Architecture des Ordinateurs

Chapitre 3 : Architecture interne

Jacques Supcik, Daniel Gachet, Luca Haab

2023-11-17

Table des matières

3 Architecture interne

Il existe une pléthore de µP et µC sur le marché. En faire le tour est totalement irréaliste. Bien que tous différents, ils ont tous beaucoup de similitudes. Ce chapitre présente l'architecture interne des processeurs ARM Cortex version 7. Pourquoi les µP ARMv7 ? Bien tout simplement parce qu'il faut bien faire un choix, mais plus sérieusement, ces processeurs se sont énormément démocratisés lors des dernières décades.

3.1 Architecture ARM

Conçus avec une architecture plus simple comparativement à d'autres familles de processeurs, ceci dans le but de réduire l'énergie nécessaire à leur fonctionnement, les processeurs ARM (acronyme de Advanced RISC Machines et anciennement de Acorn RISC Machine) équipement à ce jour grand nombre de systèmes embarqués, de systèmes sur puce (SoC) et d'objets de l'Internet (IoT). Les processeurs ARM sont fabriqués sous licence par un grand nombre de manufacturiers.

3.1.1 Evolution

Depuis la première version de leur architecture (ARMv1) en 1985, les µP et µC ARM n'ont pas cessé de faire évoluer leur architecture, d'améliorer leurs performances et d'ajouter de nouvelles fonctionnalités (figure 3.1 et figure 3.2).

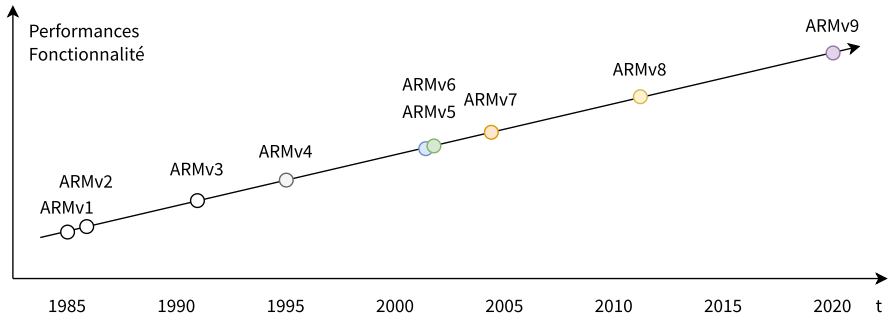


FIGURE 3.1 – Evolution de l'architecture ARM

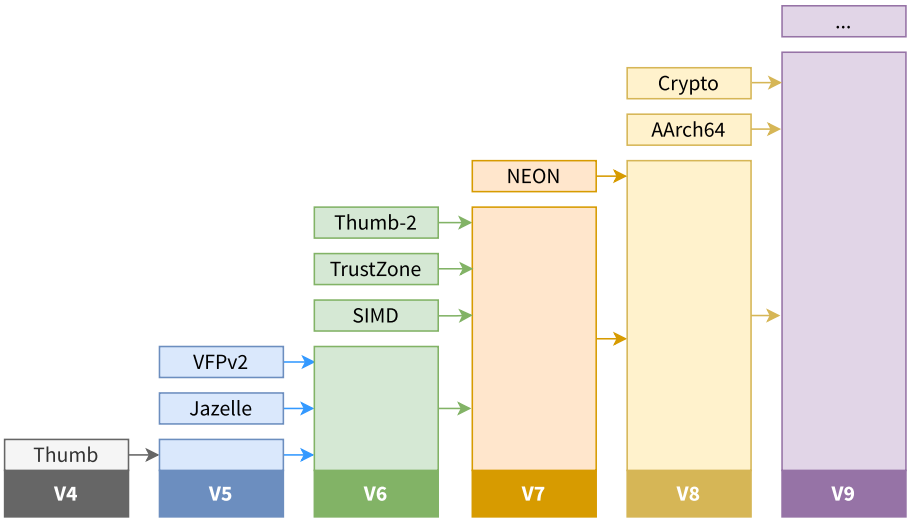


FIGURE 3.2 – Evolution de la fonctionnalité

Thumb est un sous-ensemble du jeu d'instructions ARM classique à 32 bits. Codé sur 16 bits, il permet d'obtenir un code plus compact afin de réduire l'empreinte mémoire pour son stockage. Contrairement au jeu d'instructions ARM, les instructions Thumb ne peuvent s'exécuter conditionnellement. Ce jeu d'instructions est particulièrement bien adapté aux systèmes sur puce (SoC).

Jazelle est une extension permettant l'exécution de bytecode Java.

VFP, **SIMD** et **NEON** sont une suite d'extensions permettant le calcul à virgule flottante (VFP - *Vector Floating Point*) selon la norme ANSI/IEEE Std 754-1985 et le calcul vectoriel. L'architecture SIMD (*Simple Instruction Multiple Data*) offre un vrai parallélisme lors de calcul vectoriel. L'unité NEON est le dernier moteur de calcul des processeurs ARM, intégrant les processeurs VFP et SIMD tout en améliorant leurs performances.

Thumb-2 est une extension du jeu d'instructions Thumb. Avec certaines instructions codées sur 32 bits, Thumb-2 offre plus de possibilités et une meilleure utilisation des ressources du processeur.

TrustZone est une zone sécurisée offrant un environnement d'exécution isolé. Il garantit l'intégrité des applications ainsi que la confidentialité des données.

AArch64 ajoute l'architecture 64 bits aux processeurs ARM.

Crypto est une extension offrant de nouvelles instructions cryptographiques. Ces instructions permettent d'accélérer des fonctions de chiffrement (AES - *Advanced Encryption Standard*) et de hachage (SHA - *Secure Hash Algorithm*).

3.1.2 Profils des µP ARMv7

ARM propose dans sa famille de coeurs Cortex ARMv7 trois profils :

- Cortex-A
- Cortex-R
- Cortex-M

Cortex-A - Processeurs d'application pour systèmes avec hautes exigences en performance

Le profil Cortex-A permet la réalisation d'applications exigeant la mise en oeuvre de système d'exploitation riche tel que GNU/Linux. Ce profil propose le jeu d'instructions ARM et offre une virtualisation de sa mémoire grâce à une MMU (*Memory Management Unit*). Pour augmenter ses performances, ce profil utilise des horloges à hautes fréquences, implémente un pipeline profond et propose l'unité NEON comme support multimédia. Les marchés cibles sont les applications mobiles (smart phones, smart tv, serveurs économes en énergie, équipements de réseau, récepteurs satellites, systèmes automobiles, etc.).

Cortex-R - Processeurs haute performance pour applications embarquées de contrôle temps réel

Le profil Cortex-R permet la réalisation d'applications critiques mettant en oeuvre des systèmes d'exploitation temps réel (RTOS), tel que FreeRTOS, afin de satisfaire des exigences temps réel fortes. Ce profil a mis un accent sur son architecture pour augmenter la fiabilité, la robustesse et la tolérance aux fautes. Il implémente entre autres une protection d'accès à la mémoire grâce à une MPU (*Memory Protection Unit*), un double-coeur pour augmenter la tolérance aux pannes CPU. Les marchés cibles sont les applications industrielles, telles que l'automobile, l'avionique, les automates programmables (PLC - *Programmable Logic Controller*), la robotique, l'automation.

Cortex-M - Processeurs économes en énergie pour une large palette d'applications IoT

Le profil Cortex-M permet la réalisation d'applications embarquées à basse voire très basse consommation ou d'applications exigeant une large palette de périphériques d'entrées/sorties, mettant en oeuvre des systèmes sur puce (SoC). Ce profil intègre sur une même puce les mémoires volatiles et non volatiles, ainsi que les contrôleurs de périphériques d'entrées/sorties nécessaires à l'application. Ces µC permettent de réduire au maximum la surface sur le circuit imprimé (PCB - *Printed Circuit Board*). La conception de ce profil permet aussi bien la réalisation d'applications en "bare metal" qu'avec un RTOS, tel que Mbed OS ou FreeRTOS. Les marchés cibles sont des objets de l'Internet, des appareils industriels ou de grande consommation.

3.1.3 Caractéristiques des µP ARMv7

Les µP et µC ARM sont des processeurs avec une architecture RISC à 32 bits (ARMv1 à ARMv7) et à 64 bits (ARMv8 et ARMv9). Les principales caractérisques communes à tous les profils de l'architecture

ARMv7 sont :

- Registres de taille et format uniforme (32 bits)
- Mécanismes de transfert (*Load and Store*) avec des opérations s'effectuant uniquement sur les registres et non avec la mémoire
- Modes d'adressage simples où toutes les adresses de transfert sont déterminées uniquement à partir du contenu des registres
- Modes d'adressage auto-incrémentés et auto-décrémentés afin d'optimiser l'exécution de boucles
- Jeu d'instructions orthogonal (registres communs pour les données et les adresses)

Les caractéristiques spécifiques des profils A et R sont :

- Jeu d'instructions ARM avec un format de taille fixe (32 bits) afin de simplifier son décodage
- Exécution essentiellement en un seul cycle
- Instructions s'exécutant conditionnellement afin d'améliorer les performances

Les caractéristiques spécifiques du profil M sont :

- Jeu d'instructions Thumb avec un format de taille fixe (16 bits) ou Thumb-2 avec un format de taille variable (16 et 32 bits)

3.2 Exécution d'un programme informatique

Avant d'étudier l'architecture interne des processeurs ARMv7, il est intéressant de réfléchir aux éléments de base d'un programme informatique et à comment un processeur les voit. Il existe certainement plusieurs réponses, cependant pour un processeur il est acceptable de réduire un programme informatique à une suite d'instructions manipulant des données.

Ok, bien, mais de quels outils le processeur doit-il disposer pour exécuter ce programme informatique ?

Selon l'architecture générale, le processeur utilise son unité de contrôle (CU) pour lire les instructions du programme, les décoder et finalement les exécuter. Il utilise son unité arithmétique et logique (ALU) avec sa banque de registres pour manipuler les données contenues dans sa mémoire centrale et dans les registres de ses contrôleurs de périphériques d'entrées/sorties.

Ceci dit, il reste encore quelques points ouverts :

- Comment organiser le code et ses instructions ?
- Comment lire les instructions ?
- Comment organiser les données ?
- Comment traiter les données ?

3.2.1 Organisation du code

En regardant un programme informatique, on constate rapidement que celui-ci utilise le principe de modularité consistant à fractionner le code en une suite de sous-programmes, lesquels sont finalement agrégés par un module principal. Ces sous-programmes sont généralement des collections de fonctions, pour un développement basé sur un paradigme impératif, ou de classes et méthodes, pour une réalisation orientée objet. Cette modularité apporte naturellement beaucoup d'avantages. En voici quelques-uns :

- Réduction de la taille du programme principal et des sous-programmes
- Création de bibliothèques de composants simplifiant le développement de grands programmes
- Simplification de la vérification et validation du programme en effectuant des tests unitaires sur chacun des composants
- Réutilisation de composants existants et parfaitement validés

Maintenant si l'on regarde de plus près le code exécuté par le processeur, il apparaît assez vite que quatre algorithmes de base sont nécessaires :

- Suites d'instructions linéaires s'exécutant séquentiellement (figure 3.3)
- Branchements conditionnels et inconditionnels (figure 3.4)
- Boucles (figure 3.5)
- Appels de fonction (figure 3.6)

Code C

```c
int lineaire(int a) {
    a += 10;
    a *= 213;
    return a;
}
```

Code assembleur Thumb-2

```
lineaire:
    adds r0, #10
    movs r3, #213
    muls r0, r3, r0
    bx   lr
```

Code Machine Cortex-M

```
0x080001b6: 0a 30
0x080001b8: d5 23
0x080001ba: 58 43
0x080001bc: 70 47
```

FIGURE 3.3 – Algorithme : suite linéaire d'instructions

Code C

```c
int branchement(int a) {
    if (a > 0) {
        a++;
    } else {
        a--;
    }
    return a;
}
```

Code assembleur Thumb-2

```
branchement:
    cmp  r0, #0        test
    ble  1f            1r branchement conditionnel
    adds r0, #1
    b    2f            2e branchement inconditionnel
1:  subs r0, #1
2:  bx   lr
```

Code Machine Cortex-M

```
0x080001be: 00 28
0x080001c0: 01 dd
0x080001c2: 01 30
0x080001c4: 00 e0
0x080001c6: 01 38
0x080001c8: 70 47
```

FIGURE 3.4 – Algorithme : branchements

Les algorithmes ci-dessus présentent d'abord un code C, puis sa traduction en code assembleur avec le jeu d'instructions Thumb-2 et finalement le code machine prêt à s'exécuter sur un µC Cortex-M.

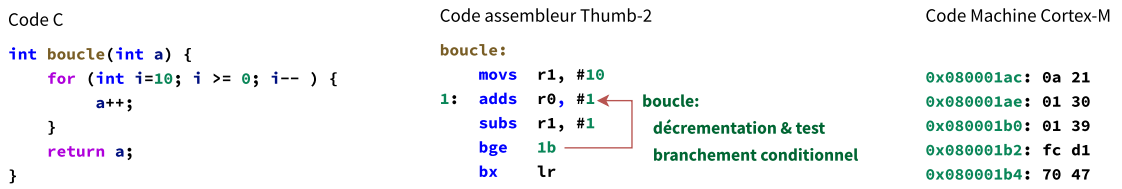


FIGURE 3.5 – Algorithme : boucle

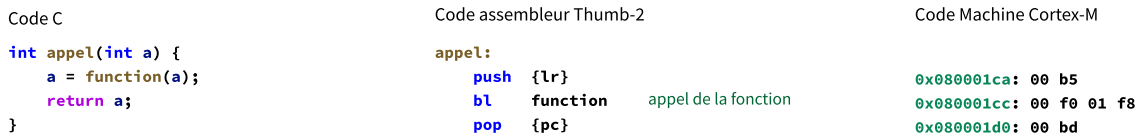


FIGURE 3.6 – Algorithme : appel de fonction

3.2.2 Compteur ordinal

Les exemples ci-dessus donnent une bonne vue d'ensemble des algorithmes nécessaires pour la réalisation d'un programme informatique, mais n'apportent pas toutes les réponses.

- Comment le processeur connaît-il la prochaine instruction à charger (figure 3.3) ?
- Comment le processeur procède-t-il lors de branchements ou de boucles (figures 3.4 et 3.5) ?
- Comment le processeur poursuit-il l'exécution des instructions placées après l'appel d'une fonction (figure 3.6) ?

Pour répondre à la première question, le processeur utilise un de ses registres pour adresser et charger la prochaine instruction à exécuter de la mémoire centrale dans son unité de contrôle. Ce registre est nommé **compteur ordinal** (PC - *Program Counter*). Le µP utilise l'adresse contenue dans ce registre pour charger l'instruction dans l'unité de contrôle (figure 3.7). Une fois l'instruction transférée dans l'unité de contrôle, le PC est incrémenté de la taille de l'instruction, ceci afin de pointer sur la prochaine instruction (figure 3.8).

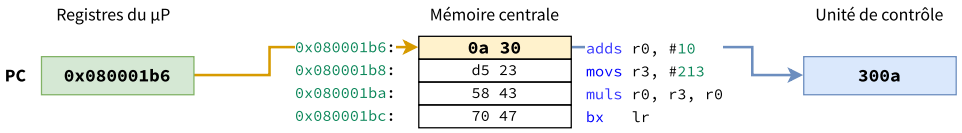


FIGURE 3.7 – PC : chargement de l'instruction (fetch)

Les branchements et les boucles provoquent une cassure dans le déroulement séquentiel de l'exécution du programme. Le processeur doit effectuer un saut (*jump*) dans le code pour pointer sur l'instruction correspondante au branchement. Pour effectuer ce saut, l'instruction de branchement remplace

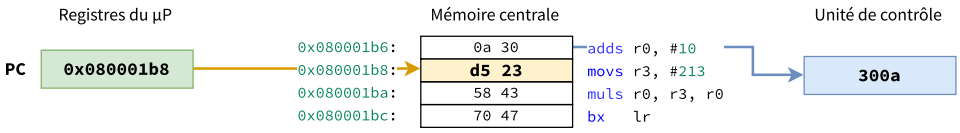


FIGURE 3.8 – PC : incrémentation

l'adresse contenue dans le registre PC par l'adresse de l'instruction pointée par le branchement.

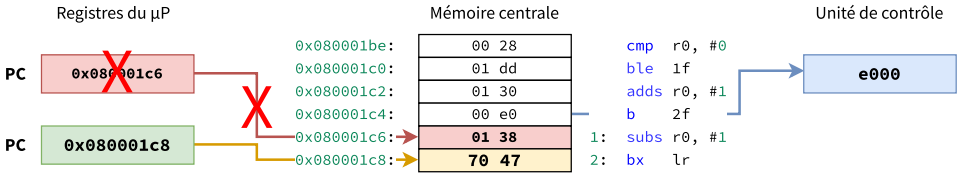


FIGURE 3.9 – PC : branchement

Dans l'exemple ci-dessus (figure 3.9), la valeur du PC, après le chargement de l'instruction "`b 2f`" située à l'adresse `0x080001c4` est de `0x080001c6`. Cependant, après l'exécution de cette instruction, la nouvelle valeur du PC est `0x080001c8`. Ce changement permet de satisfaire l'instructuction de branchement "`b 2f`" pour pointer ainsi sur l'instruction "`bx lr`" correspondante au branchement souhaité.

Reste maintenant encore à résoudre le troisième problème, l'appel de fonction. Pour l'illustrer, il est intéressant d'étudier l'implémentation de la suite de Fibonacci à l'aide d'une fonction récursive (figure 3.10). Comme le montre le code C, cette fonction effectue un double appel à elle-même.

Code C

```
long fibonacci(long n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci_c(n - 1)
         + fibonacci_c(n - 2);
}
```

Code assembleur Thumb-2

```
fibonacci:
1:   cmp   r0, #1
2:   it    le
3:   bxle  lr          retour
4:   push  {r4, lr}    sauvegarde
5:   movs  r4, r0
6:   subs  r0, #1
7:   bl    fibonacci   1r appel de la fonction
8:   movs  r1, r0
9:   subs  r0, r4, #2
10:  movs  r4, r1
11:  bl    fibonacci   2e appel de la fonction
12:  adds  r0, r4
13:  pop   {r4, pc}    restauration et retour
```

Code Machine Cortex-M

```
0x080001d2: 01 28
0x080001d4: d8 bf
0x080001d6: 70 47
0x080001d8: 10 b5
0x080001da: 04 00
0x080001dc: 01 38
0x080001de: ff f7 f8 ff
0x080001e2: 01 00
0x080001e4: a0 1e
0x080001e6: 0c 00
0x080001e8: ff f7 f3 ff
0x080001ec: 00 19
0x080001ee: 10 bd
```

FIGURE 3.10 – PC : appel de fonction (Suite de Fibonacci)

Pour le processeur, ce double appel se traduit par l'exécution des deux instructions "`bl fibonacci`", lignes 7 et 11. Le problème de l'adresse de retour apparaît très clair ici. Comment

informer la fonction `fibonacci` qu'elle doit retourner à la ligne 8 après le premier appel et à la ligne 12 après le deuxième? Il est évident que le compilateur et l'éditeur de liens ne peuvent pas le résoudre, car nous avons deux adresses différentes. Le problème se complique encore, car la fonction `fibonacci` est appelée par une ou plusieurs fonctions extérieures à ce code. Alors que faire? La solution choisie par ARM est de stocker cette adresse de retour dans un de ses registres. Ce registre est nommé **registre de lien** (LR - *Link Register*). Cette opération s'effectue lors de l'appel de la fonction, grâce à l'instruction `bl` (*branch and link*). Dès lors, la fonction appelée peut revenir au programme appelant en utilisant par exemple l'instruction "`bx lr`". Dans l'exemple ci-dessus, l'instruction conditionnelle "`bxle lr`" à la ligne 3 effectue cette opération. Un problème subsiste encore, les appels multiples. En effet, cette fonction, par sa récursivité, s'appelle elle-même plusieurs fois et à deux endroits distincts. Il ne suffit donc pas de sauver l'adresse de retour dans le registre LR. Si on se contente que de cela, l'adresse de retour sera perdue, car écrasée par les appels récursifs. Il est donc essentiel de préalablement sauver le contenu du registre LR avant d'effectuer l'appel et de le restaurer après l'appel. Le jeu d'instructions ARM et Thumb-2 implémente à cet effet les instructions `push` pour la sauvegarde de registres et `pop` pour leur restauration. Dans l'exemple ci-dessus, la sauvegarde est effectuée à la ligne 4, tandis que la restauration est effectuée à la ligne 13. La restauration s'accompagne avec le retour à la fonction appelante en plaçant la valeur sauvegardée du registre LR, "`push {r4, lr}`", ligne 3, directement dans le registre PC "`pop {r4, pc}`", ligne 13.

3.2.3 Organisation des données

Avec les langages de programmation évolués, toutes les données ont un type. Ce type définit la nature de la donnée, les valeurs qu'elle peut prendre, ainsi que les opérateurs qui permettent de les traiter et de les manipuler. Pour le processeur, il est possible de classer les données d'un programme informatique en quatre catégories (figure 3.11).

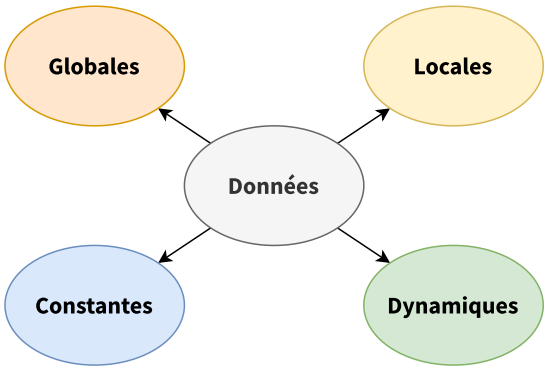


FIGURE 3.11 – Données : catégories

Les données **constantes** sont des données avec un accès en lecture seule. Sur les systèmes sur puce (SoC), ces données sont stockées avec le code dans la mémoire non volatile. Sur les systèmes embarqués, équipés d'une MMU, elles sont placées en RAM dans une zone protégée en écriture.

Les données **globales** sont des données volatiles accessibles par toutes les fonctions d'un composant logiciel voire même par l'ensemble des fonctions de l'application. Elles disposent d'un double accès en lecture et en écriture. L'éditeur de liens (*linker*) fixe leur emplacement en mémoire, leur adresse, lors de la génération de l'application. Cette adresse reste invariable durant toute la durée de l'exécution de l'application.

Les données **locales** sont des données volatiles accessibles en lecture et écriture que par la fonction dans laquelle elles sont déclarées. Ce sont des variables locales à la fonction ou des arguments passés à la fonction lors de son appel. Afin d'autoriser la récursivité et la réentrance, ces données ne sont pas placées à un endroit fixe dans la mémoire, mais au contraire sur une pile d'exécution (*call stack*), abrégée *la pile*. La location de ces données varie en fonction des appels.

Les données **dynamiques** sont des données créées à la volée par l'application lors de son exécution. L'emplacement de ces données s'obtient avec l'aide de fonctions mises à disposition par les langages de programmation. Ces fonctions gèrent une zone de la mémoire centraile, appelée le tas (*heap*). Cette zone se situe entre les données globales et la pile (voir l'architecture générale). En C, la bibliothèque standard du langage offre deux fonctions de base, `malloc` et `free`, pour la réservation et libération de l'espace mémoire nécessaire à stocker la donnée. En C++, les opérateurs **`new`** et `delete` offrent cette fonctionnalité.

3.2.4 Pile d'exécution

La pile d'exécution, communément appelée *la pile*, est une structure de données organisée sous la forme de pile. La pile (*stack*) fonctionne sur le principe "dernier arrivé, premier sorti" (LIFO - *Last In, First Out*), c'est-à-dire que la dernière donnée placée sur la pile sera la première donnée retirée. L'exemple ci-dessous (figure 3.12) montre le principe de fonctionne de la pile où l'on pousse successivement deux données avant de les retirer.

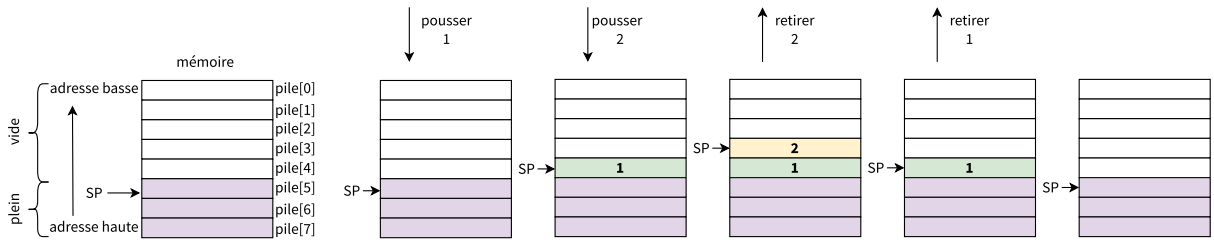


FIGURE 3.12 – Pile : principe

Le **pointeur de pile** (SP - *Stack Pointer*) contient l'adresse mémoire de la dernière donnée placée sur la pile. Sur les processeurs ARM, la pile est une pile pleine descendante, c'est-à-dire que si une donnée doit être ajoutée sur la pile, le processeur décrémente le SP avant de stocker la donnée. Si la donnée doit être retirée, le processeur lit d'abord la donnée pointée par le SP avant de l'incrémenter.

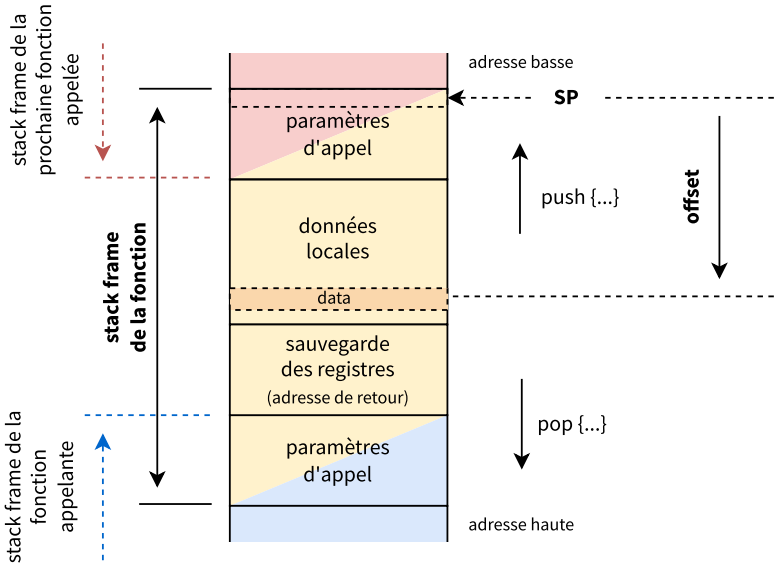


FIGURE 3.13 – Pile d'exécution

Les compilateurs utilisent la pile pour stocker le contexte (*stack frame*) des fonctions en cours d'exécution (figure 3.13). Le contexte d'une fonction consiste à :

- Paramètres d'appel
- Sauvegarde des registres du processeur et en particulier l'adresse de retour
- Données locales
- Paramètres passés aux fonctions appelées.

Le jeu d'instructions des processeurs ARM implémente deux instructions pour la sauvegarde et la restauration des registres, l'instruction `push` et respectivement l'instruction `pop`. L'accès aux données fonctionne de façon similaire à celui des données globales, à la différence prêt que l'adresse pointant sur la donnée est le résultat de l'addition de la valeur du SP et d'une offset ou décalage correspondant à l'emplacement de la donnée dans le contexte de la fonction (*stack frame*). Cette offset est fixe, car elle ne dépend que du contexte. En cas d'appels multiples, ce n'est que la valeur du SP qui varie.

Les debuggers utilisent également la pile d'exécution pour obtenir une trace des appels (*stack trace* ou *backtrace*). Cette représentation de la pile facile grandement le déboggage des applications. Elle donne une vue d'ensemble sur toutes les fonctions appelées, leurs paramètres et données locales.

3.2.5 Traitement des données

La connaissance de trois attributs est nécessaire pour un traitement approprié de ces données par le processeur, soit la taille, la location en mémoire (l'adresse) et finalement le type. Ces trois attributs sont indispensables lors du transfert de la donnée entre la mémoire et les registres du processeur. Ils permettent de choisir l'instruction transfert appropriée. Quant à la connaissance du type, il est primordial afin d'appliquer la bonne opération de traitement sur cette donnée.

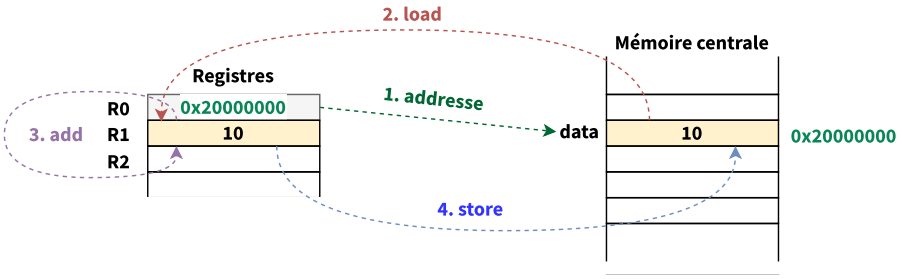


FIGURE 3.14 – "Load and Store" : principe

Avec leur architecture `Load and Store`, les processeurs ARM doivent impérativement charger les données à traiter de la mémoire centrale ou de registres des contrôleurs de périphériques d'entrées/-sorties avant de pouvoir la traiter et finalement la stocker à nouveau en mémoire (figure 3.14).

L'exemple ci-dessus montre l'incrémentation d'une donnée globale `data`. Cette simple opération exige l'exécution de quatre instructions (figure 3.15).

— Chargement de l'adresse de la donnée dans un premier registre, ligne 1
— Chargement du contenu de la donnée de la mémoire dans un deuxième registre, ligne 2
— Incrémentation de la donnée, ligne 3
— Stockage de la donnée incrémentée dans la mémoire, ligne 4

Code C

```
void incr(void) {
    data++;
}
```

Code assembleur Thumb-2

```
incr:
1:    ldr  r0, =data
2:    ldr  r1, [r0]
3:    adds r1, #1
4:    str  r1, [r0]
      bx   lr
```

Code Machine Cortex-M

```
0x08000206: 02 48
0x08000208: 01 68
0x0800020a: 01 31
0x0800020c: 01 60
0x0800020e: 70 47
```

FIGURE 3.15 – "Load and Store" : exemple

Cette architecture divise le jeu d'instructions entre des opérations d'accès avec la mémoire (lire et écrire de données entre la mémoire et les registres du µP) et des opérations avec l'unité arithmétique et logique (ALU), laquelle ne travaille qu'avec des données stockées dans ses registres.

3.3 Composants internes des µP ARMv7

Le CPU des processeurs ARM implémente une architecture légèrement différente de l'architecture générale composée d'une unité de contrôle (CU), d'une unité arithmétique et logique (ALU) et d'une banque de registres. Son architecture correspond plutôt aux trois phases du traitement de l'information, soit une unité de chargement (*Prefetch Unit*), une unité de décodage (*Decode Unit*), une unité d'exécution (*Execution Unit*). Selon les modèles, le CPU intègre également une unité de calcul à virgule flottante (FPU - *Floating Point Unit*). Cette conception facilite largement l'implémentation de l'architecture Harvard offrant de meilleures performances que l'architecture von Neumann.

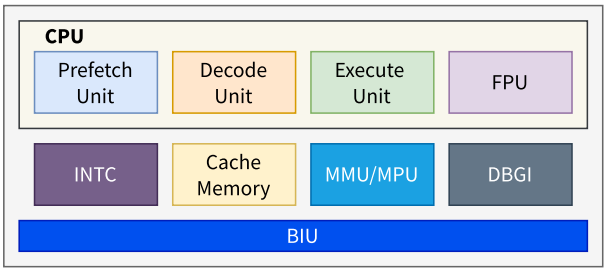


FIGURE 3.16 – Composants internes des coeurs ARMv7

Sur les processeurs ARM, le CPU s'entoure généralement de composants supplémentaires (figure 3.16) :

- Contrôleur d'interruptions (INTC / Interrupt Controller)
- Mémoire cache de 1er niveau (L1) pour les instructions et les données et éventuellement de 2^{e} niveau (L2)
- Unité de gestion de la mémoire (MMU / Memory Management Unit) ou de protection de la mémoire (MPU)
- Interface de debug et de traçage (DBGI / Debug Interface)
- Unité d'interface du bus µP (BIU / Bus Interface Unit)

3.3.1 Unité de chargement

L'unité de chargement (*Prefetch Unit*) lit les nouvelles instructions à exécuter. Elle dispose de son propre bus pour le transfert des instructions. Pour les profils A et R, l'unité est connectée à la mémoire centrale du µP par l'intermédiaire d'une mémoire cache d'instructions de premier niveau (L1 I-Cache). Par contre, les µC du profil M ne disposent pas forcément de cette mémoire cache. Dans ce cas, le bus est connecté directement avec la mémoire centrale.

3.3.2 Unité de décodage

L'unité de décodage (*Decode Unit*) analyse les instructions pour déterminer les opérandes et le type d'instruction avant de les transférer à l'unité d'exécution.

3.3.3 Unité d'exécution

L'unité d'exécution (*Execute Unit*) effectue les opérations correspondant aux instructions préalablement décodées. Cette unité intègre généralement la banque de registres, l'ALU équipé avec son barrel shifter ainsi que d'une logique pour le transfert des données entre la mémoire centrale et les registres.

Banque de registres

La banque de registres (*Register Bank*) compte 13 registres à usage général (R0 à R12), 3 registres à usage dédié (R13 à R15) et 1 registre d'état du programme (APSR) (figure 3.17). Le jeu d'instructions Thumb (codé sur 16 bits) ne permet l'accès qu'aux huit premiers registres (*Low General Purpose Registers*). L'accès aux treize premiers registres n'est possible qu'avec les jeux d'instructions Thumb-2 et ARM.

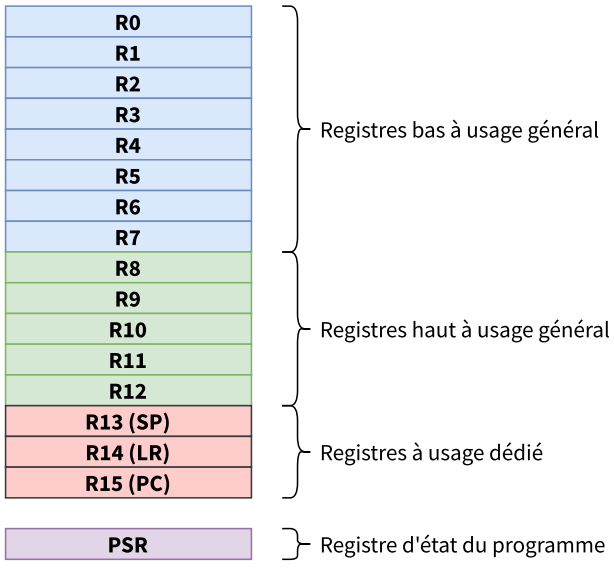


FIGURE 3.17 – Banque de registres

Le registre R13 (SP - *Stack Pointer*) pointe sur le sommet de la pile utilisée pour l'appel de fonction. Le registre R14 (LR - *Link Register*) contient l'adresse de retour lors d'appel d'une fonction. Le registre R15

(PC - *Program Counter*) pointe sur l'instruction à exécuter. Selon le mode d'opération du processeur (application, exception, etc.), le registre PSR (*Program Status Register*) contient des champs de bits différents. En mode application, mode dans lequel les applications utilisateur s'exécutent, le registre se nomme APSR (*Application Program Status Register*). Il contient principalement quatre fanions de conditions (*flags*) résultant des opérations arithmétiques et logiques (figure 3.18).

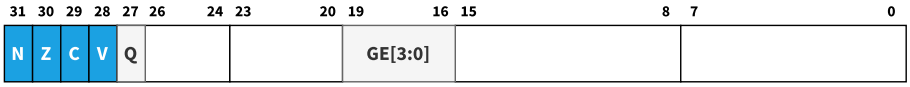


FIGURE 3.18 – Registre d'état du programme

Le fanion Z (*Zero*) indique que l'opération a donné un résultat nul. Le fanion C (*Carry*) indique un dépassement de capacité (report) résultant d'une opération mathématique sur des nombres non signés, tandis que le fanion V (*oVerflow*) l'indique sur des nombres signés. Le fanion N (*Negative*) indique que le nombre contenu dans le registre est négatif. Lors de branchements conditionnels, le µP évalue ces fanions afin de décider du branchement à effectuer.

A ces quatre fanions principaux s'ajoutent encore le fanion Q (*Sticky Overflow*) indiquant un dépassement de capacité lors de calculs arithmétiques saturés et les fanions GE (*Greater than Equal*). Ces derniers flags ne sont disponibles que si le processeur dispose d'une unité de calcul à virgule flottante.

ALU

L'ALU (*Arithmetic and Logical Unit*) (figure 3.19) exécute les opérations arithmétiques et logiques sur deux opérandes. Il place ensuite le résultat de l'opération dans un troisième registre (Rd) et met à jour les fanions dans le registre APSR.

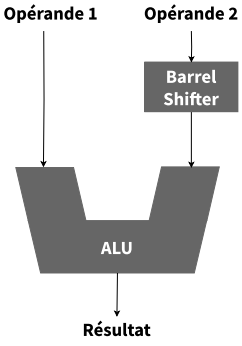


FIGURE 3.19 – Unité arithmétique et logique (ALU)

Le premier opérande, contenue dans un registre (Rn) est envoyé directement à l'ALU. Le deuxième opé-

rande, nommée opérande à décalage (Sop - *Shifter Operand*) passe d'abord à travers un barrel shifter. Elle peut venir soit du contenu d'un registre (Rm) ou d'une valeur immédiate (une constante).

Barrel shifter

Le barrel shifter est un circuit à logique combinatoire. Ce circuit est capable d'effectuer une rotation vers la gauche des bits contenus dans un registre sans logique séquentielle. La figure ci-dessous (figure 3.20) donne un exemple de barrel shifter pour des mots de 4 bits.

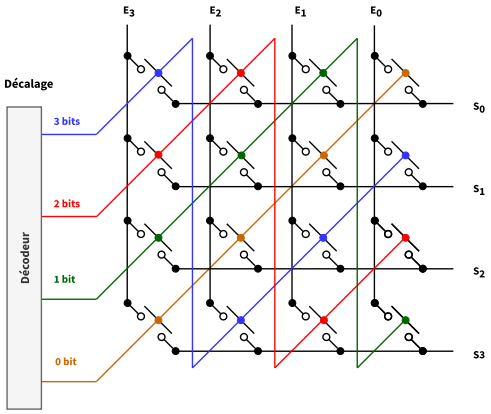


FIGURE 3.20 – Barrel Shifter

Le décodeur détermine, en fonction du décalage souhaité (le nombre de bits), la ligne à activer. Les portes placées sur la ligne activée se ferment et placent les bits d'entrée (Ex) sur les bits de sortie (Sx) correspondants. Le décalage s'effectue en 1 cycle d'horloge.

3.3.4 Unité de calcul à virgule flottante

La FPU (*Floating Point Unit*) est une unité optionnelle de calcul à virgule flottante. Elle existe en différentes variantes offrant plus ou moins de fonctionnalité, calcul sur les nombres réels à virgule flottante selon le standard IEEE754, calcul vectoriel avec des opérations avancées SIMD, traitement de signal avec des opérations DSP (*Digital Signal Processing*).

3.3.5 Contrôleur d'interruptions

Le INTC (*Interrupt Controller*) est un contrôleur pour le traitement des interruptions. Le CPU ne disposant que de très peu de lignes pour recevoir les requêtes d'interruptions levées par les contrôleurs de périphériques d'entrées/sorties. Alors, l'INTC priorise les requêtes, indique la source d'interruption, c'est-à-dire l'identité du contrôleur ayant levé l'interruption, et signale la requête au CPU.

3.3.6 Mémoire cache

La mémoire cache ou antémémoire est une mémoire très rapide placée entre la mémoire centrale et le CPU. Elle stocke temporairement les données afin d'accélérer leurs accès et d'augmenter ainsi les performances du processeur. Selon les processeurs et leur profil, la mémoire cache s'implémente sur 1, 2 voire 3 niveaux.

Le niveau 1 (L1 - *Level 1*) est le plus rapide, mais de taille relativement faible, de quelques KiB à quelques dizaines de KiB. Les µP l'utilisent en deux exemplaires. Le premier stocke les instructions (*i-cache*), tandis que le deuxième stocke les données (*d-cache*). Deux bus indépendants connectent ces mémoires caches au CPU, le premier avec l'unité de chargement et le deuxième avec l'unité d'exécution.

Si une mémoire cache de niveau 2 (L2 - *Level 2*) équipe le processeur, elle sert d'interface entre les mémoires caches de 1^{er} niveau et la mémoire centrale. Sa taille varie entre quelques centaines de KiB à quelques MiB. Elle peut également stocker temporairement les données de la FPU si le CPU en dispose d'une. Les processeurs multi-coeurs l'utilisent pour l'échange de données entre les coeurs.

Le niveau 3 (L3 - *Level 3*) est plus rare et se situe généralement à l'extérieur de la puce du processeur.

3.3.7 Unité de gestion/protection de la mémoire

La MMU (*Memory Management Unit*) a pour fonction la gestion de la mémoire. Elle équipe tous les processeurs du profil A. Elle offre aux OS une virtualisation de l'espace mémoire, c'est-à-dire elle convertit les adresses virtuelles en adresses physiques. Elle contrôle les accès à la mémoire centrale et aux registres des contrôleurs de périphériques d'entrées/sorties. Elle interagit également avec la mémoire cache afin de l'informer si les données peuvent être placées dans son antémémoire.

La MPU (*Memory Protection Unit*) a pour fonction la protection des accès à la mémoire. Elle équipe les processeurs des profils R et M.

3.3.8 Interface de debug

Le développement d'applications logicielles exige la mise en oeuvre d'outils pour déboguer, tester et valider le bon fonctionnement du programme. Les processeurs ARM offrent deux interfaces à cet effet (figure 3.21), le *debug interface* et le *trace interface*.

L'interface de debug (SWJ-DP - *Serial Wire JTAG Debug Port*) propose deux interfaces à choix, le JTAG (*Joint Test Action Group*) et le SWD (*Serial Wire Debug*). Via les unités `Run Control`, `Breakpoint` et `Memory Acces`, ces deux interfaces offrent la fonctionnalité suivante :

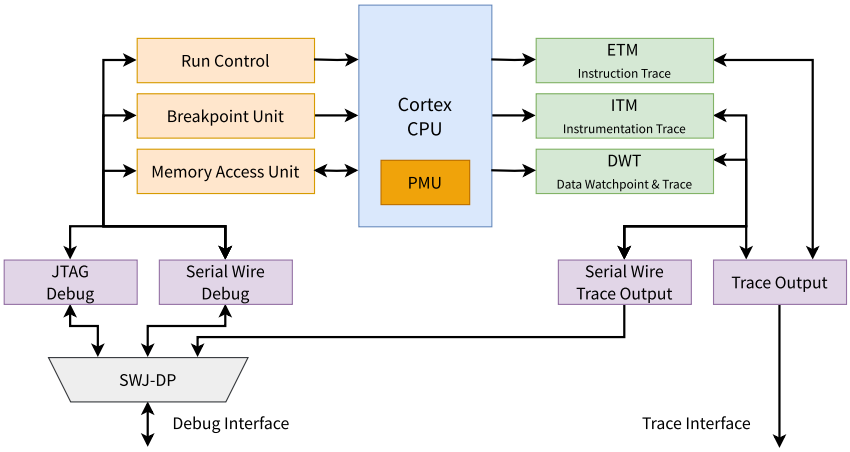


FIGURE 3.21 – Interfaces de debug et de traçage

— Démarrer et arrêter le programme
— Déboguer pas à pas (*Single Step*)
— Placer des points d'arrêt (*Breakpoint*) à la volée
— Lire/écrire le contenu de mémoire et des registres des contrôleurs de périphériques à la volée
— Programmer la mémoire Flash (interne et externe)

L'interface de traçage (ETM - *Embedded Trace Macrocell*) fournit une interface à haut débit pour le traçage des instructions. Cette interface offre la fonctionnalité suivante :

— Déboguer des historiques de séquences d'événements
— Profiler le logiciel et optimiser les algorithmes (*Profiling*)
— Analyser la couverture du code (*Code Coverage*)

Les nouvelles générations de processeurs du profil A implémentent également une unité pour monitorer les performances des applications (PMU - *Performance Monitoring Unit*). Cette unité dispose d'une série de compteurs capables de collecter différents événements du processeur, tels que le nombre de cycles d'horloge, d'instructions exécutées, d'accès à la mémoire cache, etc. Grâce à ces compteurs, il devient possible d'analyser le comportement d'applications complexes afin de trouver par exemple les goulots d'étranglement (*bottleneck*).

3.3.9 Unité d'interface du bus µP

L'unité d'interface du bus µP (BIU) connecte le CPU et ses composants avec la mémoire centrale et les contrôleurs de périphériques d'entrées/sorties. L'implémentation de cette unité dépend du processeur et de son profil.

3.3.10 Profil A

Le processeur ARM Cortex-A8 (figure 3.22), basé sur le profil A, est un µP superscalaire (*Superscalar*), c'est-à-dire en l'absence de dépendances entre deux instructions consécutives, ce µP est capable de les traiter simultanément.

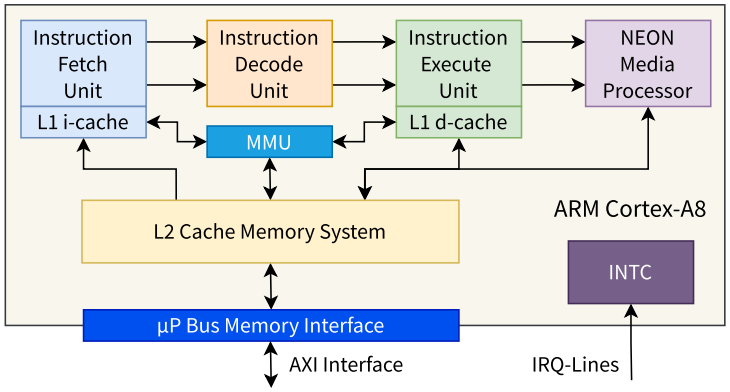


FIGURE 3.22 – ARM Cortex-A8 :

L'**unité de chargement** (*Instruction Fetch Unit*) (figure 3.23) lit les instructions. Elle comprend également une mémoire cache pour les instructions (L1 i-cache avec sa RAM, sa TLB (*Translation Lookaside Buffer*) et son interface), ainsi qu'un prédicateur de branchements (*Branch Predictor*).

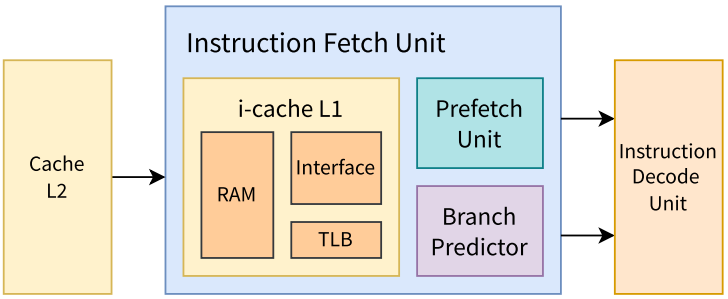


FIGURE 3.23 – Cortex-A8 : Instruction Fetch Unit

Si les instructions ne sont pas disponibles dans sa cache L1, l'unité les charge alors de la mémoire cache L2 ou finalement de la mémoire centrale. Une fois disponible les instructions sont mise à disposition de l'unité de décodage. L'unité dispose de deux interfaces avec l'unité de décodage. Le prédicateur de branchements permet de prévoir le résultat d'un branchement et ainsi d'optimiser l'utilisation du pipeline.

L'**unité de décodage** (*Instruction Decode Unit*) (figure 3.24) décode les instructions et séquence les événements du CPU (exceptions, interruptions, etc.). Elle vérifie également la dépendance entre les instructions.

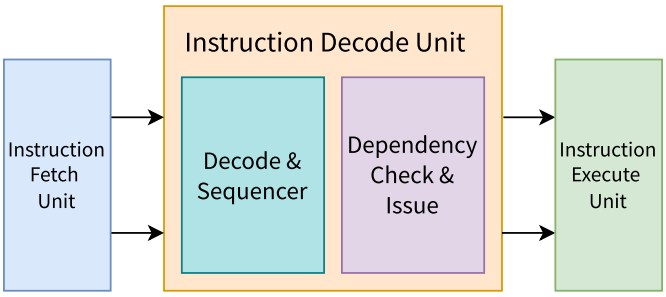


FIGURE 3.24 – Cortex-A8 : Instruction Decoce Unit

L'**unité d'exécution** (*Instruction Execute Unit*) (figure 3.25) opère sur les données stockées dans la banque de registres selon les instructions reçues de l'unité de décodage. Pour exécuter ces opérations, elle dispose de deux unités arithmétique et logique (ALU), d'une unité de multiplication (MAC) et d'une unité le transfert des données (LS). L'unité implémente également une mémoire cache de niveau 1 pour les données (L1 d-cache avec sa RAM, sa TLB (*Translation Lookaside Buffer*) et son interface).

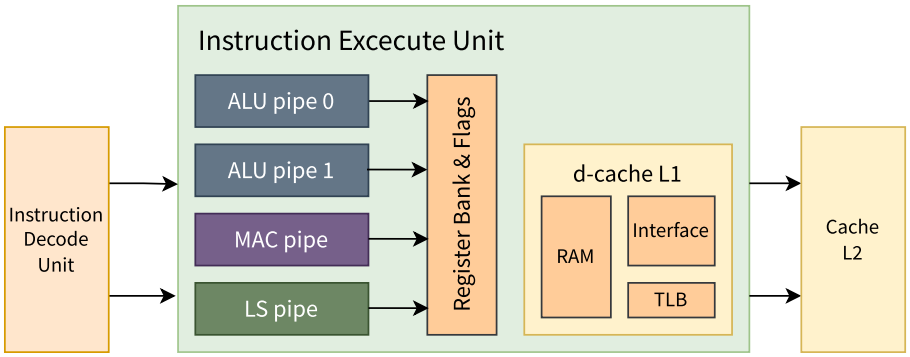


FIGURE 3.25 – Cortex-A8 : Instruction Execute Unit

L'**INTC** dispose de 128 lignes de requêtes d'interruptions afin de connecter les contrôleurs de périphériques d'entrées/sorties internes au CPU.

La **BIU** sert d'interface entre le CPU et la mémoire centrale et les contrôleurs de périphériques d'entrées/sorties. Il implémente l'interface AXI (*Advanced eXtensible Interface*), interface standard des processeurs ARM.

3.3.11 Profil M

Le processeur ARM Cortex-M4F (figure 3.26), basé sur le profil M, est un µC. Equipé d'une mémoire SRAM et d'une mémoire NOR-Flash, ce µC ne nécessite aucune mémoire supplémentaire pour stocker et exécuter les applications. Afin de réduire la taille du code, ce µC ne propose que les jeux d'instruc-

tions Thumb et Thumb-2.

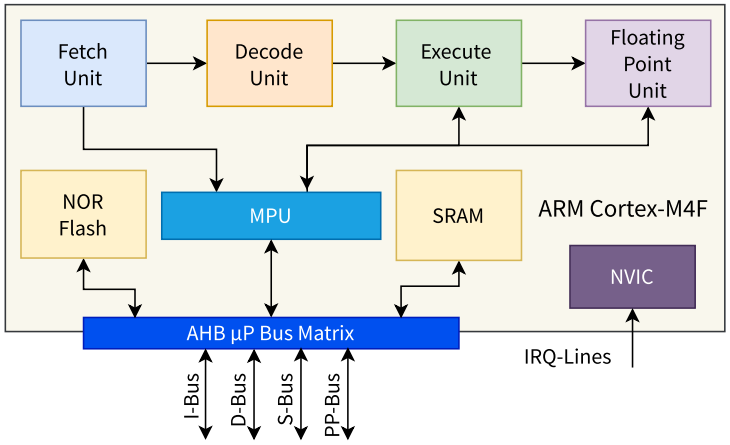


FIGURE 3.26 – ARM Cortex-M4F

Le **NVIC** (*Nested Vectored Interrupt Controller*), conçu pour un traitement des interruptions directement en C, peut connecter jusqu'à 240 sources d'interruptions différentes. Ce contrôleur peut également réveiller le µC, si celui-ci se trouve en mode veille à très faible consommation.

La **BIU** servant d'interface entre le CPU et la mémoire centrale et les contrôleurs de périphériques d'entrées/sorties implémente trois interfaces AHB (*Advanced High-performance Bus*), soit I-Bus, D-Bus et S-Bus, ainsi qu'une interface APB (*Advanced Peripheral Bus*), soit PP-Bus. Le CPU charge le code via le I-Bus (*Instruction Bus*), le transfert des données passe par le D-Bus (*Data Bus*), tandis que les données contenues dans les registres des contrôleurs de périphériques d'entrées/sorties transitent par le S-Bus (*System Bus*). Le PP-Bus (*Private Peripheral Bus*) sert quant à lui à accéder les registres de ses contrôleurs internes (internes au CPU).

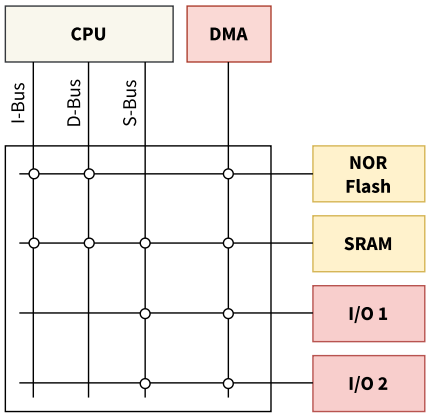


FIGURE 3.27 – AHB µP Bus Matrix

Une matrice (figure 3.27) interconnecte l'unité centrale de traitement (CPU) et les DMA (*Direct Memory Access*) avec les périphériques (Flash, RAM, I/O). Cette matrice offre un transfert non bloquant, c'est-à-dire si le CPU charge une instruction de la Flash et un écrit une donnée dans la RAM et qu'un DMA transfère des données avec des périphériques, toutes ces opérations peuvent s'effectuer simultanément.

3.4 Modes des CPU ARMv7

Avec l'évolution des systèmes d'exploitation (*OS*), la nécessité de pouvoir disposer de différents niveaux de privilèges a émergé. En effet, afin d'augmenter la robustesse et la fiabilité des logiciels, il est devenu indispensable de pouvoir contrôler l'accès aux ressources du processeur ainsi que d'isoler le code du noyau de l'OS des applications utilisateur.

3.4.1 Concept

Les processeurs connaissent deux espaces de fonctionnement principaux (figure 3.28), l'espace système ou noyau (*kernel space*), et l'espace application ou utilisateur (*user space*). Ces deux espaces correspondent à deux niveaux de privilèges, le niveau privilégié et respectivement le niveau non privilégié. L'espace *système* fonctionne au niveau privilégié, tandis que l'espace *application* travaille au niveau non privilégié.

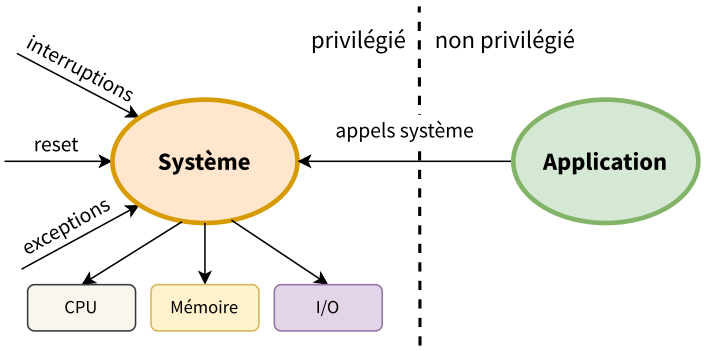


FIGURE 3.28 – Modes principaux des CPU

Idéalement et avec le support de la MMU/MPU, seul le code du noyau travaille au niveau privilégié. Cela lui accorde un accès illimité à l'ensemble des composants matériels du processeur (CPU, mémoires et contrôleurs de périphériques d'entrées/sorties). Lors du démarrage du système ou après un *reset*, le processeur entre toujours dans l'espace système afin d'obtenir tous les privilèges et de pouvoir ainsi procéder à l'ensemble des initialisations nécessaires au fonctionnement du système. Si des interruptions ou des exceptions surviennent, le processeur bascule automatiquement dans l'espace système

afin d'acquérir le niveau privilégié lui permettant d'accéder aux ressources matérielles et effectuer les traitements adéquats.

Les applications utilisateur, généralement considérées comme des programmes moins fiables que le code du noyau, s'exécutent dans un espace restreint au niveau non privilégié. Si une application a besoin d'accéder à des informations ou des ressources matérielles, elle devra impérativement passer par le noyau avec des appels système (une interruption logicielle). Les appels système ont un certain coût en temps d'exécution, lequel peut dégrader les performances du système et nuire au bon fonctionnement de certains logiciels. Dans de tels cas, il est usuel de déplacer le code du logiciel d'application dans le code du noyau. Il en va généralement ainsi pour les pilotes de périphériques, ceci d'autant plus qu'ils requièrent l'accès aux registres de contrôleurs de périphériques d'entrées/sorties.

Sur les CPU ARMv7, ces deux espaces se décomposent en différents modes de fonctionnement. Leur déclinaison dépend du profil du processeur (A, R ou M), ainsi que des fonctionnalités choisies par le fabricant du µP ou du µC.

3.4.2 Profil A

Les processeurs du profil A proposent encore deux extensions en option. La première est une extension de sécurité et la deuxième une extension de virtualisation.

Etats de sécurité

L'extension de sécurité a deux états de sécurité, l'état sécurisé et l'état non sécurisé (figure 3.29). Cette extension est une option des processeurs du profil A que les fabricants ont à choix.

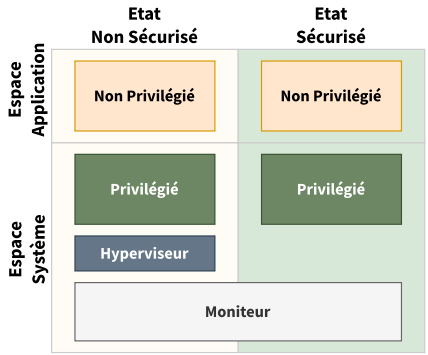


FIGURE 3.29 – Profil A : Etats de sécurité

Dans l'état sécurisé (figure 3.30), le processeur se trouve dans un espace de confiance (*Trust Zone*) afin de fournir aux applications un environnement d'exécution de confiance (TEE - *Trust Execution Environment*). Il garantit la confidentialité et l'intégrité du code et des données placés dans un environnement

d'exécution isolé. Seuls des OS de confiance sont destinés à fonctionner dans cet environnement. OP-TEE (https://www.op-tee.org/) propose un tel environnement.

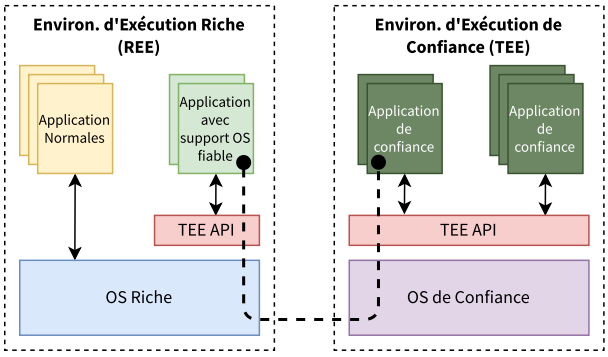


FIGURE 3.30 – Profil A : Applications sécurisées

Dans l'état non sécurisé, le processeur se trouve dans un espace dédié aux applications "normales" (REE - *Rich Execution Environment*) n'exigeant pas de sécurité spéciale hormis celle proposée par les OS riches, tels GNU/Linux ou SELinux. Une interface spécifique (*TEE API*) fournit un canal de communication entre des applications dans les deux espaces.

Virtualisation

La conception de certains systèmes exige la réalisation de plusieurs applications logicielles distinctes nécessitant chacune la mise en oeuvre de leur propre système d'exploitation. Le concept de machines virtuelles (VM - *Virtual Machine*) permet de telles réalisations (figure 3.31). La gestion et le contrôle des VM sont confiés à un logiciel spécifique, l'hyperviseur.

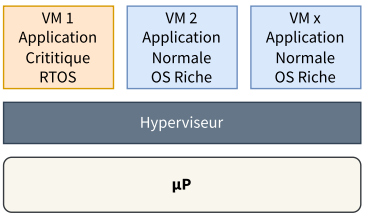


FIGURE 3.31 – Profil A : Hyperviseur

Afin de permettre une implémentation performante et efficace de l'hyperviseur, l'extension de virtualisation ajoute aux deux niveaux de privilèges standards, privilégié et non privilégié, un niveau supplémentaire, le niveau *hyperviseur*. Avec ce nouveau niveau, option des processeurs avec le profil A, l'hyperviseur s'exécute dans un espace sécurisé et isolé des machines virtuelles. Il lui permet d'intercepter les accès aux matériels du processeur effectués par les différentes VMs et ainsi de les contrôler et de les gérer efficacement. Ce niveau n'est disponible que dans l'état non sécurisé.

Modes du processeur

La gestion des espaces de fonctionnement du processeur et des niveaux de privilèges s'effectue grâce aux modes du processeur (figure 3.32).

- Le mode **User** (usr) offre un accès limité, non privilégié, aux ressources du µP. Les systèmes d'exploitation utilisent ce mode pour exécuter les applications utilisateur. Tous les µP du profil A implémentent ce mode. Si le µP dispose de l'option de sécurité, ce mode est disponible dans les deux états de sécurité.
- Le mode **FIQ** (fiq) offre un accès illimité, privilégié, aux ressources du µP. Le µP bascule automatiquement dans ce mode pour traiter une interruption de type `FIQ`. Tous les µP du profil A implémentent ce mode. Si le µP dispose de l'option de sécurité, ce mode est disponible dans les deux états de sécurité.

Mode du processeur	Abrv.	Code	Niveau de privilèges	Implémentation	Etat de sécurité
User	usr	0b1'0000	Non privilégié	Toujours	Les deux
FIQ	fiq	0b1'0001	Privilégié	Toujours	Les deux
IRQ	irq	0b1'0010	Privilégié	Toujours	Les deux
Supervisor	**svc**	**0b1'0011**	**Privilégié**	**Toujours**	**Les deux**
Monitor	mon	0b1'0110	Privilégié	Avec l'extension Sécurité	Sécurisé
Abort	abt	0b1'0111	Privilégié	Tourjours	Les deux
Hypervisor	hyp	0b1'1010	Hyperviseur	Avec l'extension Virtualisation	Non-sécuritsé
Undefined	und	0b1'1011	Privilégié	Toujours	Les deux
System	sys	0b1'1111	Privilégié	Toujours	Les deux

FIGURE 3.32 – Profil A : Modes

- Le mode **IRQ** (irq) offre un accès illimité, privilégié, aux ressources du µP. Le µP bascule automatiquement dans ce mode pour traiter une interruption de type `IRQ`. Tous les µP du profil A implémentent ce mode. Si le µP dispose de l'option de sécurité, ce mode est disponible dans les deux états de sécurité.
- Le mode **Supervisor** (svc) offre un accès illimité, privilégié, aux ressources du µP. Le µP bascule automatiquement dans ce mode pour traiter les interruptions logicielles levées avec l'instruction `SVC` (*supervision call*). `Supervisor` est le mode par défaut après un *reset* du processeur. Les systèmes d'exploitation utilisent ce mode pour exécuter le code noyau et accéder aux ressources du µP. Tous les µP du profil A implémentent ce mode. Si le µP dispose de l'option de sécurité, ce mode est disponible dans les deux états de sécurité.
- Le mode **Monitor** (mon) offre un accès illimité, privilégié, aux ressources du µP. Le µP bascule automatiquement dans ce mode pour traiter les interruptions logicielles levées avec l'instruc-

tion SMC (*Secure Monitor Call*). Indépendamment de l'état du processeur avant l'appel de l'instruction SMC, le processeur entre dans l'état de sécurité sécurisé. Le code exécuté dans ce mode a accès aussi bien aux ressources disponibles dans l'état non sécurisé que celles disponibles dans l'état sécurisé. Ce mode de l'extension de sécurité est une option du profil A au choix des fabricants. Si elle n'est pas disponible, seul l'état de sécurité non sécurisé est disponible.

— Le mode **Abort** (abt) offre un accès illimité, privilégié, aux ressources du µP. Le µP bascule automatiquement dans ce mode pour traiter une exception de type `data` ou `prefetch abort`, levée lors d'accès non autorisés sur des données ou du code. Tous les µP du profil A implémentent ce mode. Si le µP dispose de l'option de sécurité, ce mode est disponible dans les deux états de sécurité.

— Le mode **Hypervisor** (hyp) offre un accès illimité, privilégié, aux ressources du µP. Le µP bascule automatiquement dans ce mode pour traiter les interruptions logicielles levées avec l'instruction HVC (*Hypervisor Call*) ou pour traiter des exceptions levées par l'accès à des ressources non autorisées. Ce mode de l'extension de virtualisation est une option du profil A au choix des fabricants. Ce mode n'est disponible que dans l'état de sécurité non sécurisé.

— Le mode **Undefined** (und) offre un accès illimité, privilégié, aux ressources du µP. Le µP bascule automatiquement dans ce mode pour traiter une exception de type `undefined`, levée lorsque le processeur essaye d'exécuter une instruction non disponible dans son jeu d'instructions (*Undefined Instruction*). Tous les µP du profil A implémentent ce mode. Si le µP dispose de l'option de sécurité, ce mode est disponible dans les deux états de sécurité.

— Le mode **System** (sys) offre un accès illimité, privilégié, aux ressources du µP. Ce mode dispose exactement des mêmes registres que ceux du mode `User`. Aucune exception ne permet de basculer dans ce mode. Seul du code déjà exécuté dans un des niveaux de privilèges privilégié peut entrer dans ce mode. Tous les µP du profil A implémentent ce mode. Si le µP dispose de l'option de sécurité, ce mode est disponible dans les deux états de sécurité.

Registre d'état du programme

Le registre d'état du programme dans l'espace application APSR est juste un alias du registre d'état courant du programme CPSR (*Current Program Status Registre*). Il contient les fanions de conditions. En espace système, le contenu du CPSR fournit des champs de bits supplémentaires avec des informations système (figure 3.33).

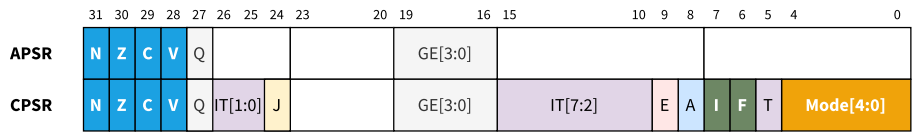


FIGURE 3.33 – Profil A : Program Status Registers

- Le champ **Mode** (bits [4 :0]) indique dans quel mode du processeur le programme s'exécute. Pour changer de mode par logiciel, le code doit impérativement s'exécuter au niveau privilégié ou `hyperviseur`.
- Les bits **T** (bit [5]) et **J** (bit [24]) déterminent le jeu d'instructions exécuté par le processeur, ARM (J=0, T=0), Thumb (J=0, T=1), Jazelle (J=1, T=0) ou ThumbEE (J=1, T=1).
- Le bit **F** (bit [6]) sert de masque au traitement des interruptions `FIQ`. Si le bit est à 1, les interruptions `FIQ` sont bloquées. S'il est à 0, les interruptions `FIQ` sont autorisées.
- Le bit **I** (bit [7]) sert de masque au traitement des interruptions `IRQ`. Si le bit est à 1, les interruptions `IRQ` sont bloquées. S'il est à 0, les interruptions `IRQ` peuvent être levées.
- Le bit **A** (bit [8]) sert de masque au traitement d'exceptions `abort` asynchrones.
- Le bit **E** (bit [9]) détermine le boutisme (*endianness*) du processeur. Si le bit est à 1, le processeur effectuera le transfert de données avec la mémoire en grand boutisme (*big-endian*) , sinon il travaillera en petit boutisme (*little-endian*).
- Les bits **IT** (bits [26 :25,15 :10]) décrivent le comportement de l'exécution de l'instruction Thumb `IT` (If-Then).

Les instructions `mrs` et `msr` donnent un accès en lecture et respectivement en écriture à ces registres.

Banques de registres

Vu des applications logicielles, le µP implémentent 17 registres, les registres à usage général R0 à R12, le pointeur de pile (`SP`), le registre de lien (`LR`) contenant l'adresse de retour lors d'appel de fonction, le compteur ordinal (`PC`) ainsi que le registre d'état du programme (`APSR`). Ces registres suffisent pour les applications exécutées en mode `User`, mais il est nécessaire de disposer de registres supplémentaires pour exécuter le code dans les autres modes (figure 3.34).

Hormis pour le mode `System`, le changement de mode s'effectue généralement sur événement (interruption ou exception). Lors de la levée de l'événement, le processeur bascule automatiquement dans le mode correspondant à l'événement. Il appelle ensuite une routine pour son traitement. Afin de pouvoir retourner au programme interrompu par le traitement de l'événement, il est impératif de sauver l'adresse de retour dans un registre `LR` spécifique au mode dans lequel le processeur a basculé. Sans cette sauvegarde, l'adresse de retour du programme interrompu serait perdue. Il en va de même pour le contenu du registre `CPSR`. En effet, le processeur change le contenu du registre `CPSR` afin de charger le nouveau mode du processeur (bits [4 :0]). Avant de modifier son contenu, le µP le sauve dans le registre `SPSR` (*Save Program Status Register*) du mode correspondant.

Le mode `System` partage les mêmes registres que ceux du mode `User`, mais au niveau privilégié. Seul le registre d'état du programme devient le registre `CPSR` avec un accès à toutes les informations.

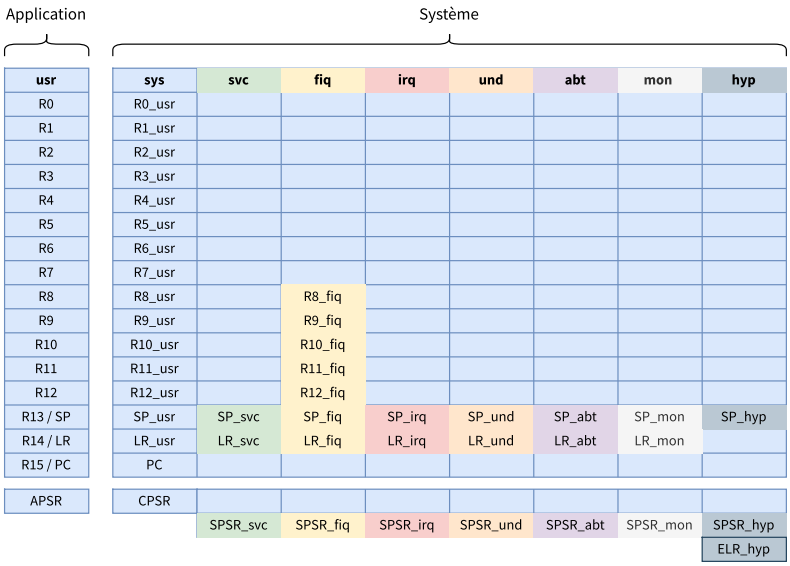


FIGURE 3.34 – Profil A : Registres

Le mode `Hypervisor` remplace le registre `SP_usr` par son propre registre `SP_hyp`. Il dispose également d'un registre `SPSR_hyp` pour la sauvegarde du CPSR ainsi que d'un registre spécial `ELR_hyp` pour la sauvegarde de l'adresse de retour. Tous les autres modes dupliquent les registres `SP_usr` et `LR_usr` et disposent également d'un registre `SPSR`. Le mode `FIQ` duplique également les registres `R8_usr` à `R12_usr`.

3.4.3 Profil M

Les processeurs du profil M sont spécialement conçus pour simplifier le développement de systèmes à basse consommation et de leurs applications. Ces processeurs ne peuvent exécuter que du code Thumb ou Thumb-2.

Modes du processeur

Les processeurs du profil M supportent deux modes, le mode `Thread` et le mode `Handler` (figure 3.35). Pour exécuter le code principal de l'application ou tâches de fond (*tasks* ou *threads*), le processeur se trouve dans le mode `Thread`. Par contre, à la levée d'une exception ou d'une interruption, le processeur bascule automatiquement dans le mode `Handler` avant de retourner dans le mode `Thread` à la fin du traitement de l'événement.

Ces processeurs supportent également deux niveaux de privilèges, le niveau privilégié et le niveau non privilégié. Lorsque le processeur entre dans le mode `Handler`, il obtient automatiquement le niveau

privilégié. Dans le mode `Thread`, l'application peut choisir entre le niveau privilégié et niveau non privilégié. Avec le niveau privilégié, le processeur donne accès à toutes ses ressources, tandis qu'avec le niveau non privilégié, en conjonction avec la MPU, il interdit l'accès à certaines zones de la mémoire et certaines opérations ne sont plus disponibles.

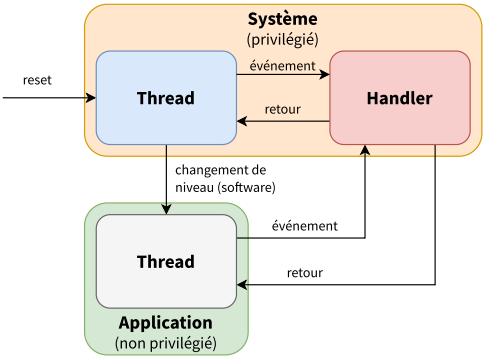


FIGURE 3.35 – Profil M : Modes

Lors du démarrage du processeur ou après un *reset*, le processeur entre toujours dans le mode `Thread` avec le niveau privilégié.

Banque de registres

Les processeurs du profil M n'ont qu'une banque de registres pour les deux modes du processeur (figure 3.36). Seul le pointeur de pile, le registre R13, existe en deux copies, le `MSP`(*Main Stack Pointer*) et le `PSP` (*Process Stack Pointer*).

Le `MSP` ou `SP_main` est le pointeur de pile par défaut après le *reset* du processeur. Le `MSP` est le SP utilisé pour le traitement des événements en mode `Handler`. Le `PSP` ou `SP_process` est un pointeur de pile à disposition des codes exécutés en mode `Thread`, ceci indépendamment du niveau de privilèges. L'usage du `PSP` devient très intéressant pour des applications multi-tâches (applications mettant en oeuvre un RTOS). Il permet en effet de limiter la taille de la pile de chacune des tâches à leurs besoins, indépendamment les unes des autres.

Les modes `Thread` et `Handler` utilisent le même registre R14 (LR). Sa duplication n'est pas nécessaire, car lors de la levée d'une interruption ou d'une exception, le processeur sauve automatiquement son contenu sur la pile. La sauvegarde effectuée, il poursuit le traitement de l'événement et l'utilisera finalement pour retourner au programme interrompu et continuer son exécution, une fois le traitement de l'événement terminé.

Les registres R0 à R12 sont à usage universel. Cependant, avec le jeu d'instructions Thumb codé sur 16-bit, beaucoup d'instructions ne peuvent utiliser que les registres R0 à R7, les registres bas. Cette limitation tombe avec le jeu d'instructions Thumb-2 codé sur 32 bits.

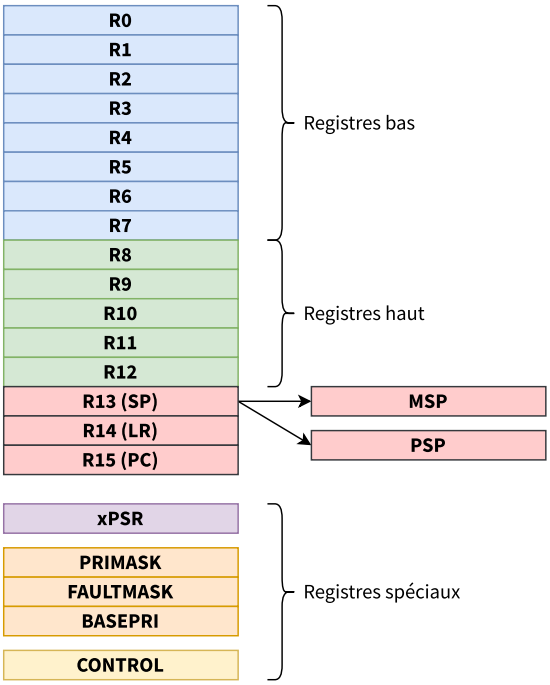


FIGURE 3.36 – Profil M : Registres

Les registres spéciaux servent à la gestion de l'état du programme (registre `xPSR`), des masques d'interruptions (registres `PRIMASK`, `FAULTMASK` et `BASEPRI`) ainsi qu'au contrôle des modes du processeur (registre `CONTROL`).

Registre d'état du programme

Le registre d'état du programme **PSR** se décline en trois sous-registres (figure 3.37).

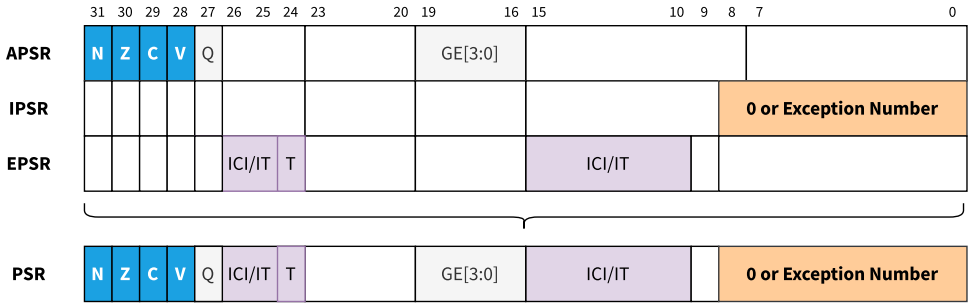


FIGURE 3.37 – Profil M : Program Status Registers

Le registre **APSR** (*Application Program Status Register*) contient les fanions de conditions résultant d'opérations effectuées par le processeur.

Le registre **IPSR** (*Interrupt Program Status Register*) fournit le numéro (bits [8 :0]) de l'exception ou de l'interruption en cours de traitement dans le mode `Handler`. Le processeur modifie ce registre à l'entrée et à la sortie de la routine de traitement. Si le processeur se trouve dans le mode `Thread`, la valeur est toujours 0.

Le registre **EPSR** (*Execution Program Status Register*) contient les fanions de conditions ainsi que le bit **T** (bit [24]) le champ superpose indiquant **ICI** / **IT** (bits[26 :24,15 :10]) l'état d'exécution du processeur.

— Le bit **T** indique au processeur le jeu d'instructions à exécuter. Il doit impérativement être à mis à 1 pour exécuter du code Thumb/Thumb-2. Lors de branchements indirects, le bit 0 de l'adresse indique au processeur le jeu d'instructions à utiliser, 0 pour ARM et 1 pour Thumb/Thumb-2. Il est dès lors impératif que ce bit soit toujours à 1.
— Le champ **ICI** (*Interrupt Continue load/store Instruction*) fournit une information sur la liste des registres en suspens pour les instructions de transfert multiple, lorsque l'exécution de l'une de ces instructions est stoppée pour traiter une exception ou une interruption. Cette fonctionnalité est une option au choix des fabricants.
— Le champ **IT** (*IT Instruction*) décrit le comportement de l'exécution de l'instruction Thumb `IT` (If-Then).

Les instructions `mrs` et `msr` donnent un accès en lecture, respectivement en écriture, à ces registres.

Registres de gestion des masques d'interruptions

Les processeurs du profil M implémentent trois registres spéciaux, le registre `PRIMASK`, le registre `FAULTMASK` et le registre `BASEPRI`, pour gérer les interruptions au niveau du CPU (figure 3.38).

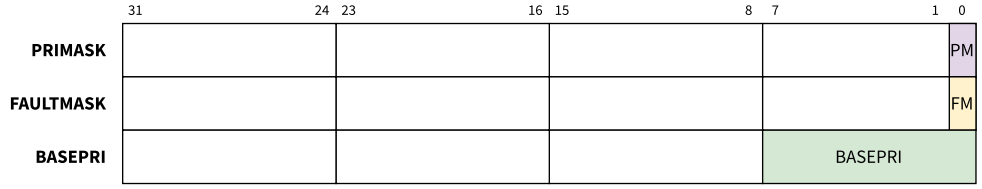


FIGURE 3.38 – Profil M : Mask Registers

Le registre **PRIMASK**, avec le `PM` mis à 1, bloque la levée de toutes les exceptions et interruptions dont le niveau de priorité peut être configuré. Ce registre correspond à un bit de désactivation globale des interruptions sur d'autres processeurs. Le bit `PM` mis à 0 autorise la levée d'interruptions. L'instruction "`cpsid i`" met le bit à 1 tandis que l'instruction "`cpsie i`" le remet à 0.

Le registre **FAULTMASK**, avec le `FM` mis à 1, bloque, en plus des exceptions et interruptions pouvant être configurées et les exceptions `HardFault`. Le bit `FM` mis à 0 autorise la levée d'interruptions. L'instruction "`cpsid f`" met le bit à 1 tandis que l'instruction "`cpsie f`" le remet à 0.

Le registre **BASEPRI**, avec le champ `BASEPRI` (bits [7 :0]), définit le niveau de priorité minimal d'une exception ou d'une interruption, afin que celle-ci puisse lever une exception. Si la valeur du registre est 0, le filtre n'a aucun effet. Les instructions `msr` et `mrs` donnent accès à son contenu.

Registre de contrôle des modes du processeur

La gestion des modes du processeur s'effectue via le registre de contrôle des modes du processeur, le registre `CONTROL` (figure 3.39).

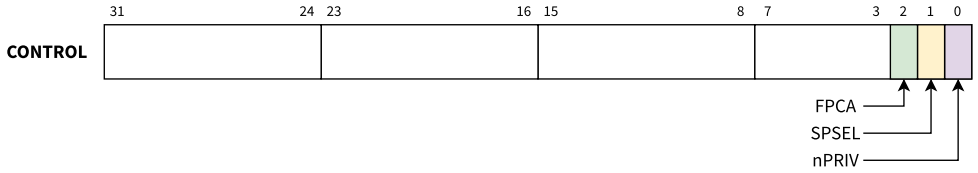


FIGURE 3.39 – Profil M : Control Register

— Le bit **mPRIV** (bit[0]) définit le niveau de privilège pour l'exécution du code dans le mode `Thread`. Avec un bit mis à 0, le code s'exécute au niveau privilégié. Si le bit est 1, le code s'exécute au niveau non privilégié.
— Le bit **SPSEL** (bit[1]) définit le pointeur de pile à utiliser pour l'exécution du code en mode `Thread`. Avec un bit mis à 0, le processeur utilise le `MSP`. Si le bit est 1, le processeur utilise le `PSP`. Si le processeur se trouve dans le mode `Handler`, le processeur ne considère pas ce bit et utilise toujours le `MSP`.
— Le bit **FPCA** (bit[2]) indique si l'extension à virgule flottante (FP) est active. Un 0 indique que l'extension FP est inactive. Un 1 indique q'elle est active.

Les instructions `msr` et `mrs` donnent accès à son contenu. Lors d'une écriture, le code doit encore utiliser l'instruction `isb` afin de garantir l'effet de l'opération avant l'exécution de la prochaine instruction.

3.5 Performances

Pour les concepteurs et architectes de processeurs, l'amélioration des performances de leurs processeurs ainsi que la diminution de leurs coûts sont des sujets permanents. La diminution de la taille des

transistors, l'augmentation de la fréquence de l'horloge système et la largeur du bus µP pour l'accès au code et aux données furent d'un apport majeur à cette amélioration.

Mais existe-t-il d'autres chemins pour réduire le temps d'exécution d'un programme ?

Côté logiciel, le choix des bonnes structures de données et des bons algorithmes pour les manipuler est évidemment déterminant afin de réaliser un programme nécessitant un minimum de ressources matérielles (µP, RAM, Flash, temps) pour remplir le cahier des charges du système. Côté matériel, suffit-il juste de réduire le temps de cycle de l'horloge avec des fréquences plus élevées ou existe-t-il d'autres approches ? Est-il possible de mieux utiliser les ressources internes du µP ?

3.5.1 Pipelining

Afin de réduire encore le temps nécessaire à l'exécution d'un programme, il faut s'intéresser à l'exécution des instructions et regarder les différents blocs de traitement qu'une instruction doit franchir avant d'être complètement exécutée.

Principe

En 1913, Henry Ford inaugura la première chaîne de montage pour des véhicules automobiles. L'objectif était de réduire le temps nécessaire au montage des modèles T de plus 12 heures à 1 heure 30.

En prenant cet exemple, il est intéressant de comparer un montage séquentiel du véhicule avec un montage parallèle (figure 3.40). Pour l'exemple, le montage du véhicule s'effectue en seulement trois étapes, première étape montage du châssis, deuxième étape montage du toit et troisième étape montage des roues.

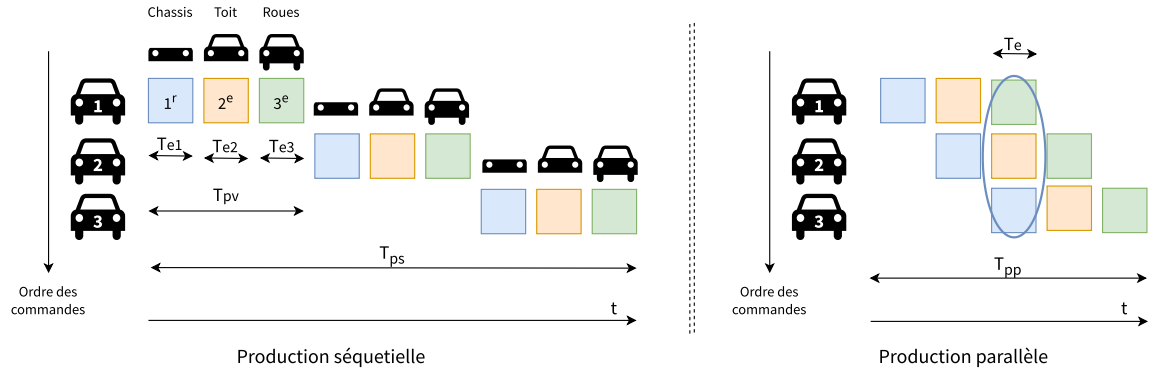


FIGURE 3.40 – Séquentiel vs Parallèle

Le temps nécessaire pour effectuer chacune de ces étapes est T_{e1}, T_{e2} et T_{e3}, soit un total T_{pv} pour la production d'un véhicule. Si l'on produit n véhicules séquentiellement le temps nécessaire est :

$$T_{ps} = n \cdot T_{pv} \text{ avec } T_{pv} = T_{e1} + T_{e2} + T_{e3}$$

Sachant que la valeur de T_e correspond au temps de l'étape prenant le plus de temps, alors si la production s'effectue en parallèle, le temps nécessaire pour produire ces n véhicules est de :

$$T_{pp} = n \cdot T_e + T_{e1} + T_{e2} \text{ avec } T_e = max(T_{e1}, T_{e2}, T_{e3})$$

Si le nombre de véhicules produits devient très grand, alors le temps de production des véhicules vaut :

$$T_{pp} = n \cdot Te$$

ce qui signifie que chaque T_e un nouveau véhicule est produit.

Pipeline idéal

D'après l'architecture générale, une instruction s'exécute en trois cycles principaux, lecture (*fetch*), décodage (*decode*) et exécution (*execute*). Ces trois cycles correspondent également aux trois unités de l'architecture des CPU ARMv7. De ce fait, l'exécution d'une instruction devra traverser chacune de ces unités.

Pour appliquer le principe de pipelining à un système électronique et plus particulièrement au CPU du µP, il est nécessaire de découpler les unités de traitement, appelées également étage du pipeline, avec des mémoires tampons (*buffer*). Ces tampons permettent de rendre les étages indépendants les uns des autres (figure 3.41). Pour un pipeline idéal, deux étages supplémentaires sont nécessaires, un étage d'accès à la mémoire et un étage de réécriture de la donnée après traitement dans la banque de registres.

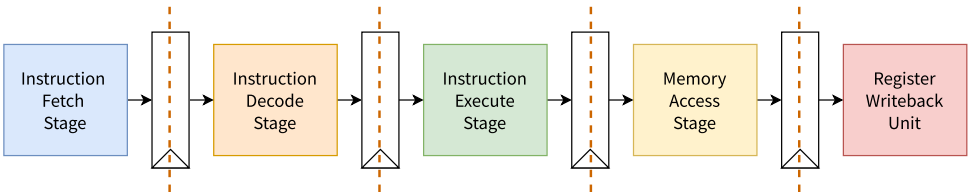


FIGURE 3.41 – Etages de traitement

En faisant l'assomption que le temps de traitement de chacun de ces étages prend un cycle d'horloge, alors le temps de traitement d'une instruction prendra cinq cycles d'horloge (figure 3.42). Il est intéressant de noter que dès le 5e cycle tous les étages du pipeline sont actifs.

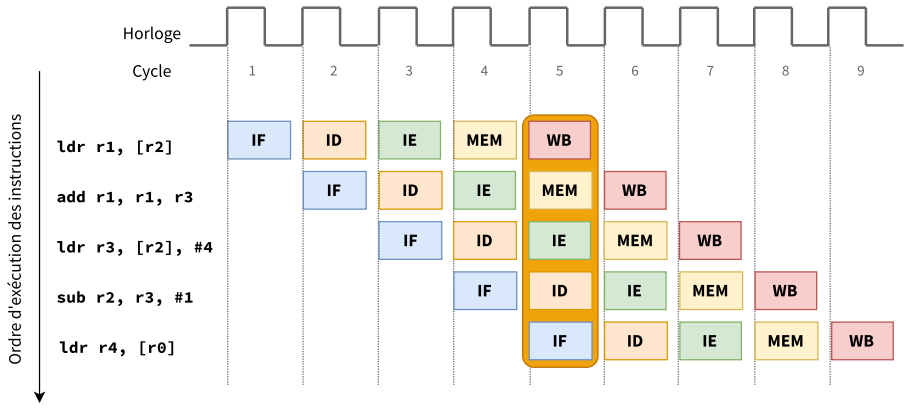


FIGURE 3.42 – Pipeline idéal

Si le programme contient un grand nombre d'instructions, alors à chaque cycle d'horloge, une instruction finit son exécution et une nouvelle est chargée. Dans ce cas idéal où le pipeline ne bloque jamais, le nombre de cycles par instruction est égal 1 (CPI = 1.0). Cependant, si des bulles (*stall*) doivent être introduites, le CPI augmentera, par exemple si le pipeline doit être bloqué durant 1 cycle pour 20% des instructions et de 3 cycles pour 5% des instructions, alors le nouveau CPI sera de 1.35.

$$\text{CPI} = 1 + 1 \cdot 0.20 + 3 \cdot 0.05 = 1.35$$

L'accès à la mémoire centrale et un exemple typique de blocage du pipeline (figure 3.43).

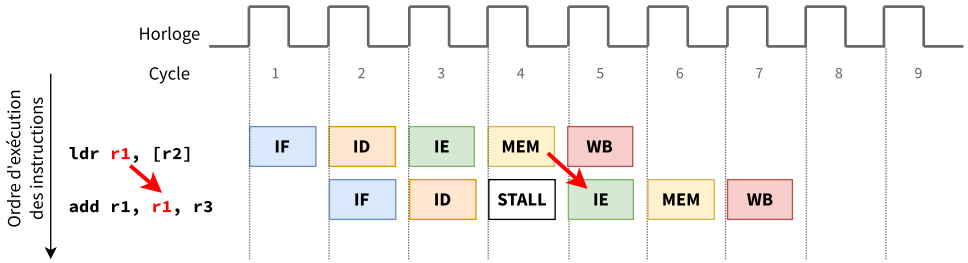


FIGURE 3.43 – Accès mémoire

En effet, la latence due un accès à une mémoire volatile de type DRAM placée à l'extérieure de la puce du µP est typiquement 10 à 100 fois plus grand que la période de l'horloge du CPU et de son pipeline. L'utilisation de mémoire cache permet de réduire drastiquement cette latence et d'éviter des blocages.

Aléas du pipeline

Le pipeline permet de lancer à chaque cycle d'horloge l'exécution d'une nouvelle instruction débute alors que d'autres sont encore en cours d'exécution dans le pipeline, c'est-à-dire l'exécution des instructions se chevauche (*overlapped*). Cependant, il existe une série de cas où l'exécution d'une instruction doit attendre, car elle entre en concurrence avec une instruction déjà en cours d'exécution pour l'accès à une ressource, tels la mémoire ou un registre. Afin garantir une exécution correcte, l'instruction est stoppée dans le pipeline, bloquant par conséquent également l'exécution des instructions qui la suivent. Ces cas sont connus sous le nom d'aléas du pipeline (*Pipeline Hazards*).

Il existe principalement trois classes d'aléas :

— Aléas structurels (*Structural Hazards*)
— Aléas de données (*Data Hazards*)
— Aléas de contrôle (*Control Hazards*)

Il est généralement possible d'éviter ces aléas en dupliquant les ressources partagées. Néanmoins, cela n'est pas toujours possible, notamment pour des accès à la mémoire. Dans ce cas, des bulles (*stall*) remplaceront le traitement des instructions dans les étages du pipeline concernés.

Aléas structurels

Ces aléas arrivent lorsque deux instructions dans des étages différents du pipeline requièrent la même ressource matérielle.

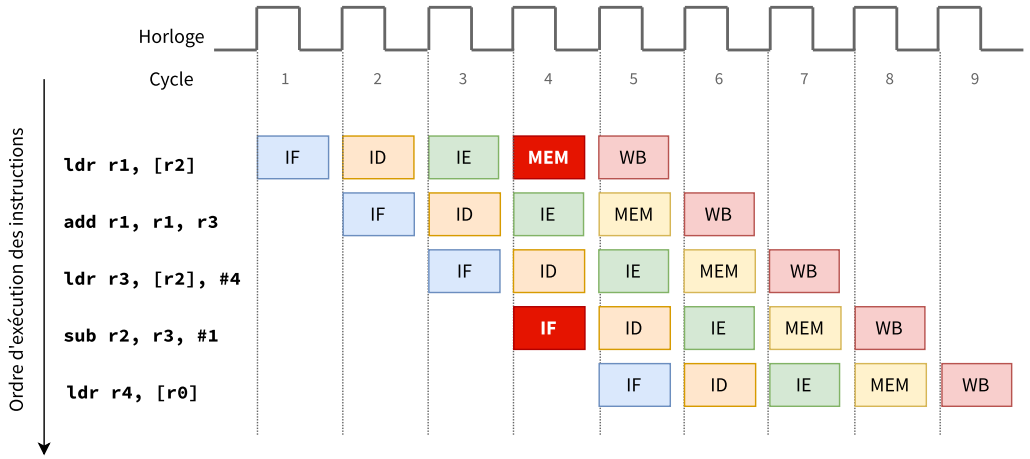


FIGURE 3.44 – Aléas structurels

Dans l'exemple ci-dessus (figure 3.44), l'exécution de la première instruction "`ldr r1, [r2]`" entre en conflit avec le chargement de la quatrième instruction "`sub r2, r2, #1`". En effet, l'étage `MEM`

et l'étage `IF` sont en concurrence pour l'accès à la mémoire, la première pour lire une donnée et la deuxième pour lire l'instruction. Une première solution pour résoudre ce conflit est d'ajouter un bulle (*stall*) afin de laisser l'étage `MEM` s'exécuter. Une meilleure solution est de mettre en oeuvre l'architecture Harvard séparant le bus de données du bus des instructions permettant un accès simultané au code et aux données.

Aléas de données

Ces aléas arrivent lorsqu'une instruction requiert une donnée qui n'est pas complètement traitée par l'exécution d'une instruction précédente encore en cours d'exécution dans le pipeline. Ce problème est la conséquence de l'exécution d'une instruction en plusieurs étapes. En effet, l'évaluation des arguments d'une instruction s'effectue dans les premiers étages du pipeline alors les derniers étages produisent le résultat.

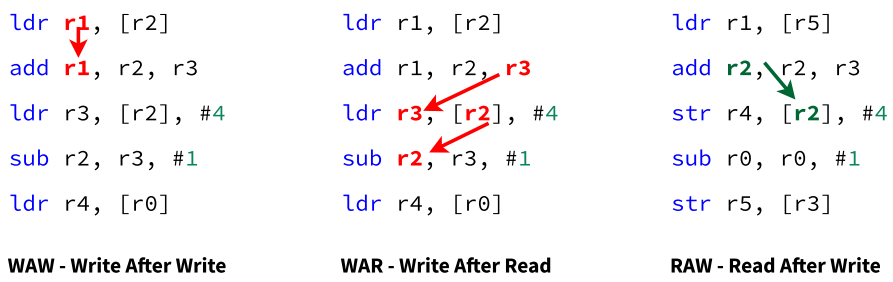


FIGURE 3.45 – Aléas de données

Le code d'exemple ci-dessus (figure 3.45) présente trois cas de figure d'aléas de données :

- WAW : Ecriture après écriture (*Write After Write*)
- WAR : Ecriture après lecture (*Write After Read*)
- RAW : Lecture après écriture (*Read After Write*)

L'aléa **WAW** survient lorsque deux instructions utilisent le même registre pour stocker (*Write*) le résultat de leur exécution, dans l'exemple il s'agit du registre `R1`. Dans un tel cas, il est essentiel de ne pas changer l'ordre des instructions afin d'éviter que l'instruction suivante reçoive la mauvaise valeur. Cet aléa est aussi connu sous le nom de "dépendance de noms" (*Name Dependency*).

L'aléa **WAR** survient lorsque deux instructions utilisent le même registre, la première instruction en lecture (*Read*) et la deuxième en écriture (*Write*). Dans l'exemple ci-dessus, il s'agit du registre `R3` pour la deuxième et troisième instruction, ainsi que du registre `R2` pour la troisième et quatrième instruction. Ces dépendances peuvent se résoudre en utilisant d'autres registres encore libres (figure 3.46).

L'aléa **RAW** survient lorsqu'une instruction utilise un registre (*Read*) dont le contenu est écrit (*Write*)

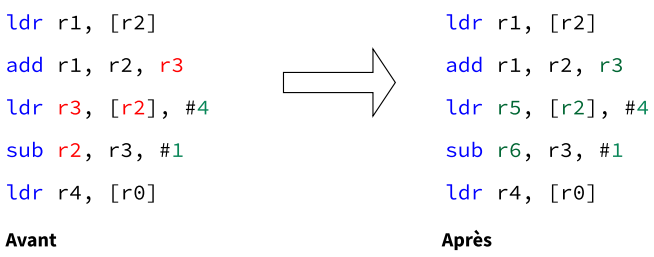


FIGURE 3.46 – Dépendances de noms

par l'exécution de l'instruction précédente. La flèche verte (figure 3.47) représente ce cas de figure typique lors de traitements de données stockées en mémoire.

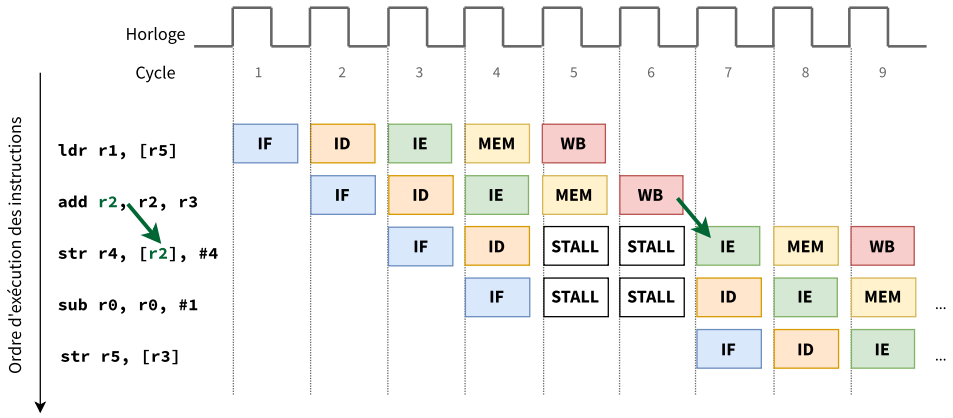


FIGURE 3.47 – Aléas RAW

Si les données doivent impérativement être communiquées par les registres, il est absolument indispensable de terminer le traitement de la deuxième instruction (ADD) avant de pouvoir poursuivre l'exécution de la troisième instruction (STR). Ce n'est qu'une fois que l'instruction ADD a traversé l'étage WB, que le résultat de l'addition sera stocké dans la banque de registres. Par contre l'instruction STR nécessite ce résultat en entrant dans l'étage IE. Ceci implique l'ajout de deux bulles (*stall*).

Est-il possible d'éviter l'ajout de ces bulles (*stall*). Si oui, que faudrait-il mettre en place ?

La solution à cet aléa réside à l'ajout de logique permettant mettre à disposition d'un étage précédent du pipeline le résultat d'un étage, ceci sans passer par la banque de registres (figure 3.48). Dans l'exemple ci-dessus, le résultat d'une instruction calculé à l'étage IE est directement mis à l'entrée de cet étage IE.

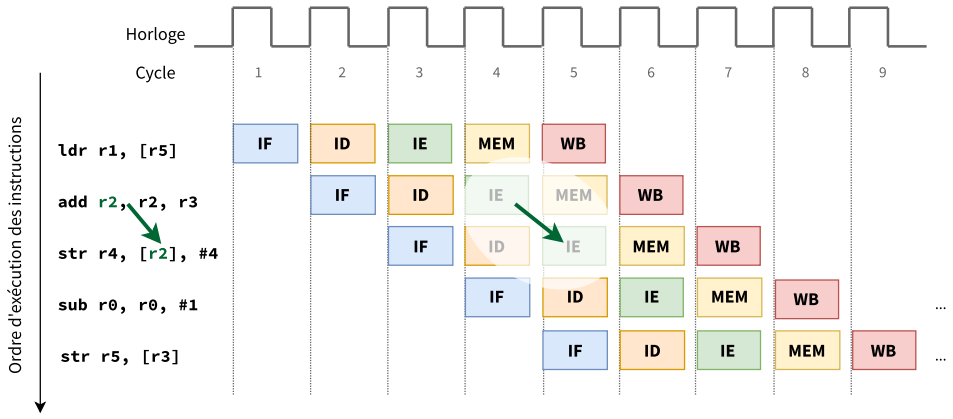


FIGURE 3.48 – Aléas RAW résolu

Aléas de contrôle

Ces aléas arrivent lorsqu'une instruction de branchement, conditionnelle ou pas, est exécutée et que le branchement est effectué. Dans le cas ù le branchement est effectif, les instructions suivantes déjà dans le pipeline ne doivent plus être exécutées (figure 3.49). Ces instructions se transforment en instruction NOP et 2 cycles sont ainsi perdus.

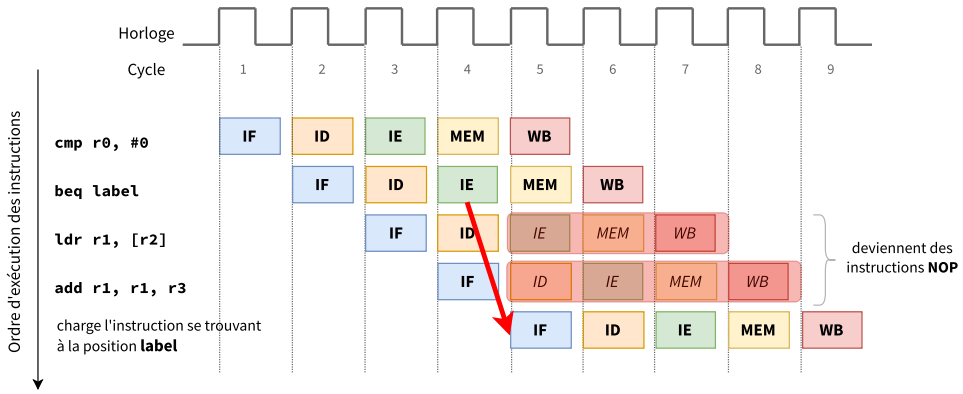


FIGURE 3.49 – Aléas de contrôle

Pour éviter ce problème, différentes stratégies sont imaginables, telle l'évaluation du branchement dans l'étage de décodage, la prédiction des branchements ou l'exécution conditionnelle d'instruction.

3.5.2 Processeur superscalaire

Les processeurs superscalaires disposent d'une unité centrale de traitement (CPU) équipée d'un pipeline capable de traiter simultanément plusieurs instructions (figure 3.50).

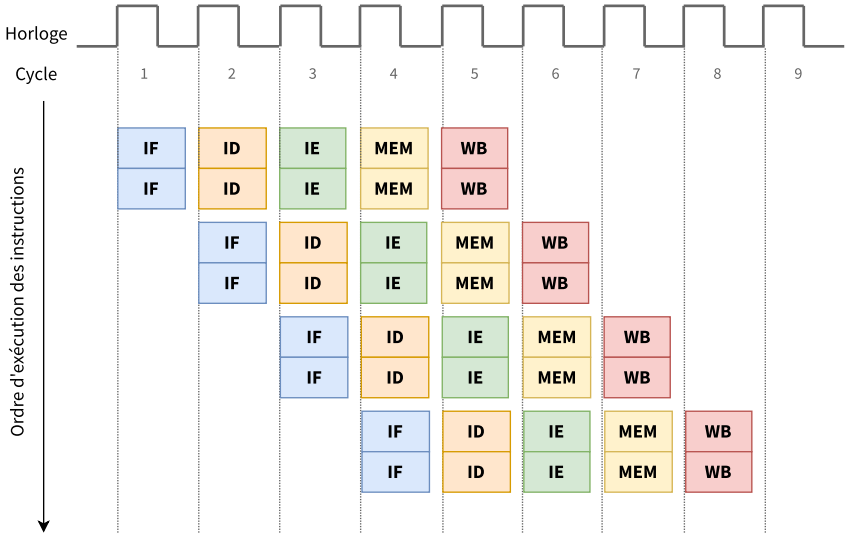


FIGURE 3.50 – Exécution superscalaire

A l'opposé des processeurs scalaires ne pouvant exécuter qu'une seule instruction par étage de leur pipeline, les processeurs superscalaires disposent d'une unité centrale de traitement (CPU) avec de plusieurs unités de traitement dans les différents étages de son pipeline (figure 3.51).

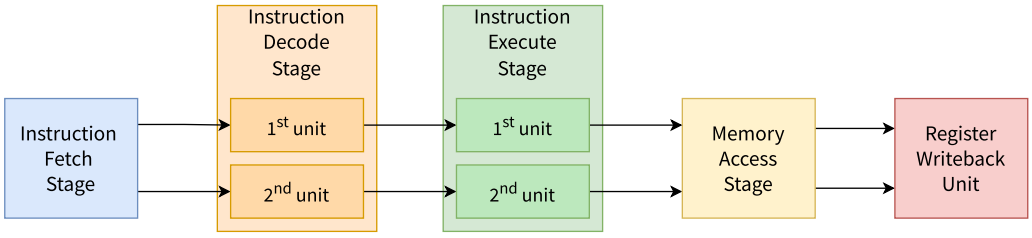


FIGURE 3.51 – Pipeline superscalaire

Le CPU de ces processeurs est apte à détecter le parallélisme existant entre les instructions d'un programme. Il l'utilise pour exécuter simultanément celles qui peuvent l'être. Cette architecture permet de réduire le temps d'exécution du programme, mais au détriment d'une complexité accrue de son design et d'une plus grande consommation d'énergie.

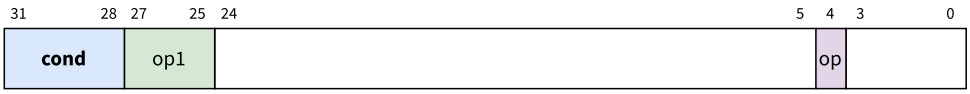
3.5.3 Profil A

Il existe une grande variée de processeurs basés sur le profil A. Cependant, tous ces processeurs implémentent un pipeline entre 8 et 15 étages et hormis le µP ARM Cortex-A5 tous sont des processeurs superscalaires.

Pour contrecarrer les effets des aléas de contrôle sur leur pipeline, les processeurs de ce profil disposent de deux outils précieux, la prédiction de branchement (*Branch Prediction*) et d'instructions pouvant s'exécuter conditionnellement.

Avec la prédiction de branchement, le CPU essaye d'évaluer le résultat d'un branchement afin de poursuivre la lecture des instructions sur la branche correspondante au résultat effectif. L'idée est d'éviter de charger des instructions qu'il ne devra pas exécuter, car pas sur la bonne branche, et lui évitant ainsi de perdre de précieux cycles d'horloge dans l'exécution du programme. Cette fonctionnalité est spécialement importante pour les processeurs avec un pipeline profond (beaucoup d'étages).

L'exécution conditionnelle d'instructions permet d'éviter la perte de cycles d'horloge due au branchement. Seule l'exécution de l'instruction concernée par la condition, si elle est négative, sera stoppée. La condition est contenue directement dans le champ cond (bits [31 :28]) de l'instruction (figure 3.52).



cond	op1	op	Classes d'instructions
≠1111	00x	-	Data-processing and miscellaneous instructions
≠1111	010	-	Load/store word and unsigned byte
≠1111	011	0	Load/store word and unsigned byte
≠1111	011	1	Media instructions
≠1111	10x	-	Branch, branch with link, and block data transfer
≠1111	11x	-	Coprocessor instructions, and Supervisor Call Includes Floating-point instructions and Advanced SIMD data transfers
= 1111	-	-	Unconditional instructions Includes Advanced SIMD instructions

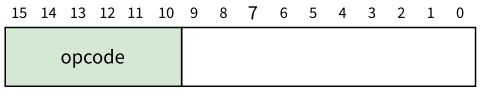
FIGURE 3.52 – ARM Instruction Set Encoding (code de condition + classes d'instructions)

Les champs cond, op1(bits[27 :25]) et op (bit [5]) servent également à subdiviser le jeu d'instructions ARM en plusieurs classes.

3.5.4 Profil M

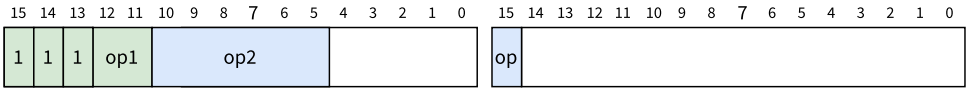
En version ARMv7, il existe actuellement trois variantes de µC basés sur le profil M, les Cortex-M3, M4 et M7. Ces µC sont des processeurs scalaires avec un pipeline de trois étages seulement pour les deux premières variantes et six étages pour la troisième. Vu la faible profondeur du pipeline des deux premières variantes, certains de ses étages nécessitent plus d'un cycle d'horloge pour effectuer le traitement correspondant à l'instruction.

Les µC Cortex-M3 et M4 n'implémentent pas de prédiction de branchement. Les jeux d'instructions Thumb (figure 3.53) et Thumb-2 (figure 3.54) ne supportent pas d'exécution conditionnelle systématique sur toutes les instructions. Par contre, ils proposent une instruction `IT` (*If-Then*) permettant d'éviter un branchement conditionnel si une, deux ou trois instructions doivent s'exécuter en fonction d'un test effectué préalablement.



opcode	Classes d'instructions
00x xxx	Shift (immediate), add, subtract, move, and compare
010 000	Data processing
010 001	Special data instructions and branch and exchange
010 01x	Load from Literal Pool, see LDR
010 1xx 011 xxx 100 xxx	Load/store single data item
101 00x	Generate PC-relative address, see ADR
101 01x	Generate SP-relative address, see ADD (SP plus immediate)
101 1xx	Miscellaneous 16-bit instructions
110 00x	Store multiple registers
110 01x	Load multiple registers
110 1xx	Conditional branch
111 00x	Unconditional Branch
111 01x 111 10x 111 11x	Instructions du jeu d'instructions Thumb-2

FIGURE 3.53 – Thumb Instruction Set Encoding



op1	op2	op	Classes d'instructions
01	00xx 0xx	x	Load Multiple and Store Multiple
01	00xx 1xx	x	Load/store dual or exclusive, table branch
01	01xx xxx	x	Data processing (shifted register)
01	1xxx xxx	x	Coprocessor instructions
10	x0xx xxx	0	Data processing (modified immediate)
10	x1xx xxx	0	Data processing (plain binary immediate)
10	xxxx xxx	1	Instructions d'addition avec le PC
11	000x xx0	x	Store single data item
11	00xx 001	x	Load byte, memory hints
11	00xx 011	x	Load halfword, memory hints
11	00xx 101	x	Load word
11	00xx 111	x	UNDEFINED
11	010x xxx	x	Data processing (register)
11	0110 xxx	x	Multiply, multiply accumulate, and absolute difference
11	0111 xxx	x	Long multiply, long multiply accumulate, and divide
11	1xxx xxx	x	Coprocessor instructions

FIGURE 3.54 – Thumb-2 Instruction Set Encoding

3.6 Exercices

Quelques exercices pour assimiler la matière de ce chapitre.

3.6.1 Exercice 1 : Profils

Citez les trois profils des processeurs ARM.
Décrivez leurs caractéristiques principales.

3.6.2 Exercice 2 : Caractéristiques des µP ARMv7

Citez les caractéristiques principales des µP ARMv7

3.6.3 Exercice 3 : Organisation du code

Quels sont les algorithmes de base traités par un processeur ?
Quel est le rôle du compteur ordinal ?
Quel est le rôle du registre de liens ?

3.6.4 Exercice 4 : Organisation des données

Quelles sont les différentes catégories de données ?
Quelles sont les caractéristiques principales de ces catégories ?
Comment les processeurs ARM procèdent-ils pour traiter des données ?

3.6.5 Exercice 5 : Pile d'exécution

Quel est le rôle de la pile d'exécution ?
Quelle est l'organisation d'une pile d'exécution ?
Quel est le rôle du pointeur de pile ?

3.6.6 Exercice 6 : Registres du processeur

Citez la liste des registres des processeurs ARM ?
Quelles différences existe-t-il entre les profils A, R et M ?
Quelle différence existe-t-il pour l'accès à ces registres entre le jeu d'instructions ARM et Thumb ?
Quelle information contient le registre APSR ?

3.6.7 Exercice 7 : Modes de fonctionnement

Quels sont les modes de fonctionnement des CPU des profils A et M ?
Quelle est leur utilité ?

3.6.8 Exercice 8 : Performances des µP

Quels mécanismes sont mis en place par les concepteurs de µP et µC pour améliorer leurs performances ?

3.6.9 Exercice 9 : Pipeline

Quel est le principe de fonctionnement d'un pipeline ?
Comment peut-on calculer le temps d'exécution d'un programme ?

3.6.10 Exercice 10 : Aléas du pipeline

La mise en oeuvre d'un pipeline se confronte à quelques problèmes, des aléas.
Citez et décrivez succinctement les 3 classes d'aléas ainsi que le moyen d'y pallier.