20000770     xPSR 0x410e000e       primask 0x00
    r1 0x20000850      r5 0x00000000      r9 0x00000009      sp 0x200001b0    fpscr 0x00000000       basepri 0x00
    r2 0x10000000      r6 0x20000850     r10 0x0000000a      lr 0x000002b1      msp 0x200001b0     faultmask 0x00
    r3 0x00000000      r7 0x00000007     r11 0x0000000b      pc 0x000006e0      psp 0x200007a8       control 0x00
 Source
 Stack
[0] from 0x000006e0
 Threads
[1] id 0 from 0x000006e0
 Variables
```

GDB : Contrôle de l'exécution

`ctrl-c`	*interrompt l'exécution du program*
`c/continue`	*reprend l'exécution*
`s/step`	*avance d'une insrustion dans une fonction*
`s 10`	*avance de 10 instructions*
`n/next`	*avance à la prochaine instruction dans la fonction*
`u/until 20`	*avance jusqu'à la ligne 20 du fichier courant*
`f/finish`	*avance jusqu'à la fin de la fonction*
`run`	*démarre le programme*
`b/break fonc`	*mets un «breakpoint» lors de l'exécution de la fonc*
`b main.c:fonc`	*mets un «breakpoint» sur fonc du fichier «main.c»*
`b main.c:18 if var > 20`	*breakpoint seulement si var > 20*
`tbreak main`	*se déclenche une fois et s'efface ensuite*
`info breakpoints`	*donne la liste des breakpoints*
`ignore 3 20`	*ignorer 20 fois le breakpoint 3*
`disable 3`	*désactive le breakpoint 3*
`delete 3`	*supprime le breakpoint 3*
`monitor reset halt`	*réinitialise le firmware dans OpenOCD et s'arrête après le reset*

GDB : observer et prendre connaissance

`info locals`	les variables locales
`info variables`	les variables globales
`info args`	les arguments de la fonction
`info registers`	les valeurs des registres

`watch var`	surveille les modifications de var
`watch montableau[10].val`	surveille le champ val de la 11ème structure
`watch *0xdeadcafe`	surveille le contenu de la mémoire par adresse
`watch var if var > 20`	surveillance conditionnelle
`watch var if var - 10 > 20`	avec une expression
`info watchpoints`	donne la liste des watchpoints
`delete 5`	supprime la 5ème surveillance

`bt`	«backtrace»: l'historique des appels de fonction
`frame`	la «frame» courante dans la pile
`up`	remonter dans la pile d'appel
`down`	descendre dans la pile d'appel

GDB : afficher et examiner les contenus

`p/print /FMT expression`	a (address)	o (octal)
	c (char)	t (binary int)
	d (decimal int)	u (unsigned decimal int)
	f (float)	x (hex int)
`p var`	affiche la valeur de var	
`p x+y`	affiche le résultat de l'expression	
`p/x &main`	affiche l'adresse de la fonction main	
`p/x $r4`	affiche le contenu du registre «r4»	
`p/a *(uint32_t[8] *) 0xdeadbabe`	affiche un tableau de 8 entier l'adresse donnée	

`x /FMT adresse`	x (hex)	b (byte)
	d (decimal)	h (halfword 2B)
	u (unsigned decimal)	w (word 4B)
	f (float)	a (address)
	i (instruction)	c (char)
	s (string)	z (padded hex)
`x var`	affiche l'adresse de var	
`x/4c 0xdeadbabe`	affiche 4 char à l'adresse indiquée	
`x/4xw &main`	affiche 4 mots en hexa à l'adresse de main	

GDB : d'autres fonctions

Afficher le source et les instructions machine

`list`	affiche le source à l'endroit courant
`list *0x12341234`	affiche le source à l'adresse indiquée
`list main.c:func`	affiche le source de la fonction définie dans le fichier main.c
`disas func`	desassemble la fonction

Rechercher dans la mémoire

`find /b 0x0,0x10000,'H','e','l','l','o'`	chercher une séquence entre 0x0 et 0x10000
⟹`0x581f 1 pattern found`	
`x/s 0x581f`	examiner la chaine à l'adresse 0x581f
⟹`"Hello world !"`	

Charger la table des Symboles

symbol-file mon_exec.elf	charge un nouveau fichier

Déboguer le SoC

DAP, «Debug Access Port» : l'interface de déboguage

- des **registres** permettant d'effectuer des opérations sur le processeur ;
- des **broches** permettant à un débogueur externe de s'y connecter ;

$\Longrightarrow$ **lire/écrire la mémoire et les registres du processeur.**

La connexion au DAP

- le **JTAG**, «*Joint Test Action Group*», standard industriel IEEE 1149.1 pour le TAP, «*Test Access Port*» :
  - teste les PCBs après leur fabrication ;
  - au moins 4 broches
    - TDI, «*Test Data In*» ;
    - TDO, «*Test Data Out*» ;
    - TCK, «*Test Clock*» ;
    - TMS, «*Test Mode Select*» ;
    - + 1 optionnelle TRST, «Test Reset» ;
  - des registres à décalage + FSM, «*Finite State Machine*» pour échanger des données ;
- le **SWD**, «Serial Wire Debug» :
  - similaire au JTAG mais avec moins de broches :
    - SWDIO : broche d'E/S échantillonée su «front montant» ;
    - SWCLK : référence de temps pour ces E/S ;

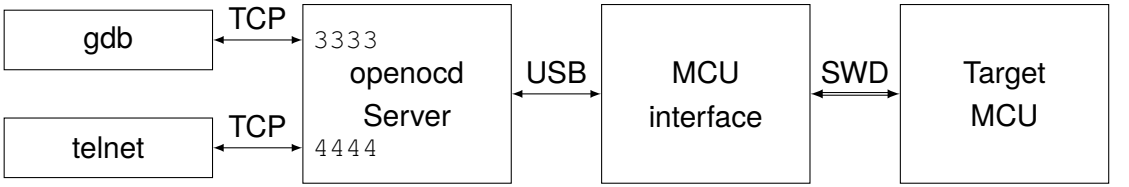
La sonde, ou «*probe*» de déboguage

- **FTDI**, «*Future Technology Devices International*» :
  - pont avec l'USB ;
  - processeur MPSSE, «*Multi Protocol Synchronous Serial Engine*» : UART, JTAG, SWD ;
- **CMSIS-DAP** : version ARM d'un DAPLink :
  - utilise un micro-contrôleur dédié avec un firmware dédié, connecté au micro-contrôleur à déboguer ;
  - le micro-contrôleur dédié assure l'interface USB et le dialogue SWD/JTAG avec l'autre MCU.

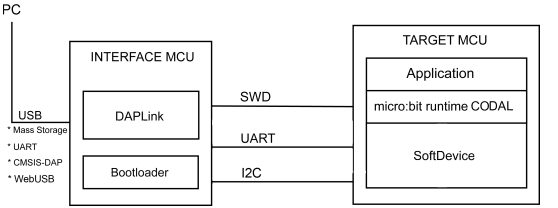
OpenOCD, «*Open On-Chip Debugger*»

OpenOCD assure le contrôle de l'interface de déboguage :

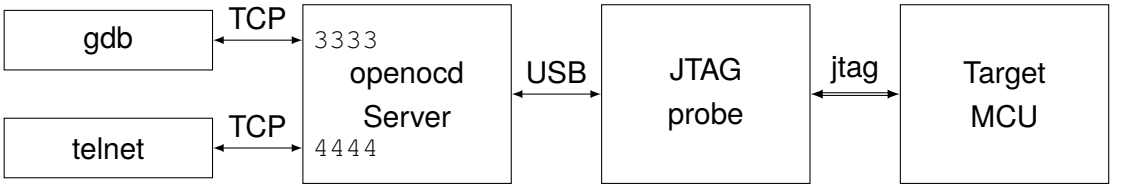
- Version SWD :



- *le SWD en détail sur le microbit v2 :*



- Version JTAG :



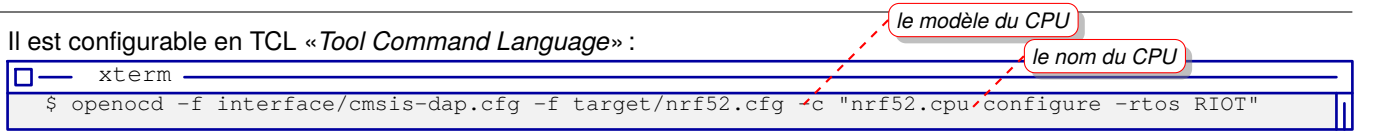
OpenOCD ouvre deux ports TCP :

- ▷ en 4444 : pour le contrôler ;
- ▷ en 3333 : pour les échanges avec gdb ;

On peut aussi contrôler OpenOCD depuis gdb avec la commande «`monitor`*».*

OpenOCD

Il est configurable en TCL «*Tool Command Language*» :



La sortie de la commande :

```
xterm
Open On-Chip Debugger 0.12.0-01004-g9ea7f3d64-dirty (2025-11-12-10:33)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "swd". To override use 'transport select <transport>'.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : Using CMSIS-DAPv2 interface with VID:PID=0x0d28:0x0204,
serial=9906360200052820fa0bc0bc9fe7a1ee000000006e052820
Info : CMSIS-DAP: SWD supported
Info : CMSIS-DAP: Atomic commands supported
Info : CMSIS-DAP: Test domain timer supported
Info : CMSIS-DAP: FW Version = 2.1.0
Info : CMSIS-DAP: Serial# = 9906360200052820fa0bc0bc9fe7a1ee000000006e052820
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 0 TDO = 0 nTRST = 0 nRESET = 0
Info : CMSIS-DAP: Interface ready
Info : clock speed 1000 kHz
Info : SWD DPIDR 0x2ba01477
Info : [nrf52.cpu] Cortex-M4 r0p1 processor detected
Info : [nrf52.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for nrf52.cpu on 3333
Info : Listening on port 3333 for gdb connections
Info : accepting 'gdb' connection on tcp/3333
Error: No symbols for RIOT
Info : nRF52833-xxAA(build code: A0) 512kB Flash, 128kB RAM
```

Ici, on lance OpenOCD avec les oprtions suivantes :

- ▷ *utilisation de l'interface* `CMSIS-DAP` ⟹ *utilisation du microbit ;*
- ▷ *le microbit v2 utilise un Cortex M0 de chez nrf, le 52833, d'où le fichier* `nrf52.cfg` *;*
- ▷ *on utilise les connaissances supplémentaires pour l'analyse d'un RTOS avec l'option* `-rtos RIOT`*.*

OpenOCD

Le script `nrf52.cfg`:

```
#
# Nordic nRF52 series: ARM Cortex-M4 @ 64 MHz
#

source [find target/swj-dp.tcl]

if { [info exists CHIPNAME] } {
    set _CHIPNAME $CHIPNAME
} else {
    set _CHIPNAME nrf52
}

# Work-area is a space in RAM used for flash programming
# By default use 16kB
if { [info exists WORKAREASIZE] } {
   set _WORKAREASIZE $WORKAREASIZE
} else {
   set _WORKAREASIZE 0x4000
}

if { [info exists CPUTAPID] } {
    set _CPUTAPID $CPUTAPID
} else {
    set _CPUTAPID 0x2ba01477
}

swj_newdap $_CHIPNAME cpu -expected-id $_CPUTAPID
dap create $_CHIPNAME.dap -chain-position $_CHIPNAME.cpu

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME cortex_m -dap $_CHIPNAME.dap

adapter speed 1000ZZ
...
```

nommage du CPU

Des usages avancés

Désassemblage et code

xterm

```
(gdb) disassemble /s main
>>> disassemble /s main
Dump of assembler code for function main:
/home/pef/PROJECTS/EMBEDDED/JTAG/RIOT/examples/networking/dtls/dtls-echo/main.c:
35      {
36          /* we need a message queue for the thread running the shell in order to
37           * receive potentially fast incoming networking packets */
38          msg_init_queue(_main_msg_queue, MAIN_QUEUE_SIZE);
   0x000007a0 <+0>:     add.w   r1, r0, #8

35      {
   0x000007a4 <+4>:     ldr.w   r0, [r3, r2, lsl #2]

36          /* we need a message queue for the thread running the shell in order to
37           * receive potentially fast incoming networking packets */
38          msg_init_queue(_main_msg_queue, MAIN_QUEUE_SIZE);
   0x000007a8 <+8>:     cbz     r0, 0x7ca <main+42>
   0x000007aa <+10>:    ldr     r0, [r0, #0]
```

```
>>> disassemble /r main
Dump of assembler code for function main:
   0x000007a0 <+0>:     f100 0108       add.w   r1, r0, #8
   0x000007a4 <+4>:     f853 0022       ldr.w   r0, [r3, r2, lsl #2]
   0x000007a8 <+8>:     b178            cbz     r0, 0x7ca <main+42>
   0x000007aa <+10>:    6800            ldr     r0, [r0, #0]
```

les octets sont en «instruction order», ici en «little endian»

```
>>> disassemble /b main
Dump of assembler code for function main:
   0x000007a0 <+0>:     00 f1 08 01     add.w   r1, r0, #8
   0x000007a4 <+4>:     53 f8 22 00     ldr.w   r0, [r3, r2, lsl #2]
   0x000007a8 <+8>:     78 b1           cbz     r0, 0x7ca <main+42>
   0x000007aa <+10>:    00 68           ldr     r0, [r0, #0]
```

les octets sont en «memory order»

Afficher les contenus des registres

xterm

```
>>> p $sp
$1 = (void *) 0x20000200 <remote>
>>> p $pc
$2 = (void (*)()) 0xb1c <_msg_send+16>
```

Modifier le contenu de registre

utiliser un symbole

```
>>> set $pc = main
>>> p $pc
$3 = (void (*)()) 0x7a0 <main>
>>> set $pc = 0x20000200
>>> p $pc
$4 = (void (*)()) 0x20000200 <remote>
```

```
>>> info files
Symbols from "/home/pef/PROJECTS/EMBEDDED/JTAG/RIOT/examples/networking/dtls/dtls-echo/bin/microbit-v2/dtls_echo.elf".
Extended remote target using gdb-specific protocol:
        `/home/pef/PROJECTS/EMBEDDED/JTAG/RIOT/examples/networking/dtls/dtls-echo/bin/microbit-v2/dtls_echo.elf', file type elf32-littlearm.
        Entry point: 0x15a8
        0x00000000 - 0x00015c80 is .text
        0x00015c80 - 0x00015c88 is .ARM.exidx
        0x20000000 - 0x20000200 is .stack
        0x20000200 - 0x20000370 is .relocate
        0x20000370 - 0x20005910 is .bss
        0x20005910 - 0x20005910 is .noinit
```

OpenOCD : support des RTOS

Dans OpenOCD, on active le support du RTOS et on indique la nature du RTOS :

xterm

```
$ openocd -f interface/cmsis-dap.cfg -f target/nrf52.cfg -c "nrf52.cpu configure -rtos RIOT"
```

FreeRTOS symbols

```
pxCurrentTCB, pxReadyTasksLists, xDelayedTaskList1, xDelayedTaskList2,
pxDelayedTaskList, pxOverflowDelayedTaskList, xPendingReadyList,
uxCurrentNumberOfTasks, uxTopUsedPriority, xSchedulerRunning.
```

RIOT symbols

```
sched_threads, sched_num_threads, sched_active_pid, max_threads,
,c_tcb_name_offset.
```

Zephyr symbols

```
_kernel, _kernel_openocd_offsets, _kernel_openocd_size_t_size
```

Obtenir des infos sur RIOT

```
Assembly
0x00000786  sched_run+10 cbnz  r3, 0x7d8 <sched_run+92>
0x00000788  sched_run+12 cbz   r5, 0x790 <sched_run+20>
0x0000078a  sched_run+14 mov   r0, r5
0x0000078c  sched_run+16 bl    0x710 <_unschedule>
0x00000790  sched_run+20 bl    0xbe4 <sched_arch_idle>
0x00000794  sched_run+24 ldr   r3, [r4, #0]
0x00000796  sched_run+26 cmp   r3, #0
0x00000798  sched_run+28 beq.n 0x790 <sched_run+20>
0x0000079a  sched_run+30 movs  r0, #0
0x0000079c  sched_run+32 ldr   r2, [pc, #68]   @ (0x7e4 <sched_run+104>)
Breakpoints
[6] break at 0x0000086c in /home/pef/RIOT/core/include/thread.h:417
     for sched.c:sched_switch
Expressions
History
$$2 = 0x200002ac <receiver_stack+12>: 536871596
$$1 = 0x200002c0 <receiver_stack+32>: 536871616
$$0 = <optimized out>
Memory
Registers
r0 0x20000e70     r4 0x200014a8     r8 0x00000008    r12 0x00000000     xPSR 0x6100000e      primask 0x01
r1 0x200014f0     r5 0x20000e70     r9 0x00000009     sp 0x200001b0    fpscr 0x00000000     basepri 0x00
r2 0x20000aa0     r6 0x200014f0    r10 0x0000000a     lr 0x00000795     msp 0x200001b0    faultmask 0x00
r3 0x00000000     r7 0x00000000    r11 0x0000000b     pc 0x00000794     psp 0x20000e08      control 0x00
Source
155             active_thread = NULL;
156         }
157
158         do {
159             sched_arch_idle();
160         } while (!runqueue_bitcache);
161     }
162
163     sched_context_switch_request = 0;
164
Stack
[0] from 0x00000794 in sched_run+24 at /home/pef/RIOT/core/sched.c:160
[1] from 0x00000b94 in isr_pendsv+12 at /home/pef/RIOT/cpu/cortexm_common/thread_arch.c:306
```

Obtenir des infos sur RIOT

```
 Threads
[1] id 0 from 0x00000794 in sched_run+24 at /home/pef/RIOT/core/sched.c:160
 Variables
loc active_thread = 0x0 <tsrb_add>: {sp = 0x20000200 <heap_top> "1\033",status = 153,priority = 12
'\f',pid = 0…, previous_thread = 0x20000e70 <cipher_stack+976>: {sp = 0x20000de4 <cipher_stack+836>
"\377\377\377\377\360\02…, nextrq = <optimized out>, next_thread = <optimized out>

>>> p active_thread
$22 = (thread_t *) 0x0 <tsrb_add>
>>> ptype tsrb_add
type = int (tsrb_t *, const uint8_t *, size_t)
>>> ptype/o tsrb_add
type = int (tsrb_t *, const uint8_t *, size_t)
>>> p *active_thread
$23 = {
  sp = 0x20000200 <heap_top> "1\033",
  status = 153,
  priority = 12 '\f',
  pid = 0,
  rq_entry = {
    next = 0xc35 <nmi_handler>
  },
  wait_data = 0xbf1 <hard_fault_default>,
  msg_waiters = {
    next = 0xc45 <mem_manage_default>
  },
  msg_queue = {
    read_count = 3157,
    write_count = 3173,
    mask = 0
  },
  msg_array = 0x0 <tsrb_add>,
  stack_start = 0x0 <tsrb_add>,
  name = 0x0 <tsrb_add>,
  stack_size = 3029
}
```

Obtenir des infos sur RIOT

```
>>> ptype/o thread_t
type = struct _thread {
/*       0       |       4 */    char *sp;
/*       4       |       1 */    thread_status_t status;
/*       5       |       1 */    uint8_t priority;
/*       6       |       2 */    kernel_pid_t pid;
/*       8       |       4 */    clist_node_t rq_entry;
/*      12       |       4 */    void *wait_data;
/*      16       |       4 */    list_node_t msg_waiters;
/*      20       |      12 */    cib_t msg_queue;
/*      32       |       4 */    msg_t *msg_array;
/*      36       |       4 */    char *stack_start;
/*      40       |       4 */    const char *name;
/*      44       |       4 */    int stack_size;

                                  /* total size (bytes):   48 */
                                }
>>>
```

ARM Semi-hosting : exécuter du code sans périphérique

- mécanisme permettant à un processeur ARM d'utiliser les ressources de l'hôte pour ses E/S ;

$\Longrightarrow$ *Très utile lors du développement, lorsque l'on ne dispose pas d'une UART, d'un écran ou de système de fichier ;*

- fonctionnement :
  - ▷ la cible charge des valeurs dans les registres :
    - ⋆ `r0` : la nature de l'ordre, par exemple `0x04` pour écrire un message sur la sortie standard ;
    - ⋆ `r1` : un paramètre pour l'ordre donné, par exemple l'adresse de la chaîne pour la sortie ;
  - ▷ la cible exécute une instructions BKPT en ARMv7 (Cortex-M) :
  - ▷ le débogueur, «*gdb*», les intercepte ;
  - ▷ le débogeur exécute les ordres demandés sur l'hôte ;
  - ▷ un résultat est inséré en retour dans les registres de la cible et un résultat peut être affiché dans la sortie d'OpenOCD (si un affichage a été demandé).
- Inconvénient : ralentit le code ;

Attention

L'instruction «`BKPT`» ne fonctionne que si un débogueur matériel comme OpenOCD+SWD est connecté.

$\Longrightarrow$Sinon, le programme plante !

Exemple de Semi-hosting

```
.syntax unified         @ Use unified assembly syntax
.cpu cortex-m4          @ Target Cortex-M4 (nRF52833)
.thumb                  @ Use Thumb instruction set

.section .text
.global _start          @ Entry point for the linker

_start:
@ Initialize stack pointer (adjust based on linker script)
ldr     r0, =0x20010000  @ Example stack top (64KB SRAM, adjust per linker)
mov     sp, r0

@ Prepare semihosting call for SYS_WRITE0
movs    r0, #0x04       @ SYS_WRITE0 operation code
ldr     r1, =message    @ Pointer to null-terminated string
bkpt    0xAB            @ Trigger semihosting breakpoint

@ Infinite loop to keep program running
loop:
b       loop

.section .rodata
message:
.asciz "Hello from micro:bit v2 via semihosting!\n"  @ Null-terminated string
```

L'appel du semi-hosting

Dans gdb :

```
xterm
>>> monitor arm semihosting enable
```

Dans OpenOCD :

```
xterm
Info : [nrf52.cpu] Cortex-M4 r0p1 processor detected
Info : [nrf52.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for nrf52.cpu on 3333
Info : Listening on port 3333 for gdb connections
Info : accepting 'gdb' connection on tcp/3333
Info : nRF52833-xxAA(build code: A0) 512kB Flash, 128kB RAM
Hello from micro:bit v2 via semihosting!
```

Écrire et exécuter un programme en RAM seulement

On va modifier le fichier utilisé par le linker :

```
MEMORY
{
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 128K
}

SECTIONS
{
    .text :
    {
        *(.text*)    /* Code */
        *(.rodata*) /* Read-only data (e.g., string for SYS_WRITE0) */
    } > RAM

    .bss :
    {
        *(.bss*)     /* Uninitialized data */
    } > RAM

    /* Stack at the top of SRAM */
    _stack_top = ORIGIN(RAM) + LENGTH(RAM);
}
```

Seule la RAM est définie

On place le début de la pile à la fin de la mémoire

- Une seule zone mémoire : la RAM ;
- placement des sections `.text`, `.rodata` et `.bss` en RAM dans cet ordre ;
- inutile d'initialiser la mémoire de la section `.bss` dans le code.

Attention

La table des vecteurs d'interruption est toujours dans la flash...

Juste en RAM

```
.syntax unified
.cpu cortex-m4
.thumb

.section .text
.global _start

_start:
/* Initialize stack pointer to top of SRAM */
ldr     r0, =_stack_top  /* Defined in linker script (0x20020000) */
mov     sp, r0

/* Semihosting call for SYS_WRITE0 */
movs    r0, #0x04        /* SYS_WRITE0 operation code */
ldr     r1, =message     /* Pointer to string in RAM */
bkpt    0xAB             /* Trigger semihosting */

/* Infinite loop */
loop:
b       loop

.section .rodata
message:
.asciz "Hello from micro:bit v2 via semihosting in RAM!\n"
```