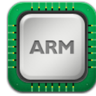


Programmation assembleur ARM

Dr. Eng. Chiheb Ameur ABID

/in/chiheb-ameur-abid
chiheb.abid@gmail.com

Plan

1. Programmation avec le coprocesseur VFP
2. Programmation avec le coprocesseur NEON

Les coprocesseurs

Les coprocesseurs

- Le processeur ARM ont implanté les opérations de la virgule flottante en utilisant une unité de traitement supplémentaire
  1. VFPv1 : Ce coprocesseur est obsolète
  2. VFPv2 : Une extension optionnelle aux provesseurss ARMv5 et ARMv6 processors. VFPv2 dispose de 16 64-bit FPU registres
  3. VFPv3 : Une extension optionnelle au processeur ARMv7. Il est compatible avec VFPv2.
  4. VFPv4 : Il est implémenté sur certains processeurs Cortex ARMv7. VFPv4 possède 32 64-bit FPU registres
  5. NEON : Utilise des instructions SIMD (Single Instruction Multiple Data). NEON supporte les calcul des virgules flottantes avec des instructions arithmétiques sur les entiers permettant d'effectuer plusieurs opérations avec une seule instruction
     - Le jeu d'instructions de Neon partageant les mêmes registres que celui de VFP

Le coprocesseur VFP en bref

Présentation du coprocesseur VFP

- Le coprocesseur VFP est un circuit additionnel au circuit du processeur ARM
  - Circuit séparé (additionnel), i.e. n'est intégré au cœur du processeur ARM
  - Il possède son propre architecture, ses propres registres et instructions
  - Avec Armv8, le VFP est devenu redondant puisque son architecture est devenue intégrée dans celle d'ARMv8
- Le jeu d'instructions d'un VFP est constitué approximativement (ça dépend de la version) de 23 instructions
- Principalement, le VFP permet les calculs en virgule flottante

Représentation en virgule flottante

Représentation en virgule flottante

- La virgule flottante est une méthode d'écriture de nombres
- Elle consiste à représenter un nombre par :
  1. Un signe
  2. Une mantisse (significande)
  3. Un exposant (entier relatif, généralement borné).
- Un nombre en virgule flottante : $signe \times mantisse \times base^{exposant}$
  - Sur les ordinateurs, la base est 2
  - Exemple de nombre en virgule flottante en base 10

$$1.3254 = \underbrace{13254}_{\text{mantisse}} \times 10^{\overbrace{-4}^{\text{exposant}}}$$

  - En faisant varier l'exposant, on fait "flotter" la virgule

Représentation en virgule flottante

Représentation en virgule flottante : la norme IEEE 754

- La norme IEEE 754 élaborée en 1985 et elle est reprise par la norme internationale CEI 605596
- Cette norme définit les spécifications suivantes :
  - Deux formats de nombres en virgule flottante (et deux formats étendus optionnels) en base 2
  - Quelques opérations associées : principalement l'addition, la soustraction, la multiplication, la division et la racine carrée
- La quasi-totalité des architectures d'ordinateurs incluent une implémentation matérielle des calculs sur flottants IEEE
- Les deux formats fixés par la norme IEEE 754 sont :
  1. Simple précision sur 32 bits
  2. Double précision sur 64 bits
- La répartition des bits avec $1 \leq M < 2$

Précision	Encodage	Signe	Exposant	Mantisse	Valeur d'un nombre
Simple précision	32 bits	1 bit	8 bits	23 bits	$(-1)^S \times M \times 2^{(E-127)}$
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^S \times M \times 2^{(E-1023)}$

Les registres du coprocesseur VFP

Les registres du coprocesseur VFP

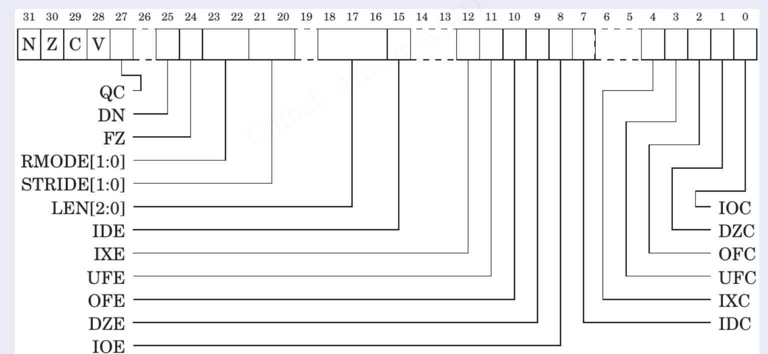
- Le coprocesseur VFP possède 32 registres en simple précision
  - Un registre peut contenir un nombre en virgule flottante avec simple précision, ou bien une valeur entière
  - Ces registres sont nommés de `S0` à `S31`
- Les mêmes registres peuvent être utilisés pour stocker des nombres en double précision
  - Chaque registre en double précision est le résultat de la fusion de deux registres en simple précision
  - Les registres en double précision sont nommés de `D0` à `D15`
- Des registres en double précision supplémentaire de `D16` à `D31` sont disponibles à partir de VFPv4
  - Ils n'ont pas de correspondance en simple précision

s1	s0	d0
s3	s2	d1
s5	s4	d2
s7	s6	d3
s9	s8	d4
s11	s10	d5
s13	s12	d6
s15	s14	d7
s17	s16	d8
s19	s18	d9
s21	s20	d10
s23	s22	d11
s25	s24	d12
s27	s26	d13
s29	s28	d14
s31	s30	d15
		d16
		d17
		d18
		d19
		d20
		d21
		d22
		d23
		d24
		d25
		d26
		d27
		d28
		d29
		d30
		d31

Les registres du coprocesseur VFP

Le registre d'état d'un coprocesseur VFP

- Le registre d'état : Floating Point Status and Control Register (FPSCR)
- Structuration du registre d'état



Programmation avec le coprocesseur VFP

Les types d'instructions du coprocesseur VFP

- Les instructions du coprocesseur VFP permettent les opérations suivantes :
  - Transférer les valeurs en virgule flottante dans les registres du coprocesseur VFP
  - Transférer les valeurs en virgule flottante entre les registres du coprocesseur et la mémoire
  - Transférer les valeurs 32-bit entre les registres du coprocesseur et les registres du processeur ARM
  - Réaliser les opérations arithmétiques : addition, soustraction, multiplication, et division impliquant deux registres sources et un registre destination
  - Calculer la racine carrée d'une valeur
  - Combiner des opérations de multiplication et accumulation
  - Réaliser des conversions entre plusieurs représentations : entier, virgule fixe, virgule flottante
  - Comparer des valeurs en virgule flottante

Programmation avec le coprocesseur VFP

Modes de fonctionnement

- Le coprocesseur VFP fonctionne en trois modes
  1. Mode scalaire
  2. Mode vectoriel
  3. Mode scalaire étendu

Programmation avec le coprocesseur VFP

Les instructions Load et Store

- Chargement/Sauvegarde d'un seul registre VFP
- Syntaxe

```
v<op>r{<cond>}{.<prec>} Fd, [Rn{,#offset}]
v<op>r{<cond>}{.<prec>} Fd, =label
```

  - `<op>` : `ld` ou `st`
  - `fd` : n'importe quel registre VFP (simple ou double précision)
  - `Rn` : peut être n'importe quel registre ARM
  - `<cond>` : suffixe optionnel de condition
  - `<prec>` : peut être soit `f32` ou `f64`

Exemple

```
VLDR S1, [R5] @Load S1 with F32 value at addr in R5
VLDR D2, [R5, #4] @Load D2 with F64 value addr+4 in R5
VSTR S3, [R6] @Store F32 value in S3 at addr in R6
VSTR.F64 D4,[R2] @Store D4 using address in r2
```

Programmation avec le coprocesseur VFP

Les instructions Load et Store

- Chargement/Sauvegarde de plusieurs registres VFP
- Syntaxe

```
v<op>m<mode>{<cond>}{.<prec>} Rn{!},<list>
vpush{<cond>}{.<prec>} <list>
vpop{<cond>}{.<prec>} <list>
```

  - `<op>` : `ld` ou `st`
  - `<mode>` : `ia` pour incrémentation après transfert, ou `db` décrémentation avant transfert
  - `Rn` : peut être n'importe quel registre ARM
  - `<cond>` : suffixe optionnel de condition
  - `<prec>` : peut être soit `f32` ou `f64`
  - `<list>` : un ensemble de registres VFP
  - Si mode `db`, alors `!` est obligatoire
  - `vpop <list>` est équivalent à `vldmia sp!,<list>`
  - `vpush <list>` est équivalent à `vstmdb sp!,<list>`

Programmation avec le coprocesseur VFP

Exemple : les isntuctions Load et Store

```
1 vstmdb.f32 sp!,{s0-s3} @ Store s0 through s3 on stack
2 vstmia.f32 r1,{s0-s31} @ Store all fp registers at address in r1
3
4 vldmia.f64 sp!,{d4-d7} @ Pop four doubles from the stack
5 vldmiaeq.f64 sp!,{d4-d7} @ If eq, then pop four doubles from the stack
```


Partie 3 Chiheb Ameur ABID
Programmation avec le coprocesseur VFP

Programmation avec le coprocesseur VFP

Instructions de traitement de données

- Instructions à deux paramètres
  - `vabs` : calcul de la valeur absolue
  - `vneg` : négation
  - `vsqrt` : calcul de la racine carrée
- Syntaxe

```
v<op>{<cond>}.<prec> Fd, Fm
```

  - `<op>` : `abs`, `neg`, ou `sqrt`
  - `<cond>` : suffixe optionnel de condition
  - `<prec>` : peut être soit `f32` ou `f64`

Exemple

```
1 vabs d3, d5 @ Store absolute value of d1 in d3
2 vnegmi s15, s15 @ if mi, then negate s15
```


Partie 3 Chiheb Ameur ABID
Programmation avec le coprocesseur VFP

Programmation avec le coprocesseur VFP

Instructions de traitement de données

- Instructions à trois paramètres
  - `vadd` : addition
  - `vsub` : soustraction
  - `vmul` : multiplication
  - `vnmul` : négation, puis multiplication
  - `vdiv` : division
- Syntaxe

```
1 v<op>{<cond>}.<prec> Fd, Fn, Fm
```

  - `<op>` : `add`, `sub`, `mul`, `nmul` ou `div`
  - `<cond>` : suffixe optionnel de condition
  - `<prec>` : peut être soit `f32` ou `f64`

Exemple

```
1 vadd.f64 d0, d1, d2 @ d0 <- d1 + d2
2 vaddgt.f32 s0, s1, s2 @ if (gt) then s0 <- s1 + s2
3 vnmul.f32 s10, s10, s14 @ s10 <-- (s10 * s14)
4 vdivlt.f64 d0, d7, d8 @ if lt, then d0 <- d7 / d8
```


Partie 3 Chiheb Ameur ABID
Programmation avec le coprocesseur VFP

Programmation avec le coprocesseur VFP

Comparaison

- L'instruction de comparaison permet de mettre à jour le registre d'état FPSCR par le biais d'une opération de soustraction
- Syntaxe

```
1 vcmp{e}{<cond>}.<prec> Fd, Fm
```

  - Si `e` est présent, une exception est levée dans le cas où le résultat est un NaN
  - `<cond>` : suffixe optionnel de condition
  - `<prec>` : peut être soit `f32` ou `f64`

Exemple

```
vcmp.f32 s0, s1 @ Subtract s1 from s0 and set FPSCR flags
```

Programmation avec le coprocesseur VFP

Transfert entre les registres

- Transfert entre des registres VFP

```
1 vmov{<cond>}{.<prec>} Fd, Fm
```

  - F peut être S ou bien D
  - Fd et Fm doivent être de même taille
  - <cond> : suffixe optionnel de condition
  - <prec> : peut être soit f32 ou f64
- Transfert entre un registre VFP (S) et un registre ARM (32bits)

```
1         vmov{<cond>} Rd,Sn
2         vmov{<cond>} Sn,Rd
```

  - Rd est un registre ARM sur 32 bits
  - Sd est un registre VFP en simple précision
  - <cond> est un suffixe optionnel de condition

Partie 3 Chiheb Ameur ABID
Programmation avec le coprocesseur VFP

Programmation avec le coprocesseur VFP

Transfert entre les registres

- Transfert entre un registre VFP et deux registres ARM

```
vmov{<cond>} destination(s), source(s)
```

  - L'un de la source ou de la destination doit être une liste de deux registres ARM, et l'autre un registre VFP en double précision ou deux registres VFP en simple précision

ARM Integer	Floating Point
Rl,Rh	Dd Sd,Sd'

  - Sd et Sd' sont adjacents tel que d'=d+1
  - <cond> est un suffixe optionnel de condition

Exemple

```
1 vmov d9,r0,r1 @ d9 <- r1:r0
2 vmov r2,r3,d12 @ r3:r2 <- d12
3 vmov s1,s2,r2,r4 @ s1 <- r2, s2 <- r4
4 vmov r5,r7,s0,s1 @ r1 <- s0, r7 <- s1
```

Chiheb Ameur ABID

Partie 3 Chiheb Ameur ABID
Programmation avec le coprocesseur VFP

Programmation avec le coprocesseur VFP

Transfert entre les registres

- Transfert du registre FPSCR

```
1 vmrs{<cond>} Rd, FPSCR
2 vmsr{<cond>} FPSCR, Rd
```

  - Rd peut être APSR_nzcv pour désigner le registre CPSR ou n'importe quel ARM registre
  - <cond> est un suffixe optionnel de condition

Exemple

```
1 vmrs APSR_nzcv,fpscr @ Copy flags from FPSCR to CPSR
2 vmrs r3, FPSCR @ Copy FPSCR flags to R3
3 vmsr FPSCR,r5 @ Copy R5 to FPSCR
```

Chiheb Ameur ABID

Partie 3 Chiheb Ameur ABID
Programmation avec le coprocesseur VFP

Programmation avec le coprocesseur VFP

Instructions de conversion

- Conversion ente le type entier et en virgule flottante

```
1 vcvt{r}{<cond>}.<type>.f64 Sd,Dm
2 vcvt{r}{<cond>}.<type>.f32 Sd,Sm
3 vcvt{<cond>}.f64.<type> Dd,Sm
4 vcvt{<cond>}.f32.<type> Sd,Sm
```

  - L'option r permet d'activer l'arrondissement
  - <cond> est un suffixe optionnel de condition
  - <type> peut être soit u32 ou s32 pour désigner un entier non signé ou signé

Exemple

```
1 vcvt.f64.u32 d5, s7 @ Convert unsigned integer to double
2 vcvt.f64.s32 d0, s4 @ Convert signed integer to double
3 vcvt.u32.f64 s0, d7 @ Convert double to unsigned integer
4 vcvt.s32.f64 s1, d4 @ Convert double to signed integer
5 @@
6 consta: .float 65536.0
7 vldr.f32 s11,consta @ Load floating point constant
8 vmul.f32 s10,s10,s11 @ Multiply equates to shift
9 vcvt.s32.f32 s10,s10 @ Convert single to signed integer
```

Programmation avec le coprocesseur VFP

Instructions de conversion

- Conversion entre virgule fixe et virgule flottante

```
1 vcvt{<cond>}.<td>.f32 Sd,Sm,#fbits
2 vcvt{<cond>}.f32.<td> Sd, Sm, #fbits
```

  - `<cond>` est un suffixe optionnel de condition
  - `<td>` spécifie le type et la taille du nombre en virgule fixe : `u16`, `u32`, `s16` et `s32`
  - L'opérande `#fbits` spécifie le nombre de bits pour la partie après la virgule

Exemple

```
1 vcvt.f32.u16 s0,s0,#4 @ Convert from U(12,4) to single
2 vcvt.s32.f32 s1,s1,#8 @ Convert from single to S(23,8)
```

Programmation avec le coprocesseur VFP

Compilation et débogage

- Pour compiler un programme utilisant le coprocesseur VFP, utiliser le flag `-mfpu=vfp` avec `g++`
  `g++ prog.s -ggdb -mfpu=vfp`
- GDB fournit des outils de débogage supportant le coprocesseur VFP
  - `info r all` visualiser tous les registres y compris ceux du coprocesseur VFP
  - `info r <list>` visualiser la liste de registres désignée par `<list>`
  - `p/fmt $registre` afficher le contenu du registre au format spécifié par `fmt`
    Exemple
    `p/d $s0` : afficher le contenu du registre `s0` sous forme d'un entier
    `p/f $s0` : afficher le contenu du registre `s0` en virgule flottante
    `p/x $s0` : afficher le contenu du registre `s0` hexadécimal

Plan

1 Programmation avec le coprocesseur VFP

2 Programmation avec le coprocesseur NEON

Programmation avec le coprocesseur NEON

Historique d'intégration de SIMD aux processeurs ARM

Armv6	Armv7-A	Armv8-A AArch64
SIMD extension	NEON	NEON
• Operates on 32-bit general purpose ARM registers • 8-bit/16-bit integer • 2x16-bit/4x8-bit operations per instruction	• Separate register bank, 32x64-bit NEON registers • 8/16/32/64-bit integer • Single precision floating point • Up to 16x8-bit operations per instruction	• Separate register bank, 32x128-bit NEON registers • 8/16/32/64-bit integer • Single precision floating point • double precision floating point, both of them are IEEE compliance • Up to 16x8-bit operations per instruction

Programmation avec le coprocesseur NEON

Le coprocesseur NEON

- Le coprocesseur NEON est un circuit spécial permettant de réaliser des opérations arithmétiques
- Un coprocessuer est un processeur simplifié qui possède ses propres registres et son jeu d'instructions
- NEON est basé sur le concept SIMD (Single Instruction Multi Data)
  - Une seule instruction peut traiter plusieurs données en entrées
  - Plusieurs opérations de même type peuvent être réalisées sur des données différents
    Exemple : addition de deux vecteurs
- Utiliser le flag `-mfpu=neon-vfpv4` pour activer NEON lors de la compilation

Programmation avec le coprocesseur NEON

Les registres du coprocesseur NEON

- Les registres du coprocesseur NEON et de l'unité de calcul flottante
- Neon possède un banc de registres qui peut être configuré en deux configurations
  - 32 registres sur 64 bits, appelés Di
  - 16 registres sur 128 bits, appelés Qi

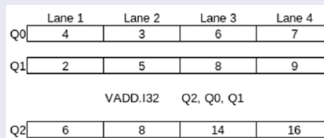
Programmation avec le coprocesseur NEON

Les registres du coprocesseur NEON

- Nombres de composants dans les registres NEON

	8-bit elements	16-bit elements	32-bit elements
64-bit D register	8	4	2
128-bit Q register	16	8	4

- Réalisation d'une opération



Programmation avec le coprocesseur NEON

Les instructions Load et Store avec NEON

- Ces instructions permettent d'échanger des données avec les registres du coprocesseur
- Chargement/Sauvegarde des données structurées sur plusieurs registres
- Syntaxe

```
1 v<op><n>.<size> <list>,[Rn{:<align>}]{!}
2 v<op><n>.<size> <list>,[Rn{:<align>}],Rm
```

  - <op> : ld ou st
  - <n> : soit 1, 2 , 3, ou 4
  - <size> : soit 8, 16, ou 32 .
  - <list> : spécifie la liste des registres. Il existe 4 formats : {Dd[x]}, {Dd[x], D(d+a)[x]}, {Dd[x], D(d+a)[x], D(d+2a)[x]}, {Dd[x], D(d+a)[x], D(d+2a)[x], D(d+3a)[x]}
  - <align> : spécifie un alignement optionnel. Si <align> n'est pas spécifié, alors l'alignement par défaut s'applique

Programmation avec le coprocesseur NEON

Combinaison des paramètres avec vld et vst

<n>	<size>	<list>	<align>	Alignment
1	8	Dd[x]		Standard only
	16	Dd[x]	16	2 byte
	32	Dd[x]	32	4 byte
2	8	Dd[x], D(d+1)[x]	16	2 byte
	16	Dd[x], D(d+1)[x]	32	4 byte
		Dd[x], D(d+2)[x]	32	4 byte
	32	Dd[x], D(d+1)[x]	64	8 byte
		Dd[x], D(d+2)[x]	64	8 byte
3	8	Dd[x], D(d+1)[x], D(d+2)[x]		Standard only
	16 or 32	Dd[x], D(d+1)[x], D(d+2)[x]		Standard only
		Dd[x], D(d+2)[x], D(d+4)[x]		Standard only
4	8	Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]	32	4 byte
	16	Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]	64	8 byte
		Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]	64	8 byte
	32	Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]	64 or 128	(<align>÷8) bytes
		Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]	64 or 128	(<align>÷8) bytes

Programmation avec le coprocesseur NEON

Exemples

```
1 vld3.8 {d0[0],d1[0],d2[0]},[r0]! @ load first element
2 vld3.8 {d0[1],d1[1],d2[1]},[r0]!
3 vld3.8 {d0[2],d1[2],d2[2]},[r0]!
4 vld3.8 {d0[3],d1[3],d2[3]},[r0]!
5 vld3.8 {d0[4],d1[4],d2[4]},[r0]!
6 vld3.8 {d0[5],d1[5],d2[5]},[r0]!
7 vld3.8 {d0[6],d1[6],d2[6]},[r0]!
8 vld3.8 {d0[7],d1[7],d2[7]},[r0]! @ load eighth element
```


Programmation avec le coprocesseur NEON

Exemple d'instruction du coprocesseur NEON

- Chaque instruction peut être exécutée en utilisant une de deux formes
  - `VADD.datatype {Qd}, Qn, Qm`
  - `VADD.datatype {Dd}, Dn, Dm`

`datatype` doit être égale à `I8`, `I16`, `I32`, `I64` ou `F32`

Programmation avec le coprocesseur NEON

Exemple multiplication d'un vecteur et d'une matrice

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} \begin{vmatrix} x \\ y \\ z \end{vmatrix} \Rightarrow \begin{vmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{vmatrix}$$

Programmation avec le coprocesseur NEON

Exemple multiplication d'un vecteur et d'une matrice (1/3)

```
1  .data
2   @ Matrice 1
3   A: .short 1, 4, 7, 0
4      .short 2, 5, 8, 0
5      .short 3, 6, 9, 0
6   @ Matrice 2
7   B: .short 9, 6, 3, 0
8      .short 8, 5, 2, 0
9      .short 7, 4, 1, 0
10  @ Matrice resultat
11  C: .fill 12, 2, 0
12  prtstr: .asciz "%3d␣%3d␣%3d\n"
13 .text
14 .global main
15 .MACRO mulcol ccol bcol
16  VMUL.I16 \ccol, D0, \bcol[0]
17  VMLA.I16 \ccol, D1, \bcol[1]
18  VMLA.I16 \ccol, D2, \bcol[2]
19 .ENDM
```

Programmation avec le coprocesseur NEON

Mise en application

Écrire un programme assembleur permettant d'effectuer la multiplication de deux matrices de même taille

Merci pour votre attention

Questions ?