CE311 Architecture des Processeurs

Architecture ARMv7-M

Nicolas Barbot

nicolas.barbot@esisar.grenoble-inp.fr

2022-2023

Présentation ARM

Les processeurs ARM dominent le marché de l'embarqué:

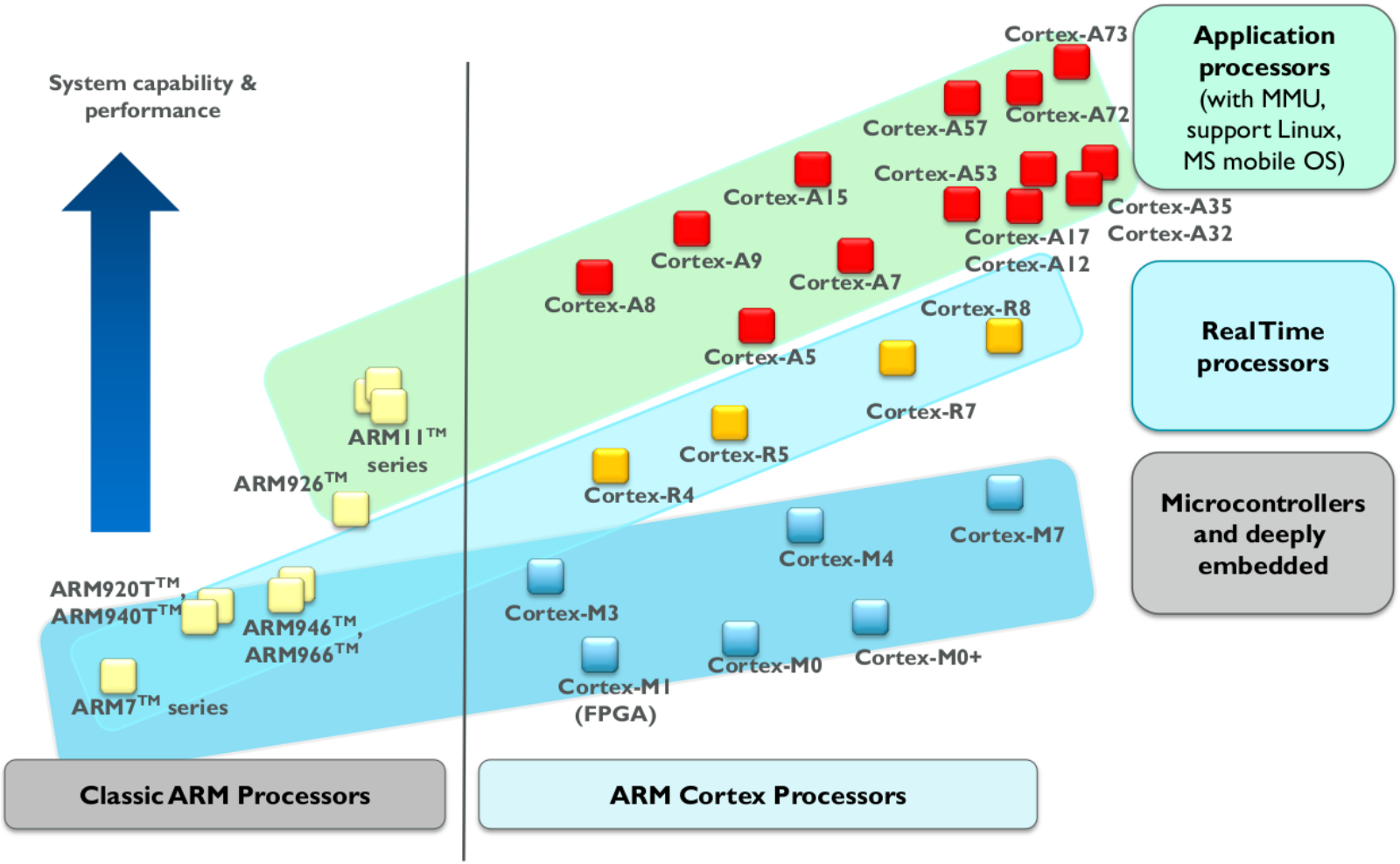
- téléphones mobiles
- tablettes
- console de jeux
- appareils photos

Processeurs RISC très faible consommation
Possibilité d'intégrer les processeurs dans des SoC
Processeurs fabriqués sous licence

Architectures et implémentations

Architecture	Implémentations (familles)
ARMv1	ARM1
ARMv2	ARM2
ARMv3	ARM7 ARM7
ARMv4	ARM8
ARMv4T	ARM7TDMI ARM9TDMI
ARMv5TE	ARM7EJ, ARM9E, ARM10E
ARMv6	ARM11
ARMv6-M	Cortex-M0, Cortex-M0+, Cortex-M1
ARMv7-A	Cortex-A8, Cortex-A9, Cortex-A5 Cortex-A7, Cortex-A12, Cortex-A15
ARMv7-M	Cortex-M3, Cortex-M4 et Cortex-M7
ARMv7-R	Cortex-R4, Cortex-R5, Cortex-R7
ARMv8	Cortex-A53, Cortex-A57

Implémentation et Performance



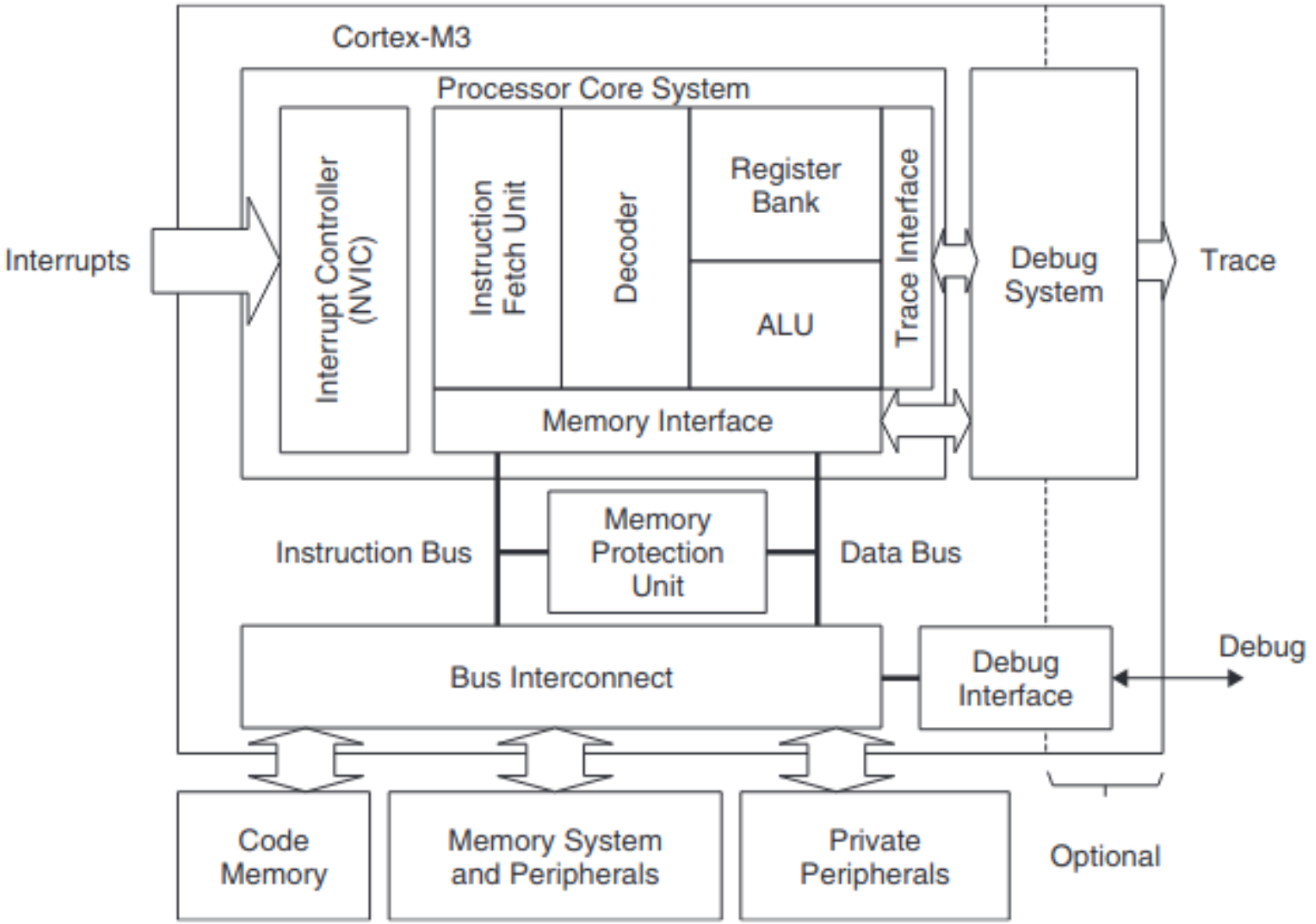
Cortex M

Cortex M0	Processeurs plus simple et faible cout (Architecture Von Neumann, instruction majoritairement 16 bits)
Cotex M1	Soft core synthétisable sur FPGA, architecure identique au Cortex M0
Cortex M3	Processeurs généralistes (Architecture Harvard, instruction 16 bits et 32 bits)
Cortex M4	Processeurs plus complexes avec FPU (Architecture Harvard, instruction 16 bits et 32 bits)
Cortex M7	Processeurs haute performances avec FPU (Architecture Harvard, instruction 16 bits et 32 bits, pipeline 6 stages, mémoire cache)

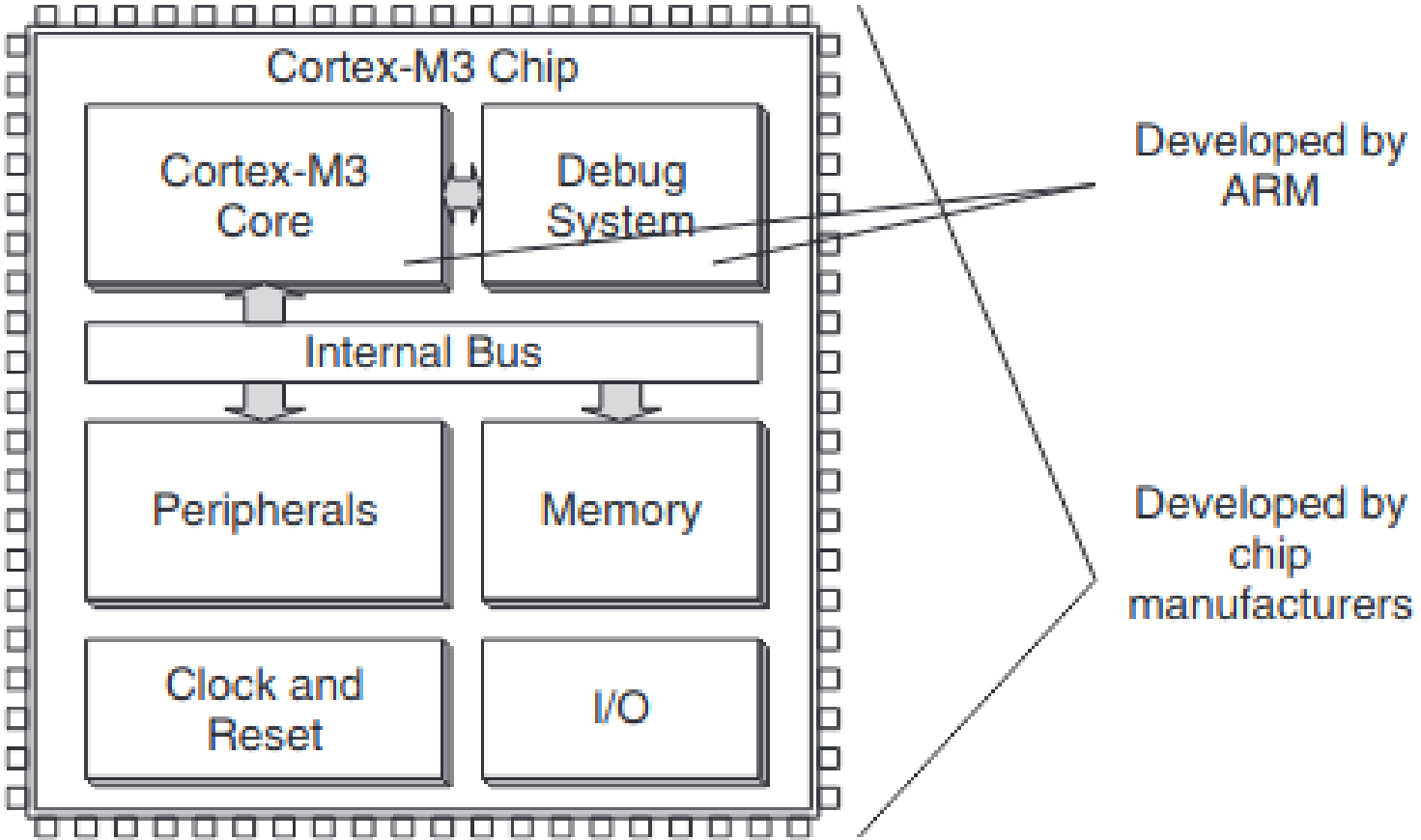
Cortex M3 et M4

- Architecture 32 bits
- Jeux d'instruction Thumb2
- Architecture du bus de type Harvard
- Pipeline en 3 stages
- Bus AMBA
- Controlleur d'interruption NVIC
- Support pour OS temps réel
- Mode veille et basse consomation
- Support pour multi-core
- MPU optionel
- Pas de cache

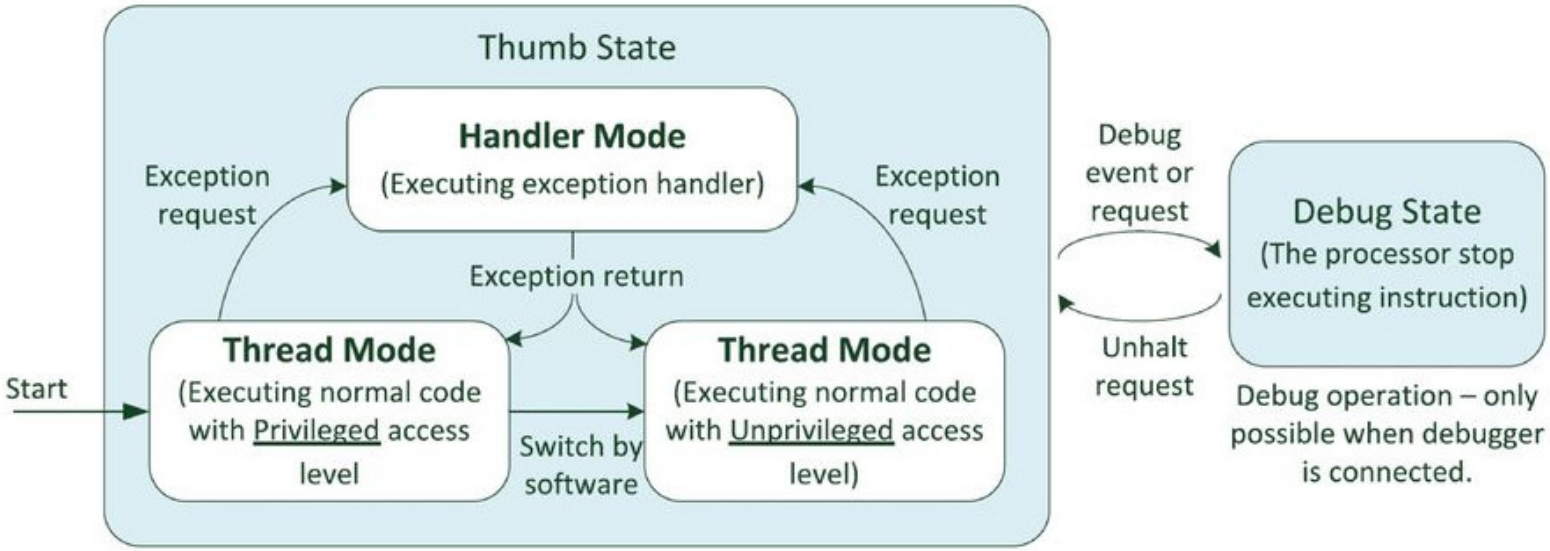
Diagramme du Cortex M3



Micro-controlleur basé sur un Cortex M3

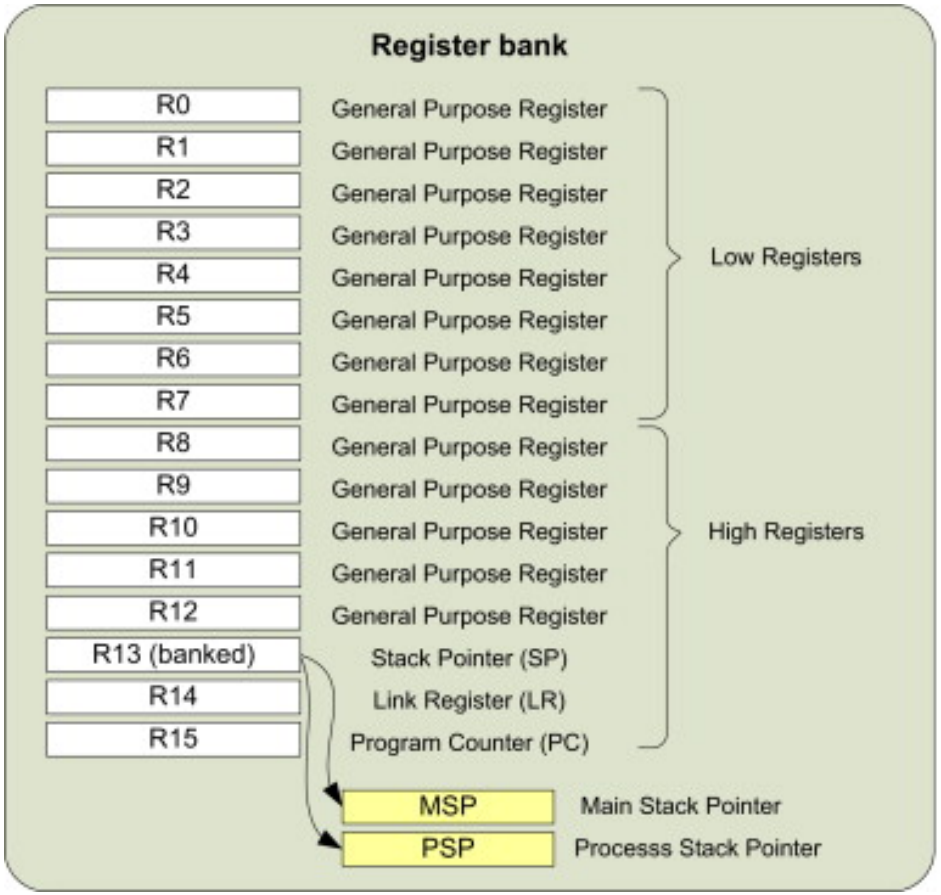


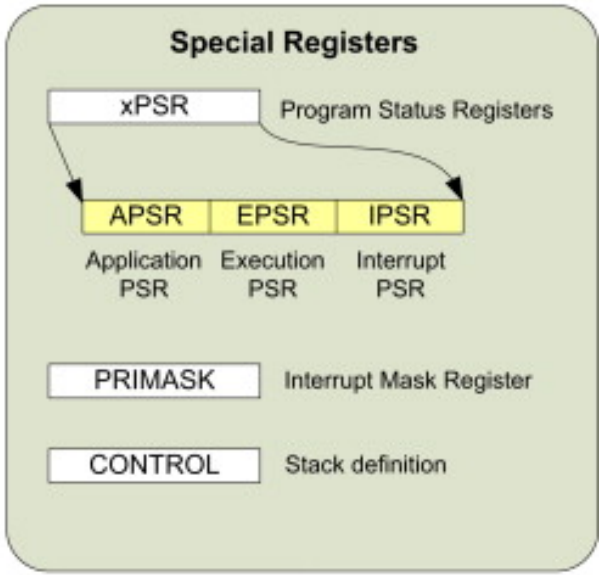
Modes d'opération



- Thread mode et Handler mode ont deux SP différent
- Unprivileged limite l'accés à la mémoire, et interdit certaines instructions
- Unprivileged vers Privileged impossible en software

Registres


Register bank
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (banked)
R14
R15
General Purpose Register
General Purpose Register
General Purpose Register
General Purpose Register
General Purpose Register
General Purpose Register
General Purpose Register
General Purpose Register
General Purpose Register
General Purpose Register
General Purpose Register
General Purpose Register
General Purpose Register
Stack Pointer (SP)
Link Register (LR)
Program Counter (PC)
Low Registers
High Registers
MSP
PSP
Main Stack Pointer
Processs Stack Pointer


Special Registers
xPSR
Program Status Registers
APSR
EPSR
IPSR
Application PSR
Execution PSR
Interrupt PSR
PRIMASK
Interrupt Mask Register
CONTROL
Stack definition

xPSR

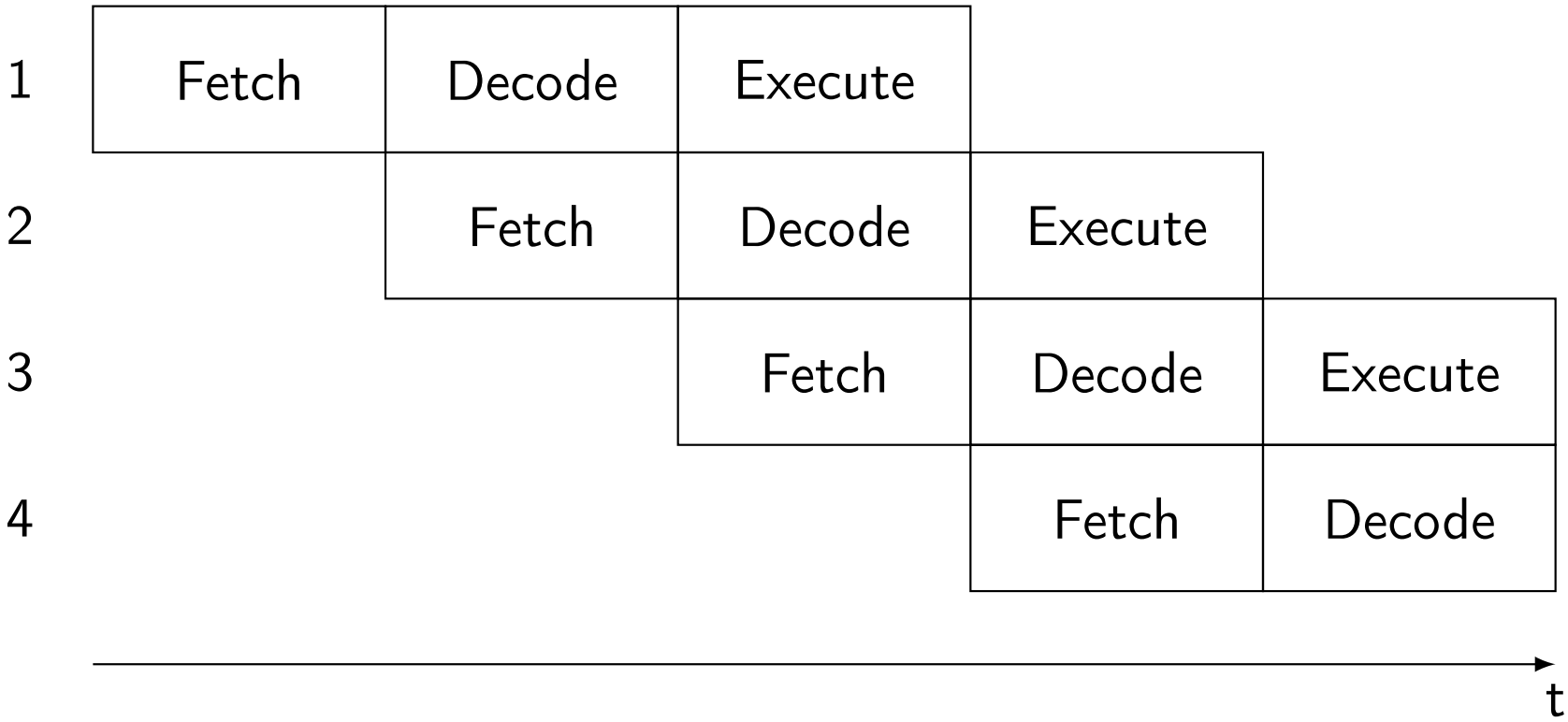
xPSR (Program Status Register) est une combinaison de

- APSR (Application PSR)
- EPSR (Execution PSR)
- IPSR (Interrupt PSR)

	31	*30*	*29*	*28*	*27*	*26:25*	*24*	*23:20*	*19:16*	*15:10*	*9*	*8*	*7*	*6*	*5*	*4:0*
APSR	N	Z	C	V	Q											
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

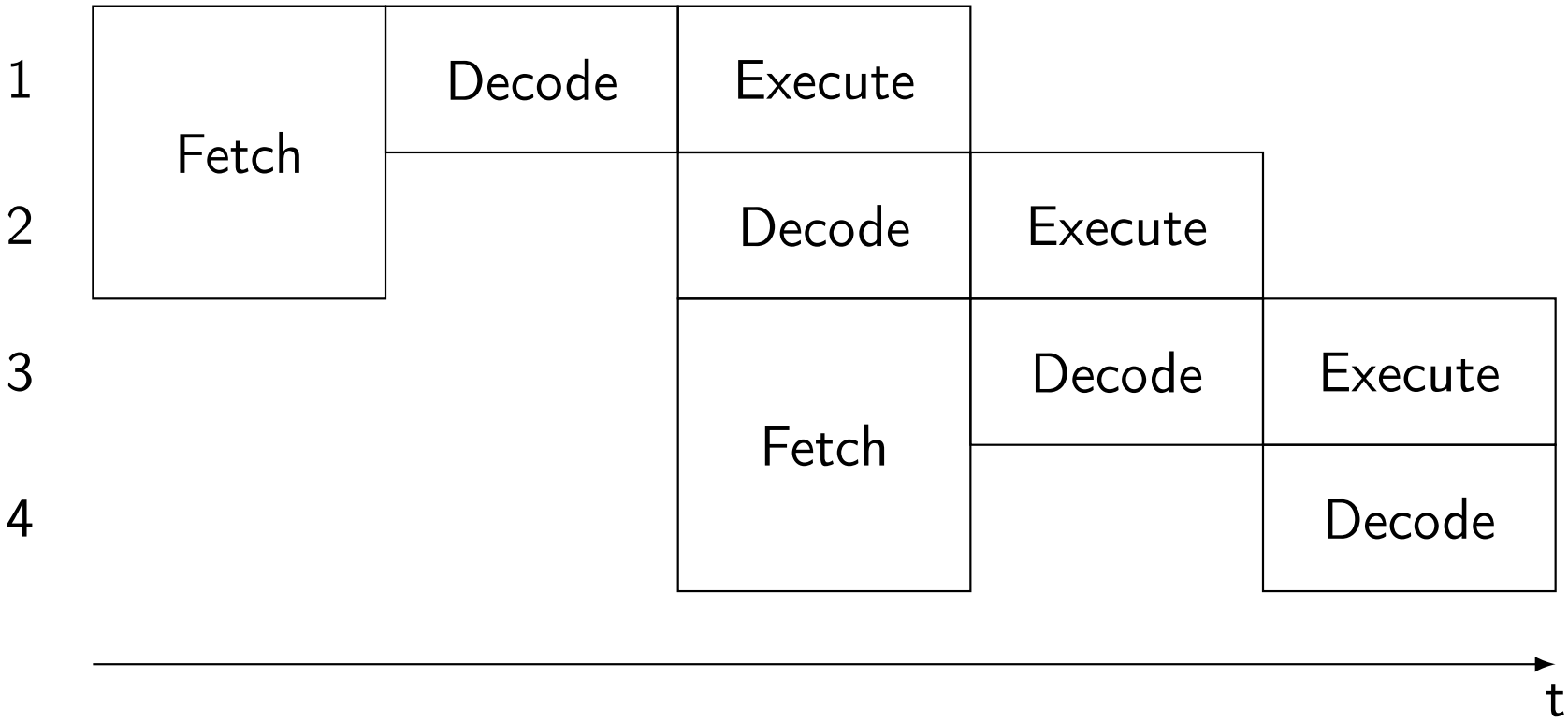
Pipeline

- Instructions 32 bits



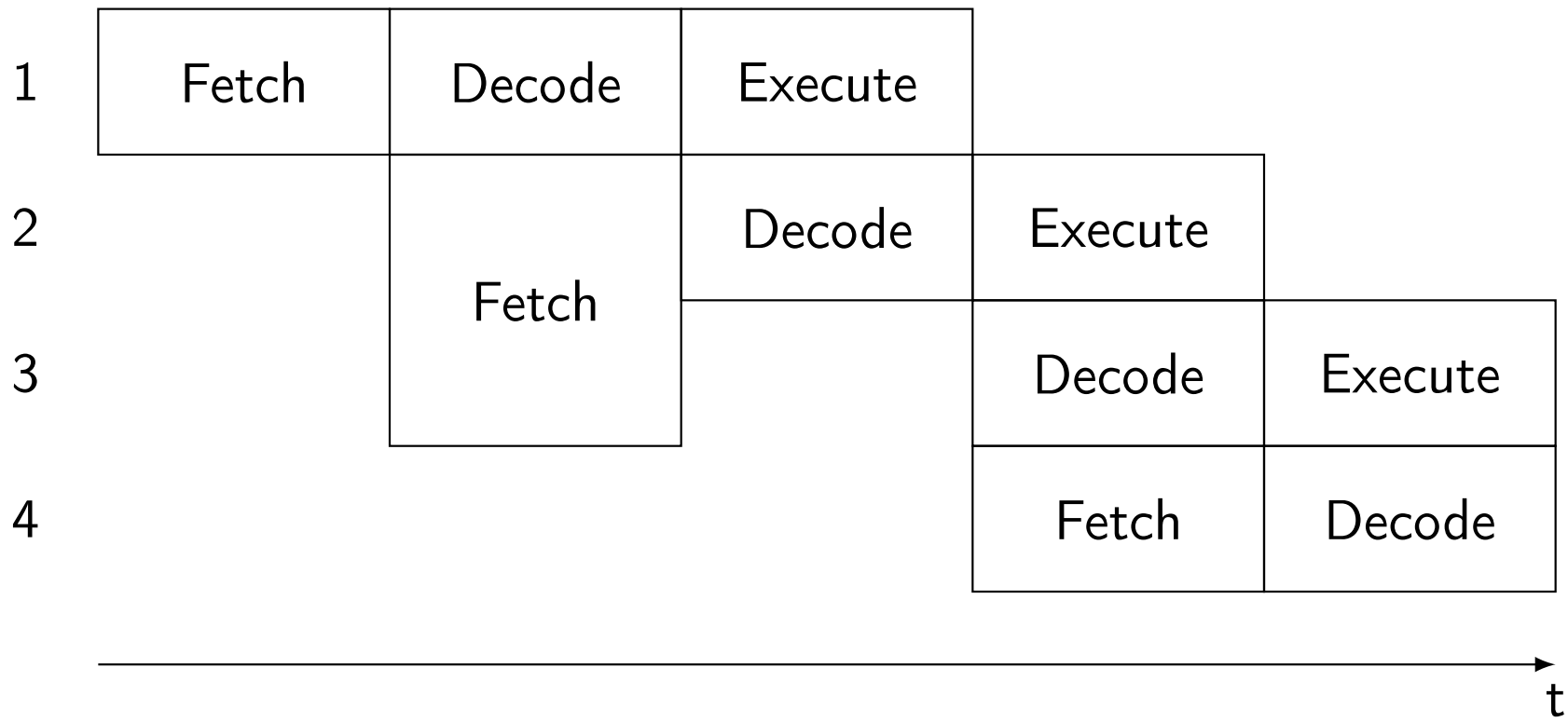
Pipeline

- Instructions 16 bits



Pipeline

- Instructions 32 et 16 bits



Mémoire

Caractéristiques générales:

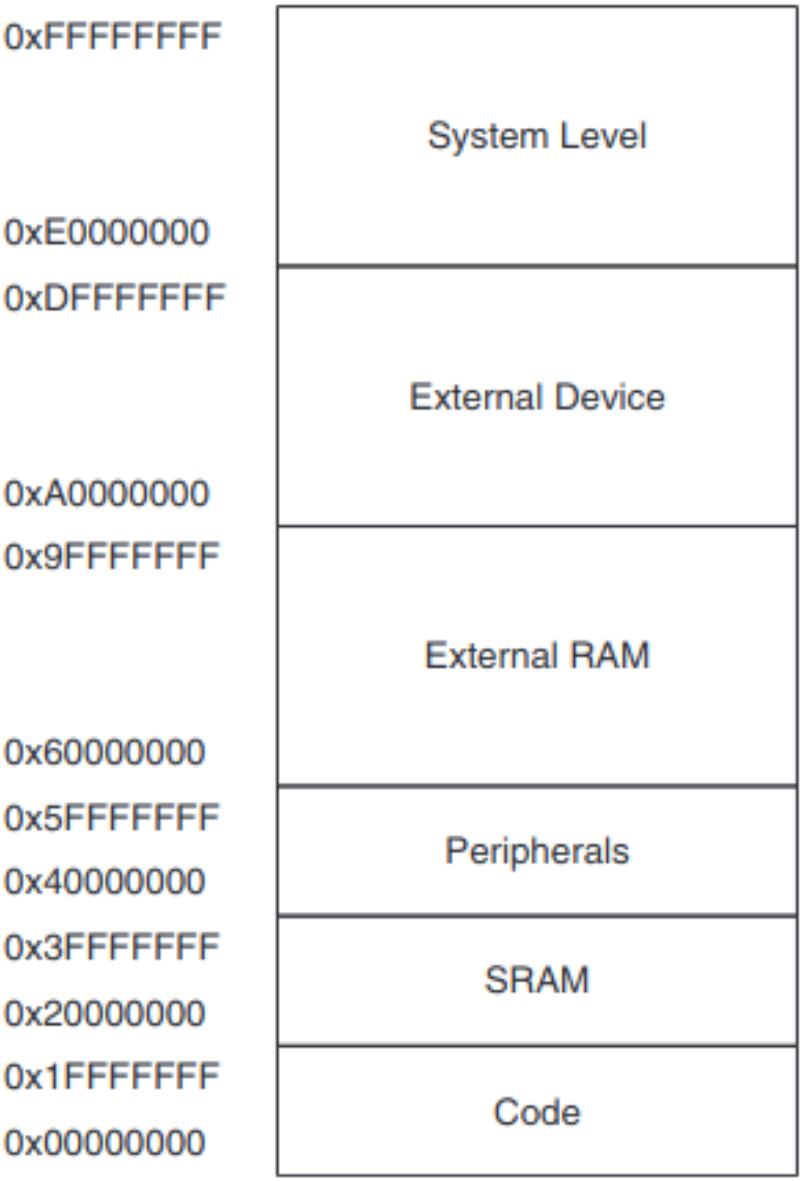
- Connectée au processeur par le bus
- Adressée à l'octet
- Espace adressable: 4 GB

La mémoire est partagée pour:

- Mémoire Flash (pour le programme)
- SRAM (pour les données)
- Périphériques
- Debug

Un MCU utilise seulement une petite partie de cet espace

Mapping Mémoire



Private peripherals, including built-in interrupt controller (NVIC), MPU control registers, and debug components

Mainly used as external peripherals

Mainly used as external memory

Mainly used as peripherals

Mainly used as static RAM

Mainly used for program code, also provides exception vector table after power-up

Jeu d'instruction

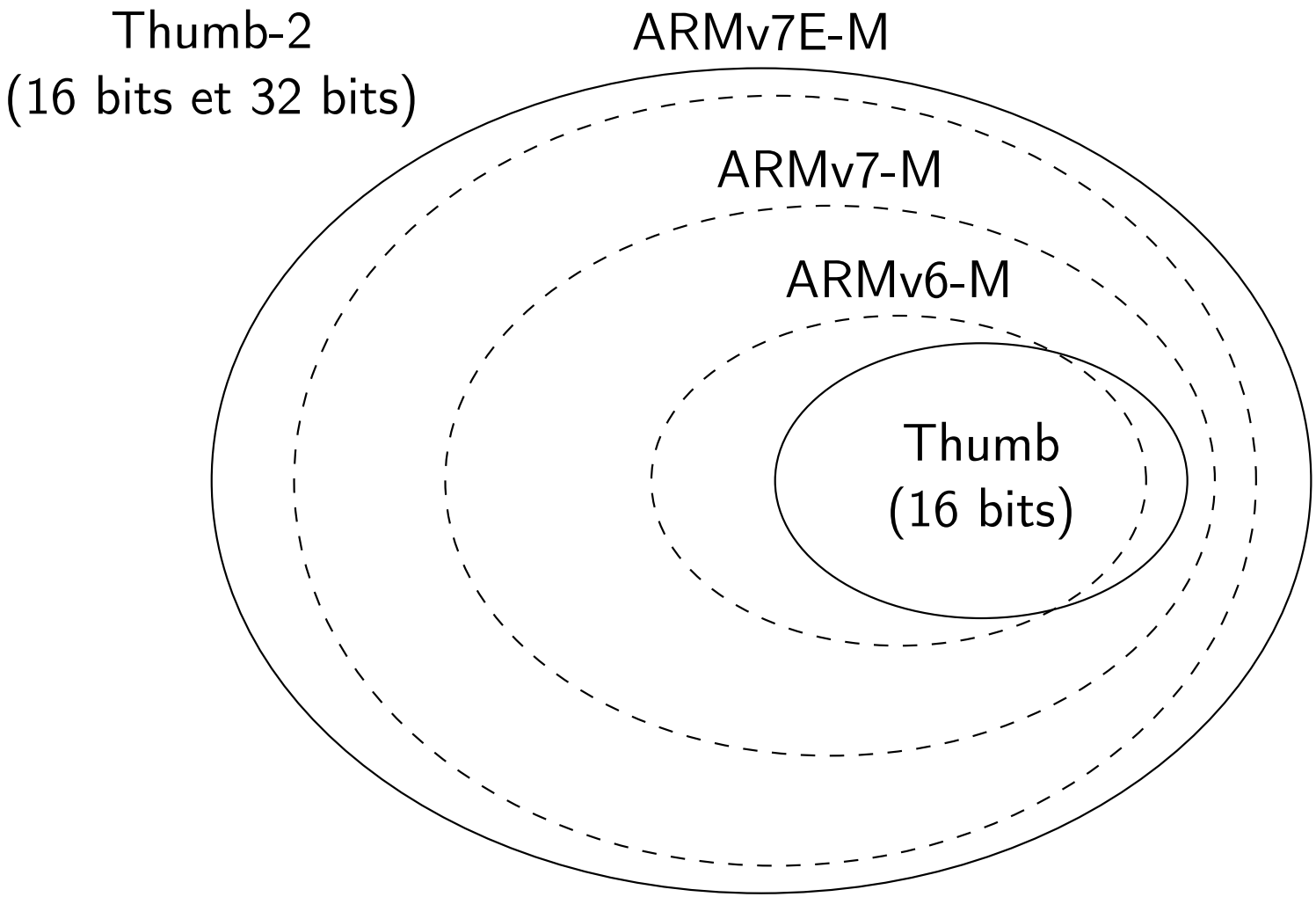
Tous les architectures de ARMv4T à ARMv6 supportent au moins deux jeux d'instruction:

- les instructions classiques ARM (sur 32 bits)
- les instructions Thumb (sur 16 bits)

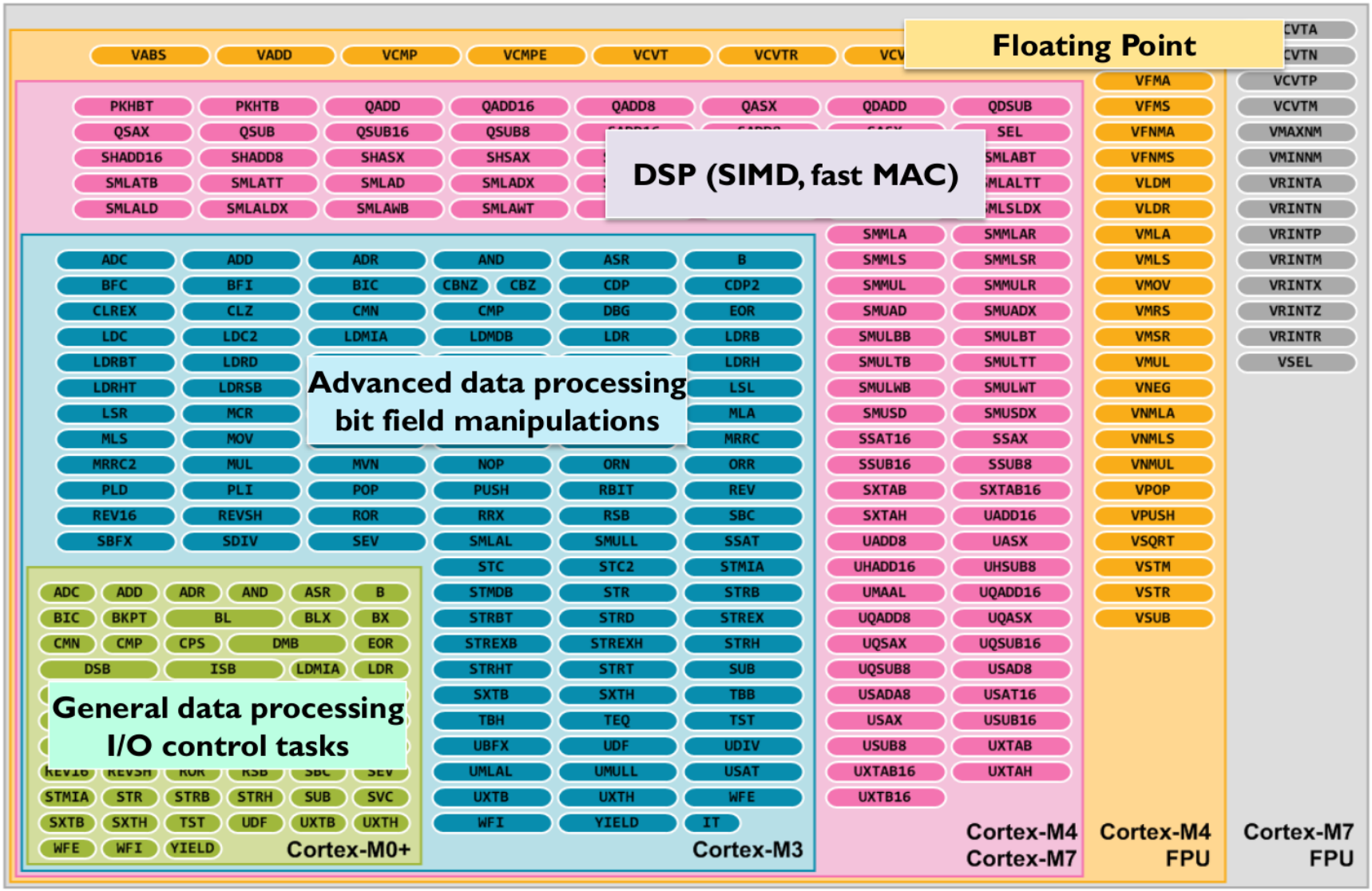
Avec l'architecture ARMv7-M (et ARMv6-M), le processeur ne supporte qu'un seul jeu d'instruction:

- les instructions Thumb-2 (16 et 32 bits)

Thumb-2


Thumb-2
(16 bits et 32 bits)
ARMv7E-M
ARMv7-M
ARMv6-M
Thumb
(16 bits)

Jeu d'instruction des Cortex M



Jeu d'instruction

Les instructions peuvent être classées en:

- Déplacement de donnée dans le processeur
- Accés mémoire
- Opération Arithmétiques
- Opérations logiques
- Décalages et rotations
- Conversions
- ...

MOV

MOV permet de déplacer une donnée d'un registre (ou d'une valeur immédiate) vers un registre.

`MOV Rd, Rm`

0	1	0	0	0	1	1	0	D	Rm	Rd

`MOV Rd, #imm8`

0	0	1	0	0	Rd	imm8

`MOV{S} Rd, #const`

11110i0	0	1	0	S	Rn	0	imm3	Rd	imm8

```
MOVW Rd, #imm16        MRS R7, PRIMASK        MVN R2, R7
MOVT Rd, #imm16        MRS CONTROL, R7
```

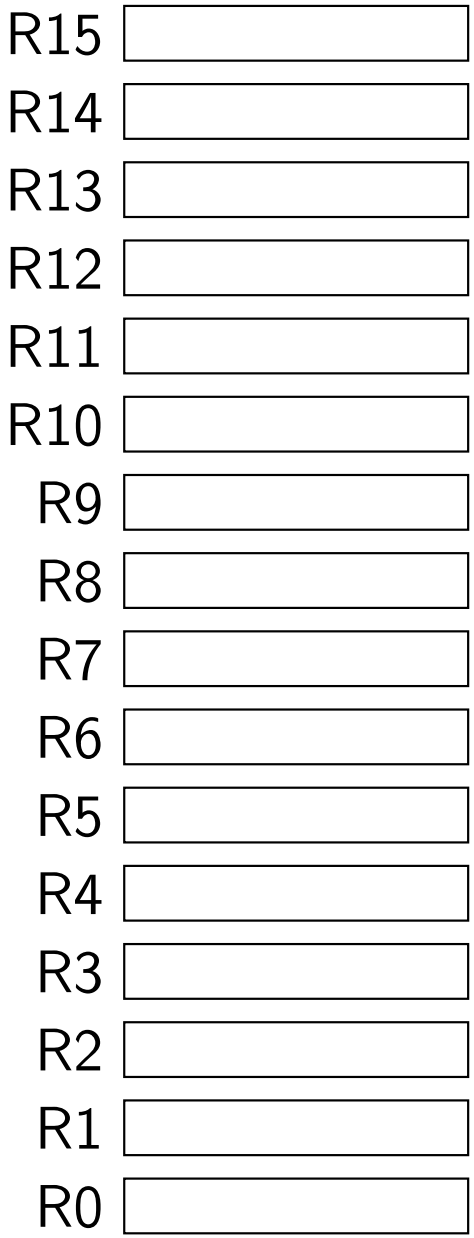
MOV

Il n'y a pas d'instruction pour charger une donnée 32 dans un registre. On peut néanmoins:

- utiliser une instruction `MOVW` et une instruction `MOVT`
- utiliser une pseudo instruction `LDR R0, =0x12345678`

Exercice

```
MOV R0, #0x5
MVN R1, #0
MOVW R2, #0x5678
MOVT R2, #0x1234
MOV R3, R1
MOVT R2, #0
```



Exercice

```
MOV R0, #0x5
MVN R1, #0
MOVW R2, #0x5678
MOVT R2, #0x1234
MOV R3, R1
MOVT R2, #0
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	0x00000005

Exercice

```
MOV R0, #0x5
MVN R1, #0
MOVW R2, #0x5678
MOVT R2, #0x1234
MOV R3, R1
MOVT R2, #0
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0xFFFFFFFF
R0	0x00000005

Exercice

```
MOV R0, #0x5
MVN R1, #0
MOVW R2, #0x5678
MOVT R2, #0x1234
MOV R3, R1
MOVT R2, #0
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00005678
R1	0xFFFFFFFF
R0	0x00000005

Exercice

```
MOV R0, #0x5
MVN R1, #0
MOVW R2, #0x5678
MOVT R2, #0x1234
MOV R3, R1
MOVT R2, #0
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x12345678
R1	0xFFFFFFFF
R0	0x00000005

Exercice

```
MOV R0, #0x5
MVN R1, #0
MOVW R2, #0x5678
MOVT R2, #0x1234
MOV R3, R1
MOVT R2, #0
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0xFFFFFFFF
R2	0x12345678
R1	0xFFFFFFFF
R0	0x00000005

Exercice

```
MOV R0, #0x5
MVN R1, #0
MOVW R2, #0x5678
MOVT R2, #0x1234
MOV R3, R1
MOVT R2, #0
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0xFFFFFFFF
R2	0x00005678
R1	0xFFFFFFFF
R0	0x00000005

LDR permet de lire une donnée en mémoire. La donnée peut être sur 1, 2, 4 ou 8 octets, signée ou non-signée.

`LDRB <Rd>, [<Rn>, <Rm>]`

0	1	0	1	1	1	0	Rm	Rn	Rd

`LDRB <Rd>, [<Rn>, #<immed_5>]`

0	1	1	1	1	imm5	Rn	Rd

`LDRB Rd, [Rn #imm12]`

1111110	0	1	0	1	1	Rn	Rd	imm12

`LDRSB <Rd>, [<Rn>, <Rm>]`
`LDRSH <Rd>, [<Rn>, <Rm>]`
`LDRM <Rd>, [<Rn>, <Rm>]`

`LDRH <Rd>, [<Rn>, <Rm>]`
`LDR <Rd>, [<Rn>, <Rm>]`
`LDRD <Rd>, [<Rn>, <Rm>]`

Mode d'adressage

Offset immédiat L'adresse mémoire est la somme d'une valeur contenue dans un registre et d'une offset. Possibilité de mettre à jour le pointeur après le transfert !
Exemple: `LDRB R0, [R1 #10]`

Literal L'adresse est générée à partir de la valeur actuelle du PC et d'un offset
Exemple: `LDR R5, [PC #10]`

Registre avec offset L'adresse corresponds à la somme de deux registres (base + index).
Exemple: `LDR R3, [R0, R2]`

Post index L'adresse est lue depuis un registre, l'offset est ajouté après le transfert.
Exemple: `LDR R1, [R0], #2`

STR

STR permet d'écrire une donnée en mémoire. La donnée peut être sur 1, 2, 4 ou 8 octets, signée ou non-signée.

`STRB <Rd>, [<Rn>, <Rm>]`

0	1	0	1	0	1	0	Rm	Rn	Rd

`STR <Rd>, [<Rn>, #imm5 ]`

0	1	1	0	0	imm5	Rn	Rd

`STRB Rd, [Rn #imm12]`

1111100	0	1	0	0	0	Rn	Rd	imm12

`STRSB <Rd>, [<Rn>, <Rm>]`
`STRSH <Rd>, [<Rn>, <Rm>]`
`STRM <Rd>, [<Rn>, <Rm>]`

`STRH <Rd>, [<Rn>, <Rm>]`
`STR <Rd>, [<Rn>, <Rm>]`
`STRD <Rd>, [<Rn>, <Rm>]`

Exercice

```
LDR R0, [R2]
LDRH R1, [R2]
LDRB R3, [R2]
LDRSH R4, [R2]
LDRSB R5, [R2]
LDR R6, [R2,#4]
LDR R7, [R2,#4]!
LDR R7, [R2,#4]!
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0xFFFFFFFF
R2	0x00005678
R1	0xFFFFFFFF
R0	0x00000005

0x22446688	0x5678
0x22222222	0x567B
0x00000001	0x567F

Exercice

```
LDR R0, [R2]
LDRH R1, [R2]
LDRB R3, [R2]
LDRSH R4, [R2]
LDRSB R5, [R2]
LDR R6, [R2,#4]
LDR R7, [R2,#4]!
LDR R7, [R2,#4]!
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0xFFFFFFFF
R2	0x00005678
R1	0xFFFFFFFF
R0	0x22446688

0x22446688	0x5678
0x22222222	0x567B
0x00000001	0x567F

Exercice

```
LDR R0, [R2]
LDRH R1, [R2]
LDRB R3, [R2]
LDRSH R4, [R2]
LDRSB R5, [R2]
LDR R6, [R2,#4]
LDR R7, [R2,#4]!
LDR R7, [R2,#4]!
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0xFFFFFFFF
R2	0x00005678
R1	0x00006688
R0	0x22446688

0x22446688	0x5678
0x22222222	0x567B
0x00000001	0x567F

Exercice

```
LDR R0, [R2]
LDRH R1, [R2]
LDRB R3, [R2]
LDRSH R4, [R2]
LDRSB R5, [R2]
LDR R6, [R2,#4]
LDR R7, [R2,#4]!
LDR R7, [R2,#4]!
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x00000088
R2	0x00005678
R1	0x00006688
R0	0x22446688

0x22446688	0x5678
0x22222222	0x567B
0x00000001	0x567F

Exercice

```
LDR R0, [R2]
LDRH R1, [R2]
LDRB R3, [R2]
LDRSH R4, [R2]
LDRSB R5, [R2]
LDR R6, [R2,#4]
LDR R7, [R2,#4]!
LDR R7, [R2,#4]!
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	0x00006688
R3	0x00000088
R2	0x00005678
R1	0x00006688
R0	0x22446688

0x22446688	0x5678
0x22222222	0x567B
0x00000001	0x567F

Exercice

```
LDR R0, [R2]
LDRH R1, [R2]
LDRB R3, [R2]
LDRSH R4, [R2]
LDRSB R5, [R2]
LDR R6, [R2,#4]
LDR R7, [R2,#4]!
LDR R7, [R2,#4]!
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	0xFFFFFF88
R4	0x00006688
R3	0x00000088
R2	0x00005678
R1	0x00006688
R0	0x22446688

0x22446688	0x5678
0x22222222	0x567B
0x00000001	0x567F

Exercice

```
LDR R0, [R2]
LDRH R1, [R2]
LDRB R3, [R2]
LDRSH R4, [R2]
LDRSB R5, [R2]
LDR R6, [R2,#4]
LDR R7, [R2,#4]!
LDR R7, [R2,#4]!
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	0x22222222
R5	0xFFFFFF88
R4	0x00006688
R3	0x00000088
R2	0x00005678
R1	0x00006688
R0	0x22446688

0x22446688	0x5678
0x22222222	0x567B
0x00000001	0x567F

Exercice

```
LDR R0, [R2]
LDRH R1, [R2]
LDRB R3, [R2]
LDRSH R4, [R2]
LDRSB R5, [R2]
LDR R6, [R2,#4]
LDR R7, [R2,#4]!
LDR R7, [R2,#4]!
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	0x22222222
R6	0x22222222
R5	0xFFFFFF88
R4	0x00006688
R3	0x00000088
R2	0x0000567B
R1	0x00006688
R0	0x22446688

0x22446688	0x5678
0x22222222	0x567B
0x00000001	0x567F

Exercice

```
LDR R0, [R2]
LDRH R1, [R2]
LDRB R3, [R2]
LDRSH R4, [R2]
LDRSB R5, [R2]
LDR R6, [R2,#4]
LDR R7, [R2,#4]!
LDR R7, [R2,#4]!
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	0x00000001
R6	0x22222222
R5	0xFFFFFF88
R4	0x00006688
R3	0x00000088
R2	0x0000567F
R1	0x00006688
R0	0x22446688

0x22446688	0x5678
0x22222222	0x567B
0x00000001	0x567F

PUSH et POP

PUSH et POP permettent d'empiler et de dépiler plusieurs registres sur la pile

`PUSH {R0, R3-R5}`

1	0	1	1	0	1	0	M	Register list

`POP {R0, R3-R5}`

1	0	1	1	1	1	0	P	Register list

`PUSH {R0, R3-R5}`

1110100	1	0	0	1	0	1	1	0	1	0	M	0	Register list

`POP {R0, R3-R5}`

1110100	0	1	0	1	1	1	1	0	1	P	M	0	Register list

Pile

La pile est une zone de mémoire (SRAM) + un registre utilisé en tant que pointeur (R13). A chaque RESET, la pile est initialisée au sommet de la SRAM.

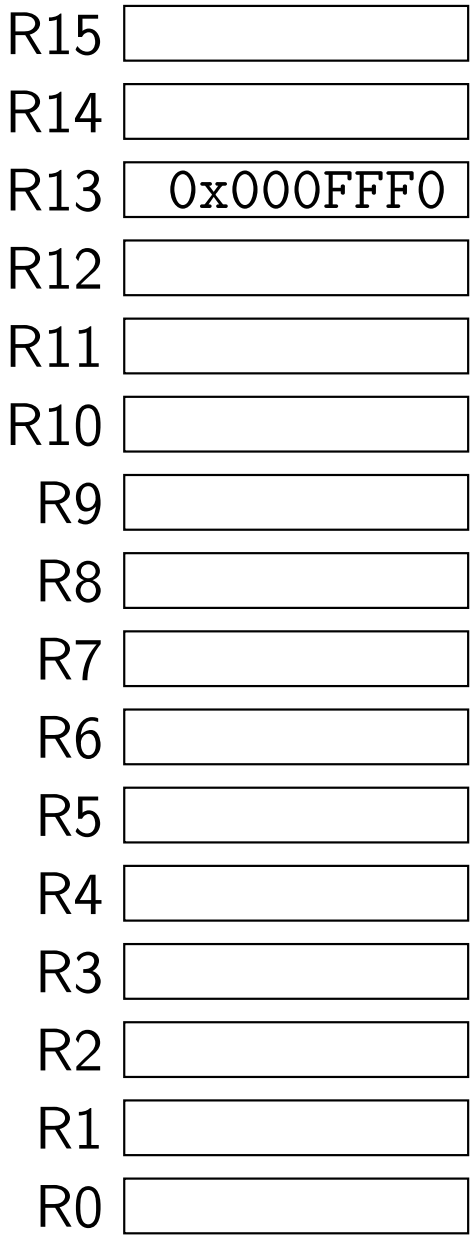
La pile est utilisée avec les instructions:

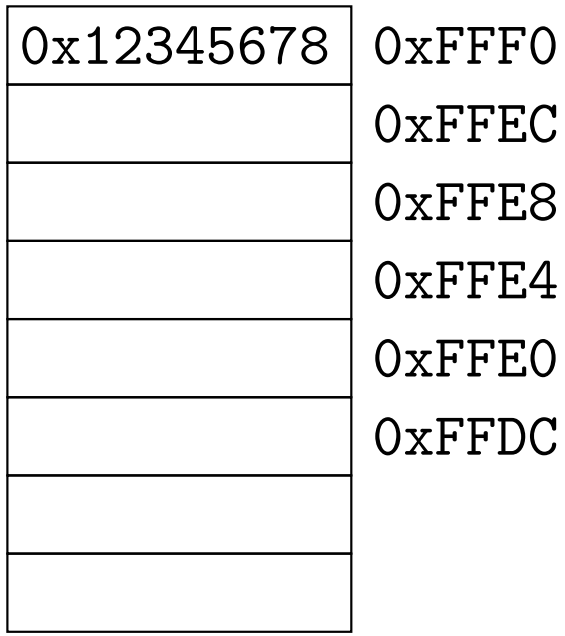
- PUSH pour placer un élément au sommet la pile
- POP pour enlever l'élément au sommet de la pile

Pour PUSH, SP est d'abord déplacé puis la donnée est écrite.
Pour POP, la donnée est d'abord lue, plus PS est déplacé.

Exercice

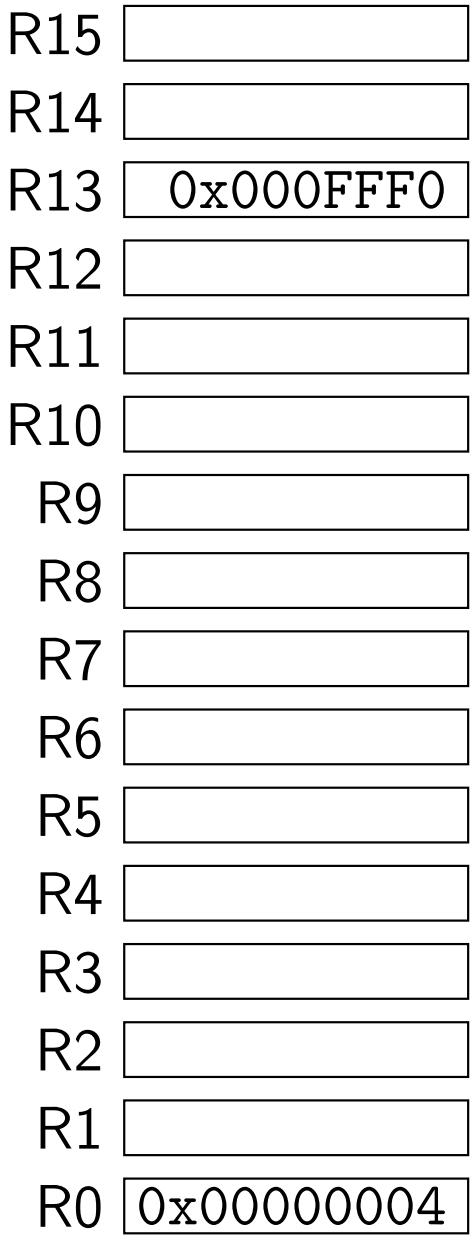
```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

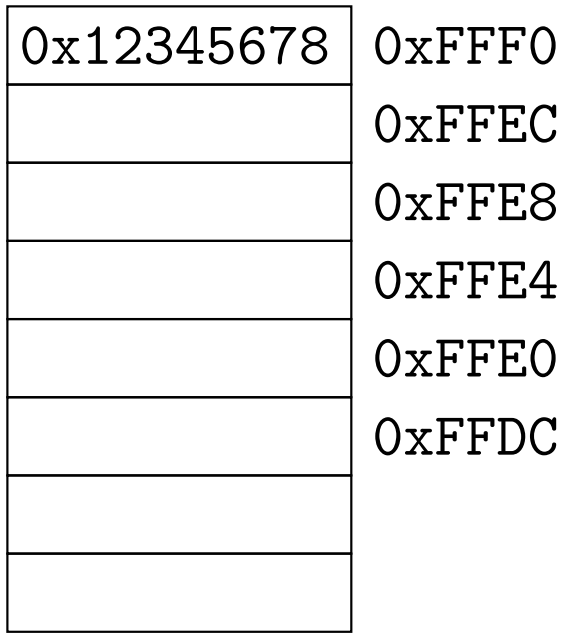




Exercice

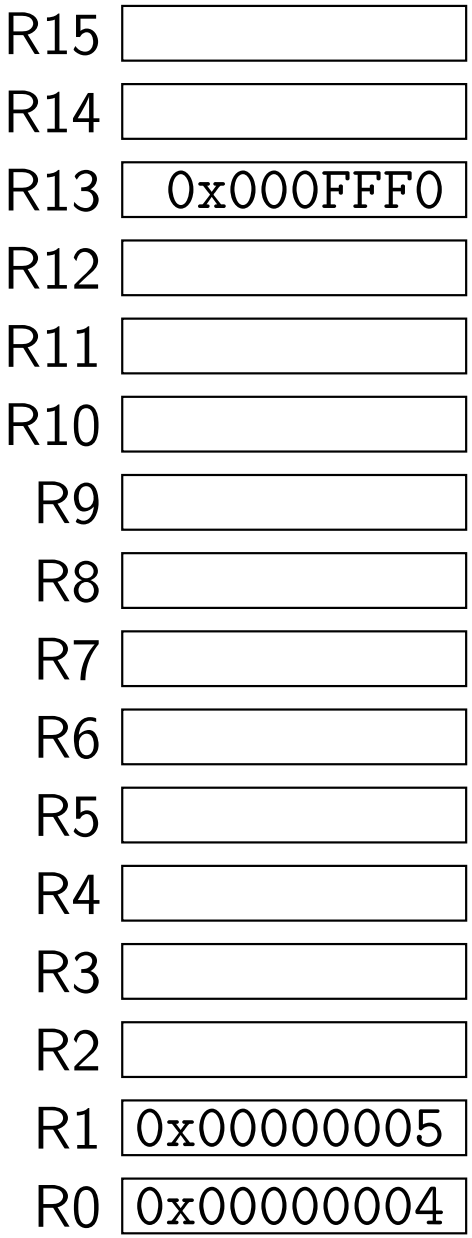
```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

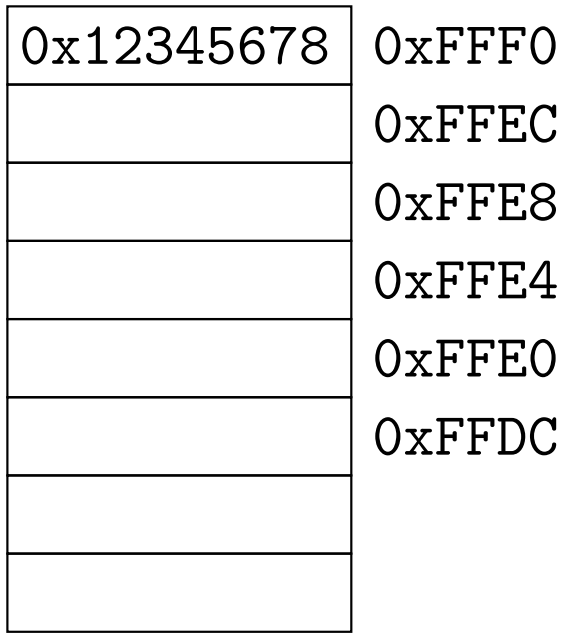




Exercice

```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```





Exercice

```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

R15	
R14	
R13	0x000FFF0
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000006
R1	0x00000005
R0	0x00000004

0x12345678	0xFFF0
	0xFFEC
	0xFFE8
	0xFFE4
	0xFFE0
	0xFFDC

Exercice

```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

R15	
R14	
R13	0x000FFF0
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x00000007
R2	0x00000006
R1	0x00000005
R0	0x00000004

0x12345678	0xFFF0
	0xFFEC
	0xFFE8
	0xFFE4
	0xFFE0
	0xFFDC

Exercice

```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

R15	
R14	
R13	0x000FFEC
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x00000007
R2	0x00000006
R1	0x00000005
R0	0x00000004

0x12345678	0xFFF0
0x00000004	0xFFEC
	0xFFE8
	0xFFE4
	0xFFE0
	0xFFDC

Exercice

```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

R15	
R14	
R13	0x000FFEC
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x00000007
R2	0x00000006
R1	0x00000005
R0	0x00000000

0x12345678	0xFFF0
0x00000004	0xFFEC
	0xFFE8
	0xFFE4
	0xFFE0
	0xFFDC

Exercice

```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

R15	
R14	
R13	0x000FFF0
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x00000007
R2	0x00000006
R1	0x00000005
R0	0x00000004

0x12345678	0xFFF0
0x00000004	0xFFEC
	0xFFE8
	0xFFE4
	0xFFE0
	0xFFDC

Exercice

```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

R15	
R14	
R13	0x000FFE4
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x00000007
R2	0x00000006
R1	0x00000005
R0	0x00000004

0x12345678	0xFFF0
0x00000005	0xFFEC
0x00000006	0xFFE8
0x00000007	0xFFE4
	0xFFE0
	0xFFDC

Exercice

```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x00000007
R2	0x00000006
R1	0x00000000
R0	0x00000004

0x12345678	0xFFF0
0x00000005	0xFFEC
0x00000006	0xFFE8
0x00000007	0xFFE4
	0xFFE0
	0xFFDC

Exercice

```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x00000007
R2	0x00000000
R1	0x00000000
R0	0x00000004

0x12345678	0xFFF0
0x00000005	0xFFEC
0x00000006	0xFFE8
0x00000007	0xFFE4
	0xFFE0
	0xFFDC

Exercice

```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x00000000
R2	0x00000000
R1	0x00000000
R0	0x00000004

0x12345678	0xFFF0
0x00000005	0xFFEC
0x00000006	0xFFE8
0x00000007	0xFFE4
	0xFFE0
	0xFFDC

Exercice

```
MOV R0, #4
MOV R1, #5
MOV R2, #6
MOV R3, #7
PUSH R0
MOV R0, #0
POP R0
PUSH {R1, R2, R3}
MOV R1, #0
MOV R2, #0
MOV R3, #0
POP {R1 R2 R3}
```

R15	
R14	
R13	0x000FFF0
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x00000007
R2	0x00000006
R1	0x00000005
R0	0x00000004

0x12345678	0xFFF0
0x00000005	0xFFEC
0x00000006	0xFFE8
0x00000007	0xFFE4
	0xFFE0
	0xFFDC

ADD

ADD permet de calculer la somme de deux registres est de stocker le resultat dans un registre

`ADD Rd, Rn, Rm`

0	0	0	1	1	0	0	Rm	Rn	Rd

`ADD Rd, Rn, #imm3`

0	0	0	1	1	1	0	imm3	Rn	Rd

`ADD Rdn, #imm8`

0	0	1	1	0	Rdn	imm8

`ADD Rdn, Rm`

0	1	0	0	0	1	0	0	D	Rm	Rdn

`ADD{S} Rd, Rn, Rm, shift`

1110101	1	0	0	0	S	Rn	0	imm3	Rd	imm2	type	Rm

ADC

ADC permet de calculer la somme de deux registres plus le bit Carry et de stocker le resultat dans un registre

`ADC Rdn, Rm`

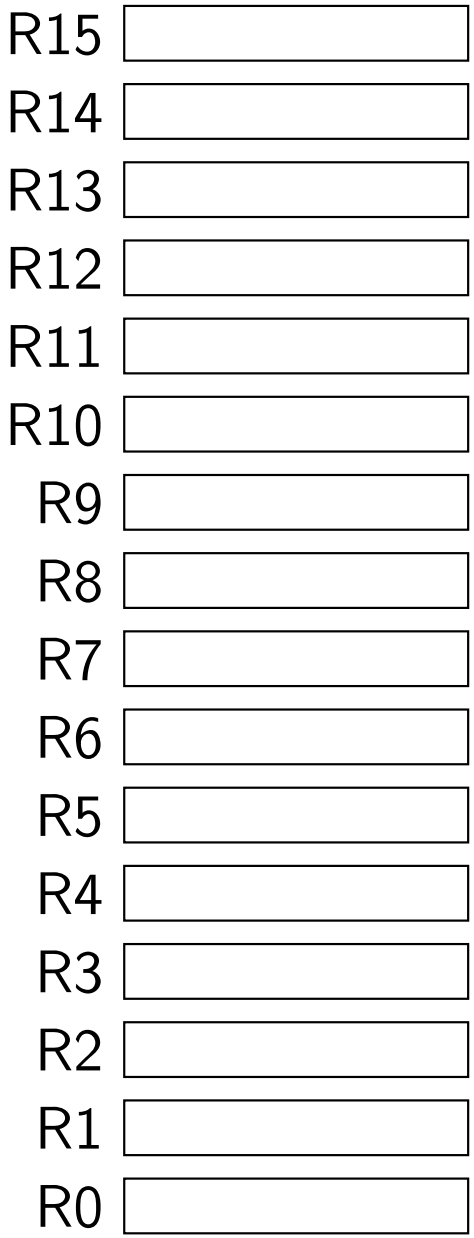
0	1	0	0	0	0	0	1	0	1	Rm	Rdn

`ADC{S} Rd, Rn, Rm, #const`

11110i0	1	0	1	0	S	Rn	0	imm3	Rd	imm8

Exercice

```
MOV R0, #0
MOV R1, #1
MOV R2, #2
ADD R0, R1, R2
ADD R0, R1, #1
ADD R0, R1
LDR R3, =0x50000000
LDR R4, =0x50000000
ADD R5, R3, R4
```



Exercice

```
MOV R0, #0
MOV R1, #1
MOV R2, #2
ADD R0, R1, R2
ADD R0, R1, #1
ADD R0, R1
LDR R3, =0x50000000
LDR R4, =0x50000000
ADD R5, R3, R4
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	0x00000000

Exercice

```
MOV R0, #0
MOV R1, #1
MOV R2, #2
ADD R0, R1, R2
ADD R0, R1, #1
ADD R0, R1
LDR R3, =0x50000000
LDR R4, =0x50000000
ADD R5, R3, R4
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x00000001
R0	0x00000000

Exercice

```
MOV R0, #0
MOV R1, #1
MOV R2, #2
ADD R0, R1, R2
ADD R0, R1, #1
ADD R0, R1
LDR R3, =0x50000000
LDR R4, =0x50000000
ADD R5, R3, R4
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000002
R1	0x00000001
R0	0x00000000

Exercice

```
MOV R0, #0
MOV R1, #1
MOV R2, #2
ADD R0, R1, R2
ADD R0, R1, #1
ADD R0, R1
LDR R3, =0x50000000
LDR R4, =0x50000000
ADD R5, R3, R4
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000002
R1	0x00000001
R0	0x00000003

Exercice

```
MOV R0, #0
MOV R1, #1
MOV R2, #2
ADD R0, R1, R2
ADD R0, R1, #1
ADD R0, R1
LDR R3, =0x50000000
LDR R4, =0x50000000
ADD R5, R3, R4
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000002
R1	0x00000001
R0	0x00000002

Exercice

```
MOV R0, #0
MOV R1, #1
MOV R2, #2
ADD R0, R1, R2
ADD R0, R1, #1
ADD R0, R1
LDR R3, =0x50000000
LDR R4, =0x50000000
ADD R5, R3, R4
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000002
R1	0x00000001
R0	0x00000003

Exercice

```
MOV R0, #0
MOV R1, #1
MOV R2, #2
ADD R0, R1, R2
ADD R0, R1, #1
ADD R0, R1
LDR R3, =0x50000000
LDR R4, =0x50000000
ADD R5, R3, R4
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x50000000
R2	0x00000002
R1	0x00000001
R0	0x00000003

Exercice

```
MOV R0, #0
MOV R1, #1
MOV R2, #2
ADD R0, R1, R2
ADD R0, R1, #1
ADD R0, R1
LDR R3, =0x50000000
LDR R4, =0x50000000
ADD R5, R3, R4
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	0x50000000
R3	0x50000000
R2	0x00000002
R1	0x00000001
R0	0x00000003

Exercice

```
MOV R0, #0
MOV R1, #1
MOV R2, #2
ADD R0, R1, R2
ADD R0, R1, #1
ADD R0, R1
LDR R3, =0x50000000
LDR R4, =0x50000000
ADD R5, R3, R4
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	0xA0000000
R4	0x50000000
R3	0x50000000
R2	0x00000002
R1	0x00000001
R0	0x00000003

Exercice

Un tableau de 4 éléments (32 bits) est placé en mémoire à l'adresse `0x500`

Ecrire un programme pour calculer la somme du tableau.

SUB

SUB permet de calculer la différence entre le second registre et le troisième registre est de stocker le résultat dans le premier registre

`SUB Rd, Rn, Rm`

0	0	0	1	1	0	1	Rm	Rn	Rd

`SUB Rd, Rn, #imm3`

0	0	0	1	1	1	1	imm3	Rn	Rd

`SUB Rdn, #imm8`

0	0	1	1	1	Rdn	imm8

`SUB{S} Rn, Rd, #const`

11110i0	1	1	0	1	S	Rn	0	imm3	Rd	imm8

SBC

SBC permet de calculer la différence entre le second registre et le troisième registre plus le bit Carry et de stocker le résultat dans un registre

`SBC Rdn, Rm`

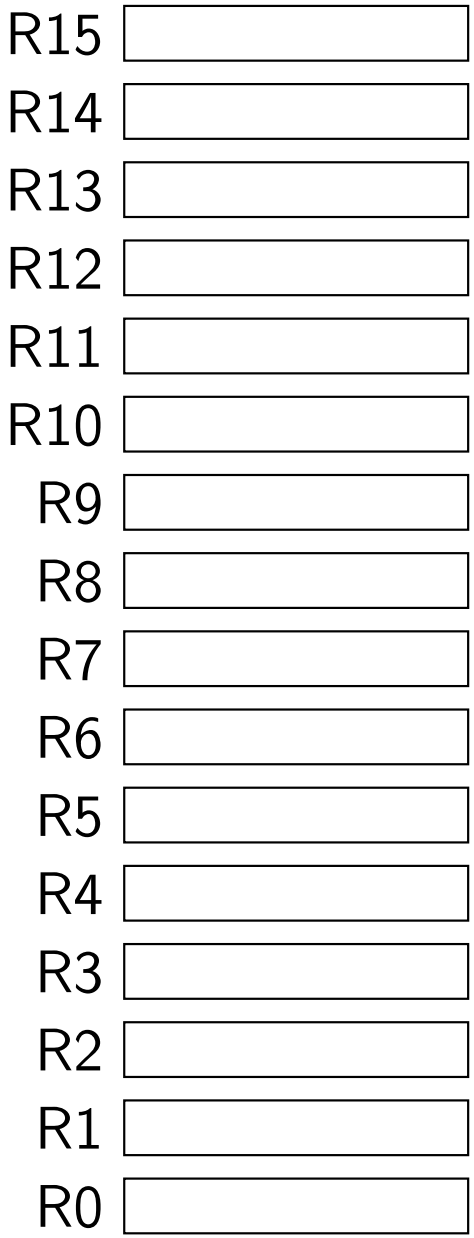
0	1	0	0	0	0	0	1	1	0	Rm	Rdn

`SBC{S} Rd, Rn, Rm, #const`

11110i0	1	0	1	0	S	Rn	0	imm3	Rd	imm8

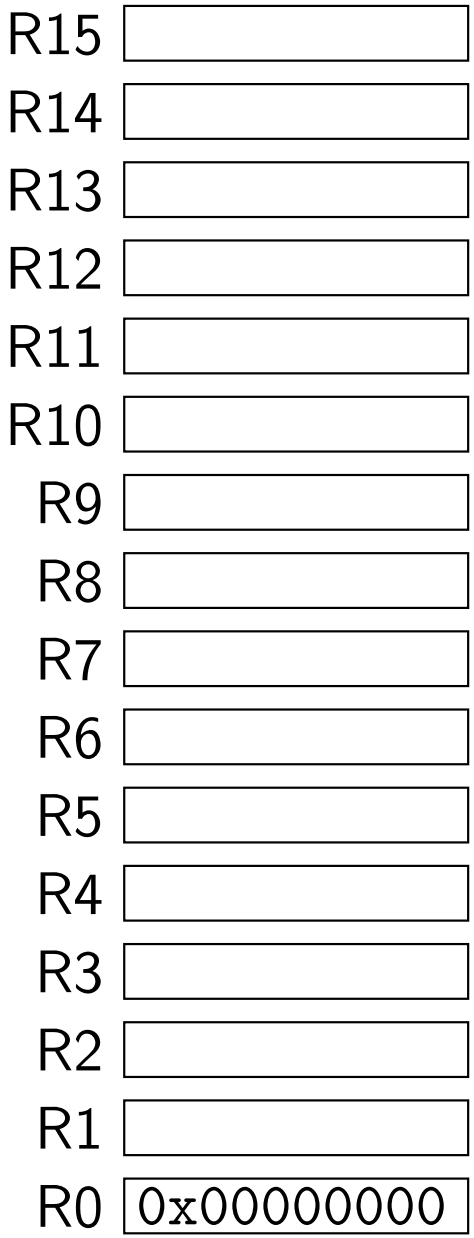
Exercice

```
MOV R0, #0
MOV R1, #2
MOV R2, #3
SUB R0, R2, R1
SUB R0, R2, #1
RSB R0, R2, #2
SUB R0, R1, R3
```



Exercice

```
MOV R0, #0
MOV R1, #2
MOV R2, #3
SUB R0, R2, R1
SUB R0, R2, #1
RSB R0, R2, #2
SUB R0, R1, R3
```



Exercice

```
MOV R0, #0
MOV R1, #2
MOV R2, #3
SUB R0, R2, R1
SUB R0, R2, #1
RSB R0, R2, #2
SUB R0, R1, R3
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x00000002
R0	0x00000000

Exercice

```
MOV R0, #0
MOV R1, #2
MOV R2, #3
SUB R0, R2, R1
SUB R0, R2, #1
RSB R0, R2, #2
SUB R0, R1, R3
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000003
R1	0x00000002
R0	0x00000000

Exercice

```
MOV R0, #0
MOV R1, #2
MOV R2, #3
SUB R0, R2, R1
SUB R0, R2, #1
RSB R0, R2, #2
SUB R0, R1, R3
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000003
R1	0x00000002
R0	0x00000001

Exercice

```
MOV R0, #0
MOV R1, #2
MOV R2, #3
SUB R0, R2, R1
SUB R0, R2, #1
RSB R0, R2, #2
SUB R0, R1, R3
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000003
R1	0x00000002
R0	0x00000002

Exercice

```
MOV R0, #0
MOV R1, #2
MOV R2, #3
SUB R0, R2, R1
SUB R0, R2, #1
RSB R0, R2, #2
SUB R0, R1, R3
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000003
R1	0x00000002
R0	0xFFFFFFFF

Exercice

```
MOV R0, #0
MOV R1, #2
MOV R2, #3
SUB R0, R2, R1
SUB R0, R2, #1
RSB R0, R2, #2
SUB R0, R1, R3
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0xFFFFFFFF
R2	0x00000003
R1	0x00000002
R0	0xFFFFFFFF

MUL

MUL permet de calculer le produit de deux registres est de stocker le résultat dans un registre

`MUL Rdn, Rm`

0	1	0	0	0	0	1	1	0	1	Rm	Rdn

`MUL{S} Rd, Rn, Rm`

1111101	1	0	0	0	0	Rn	1	1	1	1	Rd	0	0	0	0	Rm

UDIV et SDIV

UDIV et SDIV permettent réaliser une division de deux registres (non signé ou signé) est de stocker le résultat dans un registre

`UDIV{S} Rd, Rn, Rm`

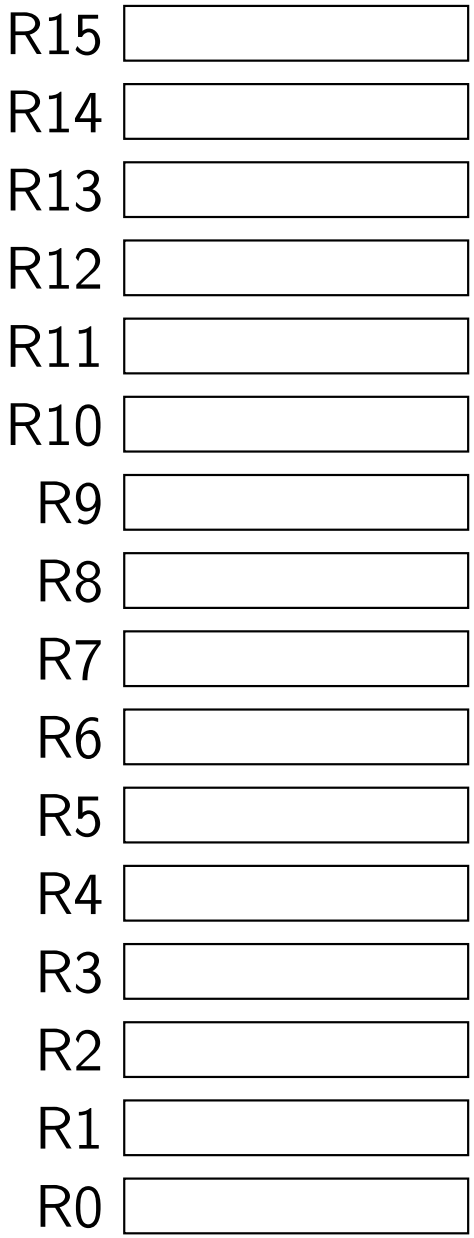
1111101	1	1	0	1	1	Rn	1	1	1	1	Rd	1	1	1	1	Rm

`SDIV{S} Rd, Rn, Rm`

1111101	1	1	0	0	1	Rn	1	1	1	1	Rd	1	1	1	1	Rm

Exercice

```
MOV R0, #10
MOV R1, #3
MOV R2, #-3
MUL R3, R0, R1
SDIV R4, R0, R1
SDIV R6, R0, R2
UDIV R5, R0, R1
UDIV R7, R0, R2
```



Exercice

```
MOV R0, #10
MOV R1, #3
MOV R2, #-3
MUL R3, R0, R1
SDIV R4, R0, R1
SDIV R6, R0, R2
UDIV R5, R0, R1
UDIV R7, R0, R2
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	0x0000000A

Exercice

```
MOV R0, #10
MOV R1, #3
MOV R2, #-3
MUL R3, R0, R1
SDIV R4, R0, R1
SDIV R6, R0, R2
UDIV R5, R0, R1
UDIV R7, R0, R2
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x00000003
R0	0x0000000A

Exercice

```
MOV R0, #10
MOV R1, #3
MOV R2, #-3
MUL R3, R0, R1
SDIV R4, R0, R1
SDIV R6, R0, R2
UDIV R5, R0, R1
UDIV R7, R0, R2
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0xFFFFFFFD
R1	0x00000003
R0	0x0000000A

Exercice

```
MOV R0, #10
MOV R1, #3
MOV R2, #-3
MUL R3, R0, R1
SDIV R4, R0, R1
SDIV R6, R0, R2
UDIV R5, R0, R1
UDIV R7, R0, R2
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	0x0000001E
R2	0xFFFFFFFD
R1	0x00000003
R0	0x0000000A

Exercice

```
MOV R0, #10
MOV R1, #3
MOV R2, #-3
MUL R3, R0, R1
SDIV R4, R0, R1
SDIV R6, R0, R2
UDIV R5, R0, R1
UDIV R7, R0, R2
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	0x00000003
R3	0x0000001E
R2	0xFFFFFFFD
R1	0x00000003
R0	0x0000000A

Exercice

```
MOV R0, #10
MOV R1, #3
MOV R2, #-3
MUL R3, R0, R1
SDIV R4, R0, R1
SDIV R6, R0, R2
UDIV R5, R0, R1
UDIV R7, R0, R2
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	0xFFFFFFFD
R4	0x00000003
R3	0x0000001E
R2	0xFFFFFFFD
R1	0x00000003
R0	0x0000000A

Exercice

```
MOV R0, #10
MOV R1, #3
MOV R2, #-3
MUL R3, R0, R1
SDIV R4, R0, R1
SDIV R6, R0, R2
UDIV R5, R0, R1
UDIV R7, R0, R2
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	0x00000003
R5	0xFFFFFFFD
R4	0x00000003
R3	0x0000001E
R2	0xFFFFFFFD
R1	0x00000003
R0	0x0000000A

Exercice

```
MOV R0, #10
MOV R1, #3
MOV R2, #-3
MUL R3, R0, R1
SDIV R4, R0, R1
SDIV R6, R0, R2
UDIV R5, R0, R1
UDIV R7, R0, R2
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	0x00000000
R6	0x00000003
R5	0xFFFFFFFD
R4	0x00000003
R3	0x0000001E
R2	0xFFFFFFFD
R1	0x00000003
R0	0x0000000A

AND

AND permet de réaliser un ET bit à bit entre deux registres (ou une constante)

`AND Rdn, Rm`

0	1	0	0	0	0	0	0	0	0	Rm	Rdn

`AND{S} Rd, Rn, #const`

11110i0	0	0	0	0	S	Rn	0	imm3	Rd	imm8

`AND{S} Rd, Rn, shift`

1110101	0	0	0	0	S	Rn	0	imm3	Rd	imm2	type	Rm

ORR

ORR permet de réaliser un OU bit à bit entre deux registres (ou une consante)

`OOR{S} Rd, Rn, #const`

11110i0	0	0	1	0	S	Rn	0	imm3	Rd	imm8

`OOR{S} Rd, Rn, shift`

1110101	0	0	1	0	S	Rn	0	imm3	Rd	imm2	type	Rm

EOR

EOR permet de réaliser un OU exclusif entre deux registres (ou un constante)

`EOR Rdn, Rm`

0	1	0	0	0	0	0	0	0	1	Rm	Rdn

`EOR{S} Rd, Rn, #const`

11110i0	0	1	0	0	S	Rn	0	imm3	Rd	imm8

`EOR{S} Rd, Rn, shift`

1110101	0	1	0	0	S	Rn	0	imm3	Rd	imm2	type	Rm

ORN

ORN permet de réaliser un ET bit à bit entre un registre et le complément d'un autre (ou d'une constante)

`ORN Rdn, Rm`

0	1	0	0	0	0	0	0	0	0	Rm	Rdn

`ORN{S} Rd, Rn, #const`

11110i0	0	0	1	1	S	Rn	0	imm3	Rd	imm8

`ORN{S} Rd, Rn, shift`

1110101	0	0	1	1	S	Rn	0	imm3	Rd	imm2	type	Rm

BIC

BIC permet de mettre à zéro les bits communs de deux registres (ou d'une constante)

`BIC Rdn, Rm`

0	1	0	0	0	0	1	1	1	0	Rm	Rdn

`BIC{S} Rd, Rn, #const`

11110i0	0	0	0	1	S	Rn	0	imm3	Rd	imm8

`BIC{S} Rd, Rn, shift`

1110101	0	0	0	1	S	Rn	0	imm3	Rd	imm2	type	Rm

LSL

LSL permet d'effectuer un décalage logique vers la gauche

`LSL Rd, Rm, #imm5`

0	0	0	0	0	imm5	Rm	Rd

`LSL Rdn, Rm`

0	1	0	0	0	0	0	0	1	0	Rm	Rdn

`LSL{S} Rd, Rn, Rm`

1111101	0	0	0	0	S	Rn	1	1	1	1	Rd	0	0	0	0	Rm

LSR

LSR permet d'effectuer un décalage logique vers la droite

`LSR Rd, Rm, #imm5`

0	0	0	0	1	imm5	Rm	Rd

`LSR Rdn, Rm`

0	1	0	0	0	0	0	0	1	1	Rm	Rdn

`LSR{S} Rd, Rn, Rm`

1111101	0	0	0	1	S	Rn	1	1	1	1	Rd	0	0	0	0	Rm

ASR

ASR permet d'effectuer un décalage arithmétique vers la droite

`ASR Rd, Rm, #imm5`

0	0	0	1	0	imm5	Rm	Rd

`ASR Rdn, Rm`

0	1	0	0	0	0	0	1	0	0	Rm	Rdn

`ASR{S} Rd, Rn, Rm`

1111101	0	0	1	0	S	Rn	1	1	1	1	Rd	0	0	0	0	Rm

- Le décalage arithmétique à gauche peut être effecué avec un décalage logique à gauche

ROR

ROR permet d'effectuer une rotation vers la droite

`ROR Rdn, Rm`

0	1	0	0	0	0	0	1	1	1	Rm	Rdn

`ROR{S} Rd, Rn, Rm`

1111101	0	0	1	1	S	Rn	1	1	1	1	Rd	0	0	0	0	Rm

- La rotation a droite de N bits n'est pas implémentée mais peut être effecuée en faisant une rotation de $32 - N$

CMP

CMP permet de comparer les valeurs de deux registres (ou d'un constante)

`CMP Rd, #imm8`

0	0	1	0	1	Rd	imm8

`CMP Rm, Rn`

0	1	0	0	0	0	1	0	1	0	Rm	Rn

`CMP Rn, #const`

11110i0	1	1	0	1	1	Rn	0	imm3	Rd	imm8

B (Branch) permet de modifier la valeur du registre R15 (PC)

`B #const`

1	1	1	0	0	imm11

`B #const`

11110Si	imm10	1	0	J1	0	J2	imm11

Exercice

```
s: MOV R0, #1
   MOV R1, #2
   B m1
   MOV R2, #3
   MOV R3, #4
m: MOV R4, #5
   MOV R5, #6
   B e
   MOV R6, #7
   MOV R7, #8
e: B e
```

R15	0x00053470
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	

Exercice

```
s: MOV R0, #1
   MOV R1, #2
   B m1
   MOV R2, #3
   MOV R3, #4
m: MOV R4, #5
   MOV R5, #6
   B e
   MOV R6, #7
   MOV R7, #8
e: B e
```

R15	0x00053474
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	0x00000001

Exercice

```
s: MOV R0, #1
   MOV R1, #2
   B m1
   MOV R2, #3
   MOV R3, #4
m: MOV R4, #5
   MOV R5, #6
   B e
   MOV R6, #7
   MOV R7, #8
e: B e
```

R15	0x00053478
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x00000002
R0	0x00000001

Exercice

```
s: MOV R0, #1
   MOV R1, #2
   B m1
   MOV R2, #3
   MOV R3, #4
m: MOV R4, #5
   MOV R5, #6
   B e
   MOV R6, #7
   MOV R7, #8
e: B e
```

R15	0x00053480
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x00000002
R0	0x00000001

Exercice

```
s: MOV R0, #1
   MOV R1, #2
   B m1
   MOV R2, #3
   MOV R3, #4
m: MOV R4, #5
   MOV R5, #6
   B e
   MOV R6, #7
   MOV R7, #8
e: B e
```

R15	0x00053484
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	0x00000005
R3	
R2	
R1	0x00000002
R0	0x00000001

Exercice

```
s: MOV R0, #1
   MOV R1, #2
   B m1
   MOV R2, #3
   MOV R3, #4
m: MOV R4, #5
   MOV R5, #6
   B e
   MOV R6, #7
   MOV R7, #8
e: B e
```

R15	0x00053484
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	0x00000006
R4	0x00000005
R3	
R2	
R1	0x00000002
R0	0x00000001

Exercice

```
s: MOV R0, #1
   MOV R1, #2
   B m1
   MOV R2, #3
   MOV R3, #4
m: MOV R4, #5
   MOV R5, #6
   B e
   MOV R6, #7
   MOV R7, #8
e: B e
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	0x00000006
R4	0x00000005
R3	
R2	0x00005678
R1	0x00000002
R0	0x00000001

B.. (Branch) permet de modifier la valeur du registre R15 (PC) suivant une condition particulière

`B.. #const`

1	1	0	1	cond	imm8

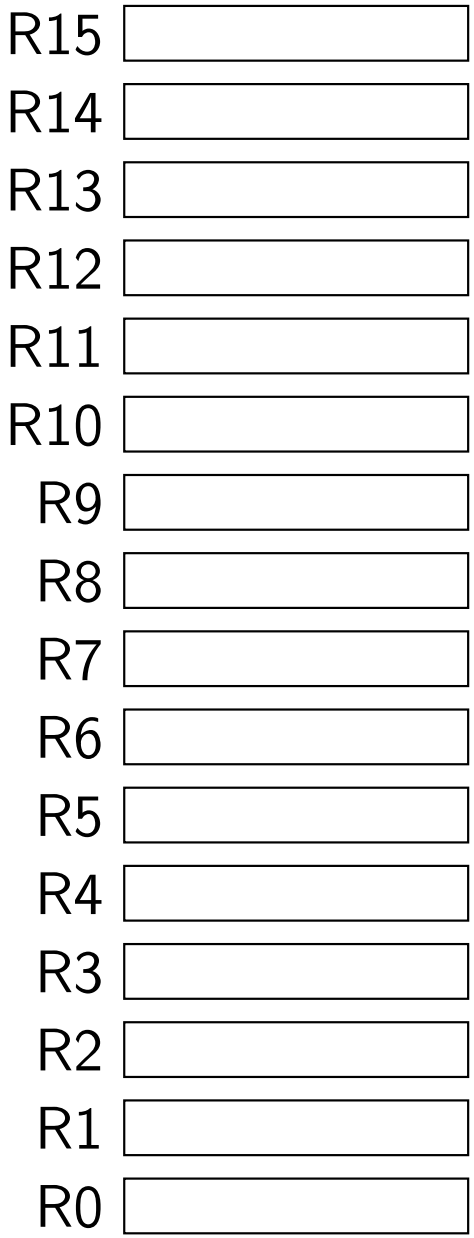
`B #const`

11110Si	cond	imm6	1	0	J1	0	J2	imm11

Condition	Suffixe	Description	Flags
0000	EQ	EQual	Z=1
0001	NE	Not Equal	Z=0
0010	CS/HS	unsigned Higher or Same	C=1
0011	CC/LO	unsigned LOwer	C=0
0100	MI	MInus (negative)	N=1
0101	PL	PLus (positive or zero)	N=0
0110	VS	oVerflow Set	V=1
0111	VC	oVerflow Clear	V=0
1000	HI	unsigned HIgher	C=1, Z=0
1001	LS	unsigned Lower or Same	C=0, Z=1
1010	GE	signed Greater or Equal	N=V
1011	LT	signed Less Than	N!=V
1100	GT	signed Greater Than	Z=0 et N=V
1101	LE	signed Less or Equal	Z=1 ou N!=V
1110	AL	ALways	

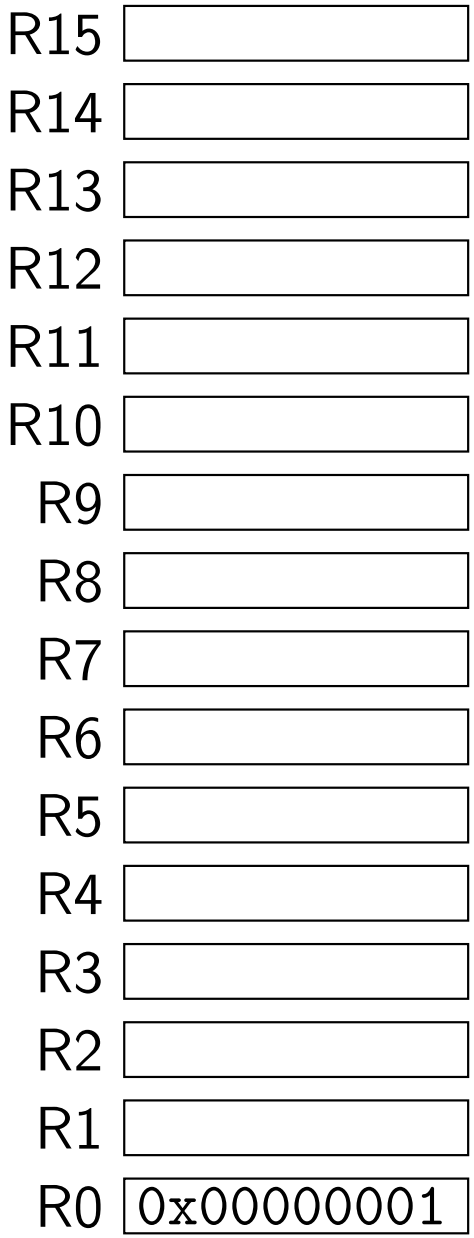
Exercice

```
s: MOV R0, #1
   ADD R1 R0 R0
   CMP R1, #3
   BEQ e
   MOV R0, #0
   CMP R2, #2
   BEQ e
   MOV R0, #1
e: B e
```



Exercice

```
s: MOV R0, #1
   ADD R1 R0 R0
   CMP R1, #3
   BEQ e
   MOV R0, #0
   CMP R2, #2
   BEQ e
   MOV R0, #1
e: B e
```



Exercice

```
s: MOV R0, #1
   ADD R1 R0 R0
   CMP R1, #3
   BEQ e
   MOV R0, #0
   CMP R2, #2
   BEQ e
   MOV R0, #1
e: B e
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x00000002
R0	0x00000001

Exercice

```
s: MOV R0, #1
   ADD R1 R0 R0
   CMP R1, #3
   BEQ e
   MOV R0, #0
   CMP R2, #2
   BEQ e
   MOV R0, #1
e: B e
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x00000002
R0	0x00000001

Exercice

```
s: MOV R0, #1
   ADD R1 R0 R0
   CMP R1, #3
   BEQ e
   MOV R0, #0
   CMP R2, #2
   BEQ e
   MOV R0, #1
e: B e
```

R15	
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x00000002
R0	0x00000000

BL

BL (Branch and Link) permet de sauvegarder l'adresse de retour dans R14 (LR) modifier la valeur du registre R15 (PC)

`BL #const`

11110Si	imm10	1	1	J1	1	J2	imm11

BX

BX (Branch and eXchange) permet charger l'adresse contenue dans un registre dans le PC

`BX #const`

0	1	0	0	0	1	1	1	0	Rm	0	0	0

Exercice

```
s:  MOV R0, #1
    MOV R1, #1
    BL add
    MOV R0, #0
    MOV R1, #0
e:  B e
add:ADD R2, R1, R0
    BX LR
```

R15	0x00058390
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	

Exercice

```
s: MOV R0, #1
   MOV R1, #1
   BL add
   MOV R0, #0
   MOV R1, #0
e: B e
add:ADD R2, R1, R0
   BX LR
```

R15	0x00058394
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	0x00000001

Exercice

```
s:  MOV R0, #1
    MOV R1, #1
    BL add
    MOV R0, #0
    MOV R1, #0
e:  B e
add:ADD R2, R1, R0
    BX LR
```

R15	0x00058398
R14	
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x00000001
R0	0x00000001

Exercice

```
s: MOV R0, #1
   MOV R1, #1
   BL add
   MOV R0, #0
   MOV R1, #0
e: B e
add:ADD R2, R1, R0
   BX LR
```

R15	0x000583A0
R14	0x00058398
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x00000001
R0	0x00000001

Exercice

```
s: MOV R0, #1
   MOV R1, #1
   BL add
   MOV R0, #0
   MOV R1, #0
e: B e
add:ADD R2, R1, R0
   BX LR
```

R15	0x000583A4
R14	0x00058398
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000002
R1	0x00000001
R0	0x00000001

Exercice

```
s: MOV R0, #1
   MOV R1, #1
   BL add
   MOV R0, #0
   MOV R1, #0
e: B e
add:ADD R2, R1, R0
   BX LR
```

R15	0x00058398
R14	0x00058398
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000002
R1	0x00000001
R0	0x00000001

Exercice

```
s: MOV R0, #1
   MOV R1, #1
   BL add
   MOV R0, #0
   MOV R1, #0
e: B e
add:ADD R2, R1, R0
   BX LR
```

R15	0x000583A0
R14	0x00058398
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000002
R1	0x00000001
R0	0x00000000

Exercice

```
s:  MOV R0, #1
    MOV R1, #1
    BL add
    MOV R0, #0
    MOV R1, #0
e:  B e
add:ADD R2, R1, R0
    BX LR
```

R15	0x000583A4
R14	0x00058398
R13	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	0x00000002
R1	0x00000000
R0	0x00000000

Interruption

Exemple

On souhaite détecter l'appui sur une touche d'un clavier:

- Polling actif
- Interruptions

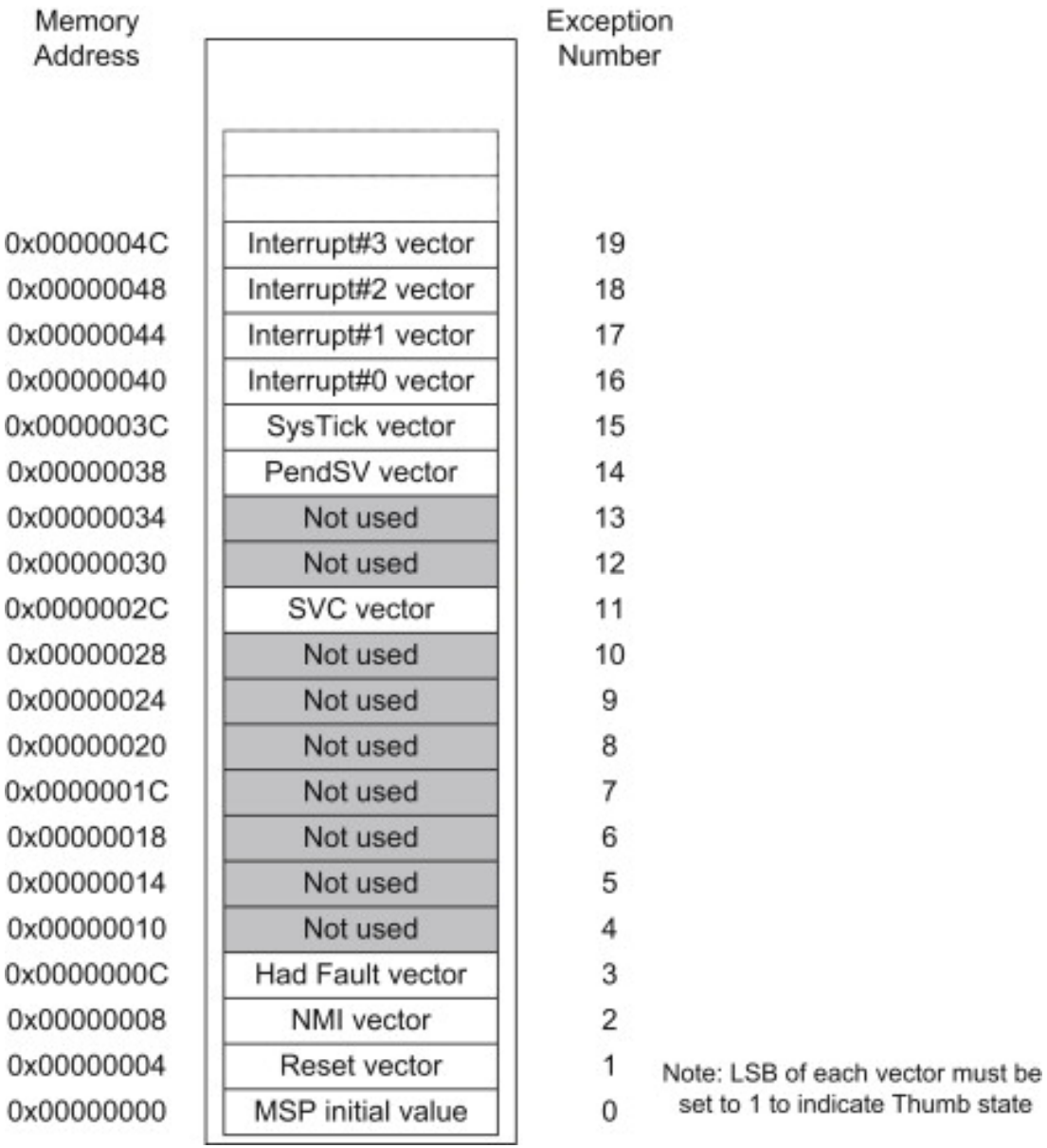
Une interruption est une fonction appelée à la suite d'un évènement externe.

Après l'interruption, le MCU doit retourner à l'endroit exact du programme ou il a été interrompu.

Interruptions

- Les Cortex M3 et M4 supportent 240 interruptions différentes.
- Chaque interruption possède un numéro pour être identifiée de manière unique
- Chaque interruption possède une priorité (configurable en software)
- Chaque interruption peut être préemptive ou non (configurable en software)
- Le numéro d'interruption est en fait un index dans la table des vecteur d'interruption (stockée en mémoire)
- Chaque élément de la table contient l'adresse de la fonction d'interruption

Interruptions



Interruptions

A chaque interruption reçue, le processeur effectue les taches suivantes:

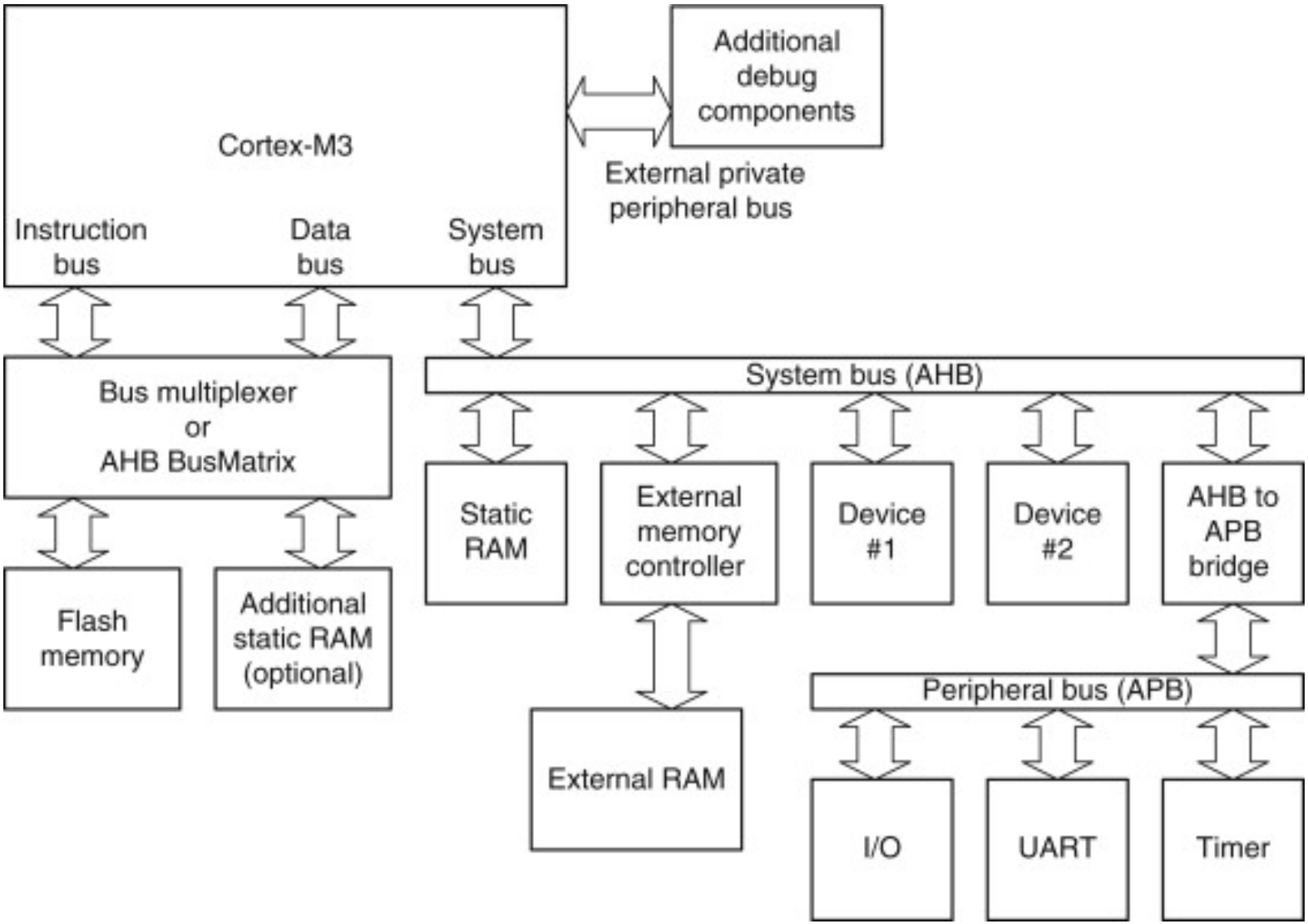
1. Stockage des registres xPSR, PC, LR, R12, R3, R2, R1, R0 sur la pile
2. PC = MEM[NISR]
3. Execution de l'interruption
4. Déstockage des registres xPSR, PC, LR, R12, R3, R2, R1, R0 sur la pile
5. Retour au programme

Boot

Le processus de démarrage (ou de reset) d'un Cortex est relativement simple:

1. Récupérer la donnée sur 32 bit située à l'adresse 0x0
2. Initialiser SP avec cette valeur
3. Récupérer la donnée sur 32 bit située à l'adresse 0x4
4. Initialiser PC avec cette valeur
5. Reset Handler va a son tour lancer la fonction `main()`

Périphériques



GPIO

- Une GPIO correspond simplement à une pin du MCU.
- Configurable en entrée ou en sortie (en software, dynamiquement pendant l'exécution)
- Pin digitale: 0 (0 V) ou 1 (3.3 V)

In Input:

- Pull-Up
- Pull-Down

In Output:

- Push-Pull
- Open-Drain

Timer

- Compteur incrémenté à une fréquence donnée
- Base de temps: exécution d'une fonction chaque seconde, milliseconde...
- Output Compare: commande d'une GPIO à un instant particulier
- Input Capture: mesure du temps à l'apparition d'un évenement

UART

- Liaison série asynchrone 2 fils (TX, RX + GND)
- Permet de transmettre ou de recevoir un charactère à la fois
- Débit fixe (généralement 9600 ou 115200 bauds)
- Bit de start, de stop, et de parité (optionel)
- Pas d'accès multiple

SPI

- Liaison série synchrone 4 fils (MOSI, MISO, SCLK, SS + GND)
- Chaque appareil possède une adresse sur 7 bits
- Débit jusqu'a quelques MHz
- Accès multiple

- Liaison série synchrone 2 fils (SDA, SCL + GND)
- Chaque appareil possède une adresse sur 7 bits
- Débit: 100kb/s, 400kb/s, 1Mb/s...
- Bit de start, de stop, et de parité (optionel)
- Accès multiple

CNA/CAN

CNA

Permet de générer une tension analogique sur 2^N niveaux à partir d'un nombre sur N bits

CAN

Permet d'obtenir d'un nombre sur N bits à partir d'une tension analogique