PLM

Pierre Molinaro

24 mai 2015

Table des matières

Liste des tableaux

Liste des figures

Chapitre 1

Introduction

1.1 Cible

Chapitre 2

Le type booléen

2.1 Le type `Bool`

L'identificateur `Bool` dénote le type booléen. Sa taille est fixée par la définition de la cible.

2.2 Les mots réservés `true` et `false`

Les mots réservés `true` et `false` dénotent respectivement la valeur logique *vraie* et la valeur logique *fausse*.

2.3 Les opérateurs infix de comparaison

Les valeurs booléennes sont comparables, les six opérateurs `==`, `!=`, `>=`, `>`, `<=` et `<` sont acceptés, avec `false` < `true`.

2.4 Les opérateurs infixes `and`, `or` et `xor`

Les opérateurs infixes `and`, `or` et `xor` implémentent respectivement le *et* logique, *ou* logique, *ou exclusif* logique. Les deux premiers évaluent les opérandes en *court-circuit*, c'est-à-dire que si la valeur de l'opérande de gauche détermine la valeur de l'expression, alors l'opérande de droite n'est pas évalué.

Noter que les opérateurs infixes `&`, `|` et `^` sont des opérateurs bit-à-bit sur les entiers non signés.

2.5 L'opérateur préfixé `not`

L'opérateur préfixé `not` est la complémentation booléenne. Noter que l'opérateur préfixé `~` effectue la complémentation bit-à-bit d'un entier non signé.

2.6 Conversion en une valeur entière

2.7 Conversion d'une valeur entière en booléen

Il n'y a pas d'opérateur dédié à la conversion d'une valeur entière vers un booléen. Il suffit d'utiliser des opérateurs entre entiers comme `==` ou `!=` pour réaliser une conversion :

```
let result : Bool = x != 0 // x est une expression entière
```

Chapitre 3

Les types entiers

Chapitre 4

Les registres de contrôle

La déclaration d'un registre de contrôle obéit à une syntaxe particulière, ne serait-ce que parce que son adresse absolue doit y être spécifiée. Pour de nombreux registres, un bit ou un groupe de bits ont une signification particulière, et obtenir la valeur d'un champ ou modifier sa valeur est une opération courante.

À titre d'exemple, nous allons nous intéresser au registre `ICSR` du processeur ARMv7-M. Le *manuel de référence de l'architecture ARMv7-M* [1] décrit ce registre comme indiqué à la figure 4.1, et indique que son adresse est `0xE000ED04`.

4.1 Simple déclaration d'un registre

Pour déclarer le registre `ICSR` (figure 4.1), on écrira simplement :

```
register ICSR at 0xE000_ED04 : UInt32
```

Le type `UInt32` qui est mentionné signifie que les valeurs écrites et lues de ce registre sont des entiers non signés de 32 bits. Tout type entier, signé ou non signé est autorisé.

Pour lire ou écrire ce registre, on le nomme comme s'il s'agissait d'une simple variable. Par exemple, pour activer l'interruption `PendSV`, il faut mettre à 1 le bit `PENDSVSET`. On écrit donc :

```
ICSR = 1 << 28
```

Si on voulait activer simultanément les interruptions `PendSV` et `SysTick`, il faut mettre à 1 les bits `PENDSVSET` et `PENDSTSET`. On écrit donc :

1. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403e.b/index.html

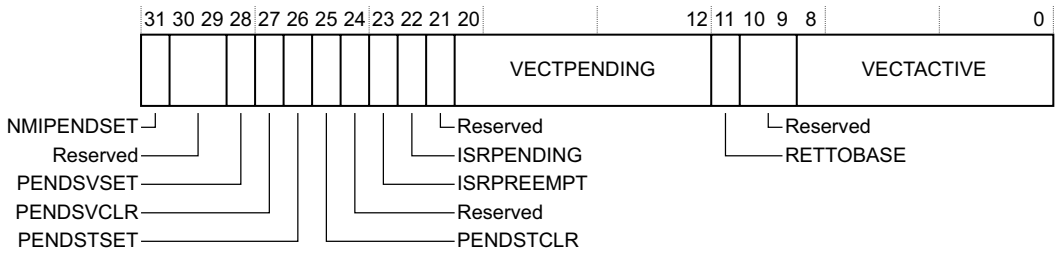

***Figure 4.1** – Registre de contrôle* `ICSR` *intégré dans l'ARMv7*

```
ICSR = (1 << 28) | (1 << 26)
```

Pour savoir si le bit `RETTOBASE` est activé, on écrit :

```
let RETTOBASEActif : Bool = (ICSR & (1 << 11)) != 0
```

Pour accéder à la valeur du champ `VECTPENDING`, on réalise un masquage :

```
let vectPending : UInt32 = ICSR & 0x1F_F000
```

Et si on veut la valeur de champ justifiée à droite :

```
let vectPending : UInt32 = (ICSR & 0x1F_F000) >> 12
```

Ces écritures peuvent être rendues plus intelligibles en précisant la composition du registre `ICSR` dans sa déclaration. C'est ce qui va être réalisé dans la section suivante.

4.2 Déclaration d'un registre et de ses champs

Lors de la déclaration d'un registre, il est possible de préciser la composition de ses champs entiers et booléens :

```
register ICSR at 0xE000_ED04 : UInt32 {
  NMIPENDSET, 2, PENDSVSET, PENDSVCLR, PENDSTSET, PENDSTCLR, 1,
  ISRPREEMPT, ISRPENDING, 1, VECTPENDING[9], RETTOBASE, 2, VECTACTIVE[9]
}
```

Entre accolades, trois définitions différentes peuvent apparaître :

— un nombre indique le nombre de bits consécutifs inutilisés ;

— un identificateur (par exemple `NMIPENDSET`) nomme un champ booléen ;

— un identificateur suivi d'un nombre entre crochets (par exemple `VECTPENDING[9]`) nomme un champ entier constitué du nombre indiqué de bits consécutifs.

La description commence par le bit le plus significatif : comme le type du registre est `UInt32` (entier non signé sur 32 bits), le premier bit nommé `NMIPENDSET` porte le n°31, `PENDSVSET` le n°28, ...

Cette écriture n'est autorisée que si le type nommé (ici `UInt32`) est une type entier non signé. Les types signés (`Int32`, ...) sont interdits. Le compilateur vérifie que la description des champs définit exactement le nombre de bits du type nommé, ici les 32 bits du type `UInt32`.

4.2.1 Accès en lecture aux champs booléens

Pour accéder à la valeur d'un champ, on utilise la notation pointée en nommant ce champ :

```
let x : UInt32 = ICSR.ISRPENDING // 0 ou 2**22
```

Ceci effectue simplement un masquage de façon à isoler le bit demandé. Aucun décalage n'est réalisé. Comme le bit `ISRPENDING` est le 22^{e}, le résultat est 0 ou 2^{22}.

Si l'on veut obtenir un résultat justifié à droite, on utilise en plus l'accesseur `.shift` :

```
let x : UInt32 = ICSR.ISRPENDING.shift // 0 ou 1
```

L'accesseur `.bool` permet d'obtenir une valeur booléenne correspondant à la valeur d'un bit d'un registre :

```
let x : Bool = ICSR.ISRPENDING.bool // false ou true
```

4.2.2 Accès en lecture aux champs entiers

Comme pour un champ booléen, l'accès à un champ entier s'effectue par la notation pointée `.`. Par exemple :

```
let x : UInt32 = ICSR.VECTPENDING
```

`ICSR.VECTPENDING` applique un masquage de la valeur de `ICSR` pour ne conserver que les bits correspondants au champ `VECTPENDING`. Aucun décalage n'est effectué.

Pour obtenir une valeur justifiée à droite, on ajoute l'accesseur `.shift` :

```
let x : UInt32 = ICSR.VECTPENDING.shift
```

La valeur renvoyée est alors comprise en 0 et $2^9 - 1$.

4.2.3 Constantes associées aux champs booléens

Le délimiteur `::` permet de définir des constantes correspondant aux bits d'un registre. Par exemple, pour activer l'interruption `PendSV`, il faut mettre à 1 le bit `PENDSVSET`. On écrit donc :

```
ICSR = ICSR::PENDSVSET
```

La constante `ICSR::PENDSVSET` a le type du registre `ICSR`, c'est-à-dire `UInt32`.

De façon analogue, si on voulait activer simultanément les interruptions `PendSV` et `SysTick`, il faut mettre à 1 les bits `PENDSVSET` et `PENDSTSET`. On écrit donc :

```
ICSR = ICSR::PENDSVSET | ICSR::PENDSTSET
```

4.2.4 Expressions associées aux champs entiers

Le délimiteur `::` permet de simplifier la composition d'une valeur correspondant à un champ entier. Par exemple :

```
let y : UInt32 = ICSR::VECTPENDING (x) // x : expression entière non si-
gnée
```

Le champ `VECTPENDING` est un champ de 9 bits, il peut donc accepter une valeur entière entre 0 et 511. L'expression `x` doit être de type entier non signé. Plusieurs cas sont à considérer :

- si `x` est une expression statique, alors le compilateur vérifie que sa valeur est comprise entre 0 et 511 (un message d'erreur de compilation est émis dans le cas contraire) ; l'expression `ICSR::VECTPENDING (x)` est alors aussi une expression statique ;
- si `x` est une expression non calculable statiquement :
  - si `x` est du type `UInt8`, alors toute valeur de `x` est acceptable : le code engendré se borne à faire le décalage à gauche de la valeur de `x` ;
  - sinon, une assertion (code : voir tableau 5.1 page 15) est engendrée ; la valeur de `x` est ensuite masquée pour pallier un éventuel débordement, puis décalée.

D'une manière générale, si `x` n'est pas calculable statiquement, le code engendré ne comprendra pas d'assertion si le nombre de bits du champ est supérieur ou égal au nombre de bits du type entier de l'expression.

4.2.5 Masques associés aux champs entiers

Le délimiteur `::` permet de simplifier la composition d'un masque correspondant à un champ entier. Par exemple :

```
let y : UInt32 = ICSR::VECTPENDING // y vaut 0x1F_F000
```

Le champ `VECTPENDING` est un champ de 9 bits commençant au bit 12. La valeur du masque `ICSR::VECTPENDING` est donc $(2^9 - 1)^{12}$, soit `0x1F_F000`.

Les masques ainsi obtenus sont des expressions statiques.

4.3 Attribut `@ro`

La déclaration d'un registre accepte l'attribut `@ro`, qui signifie qu'il est en lecture seule. Par exemple :

```
register SYST_CALIB @ro at 0xE000_E01C : UInt32
```

Toute tentative de faire figurer ce registre dans une construction qui provoque une écriture de celui-ci entraîne l'apparition d'une erreur de compilation.

Chapitre 5

Exceptions

Numéros	Signification	Lien
0 à 999	Exceptions liées aux vecteurs d'interruption	
1000	Dépassement de capacité de l'incrémentation (`++`)	
1001	Dépassement de capacité de l'incrémentation (`--`)	
1002	Dépassement de capacité de la négation (`-`)	
1003	Dépassement de la construction d'un champ entier d'un registre (`registre::champ (...)`)	section 4.2.4 page 13
1010	Dépassement de capacité de l'addition (`+`)	
1011	Dépassement de capacité de la soustraction (`-`)	
1012	Dépassement de capacité de la multiplication (`*`)	
1013	Dépassement de capacité de la division (`/`)	
1020	Échec de l'instruction `assert`	
1021	Instruction `throw`	

Tableau 5.1 – *Code des exceptions*

Chapitre 6

Configuration d’une cible