Chapitre 1. Introduction aux processeurs ARM-Cortex

- Les différents types de processeur ARM-Cortex
- Architecture des processeurs ARM Cortex
- Les registres
- La mémoire
- Le pipeline
- Les interruptions et les exceptions
- Le jeu d'instructions
- Les performances
- Introduction aux familles STM32.
- L'environnement de développement SM32CubeIDE.
- Aperçu sur la carte de développement Nucléo.

1.1 Introduction

Dans un monde où les systèmes embarqués dominent l'innovation technologique – des objets connectés IoT aux dispositifs médicaux intelligents et aux solutions agricoles automatisées –, les processeurs ARM Cortex se positionnent comme le cœur battant de cette révolution. Développés par ARM Holdings, ces processeurs RISC (Reduced Instruction Set Computer) offrent un équilibre optimal entre performances, consommation énergétique et coût, rendant possible l'intégration de l'intelligence artificielle et des capteurs avancés dans des appareils compacts et autonomes.

Ce chapitre pose les fondations essentielles pour la maîtrise de l'architecture interne des processeurs ARM Cortex, en se focalisant sur la famille Cortex-M, omniprésente dans les microcontrôleurs STM32 de STMicroelectronics. L'architecture RISC de ces cœurs exploite des mécanismes avancés comme le pipeline, les interruptions NVIC et le jeu d'instructions Thumb-2 pour répondre à des contraintes temps réel critiques, tout en minimisant la consommation – un atout clé pour les prototypes IoT en santé (monitoring biomédical) ou agriculture (optimisation bioénergétique).

Objectifs principaux :

- Comprendre les familles Cortex (M, R, A) et leur architecture RISC.
- Maîtriser les registres, la mémoire mappée, le pipeline et les interruptions.
- Explorer le jeu d'instructions et les métriques de performance (DMIPS/MHz).
- S'initier aux STM32, à STM32CubeIDE et aux cartes Nucleo pour un prototypage rapide.

À travers des exemples concrets (comme la gestion d'une interruption GPIO sur une carte Nucleo), ce chapitre prépare à la programmation pratique en assembleur et C, ainsi qu'à des projets réels tels qu'un capteur IoT ou un système de monitoring santé. Le détail suit ci-après.

1.2 Différents types de processeurs ARM-Cortex

La famille ARM Cortex regroupe plusieurs types de processeurs conçus pour répondre à des besoins différents en matière de performance, de consommation, de temps réel et de complexité logicielle. Chaque famille possède des caractéristiques propres qui orientent son utilisation vers des domaines bien précis, allant des microcontrôleurs simples aux systèmes embarqués les plus puissants. On distingue principalement trois grandes catégories de processeurs Cortex : Cortex-M, Cortex-R et Cortex-A. Cette classification permet de sélectionner l'architecture la plus adaptée selon les contraintes de l'application à développer.

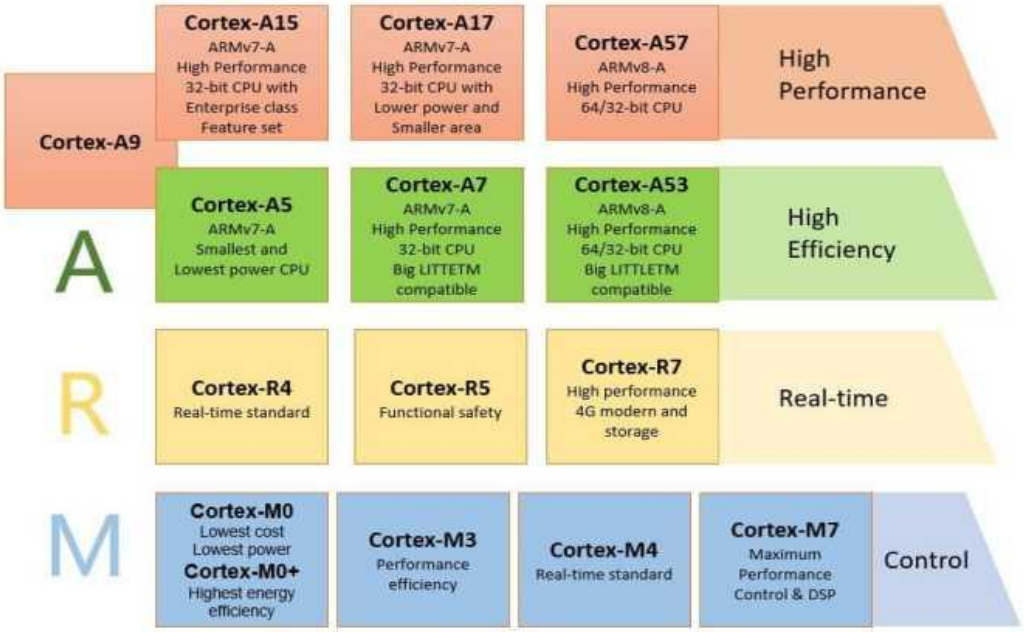

Figure 1.1 : types de processeurs ARM-Cortex.

Les processeurs Cortex-M (Microcontroller Profile) sont destinés aux microcontrôleurs embarqués et aux systèmes à faible consommation. Ils sont largement utilisés dans les capteurs intelligents, les objets connectés, la domotique et de nombreux systèmes industriels simples. Leur principal avantage réside dans leur simplicité de mise en œuvre, leur faible consommation énergétique et leur très bon comportement en temps réel. Ils sont très utilisés dans les systèmes embarqués temps réel et les applications industrielles.Les processeurs **Cortex-M** sont destinés principalement aux **microcontrôleurs**. Ils sont optimisés pour :

- une **faible consommation d'énergie**,
- une **architecture simple**,
- une **réponse rapide aux interruptions**.

Ils sont très utilisés dans les **systèmes embarqués temps réel** et les applications industrielles.

Caractéristiques principales :

- Architecture **RISC 32 bits**
- Gestion efficace des interruptions via le **NVIC**
- Exécution déterministe (temps réel)
- Faible coût matériel
- Mémoire limitée (Flash et SRAM)

Exemples d'applications :

- Automates industriels
- Systèmes embarqués éducatifs
- Capteurs intelligents (IoT)
- Cartes STM32, Arduino compatibles ARM

Exemples de cœurs : Cortex-M0, M3, M4, M7

❖ **Les processeurs Cortex-R (Real-Time Profile)** sont conçus pour les applications temps réel critiques. Ils sont employés dans les domaines où la fiabilité et la rapidité de réponse sont essentielles, comme l'automobile, le contrôle industriel ou certaines applications aéronautiques. Leur architecture est pensée pour garantir un comportement déterministe et une grande robustesse face aux contraintes temporelles.

Caractéristiques principales :

- Temps de réponse **strictement déterministe**
- Support de mécanismes de **tolérance aux fautes**
- Performances supérieures aux Cortex-M
- Gestion avancée de la mémoire
- Forte robustesse

Exemples d'applications :

- Systèmes automobiles (freinage ABS, airbag)
- Contrôle moteur
- Systèmes ferroviaires
- Équipements aéronautiques

Les Cortex-R sont moins courants dans l'enseignement, mais très présents dans l'industrie critique.

- **Les processeurs Cortex-A (Application Profile)** sont orientés vers les applications hautes performances. Ils sont utilisés dans les smartphones, les tablettes, les systèmes multimédias et les plateformes embarquées exécutant un système d'exploitation complet. Ils offrent une puissance de calcul élevée ainsi que des fonctionnalités avancées, mais avec une consommation plus importante que celle des Cortex-M.

Le choix entre ces trois familles dépend donc des objectifs du système à concevoir. Lorsqu'il s'agit d'un microcontrôleur simple, peu coûteux et peu énergivore, le Cortex-M constitue la solution la plus adaptée. Pour des applications critiques nécessitant une réponse stricte dans le temps, le Cortex-R est plus approprié. Enfin, pour les systèmes embarqués complexes exigeant de fortes capacités de traitement, le Cortex-A est le plus indiqué.

Dans le cadre des microcontrôleurs STM32, la famille Cortex-M occupe une place centrale, car elle répond parfaitement aux besoins des applications embarquées modernes. Elle constitue ainsi la base idéale pour l'apprentissage, le prototypage et le développement de projets industriels ou académiques.

1.3. Architecture générale des processeurs ARM Cortex

L'architecture générale des processeurs ARM Cortex repose sur le principe RISC, c'est-à-dire une organisation fondée sur un ensemble réduit d'instructions simples, rapides à décoder et à exécuter. Cette approche permet d'optimiser le compromis entre performances, consommation énergétique et surface matérielle, ce qui explique son adoption massive dans les systèmes embarqués.

Dans cette architecture, le processeur est structuré autour d'une unité de traitement arithmétique et logique, d'un banc de registres internes, d'unités de contrôle chargées de la gestion des flux d'exécution, ainsi que de mécanismes dédiés à la gestion des interruptions et, selon la famille considérée, à la protection mémoire et à la virtualisation.

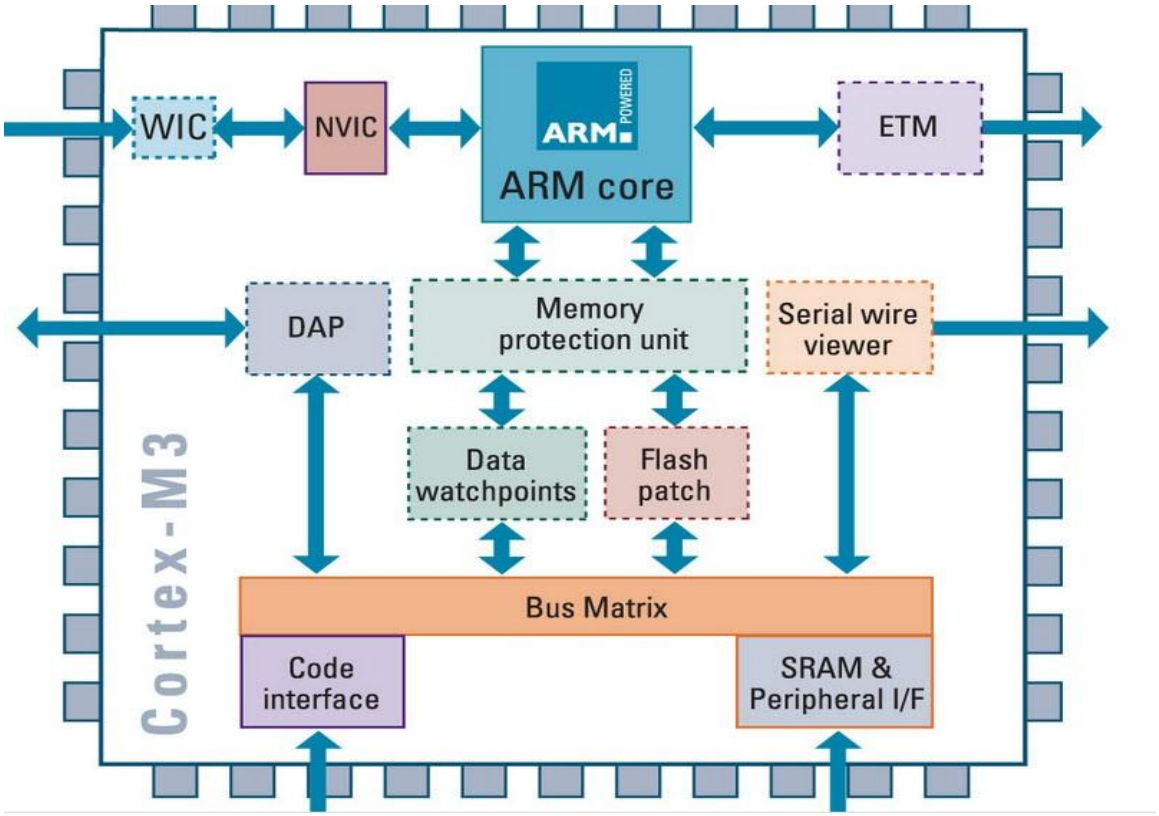

Figure 1.2 : Architecture générale des processeurs ARM Cortex.

L'organisation interne du cœur Cortex vise à réduire la complexité du matériel tout en maintenant une exécution efficace des instructions, notamment grâce à une forte intégration entre les blocs fonctionnels.

Le Cortex-M, en particulier, est conçu pour les microcontrôleurs à faible consommation, avec une architecture simplifiée mais très réactive, adaptée aux contraintes du temps réel. Cette structure matérielle est complétée par une gestion hiérarchisée des accès mémoire et par des mécanismes de pipeline qui permettent de chevaucher plusieurs étapes d'exécution afin d'augmenter le débit de traitement. Ainsi, l'architecture ARM Cortex se distingue par sa capacité à offrir une base matérielle souple, évolutive et particulièrement bien adaptée aux besoins variés des applications embarquées modernes. Cela signifie que le processeur utilise un ensemble réduit d'instructions simples, ce qui permet :

- Une exécution rapide
- Une faible consommation énergétique
- Une meilleure optimisation du pipeline
- L'architecture Cortex intègre généralement :
- Une unité de calcul (ALU)
- Des registres internes
- Un contrôleur d'interruptions
- Un bus de communication interne
- Une unité de gestion mémoire (selon le modèle)

Le modèle Cortex-M est particulièrement optimisé pour les microcontrôleurs.

L'architecture RISC se caractérise par :

- un **jeu d'instructions réduit et homogène**,
- des instructions de **longueur fixe** (ou quasi fixe),
- une exécution des instructions généralement en **un nombre réduit de cycles d'horloge**,
- une séparation claire entre **calculs** et **accès mémoire** (architecture *load/store*).

Dans les processeurs ARM Cortex :

- seules les instructions **Load** et **Store** accèdent à la mémoire,
- les opérations arithmétiques et logiques s'effectuent **exclusivement sur les registres**.

Cela permet de simplifier le matériel et d'augmenter la vitesse d'exécution. Un processeur ARM Cortex est organisé autour des blocs suivants :

1. **Unité de traitement (CPU Core)**
   Elle assure l'exécution des instructions et comprend :
   - l'**ALU** (Arithmetic Logic Unit),
   - l'unité de calcul logique et arithmétique,
   - le décodeur d'instructions.
2. **Banque de registres**
   Elle contient :
   - les registres généraux (R0 à R12),
   - le **compteur ordinal (PC)**,
   - le **registre d'état (xPSR)**,
   - les registres de pile.
3. **Unité de contrôle**
   Elle coordonne :
   - le séquencement des instructions,

- la gestion du pipeline,
- les sauts et branchements.

4. **Interface mémoire**
   Elle permet l'accès à :
   - la mémoire Flash (programme),
   - la SRAM (données),
   - les périphériques mappés mémoire.

1.4 Les registres

Dans l'architecture ARM Cortex, les registres constituent la zone de stockage la plus rapide du processeur et occupent une place essentielle dans l'exécution des instructions. Ils servent à conserver temporairement les données, les adresses, les résultats intermédiaires et certaines informations de contrôle, ce qui permet de limiter les accès à la mémoire principale et d'améliorer les performances du système.

Le banc de registres d'un Cortex-M comprend seize registres principaux, notés R0 à R15. Les registres R0 à R12 sont des registres généraux utilisés pour stocker les paramètres, les variables temporaires et les résultats de calcul. Le registre R13 correspond au Stack Pointer, chargé de repérer le sommet de la pile et de gérer les appels de fonctions, les variables locales et les interruptions. Le registre R14, appelé Link Register, conserve l'adresse de retour lorsqu'une fonction est appelée. Enfin, le registre R15, ou Program Counter, contient l'adresse de la prochaine instruction à exécuter et assure le déroulement séquentiel du programme.

En plus de ces registres généraux, l'architecture Cortex-M comporte des registres spéciaux qui participent à la gestion de l'état du processeur et de ses modes de fonctionnement. Le registre xPSR rassemble plusieurs indicateurs d'état liés à l'exécution des instructions, tandis que le registre CONTROL permet de gérer certains aspects du mode d'exécution et de l'organisation de la pile. Ces registres deviennent particulièrement importants lors du traitement des interruptions et des exceptions, où le processeur doit sauvegarder puis restaurer rapidement son contexte.

Donc pour résumer :

Le cœur ARM Cortex-M comprend :

- 13 registres généraux (R0–R12)

- Registre SP (Stack Pointer)
- Registre PC (Program Counter)
- Registre xPSR (Program Status Register)

Dans un processeur **ARM Cortex-M**, les registres constituent des **mémoires internes très rapides** utilisées par le cœur pour stocker temporairement les données, les adresses et les informations de contrôle.

Leur rôle est fondamental, car **toutes les opérations arithmétiques, logiques et de contrôle** s'effectuent principalement à partir des registres.

Le cœur ARM Cortex-M dispose d'un ensemble de **registres 32 bits**, organisés pour assurer à la fois **performance**, **simplicité de programmation** et **gestion efficace des interruptions**.

Les registres généraux (R0 à R12)

Les registres R0 à R12 sont appelés registres généraux (General Purpose Registers).
Ils sont utilisés pour :

- stocker les **opérandes** des calculs,
- contenir des **résultats intermédiaires**,
- passer des **paramètres aux fonctions**,
- manipuler des **adresses mémoire**.

Caractéristiques principales :

- Nombre : **13 registres**
- Taille : **32 bits**
- Accès très rapide (un cycle d'horloge)

Utilisation particulière :

R0 à R3 :

- très utilisés pour le passage des paramètres aux fonctions,
- souvent employés pour les calculs temporaires.

R4 à R11 :

- utilisés pour stocker des variables locales,

- généralement sauvegardés lors des appels de fonctions.

R12 (IP – Intra-Procedure call) :
registre temporaire utilisé par le compilateur.

Ces registres rendent l'exécution des programmes très rapide en réduisant les accès à la mémoire.

Le registre SP – Stack Pointer

Le **SP (Stack Pointer)** est un registre spécial qui pointe vers le **sommet de la pile (stack)**.

La pile est une zone mémoire utilisée pour :

- stocker les variables locales,
- sauvegarder le contexte lors des appels de fonctions,
- mémoriser automatiquement les registres lors des interruptions.

Dans les Cortex-M, il existe **deux pointeurs de pile** :

- **MSP (Main Stack Pointer)** : utilisé au démarrage et pour les interruptions,
- **PSP (Process Stack Pointer)** : utilisé par les applications utilisateur.

Le choix du pointeur de pile permet une **meilleure séparation entre système et application**, notamment avec un système d'exploitation temps réel (RTOS).

Le registre PC – Program Counter

Le **PC (Program Counter)** est un registre essentiel qui contient :

- l'**adresse de la prochaine instruction à exécuter**.

À chaque cycle :

- le PC est automatiquement incrémenté,
- ou modifié lors d'un saut, appel de fonction ou interruption.

Rôle principal :

- assurer le **séquencement du programme**,
- permettre les branchements conditionnels et inconditionnels.

Toute modification du PC entraîne un changement du flot d'exécution du programme.

Le registre xPSR – Program Status Register

Le registre **xPSR** regroupe plusieurs informations relatives à l'état du processeur.

Il est en réalité la combinaison de trois registres logiques :

1. APSR (Application Program Status Register)

Contient les indicateurs de résultat :

- ✓ Z (Zero)
- ✓ N (Negative)
- ✓ C (Carry)
- ✓ V (Overflow)

2. IPSR (Interrupt Program Status Register)

Indique :

- ➢ le numéro de l'exception ou de l'interruption en cours.

3. EPSR (Execution Program Status Register)

Contient :

- ➢ l'état du jeu d'instructions (Thumb),
- ➢ des informations sur l'exécution.

Le xPSR permet au processeur de **prendre des décisions conditionnelles** et de gérer correctement les interruptions.

Rôle des registres lors des interruptions

Lorsqu'une interruption survient :

- ➢ le processeur sauvegarde automatiquement certains registres sur la pile :
- ✓ R0 à R3
- ✓ R12
- ✓ PC
- ✓ xPSR
- ✓ le SP est ajusté automatiquement,
- ✓ l'exécution de la routine d'interruption commence.

Ce mécanisme matériel garantit :

- une **faible latence**,
- une **reprise correcte du programme** après l'interruption.

Les registres du cœur ARM Cortex-M :

- constituent la base de l'exécution des programmes,
- permettent des calculs rapides sans accès mémoire,
- assurent la gestion du flot d'exécution (PC),
- facilitent la gestion des interruptions (SP, xPSR).

La maîtrise des registres est **indispensable** pour comprendre :

- la programmation bas niveau,
- le débogage,
- le fonctionnement interne des microcontrôleurs **STM32**.

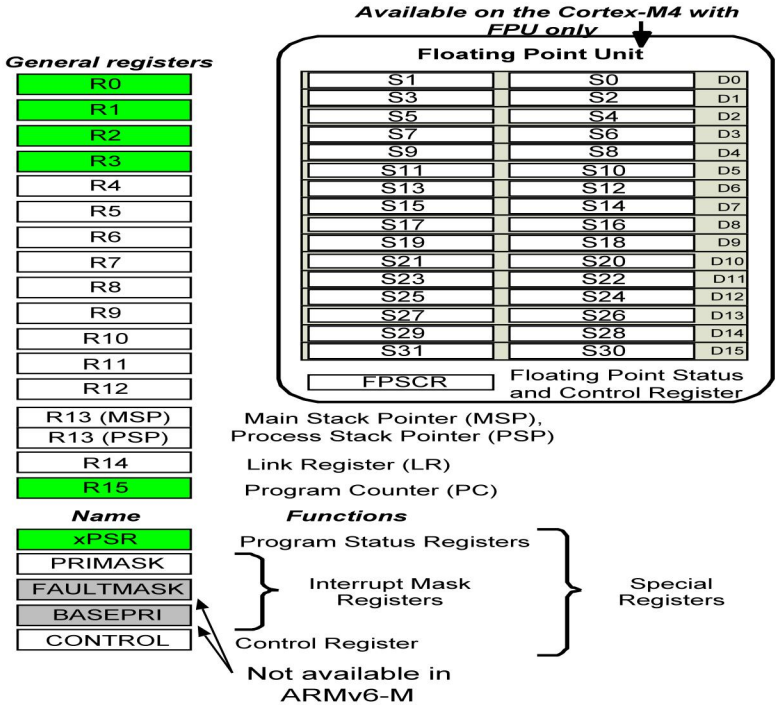

Figure 1.3 : Les registres des processeurs ARM Cortex

1.5. La mémoire

L'organisation de la mémoire dans un processeur ARM Cortex repose sur une structure hiérarchisée qui sépare les différents types d'espaces mémoire selon leur rôle dans le fonctionnement du microcontrôleur. Cette organisation permet d'optimiser à la fois la rapidité d'exécution, la gestion des données et l'accès aux périphériques. Dans les systèmes basés sur ARM Cortex, la mémoire n'est pas seulement un espace de stockage ; elle constitue également un moyen d'interaction direct avec le matériel.

La mémoire d'un microcontrôleur est généralement répartie en plusieurs zones principales. La mémoire Flash est non volatile et conserve le programme même en l'absence d'alimentation. Elle contient le code source compilé, les constantes et parfois certaines tables de données. La SRAM, quant à elle, est une mémoire volatile utilisée pour stocker les variables, la pile et les données temporaires pendant l'exécution du programme. Sa rapidité d'accès en fait un élément essentiel pour le traitement en temps réel. À ces deux zones s'ajoutent les espaces réservés aux périphériques, qui sont mappés en mémoire afin de permettre leur contrôle à travers des adresses spécifiques.

L'une des caractéristiques importantes de l'architecture ARM Cortex est l'utilisation d'une mémoire mappée. Cela signifie que les périphériques matériels, tels que les ports d'entrée-sortie, les timers, les convertisseurs analogique-numérique ou les interfaces de communication, sont accessibles comme s'il s'agissait de cases mémoire ordinaires. Cette approche simplifie la programmation, car il suffit de lire ou d'écrire à une adresse déterminée pour piloter un composant matériel. Ainsi, la mémoire devient une interface directe entre le logiciel et le matériel.

ARM Cortex adopte également une architecture de type Harvard modifiée. Dans cette organisation, les instructions et les données peuvent circuler sur des chemins distincts, ce qui améliore les performances en permettant certains accès simultanés. Cette séparation réduit les conflits d'accès et favorise une exécution plus fluide des programmes. Elle est particulièrement avantageuse dans les applications embarquées où la rapidité de réaction et l'efficacité énergétique sont essentielles.

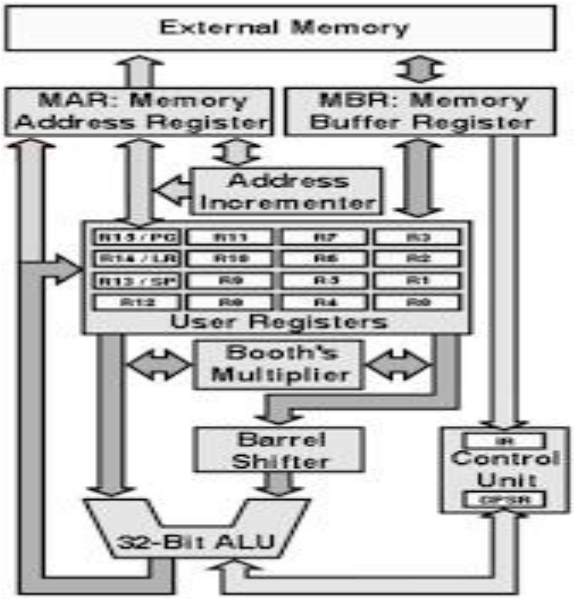

Figure 1.4 : La mémoire des processeurs ARM Cortex

Enfin, la mémoire joue un rôle fondamental dans la gestion de la pile, du contexte d'exécution et des interruptions. Les variables locales, les adresses de retour et les informations temporaires y sont stockées lors de l'exécution des fonctions. Une bonne compréhension de cette organisation est indispensable pour programmer efficacement un microcontrôleur STM32 et pour exploiter correctement ses ressources matérielles.

Donc et pour résumer :

L'architecture mémoire des processeurs **ARM Cortex-M** est conçue pour offrir une **grande simplicité de programmation**, une **rapidité d'accès** et une **compatibilité avec les applications temps réel**.

Elle repose sur un **espace mémoire linéaire**, dans lequel **le programme, les données et les périphériques** sont accessibles par des **adresses mémoire uniques**.

L'espace mémoire est linéaire et comprend :

- Flash : stockage du programme
- SRAM : données et pile
- Périphériques mappés mémoire

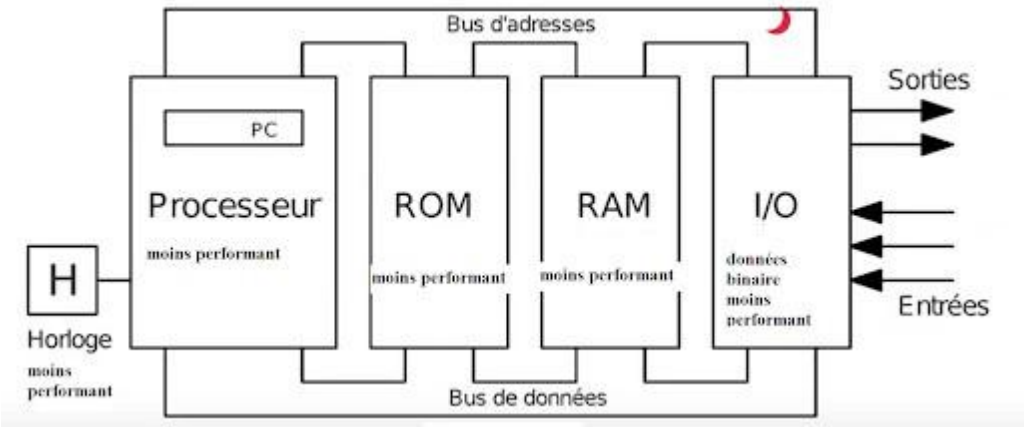

Figure 1.5 : Schéma simplifié de la mémoire.

Espace mémoire linéaire

Dans un Cortex-M, l'espace mémoire est organisé sous la forme d'un **adressage linéaire 32 bits**, ce qui signifie que :

- chaque case mémoire possède une **adresse unique**,
- le processeur accède à la mémoire de la même manière, quel que soit le type de ressource,
- les périphériques matériels sont accessibles comme des variables mémoire.

Ce principe simplifie fortement la programmation bas niveau et l'accès au matériel.

❖ **La mémoire Flash**

La **mémoire Flash** est une mémoire **non volatile**, c'est-à-dire qu'elle conserve son contenu même lorsque l'alimentation est coupée.

Rôle principal :

- stockage du **programme (firmware)**,
- stockage des constantes,
- stockage du vecteur d'interruptions.

Caractéristiques :

- accès plus lent que la SRAM,
- capacité généralement élevée (de quelques dizaines de Ko à plusieurs Mo),
- écriture limitée en nombre de cycles.

Utilisation typique :

- le code compilé est chargé en Flash,
- le processeur démarre l'exécution à partir de la Flash après un reset.

Dans les microcontrôleurs **STM32**, la Flash contient aussi la **table des vecteurs d'interruptions**.

❖ **La mémoire SRAM**

La **SRAM (Static Random Access Memory)** est une mémoire **volatile**, rapide, utilisée pour le stockage des données pendant l'exécution du programme.

Rôles principaux :

- stockage des **variables globales et locales**,
- gestion de la **pile (stack)**,
- gestion du **tas (heap)** pour l'allocation dynamique.

Caractéristiques :

- accès très rapide,
- capacité plus faible que la Flash,
- contenu perdu à la coupure d'alimentation.

La pile (stack), pointée par le registre **SP**, est située en SRAM et joue un rôle clé dans :

- les appels de fonctions,
- les interruptions,
- le multitâche.

❖ **Les périphériques mappés en mémoire**

Dans l'architecture ARM Cortex-M, les **périphériques sont mappés en mémoire**, ce qui signifie que :

- chaque périphérique possède une **adresse mémoire spécifique**,
- ses registres de contrôle sont accessibles par des lectures/écritures mémoire.

Exemples de périphériques mappés mémoire :

- GPIO (entrées/sorties),
- timers,
- UART, SPI, I²C,
- ADC, DAC.

Avantages :

- accès simple via des pointeurs,
- unification du modèle de programmation,
- facilité de débogage.

Par exemple, écrire dans un registre GPIO revient à écrire dans une adresse mémoire donnée.

❖ **Architecture Harvard modifiée**

Les processeurs ARM Cortex-M utilisent une **architecture Harvard modifiée** :

➢ séparation logique entre :
  - ✓ mémoire programme (Flash),
  - ✓ mémoire données (SRAM),

- possibilité d'accès simultané pour améliorer les performances,
- partage partiel des bus pour réduire le coût matériel.

Cette architecture constitue un **compromis efficace** entre performance et simplicité.

L'organisation mémoire des processeurs ARM Cortex-M repose sur :

- un **adressage linéaire 32 bits**,
- une séparation claire entre Flash et SRAM,
- une intégration directe des périphériques dans l'espace mémoire,
- une architecture adaptée aux **applications temps réel**.

Cette organisation facilite :

- la programmation bas niveau,
- l'accès direct au matériel,
- la compréhension du fonctionnement interne des microcontrôleurs **STM32**.

1.6 Le pipeline

Le pipeline est une technique d'organisation interne du processeur qui permet d'améliorer les performances en découpant l'exécution d'une instruction en plusieurs étapes successives. Au lieu de traiter une instruction entièrement avant de passer à la suivante, le processeur peut superposer plusieurs phases d'exécution, ce qui augmente le débit global. Dans une architecture simple, ces étapes comprennent généralement le fetch, qui correspond à la lecture de l'instruction en mémoire, le decode, qui consiste à interpréter cette instruction, et l'execute, qui réalise l'opération demandée. Ainsi, pendant qu'une instruction est exécutée, la suivante peut déjà être lue, et une troisième peut être en cours de décodage. Cette organisation rend le processeur plus efficace et mieux adapté aux exigences des systèmes embarqués. Les Cortex-M3 et Cortex-M4 utilisent un pipeline à trois étages, tandis que les Cortex-A peuvent intégrer des pipelines beaucoup plus profonds, allant jusqu'à huit ou douze étages selon l'architecture. Plus le pipeline est profond, plus le débit peut être élevé, mais plus la gestion des sauts et des aléas devient complexe.

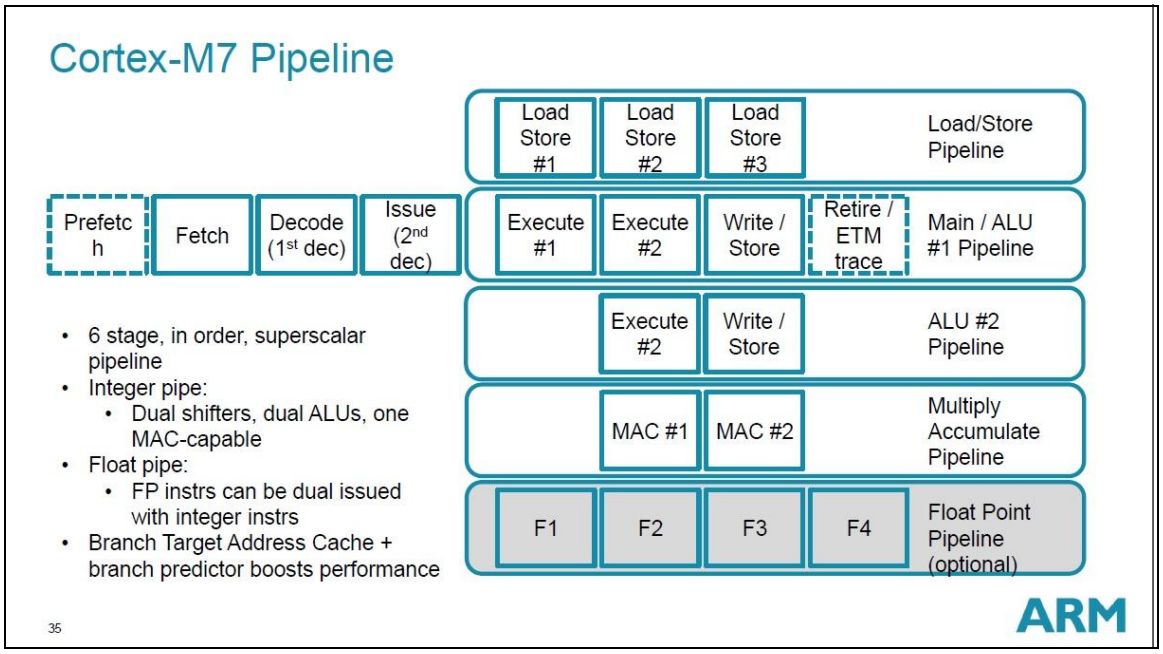

Figure 1.6 : Le pipeline des processeurs ARM Cortex

Donc pour résumer

Le pipeline permet l'exécution parallèle des instructions.
Exemple pipeline 3 étages (Cortex-M3/M4) :

1. **Fetch**
2. **Decode**
3. **Execute**

Le **pipeline** est une technique d'architecture des processeurs qui permet d'augmenter les **performances** en divisant l'exécution d'une instruction en plusieurs **étapes successives**, exécutées de manière **chevauchée**.

Ainsi, pendant qu'une instruction est en cours d'exécution, d'autres instructions peuvent être **lues** ou **décodées** en parallèle.

Dans les processeurs **ARM Cortex-M**, le pipeline est conçu pour offrir un **bon compromis entre simplicité, rapidité et faible consommation**, ce qui est essentiel pour les systèmes embarqués temps réel.

❖ **Principe général du pipeline**

Sans pipeline :

- une instruction doit être entièrement exécutée avant que la suivante ne commence,
- le processeur reste inactif pendant certaines phases.

Avec pipeline :

- chaque instruction est découpée en étapes,
- plusieurs instructions sont traitées simultanément,
- le débit d'exécution est fortement amélioré.

Le pipeline permet d'augmenter le **nombre d'instructions exécutées par unité de temps** sans augmenter la fréquence d'horloge.

❖ **Pipeline à 3 étages dans les Cortex-M3 / Cortex-M4**

Les processeurs **Cortex-M3** et **Cortex-M4** utilisent généralement un **pipeline à trois étages** :

❖ **Fetch (Lecture de l'instruction)**

- Le processeur lit l'instruction depuis la **mémoire Flash**.
- L'adresse de l'instruction est fournie par le **PC (Program Counter)**.
- L'instruction est placée dans un registre interne.

Pendant cette étape, le PC est automatiquement mis à jour pour pointer vers l'instruction suivante.

➢ **Decode (Décodage de l'instruction)**

- L'instruction lue est analysée par le **décodeur**.
- Le processeur identifie :
  - le type d'instruction,
  - les registres source et destination,
  - l'opération à effectuer.
- Les signaux de contrôle internes sont générés.

Cette étape prépare le matériel à exécuter correctement l'instruction.

➢ **Execute (Exécution)**

- L'opération est réalisée :
  - ✓ calcul arithmétique ou logique (ALU),
  - ✓ accès mémoire (Load / Store),
  - ✓ branchement ou saut.
- Les résultats sont écrits dans les registres ou en mémoire.
- Les indicateurs du registre **xPSR** peuvent être mis à jour.

À la fin de cette étape, l'instruction est considérée comme terminée.

❖ **Fonctionnement parallèle du pipeline**

Le fonctionnement du pipeline peut être illustré par un chevauchement des instructions :

Cycle	Instruction 1	Instruction 2	Instruction 3
1	Fetch	–	–
2	Decode	Fetch	–
3	Execute	Decode	Fetch
4	–	Execute	Decode
5	–	–	Execute

Après le remplissage du pipeline, le processeur peut idéalement **terminer une instruction par cycle d'horloge**.

❖ **Avantages du pipeline**

Le pipeline offre plusieurs avantages majeurs :

- augmentation du **débit d'instructions**,
- meilleure utilisation des ressources internes,
- amélioration des performances sans augmentation de la fréquence,
- réduction de la consommation énergétique par instruction.

Ces avantages sont essentiels pour les **applications embarquées temps réel**.

❖ **Limites et perturbations du pipeline**

Malgré ses avantages, le pipeline peut être perturbé dans certains cas :

1. **Branchements et sauts**
   - ✓ un changement du PC peut invalider les instructions déjà chargées,
   - ✓ le pipeline doit être vidé (*pipeline flush*).
2. **Interruptions**
   - ✓ lors d'une interruption, le pipeline est interrompu,
   - ✓ le contexte est sauvegardé avant l'exécution de la routine d'interruption.
3. **Accès mémoire lents**
   - ✓ certains accès à la Flash peuvent nécessiter plusieurs cycles,
   - ✓ cela peut ralentir temporairement le pipeline.

Les Cortex-M intègrent des mécanismes matériels pour limiter ces effets.

❖ **Pipeline et temps réel**

Dans les systèmes temps réel :

- le comportement du pipeline doit être **prévisible**,
- la latence des interruptions doit être minimale.

Les processeurs ARM Cortex-M sont conçus pour garantir :

- une **latence d'interruption faible et déterministe**,
- une reprise rapide de l'exécution normale.

Le pipeline dans les processeurs ARM Cortex-M :

- découpe l'exécution en **Fetch, Decode et Execute**,
- permet une **exécution parallèle des instructions**,
- améliore significativement les performances,
- reste simple et adapté aux contraintes temps réel.

La compréhension du pipeline est essentielle pour :

- analyser les performances,
- comprendre les latences,
- optimiser les programmes embarqués.

1.7 Interruptions et exceptions

Les interruptions et les exceptions constituent des mécanismes essentiels pour permettre au processeur de réagir rapidement aux événements internes ou externes. Une interruption est généralement déclenchée par un événement externe, comme l'appui sur un bouton, la réception d'une donnée sur une interface UART ou le débordement d'un timer. Elle interrompt momentanément l'exécution normale du programme afin de traiter l'événement avec priorité. Les exceptions, quant à elles, sont liées à des événements internes au processeur, comme une erreur de type HardFault, un défaut d'accès mémoire de type MemManage fault ou une erreur d'exécution de type Usage fault. Dans l'architecture ARM Cortex-M, la gestion de ces événements est assurée par le NVIC, un contrôleur dédié aux interruptions imbriquées et vectorisées. Ce mécanisme permet d'attribuer des priorités différentes aux interruptions, de gérer plusieurs requêtes simultanément et de garantir un temps de réponse très court. Cette capacité est particulièrement importante dans les applications temps réel, où la rapidité de réaction du système est un critère fondamental.

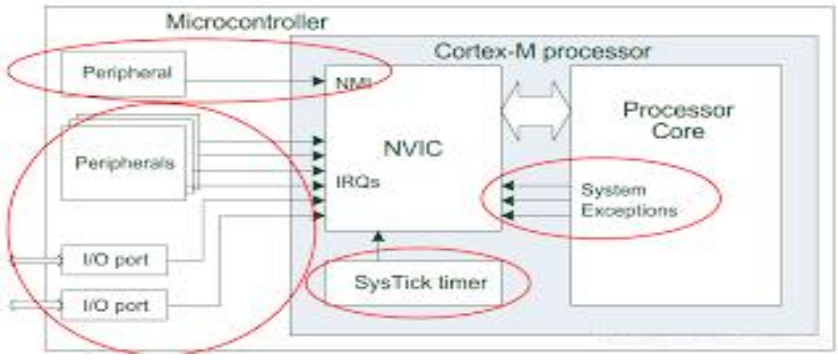

Figure 1.7: core interrupt/exception system AARM Cortex-M

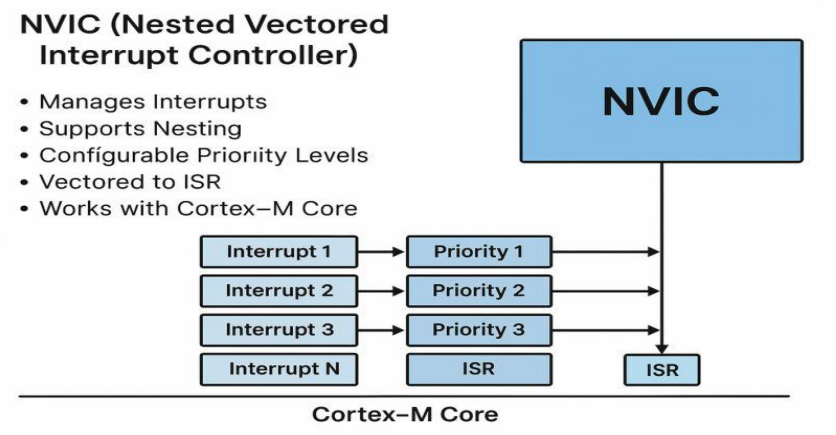

Figure 1.8 : contrôleur dédié aux interruptions imbriquées et vectorielles.

Donc pour résumer :

❖ **Interruptions et exceptions (NVIC)**

Les processeurs **ARM Cortex-M** intègrent un contrôleur matériel d'interruptions appelé **NVIC** (*Nested Vectored Interrupt Controller*).

Le NVIC est un composant clé de l'architecture, spécialement conçu pour les **applications temps réel**, où la rapidité et la hiérarchisation des événements sont essentielles.

➢ **Rôle du NVIC**

Le NVIC assure :

- la **détection des interruptions** provenant des périphériques,
- la **priorisation** des interruptions,
- la **préemption** (interruption d'une interruption),
- la transmission rapide du contrôle au processeur.

Il est directement intégré au cœur du processeur, ce qui garantit une **faible latence**.

➢ **Interruptions et exceptions**

- **Exception** : événement interne au processeur (reset, fault, division par zéro, etc.)
- **Interruption** : événement externe généré par un périphérique (timer, GPIO, UART, ADC…)

Toutes les exceptions et interruptions sont gérées de manière uniforme par le NVIC.

➢ **Priorités et préemption**

Chaque interruption possède :

- un **niveau de priorité programmable**,
- la possibilité de **préempter** une interruption de priorité inférieure.

Cela permet de garantir qu'un événement critique sera toujours traité en premier.

Exemple :

- Une interruption Timer (priorité élevée) peut interrompre une communication UART (priorité faible).

➢ **Faible latence**

Lorsqu'une interruption survient :

- le contexte du processeur est **sauvegardé automatiquement**,
- le PC est redirigé vers la routine d'interruption,
- l'exécution reprend rapidement après la fin de l'interruption.

Cette gestion matérielle assure un **temps de réponse très court**, indispensable aux systèmes temps réel.

1.8 Jeu d'instructions ARM (voir Annex)

Le jeu d'instructions ARM définit l'ensemble des opérations que le processeur est capable d'exécuter directement. Deux grands formats sont traditionnellement associés à l'architecture ARM : le format ARM en 32 bits et le format Thumb ou Thumb-2, qui combine des instructions en 16 et 32 bits. Dans les Cortex-M, le jeu Thumb-2 est privilégié parce qu'il offre un bon compromis entre compacité du code et efficacité d'exécution.

Cela permet de réduire la taille du programme en mémoire tout en conservant de bonnes performances. Parmi les instructions les plus courantes, on trouve **MOV** pour le transfert de données, **ADD et SUB** pour les opérations arithmétiques, **LDR et STR** pour les accès mémoire, ainsi que **B et BL** pour les branchements et les appels de sous-programmes. Ce jeu d'instructions est largement utilisé dans les systèmes embarqués et les applications temps réel, car il offre un contrôle précis du matériel et une grande efficacité dans la gestion des ressources.

Principales caractéristiques du jeu d'instructions Thumb-2 (ARMv7-M)

❖ **Performance et Taille :** Combine la compacité des instructions Thumb 16 bits avec la puissance des instructions ARM 32 bits.

❖ **Architecture Load-Store :** Les opérations de traitement de données s'effectuent uniquement entre les registres. Les instructions de chargement (LDR) et de stockage (STR) sont nécessaires pour accéder à la mémoire.

❖ **Registres :** Le Cortex-M3 possède 13 registres de données à usage général (R0-R12) de 32 bits, un pointeur de pile (SP, R13), un registre de lien (LR, R14) et un compteur de programme (PC, R15).

➢ **Catégories d'instructions clés :**

Ce qu'il faut savoir : Le Cortex-M3 est une architecture **Load/Store**. Cela signifie qu'on ne peut pas faire d'opération arithmétique directement sur la mémoire ; il faut d'abord charger la valeur dans un registre, calculer, puis renvoyer le résultat en mémoire.

- **Transfert de données :** LDR (load), STR (store), MOV (move).
- **Arithmétique et Logique :** ADD, SUB, MUL, AND, ORR, EOR.
- **Branchements (Contrôle de flux) :** B (branchement), BL (branchement avec lien pour les fonctions), BX (branchement et échange d'état).
- **Gestion de la pile :** PUSH, POP.

Liste des suffixes “condition”possibles

Suffixe	Condition	Fanions
EQ	Égalité (EQual)	Z = 1
NE	Non égalité (Non Equal)	Z = 0
CS **HS**	Dépassement de capacité (Carry Set) Plus grand ou égal (non signé) (Higher or Same)	C = 1
CC **LO**	Pas de dépassement de capacité (Carry Clear) Plus petit (non signé) (Lower)	C = 0
MI	Négatif (Minus)	N = 1
PL	Positif (PLus)	N = 0
VS	Dépassement (signé) (oVerflow Set)	V = 1
VC	Pas de dépassement (signé) (signé)(oVerflow Clear)	V = 0
HI	Plus grand (non signé) (Unsigned HIgher)	C = 1 ET Z = 0
LS	Plus petit ou égal (non signé) (Unsigned Lower or Same)	C = 0 OU Z = 1
GE	Plus grand ou égal(signé) (Signed Greater than or Equal)	N = V
LT	Plus petit (signé) (Signed Less Than)	N ≠ V
GT	Plus grand (signé) (Signed Greater Than)	Z = 0 ET N = V
LE	Plus petit ou égal(signé) (Signed Less than or Equal)	Z = 1 OU N ≠ V

CMP a,b ... B??		
	a,b signés	a,b non signés
=	EQ	EQ
<	MI	LO
≤	LE	LS
>	GT	HI
≥	GE	HS

Grandes catégories d'instructions à retenir

1. Transfert de données

- **LDR / STR** : Charger/Stocker un mot (32 bits) depuis ou vers la mémoire.

- **LDRB / STRB** : Transfert d'un octet (8 bits).
- **LDM / STM** : Transfert multiple (très utile pour sauvegarder les registres sur la pile).
- **PUSH / POP** : Gestion de la pile (Stack).

2. Opérations Arithmétiques et Logiques

- **ADD / SUB** : Addition et soustraction.
- **MUL / UDIV / SDIV** : Multiplication et division (le Cortex-M3 gère la division matérielle).
- **AND / ORR / EOR / BIC** : Opérations logiques (ET, OU, OU exclusif, Bit Clear).
- **LSL / LSR** : Décalages logiques à gauche ou à droite.

3. Contrôle de flux (Branchements)

- **B** : Branchement inconditionnel.
- **BL** : Branchement avec lien (appel de fonction, sauve l'adresse de retour dans LR).
- **BX / BLX** : Branchement vers une adresse contenue dans un registre.
- **CBZ / CBNZ** : Branchement si le registre est à zéro (ou non-zéro), idéal pour les boucles courtes.

4. Instructions Spécifiques (Thumb-2)

- **IT (If-Then)** : Permet de rendre conditionnelles jusqu'à 4 instructions suivantes (ex: "exécute ces 3 lignes seulement si le résultat précédent est négatif").
    - **REV** : Inverse l'ordre des octets (utile pour le passage de Big-Endian à Little-Endian).
- **BITFIELD (BFI, UBFX)** : Manipulation directe de groupes de bits à l'intérieur d'un registre.

5. Accès au Système

- **MRS / MSR** : Transférer des données entre un registre général et un registre spécial (comme PSR, CONTROL ou MSP).
- **SVC** : Appel de supervision (interruption logicielle pour solliciter l'OS).

Les instructions de transfert interne

ADR	16/32	Chargement d'adresse CODE
ADR<c> <Rd>, <label>		Rd ← adresse du label
MOV	16/32	Transfert interne de registre
MOV{S}<c> <Rd>, #<const>		Rd ← const
MOV{S}<c> <Rd>, <Rm>		Rd ← Rm
MOVT	32	Affectation des 16 bits de pf d'un registre
MOVT<c> <Rd>, #<imm16>		Rd[16 : 31] ← imm16
MRS	32	Lecture d'un registre spécial
MSR<c> <Rn>,<spec_reg>		Rn ← spec_reg
MSR	32	Ecriture sur un registre spécial

Les instructions arithmétiques

NOP	16/32	**Pas d'opération**
NOP<c>		ne fait rien
ADC	16/32	**Addition avec Carry**
ADC{S}<c> {<Rd>,} <Rn>, #<const>		Rd ← Rn + const + fanion C
ADC{S}<c> {<Rd>,} <Rn>, <Rm> {,<shift>}		Rd ← Rn +shift(Rm) + fanion C
ADD	16/32	**Addition simple**
ADD{S}<c> {<Rd>,} <Rn>,#<const>		Rd ← Rn + const
ADD{S}<c> {<Rd>,} <Rn>, <Rm> {,<shift>}		Rd ← Rn + shift(Rm)
ADD{S}<c> {<Rd>,} SP, #<const>		Rd ← SP + const
ADD{S}<c> {<Rd>,} SP, <Rm>{,<shift>}		Rd ← SP + shift(Rm)
MLA	32	**Multiplication et addition**
MLA<c> <Rd>, <Rn>, <Rm>, <Ra>		Rd ← (Rn * Rm) + Ra
MLS	32	**Multiplication et soustraction**
MUL	16/32	**Multiplication - Résultats sur 32 bits**
MUL{S}<c> {<Rd>,} <Rn>, <Rm>		Rd ← Rn * Rm
RSB	16/32	**Soustraction inversée**
RSB{S}<c> {<Rd>,} <Rn>, #<const>		Rd ← - Rn + const
RSB{S}<c> {<Rd>,} <Rn>, <Rm> {,<shift>}		Rd ← -Rn + shift(Rm)
SBC	16/32	**Soustraction avec Carry**
SBC{S}<c> {<Rd>,} <Rn>, #<const>		Rd ← Rm - const + fanion C
SBC{S}<c> {<Rd>,} <Rn>, <Rm> {,<shift>}		Rd ← Rm - shift(Rm) + fanion C
SDIV	32	**Division signée**
SDIV<c> {<Rd>,} <Rn>, <Rm>		Rd ← Rn ÷ Rn
SMLAL	32	**Multiplication signée et addition 64 bits**
SMLAL<c> <Rd_{pf}>, <Rd_{PF}>, <Rn>, <Rm>		[Rd_{PF} : Rd_{pf}] ← Rn * Rm +[Rd_{PF} : Rd_{pf}]
SMULL	32	**Multiplication signée - résultats sur 64 bits**
SMULL<c> <Rd_{pf}>, <Rd_{PF}>, <Rn>, <Rm>		[Rd_{PF} : Rd_{pf}] ← Rn * Rm
SSAT	32	**Saturation signée**
SSAT<c> <Rd>,#<imm5>,<Rn>{,<shift>}		si (Rn < 0) Rd ← $\min(-2^{(imm5-1)}$,shift(Rn)) si (Rn >0) Rd ← $\max(2^{(imm5-1)} - 1$,shift(Rn))

SUB	16/32	Soustraction simple
SUB{S}<c> {<Rd>} ,<Rn>,#<const>		Rd ← Rn - const
SUB{S}<c> {<Rd>} ,<Rn>, <Rm> {,<shift>}		Rd ← Rn - shift(Rm)
SUB{S}<c> {<Rd>} ,SP,#<const>		Rd ← SP - const
SUB{S}<c> {<Rd>} ,SP,<Rm> {,<shift>}		Rd ← SP - shift(Rm)
UDIV	32	Division non signée
UDIV<c> {<Rd>,} <Rn>, <Rm>		Rd ← Rn ÷ Rm
UMLAL	32	Multiplication non signée et addition 64 bits
UMLAL<c> <Rd_{pf}>, <Rd_{PF}>, <Rn>, <Rm>		[Rd_{PF} : Rd_{pf}] ← Rn * Rm + [Rd_{PF} : Rd_{pf}]
UMULL	32	Multiplication non signée - Résultats sur 64 bits
UMUL<c> <Rd_{pf}>, <Rd_{PF}>, <Rn>, <Rm>		[Rd_{PF} : Rd_{pf}] ← Rn * Rm
USAT	32	Saturation non signée
USAT<c> <Rd>,#<imm5>,<Rn>{,<shift>}		Rd ← $\max(2^{(imm5-1)} - 1 , \text{shift(Rn)})$

Les instructions logique et de manipulation de bits

AND	16/32	ET logique
AND{S}<c> {<Rd>,} <Rn>, #<const>		Rd ← Rn AND const
AND{S}<c> {<Rd>,} <Rn>, <Rm> {,<shift>}		Rn ← Rn AND shift(Rm)
ASR	16/32	Décalage arithmétique à droite
ASR{S}<c> <Rd>, <Rm>, #<imm5>		Rd ← Rm >>$_{imm5}$
ASR{S}<c> <Rd>, <Rn>, <Rm>		Rd ← Rn >>$_{Rm}$
BFC	32	Effacement de champs de bits

BFC<c> <Rd>, #<pf>, #<Nb>		Rd[pf+Nb-1 : pf] ← 0
BFI	32	Recopie de champs de bits
BFI<c> <Rd>,<Rn>, #<pf>, #<Nb>		Rd[pf+Nb-1 : pf] ← Rn[Nb : 0]
BIC	16/32	Effacement de bits par masque ET
BIC{S}<c> {<Rd>,} <Rn>, #<const>		Rd ← Rn AND $\overline{\text{(const)}}$
BIC{S}<c> {<Rd>,} <Rn>, <Rm> {,<shift>}		Rd ← Rn AND $\overline{\text{(shift(Rm))}}$
CLZ	32	Dénombre les bits de PF à 0 devant le premier bit à 1
CLZ<c> <Rd>, <Rm>		Rd ← CLZ(Rn)
EOR	16/32	OU Exclusif
EOR{S}<c> {<Rd>,} <Rn>, #<const>		Rd ← Rn XOR const
EOR{S}<c> {<Rd>,} <Rn>, <Rm> {,<shift>}		Rd ← Rn XOR shift(Rm)

LSL	16/32	Décalage logique à gauche
LSL{S}<c> <Rd>, <Rm>, #<imm5>		Rd ← Rm << imm5
LSL{S}<c> <Rd>, <Rn>, <Rm>		Rd ← Rn << Rm
LSR	16/32	Décalage logique à droite
LSR{S}<c> <Rd>, <Rm>, #<imm5>		Rd ← Rm >> imm5
LSR{S}<c> <Rd>, <Rn>, <Rm>		Rd ← Rn >> Rm
MVN	16/32	Complément à 1 logique
MVN{S}<c> <Rd>, #<const>		Rd ← NOT(const)
MVN{S}<c> <Rd>, <Rm> {, <shift>}		Rd ← NOT(shift(Rn))
NEG	16/32	Complément à 2
NEG<c> {<Rd>,} <Rm>		Rd ← -Rm
ORN	16/32	OU logique complémenté
ORN{S}<c> {<Rd>,} <Rn>, #<const>		Rd ← Rm OU NOT(const)
ORN{S}<c> {<Rd>,} <Rn>, <Rm> {,<shift>}		Rd ← Rm OU NOT(shift(Rm))
ORR	16/32	OU logique
ORR{S}<c> {<Rd>,} <Rn>, #<const>		Rd ← Rm OR const
ORR{S}<c> {<Rd>,} <Rn>, <Rm> {,<shift>}		Rd ← Rm OR shift(Rm)
RBIT	32	Transposition de bits
RBIT<c> <Rd>, <Rm>		Rd[31-k] ← Rm[k] avec k = 0 ... 31
REV	16/32	Inversion octets PF et pf
REV<c> <Rd>, <Rm>		Rd[31 : 24] ← Rm[7 : 0] Rd[23 : 16] ← Rm[15 : 8] Rd[15 : 8] ← Rm[23 : 16] Rd[7 : 0] ← Rm[31 : 24]
REV16	16/32	Inversion octet PF et pf par $\frac{1}{2}$ mot
REV16<c> <Rd>, <Rm>		Rd[31 : 24] ← Rm[23 : 16] Rd[23 : 16] ← Rm[31 : 24] Rd[15 : 8] ← Rm[7 : 0] Rd[7 : 0] ← Rm[15 : 8]
REVSH	16/32	Inversion signée d'un $\frac{1}{2}$ mot
REVSH<c> <Rd>, <Rm>		Rd[31 : 8] ← Promotion signée (Rm[7 : 0]) Rd[7 : 0] ← Rm[15 : 8]
ROR	16/32	Rotation vers la droite
ROR{S}<c> <Rd>, <Rm>, #<imm5>		Rd ← rotation(Rm, imm5 bits)
ROR{S}<c> <Rd>, <Rn>, <Rm>		Rd ← rotation(Rn, Rm bits)

RRX	32	**Rotation étendue vers la droite**
ROR{S}<c> <Rd>, <Rn>, <Rm>		Rd ← rotation([Rn,C], Rm bits
SBFX	32	**Promotion signée sur 32 bits d'un champs de bits**
SBFX<c> <Rd>, <Rn>, #<pf>, #<Nb>		Rd[Nb-1 : 0] ← Rn[pf+Nb-1 : pf] Rd[31 : Nb] ← Rd[pf+Nb-1]
SXTB	16/32	**Promotion signée sur 32 bits d'un octet**
SXTB<c> <Rd>, <Rm> {, <rotation>}		Rd ← rotation_{32}(Rn)[7 : 0] Rd[31 : 8] ← Rd[7]
SXTH	16/32	**Promotion signée sur 32 bits d'un $\frac{1}{2}$ mot**
SXTH<c> <Rd>, <Rm> {, <rotation>}		Rd ← rotation_{32}(Rn)[15 : 0] Rd[31 : 8] ← Rd[15]
UBFX	16/32	**Promotion non signée sur 32 bits d'un champs de bits**
UBFX<c> <Rd>, <Rn>, #<pf>, #<Nb>		Rd[Nb-1 : 0] ← Rn[pf+Nb-1 : pf] Rd[31 : Nb] ← 0
UXTB	16/32	**Promotion non signée sur 32 bits d'un octet**
UXTB<c> <Rd>, <Rm> {, <rotation>}		Rd ← rotation_{32}((Rn)[7 : 0] Rd[31 : 8] ← 0
UXTH	16/32	**Promotion non signée sur 32 bits d'un $\frac{1}{2}$ mot**
UXTH<c> <Rd>, <Rm> {, <rotation>}		Rd ← rotation_{32}((Rn)[15 : 0] Rd[31 : 8] ← 0

Les instructions de transfert interne

ADR	16/32	**Chargement d'adresse CODE**
ADR<c> <Rd>, <label>		Rd ← adresse du label
MOV	16/32	**Transfert interne de registre**
MOV{S}<c> <Rd>, #<const>		Rd ← const
MOV{S}<c> <Rd>, <Rm>		Rd ← Rm
MOVT	32	**Affectation des 16 bits de pf d'un registre**
MOVT<c> <Rd>, #<imm16>		Rd[16 : 31] ← imm16
MRS	32	**Lecture d'un registre spécial**
MSR<c> <Rn>,<spec_reg>		Rn ← spec_reg
MSR	32	**Ecriture sur un registre spécial**

Les instructions de test

CMN	16/32	Addition sans affectation - Modification des fanions
CMN<c> <Rn>, #<const>		Fanions ← test(Rn + const)
CMN<c> <Rn>, <Rm>{,<shift>}		Fanions ← test(Rn + shift(Rm))
CMP	16/32	Soustraction sans affectation - Modification des fanions
CMP<c> <Rn>, #<const>		Fanions ← test(Rn - const)
CMP<c> <Rn>, <Rm>{,<shift>}		Fanions ← test(Rn - shift(Rm))
TEQ	32	OU exclusif sans affectation - Modification des fanions
TEQ<c> <Rn>, #<const>		Fanions ← test(Rn XOR const)
TEQ<c> <Rn>, <Rm>{,<shift>}		Fanions ← test(Rn XOR shift(Rm))
TST	16/32	ET logique sans affectation - Modification des fanions
TST<c> <Rn>, #<const>		Fanions ← test(Rn AND const)
TST<c> <Rn>, <Rm>{,<shift>}		Fanions ← test(Rn AND shift(Rm))

1 Les instructions de saut

B	16/32	Branchement simple
B<c> <label>		PC ← label
BL	32	Branchement avec lien
BL<c> <label>		LR ← @ de retour PC ← label
BLX	16	Branchement avec lien par registre
BLX<c> <Rm>		LR ← @ de retour PC ← Rm
BX	16	Branchement par registre
BX<c> <Rm>		PC ← Rm
CBZ, CBNZ	16	Branchement conditionné sur la nullité d'un registre
CBZ<c> <Rm> <label>		PC ← label si (Rm = 0)
CBNZ<c> <Rm> <label>		PC ← label si (Rm ≠ 0)
IT	16	Condition et saut de type si...alors
IT{x{y{z}}} <firstcond>		Fixe l'exécution du bloc d'instructions suivantes (max 4)
TBB, TBH	32	Table de saut relatif
TBB<c> [<Rn>, <Rm>] TBH<c> [<Rn>, <Rm>, LSL #1]		Pc ← PC + Rn[Rm] PC + Rn[Rm]

2 Les instructions de load/store

LDR et STR sont exprimées ici en version 32 bits mais se déclinent en 8 ou 16 bit en ajoutant 'B' ou 'H' au mnémonique.

LDR	Chargement d'un registre avec un mot mémoire	
LDR<c> <Rt>, [<Rn> {, #±<imm>}]		Rt←$\mathcal{M}_{32}$(Rn±imm)
LDR<c> <Rt>, [<Rn>, #±<imm>]!		Rn←Rn+imm puis Rt← $\mathcal{M}_{32}$(Rn)
LDR<c> <Rt>, [<Rn>], #±<imm>		Rt←$\mathcal{M}_{32}$(Rn) puis Rn←Rn+imm
LDR<c> <Rt>, <label>		Rt ← label
LDR<c> <Rt>, [PC, #±<imm>]		Rt←$\mathcal{M}_{32}$(Pc±imm)
LDR<c> <Rt>, [<Rn>, <Rm> {LSL,#<shift>}]		Rt←$\mathcal{M}_{32}$(Rn+shift(Rm))
STR	**Déchargement d'un registre vers un mot mémoire**	
STR<c> <Rt>, [<Rn> {,#±<imm>}]		$\mathcal{M}_{32}$(Rn±imm)←Rt
STR<c> <Rt>, [<Rn>, #±<imm>]!		Rn←Rn+imm puis $\mathcal{M}_{32}$(Rn)←Rt
STR<c> <Rt>, [<Rn>], #±<imm>		$\mathcal{M}_{32}$(Rn)←Rt puis Rn←Rn+imm
STR<c> <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]		$\mathcal{M}_{32}$(Rn+shift(Rm))←Rt
LDM	16/32	**Chargement multiple à partir d'adresses ascendantes**
LDM<c> <R_k>,{R_i-R_j}		$R_k \leftarrow \mathcal{M}_{32}(Rn + 4*(k-i))$ avec $k = i...j$
LDM<c> <R_k>!,{R_i-R_j}		$R_k \leftarrow \mathcal{M}_{32}(Rn + 4*(k-i))$ avec $k = i...j$ puis Rn← Rn + $4*(j-i)$
LDMDB	32	**Chargement multiple à partir d'adresses descendantes**
LDMDB<c> <R_k>,{R_i-R_j}		$R_k \leftarrow \mathcal{M}_{32}(Rn - 4*(k-i+1))$ avec $k = i...j$
LDMDB<c> <R_k>!,{R_i-R_j}		$R_k \leftarrow \mathcal{M}_{32}(Rn - 4*(k-i+1))$ avec $k = i...j$
		puis Rn← Rn - $4*(j-i)$
LDRD	32	**Chargement double**
LDRD<c> <Rt>, <Rt2>, <litteral>		Rt← $\mathcal{M}_{32}$(litteral) Rt2← $\mathcal{M}_{32}$(litteral+4)
LDRD<c> <Rt>, <Rt2>, [PC, #±<imm>]		Rt← $\mathcal{M}_{32}$(PC+imm) Rt2 ←$\mathcal{M}_{32}$(PC+imm+4)
LDRD<c> <Rt>,<Rt2>,[<Rn>{,#±<imm>}]		Rt← $\mathcal{M}_{32}$(Rn+imm) Rt2← $\mathcal{M}_{32}$(Rn+imm+4)
LDRD<c> <Rt>,<Rt2>,[<Rn>,#±<imm>]!		Rn←Rn+imm puis Rt← $\mathcal{M}_{32}$(Rn+imm) et Rt2← $\mathcal{M}_{32}$(Rn+imm+4)
LDRD<c> <Rt>,<Rt2>,[<Rn>],#±<imm>		Rt← $\mathcal{M}_{32}$(Rn+imm) Rt2← $\mathcal{M}_{32}$(Rn+imm+4) puis Rn←Rn+imm

STM	16/32	Déchargement multiple vers des adresses ascendantes
STM<c> <R_k>,{R_i-R_j}		$\mathcal{M}_{32}(\text{Rn} + 4 * (k-i)) \leftarrow R_k$ avec $k = i...j$
STM<c> <R_k>!,{R_i-R_j}		$\mathcal{M}_{32}(\text{Rn} + 4 * (k-i)) \leftarrow R_k$ avec $k = i...j$ puis Rn$\leftarrow$ Rn $+ 4 * (j-i)$
STMDB	32	**Déchargement multiple vers des adresses descendantes**
STMDB<c> <R_k>,{R_i-R_j}		$\mathcal{M}_{32}(\text{Rn} - 4 * (k-i+1)) \leftarrow R_k$ avec $k = i...j$
STMDB<c> <R_k>!,{R_i-R_j}		$\mathcal{M}_{32}(\text{Rn} - 4 * (k-i+1)) \leftarrow R_k$ avec $k = i...j$ puis Rn$\leftarrow$ Rn - $4 * (j-i)$
STRD	32	**Déchargement double**
STRD<c> <Rt>,<Rt2>,[<Rn>{,#±<imm>}]		$\mathcal{M}_{32}(\text{Rn+imm}) \leftarrow$ Rt $\mathcal{M}_{32}(\text{Rn+imm+4}) \leftarrow$ Rt2
STRD<c> <Rt>,<Rt2>,[<Rn>,#±<imm>]!		Rn$\leftarrow$Rn+imm puis $\mathcal{M}_{32}$(Rn+imm)$\leftarrow$Rt et $\mathcal{M}_{32}$(Rn+imm+4)$\leftarrow$Rt2
STRD<c> <Rt>,<Rt2>,[<Rn>],#±<imm>		$\mathcal{M}_{32}$(Rn+imm)$\leftarrow$Rt $\mathcal{M}_{32}$(Rn+imm+4)$\leftarrow$Rt2 puis Rn$\leftarrow$Rn+imm
POP	16/32	**Déstockage depuis la pile système**
POP<c> {R_i-R_j}		$R_k \leftarrow \mathcal{M}_{32}(\text{SP} + 4 * (k-i))$ avec $k = i...j$ puis SP$\leftarrow$ SP $+ 4 * (j-i)$
PUSH	16/32	**Stockage vers la pile système**
PUSH<c> {R_i-R_j}		$\mathcal{M}_{32}(\text{SP} - 4 * (k-i+1)) \leftarrow R_k$ avec $k = i...j$ puis SP$\leftarrow$ SP - $4 * (j-i)$

1.9 Performances des processeurs ARM

Les performances d'un processeur ARM dépendent de plusieurs paramètres liés à son architecture et à son environnement d'exécution.

La fréquence d'horloge influe directement sur le nombre d'instructions traitées par seconde, tandis que la profondeur du pipeline améliore le débit d'exécution.

La présence d'une mémoire cache, surtout dans les Cortex-A, permet de réduire les temps d'accès à la mémoire et d'augmenter la rapidité globale du système. Les performances dépendent aussi de la qualité de l'optimisation du compilateur et de la manière dont le processeur gère les interruptions et les accès mémoire. Un indicateur souvent utilisé est le DMIPS par MHz, qui donne une idée de l'efficacité relative du cœur processeur.

Les Cortex-M sont généralement optimisés pour la faible consommation et la simplicité, alors que les Cortex-A visent une puissance de calcul plus élevée pour les systèmes complexes. Le choix d'un processeur ARM repose donc sur un équilibre entre performance, consommation, coût et contraintes applicatives.

❖ **Optimisation de la consommation énergétique**

L'architecture ARM Cortex est conçue pour minimiser la consommation grâce à :

- des modes basse consommation,
- une exécution efficace des instructions,
- un nombre réduit de cycles par instruction,
- des mécanismes d'arrêt partiel du cœur (sleep, deep sleep).

Ces caractéristiques rendent les processeurs ARM Cortex particulièrement adaptés aux systèmes autonomes et aux applications sur batterie.

1.10 Introduction aux familles STM32

La famille STM32 regroupe des microcontrôleurs développés par STMicroelectronics et fondés sur des cœurs ARM Cortex-M. Cette famille couvre un large éventail de besoins, depuis les applications simples jusqu'aux systèmes embarqués les plus exigeants. La série STM32F0 repose sur le Cortex-M0 et vise les applications d'entrée de gamme. La série STM32F1, basée sur le Cortex-M3, est très utilisée pour les usages généraux. La série STM32F4 intègre un Cortex-M4, souvent choisi pour les applications intégrant du traitement numérique du signal ou une unité de calcul flottant. La série STM32H7, fondée sur le Cortex-M7, vise les hautes performances, tandis que la série STM32L4 privilégie la basse consommation. Ces microcontrôleurs sont largement utilisés dans la domotique, la robotique, l'industrie et l'Internet des objets, grâce à leur flexibilité, leur coût raisonnable et leur riche écosystème logiciel et matériel.

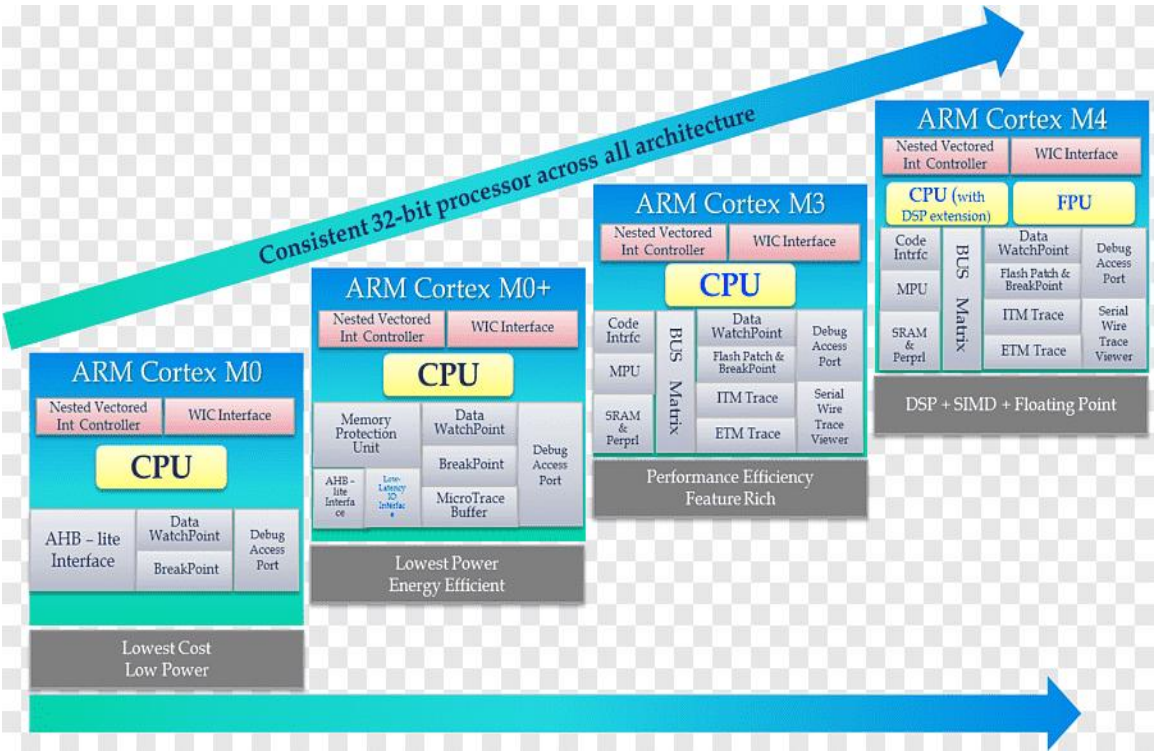

Figure 1.9 : familles STM32.

Les microcontrôleurs **STM32** sont basés sur les cœurs **ARM Cortex-M** et sont développés par la société STMicroelectronics.

Ils sont largement utilisés dans l'industrie, l'enseignement et la recherche.

❖ **Principales familles STM32**

- **STM32F** : usage général, bonnes performances
- **STM32L** : basse consommation
- **STM32G** : nouvelle génération, performances optimisées
- **STM32H** : hautes performances
- **STM32WB** : applications sans fil (Bluetooth, IoT)

Chaque famille est adaptée à un **type d'application spécifique**.

1.11 Environnement de développement STM32CubeIDE

STM32CubeIDE est l'environnement de développement officiel proposé par STMicroelectronics pour la programmation des microcontrôleurs STM32. Il regroupe dans un seul outil la configuration matérielle, la génération de code, la compilation et le débogage. L'intégration de CubeMX permet de configurer graphiquement les périphériques internes comme les GPIO, les timers, les interfaces UART, SPI ou I2C, ainsi que les convertisseurs ADC et DAC. Une fois la configuration terminée, l'outil

génère automatiquement la structure de base du projet en langage C ou C++. STM32CubeIDE facilite également le débogage grâce à la prise en charge de ST-Link, ce qui permet d'observer l'exécution du programme en temps réel, d'inspecter les variables et de suivre les appels de fonctions. Cet environnement constitue ainsi l'outil central pour le développement sur STM32, aussi bien pour l'apprentissage que pour les projets professionnels.

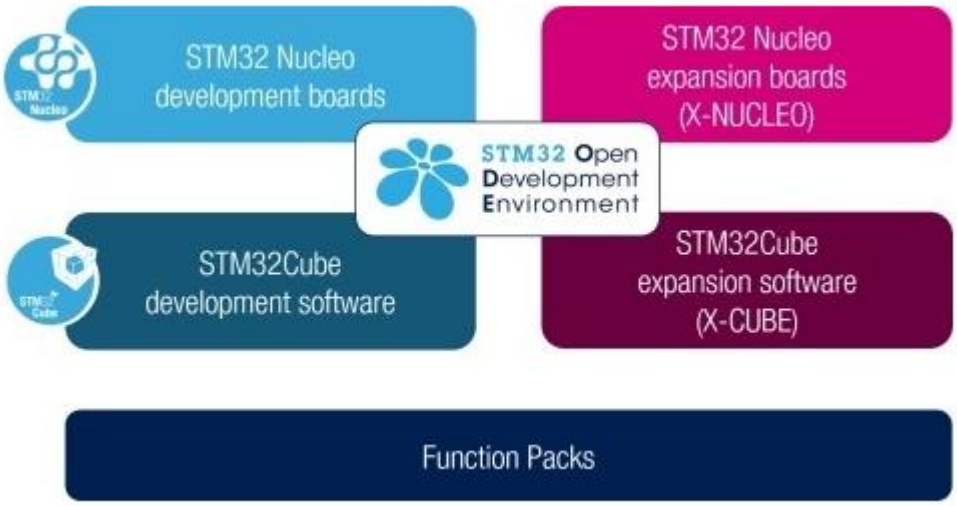

Figure 1.10 : Environnement de développement STM32CubeIDE.

1.12 Carte de développement Nucleo

Les cartes Nucleo sont des cartes de développement conçues pour le prototypage rapide autour des microcontrôleurs STM32. Elles intègrent un microcontrôleur STM32, un programmateur-débogueur ST-Link embarqué, ainsi qu'une connectique compatible avec les formats Arduino Uno ou Morpho selon les modèles. Cette conception facilite grandement la mise en œuvre de premiers essais matériels et logiciels. Les cartes Nucleo sont particulièrement appréciées pour leur faible coût, leur simplicité d'utilisation et leur compatibilité directe avec STM32CubeIDE. Elles constituent un support très adapté à l'apprentissage des bases du développement embarqué, tout en restant suffisamment puissantes pour des projets plus avancés. À titre d'exemple, une carte Nucleo peut être utilisée pour allumer une LED via un port GPIO ou pour lire un capteur externe à travers une interface I2C.

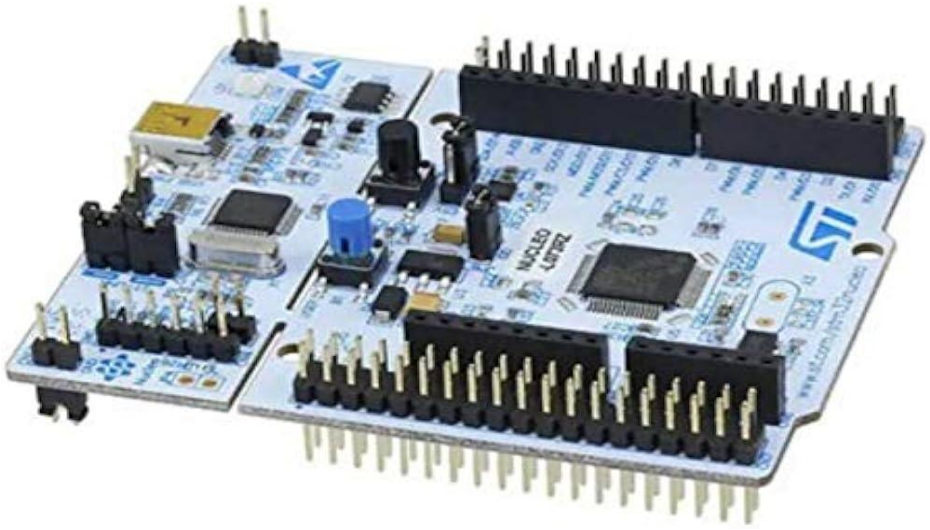

Figure 1.11 : STMicroelectronics STM32 Nucleo-64 MCU Carte de développement ARM Cortex M0+ STM32L073RZT6 .

1.13 Conclusion

Ce premier chapitre a présenté les bases des processeurs ARM Cortex, leur architecture interne, leur mode de fonctionnement ainsi que leur intégration dans les microcontrôleurs STM32. Les notions de pipeline, d'interruptions, de jeu d'instructions, de performances, d'environnement de développement et de carte de prototypage constituent un socle indispensable pour la suite du cours. Les chapitres suivants approfondiront la programmation pratique, l'utilisation des périphériques STM32 et la réalisation de projets embarqués complets.