Memory Access

ARO
5 - Execute & Memory Access

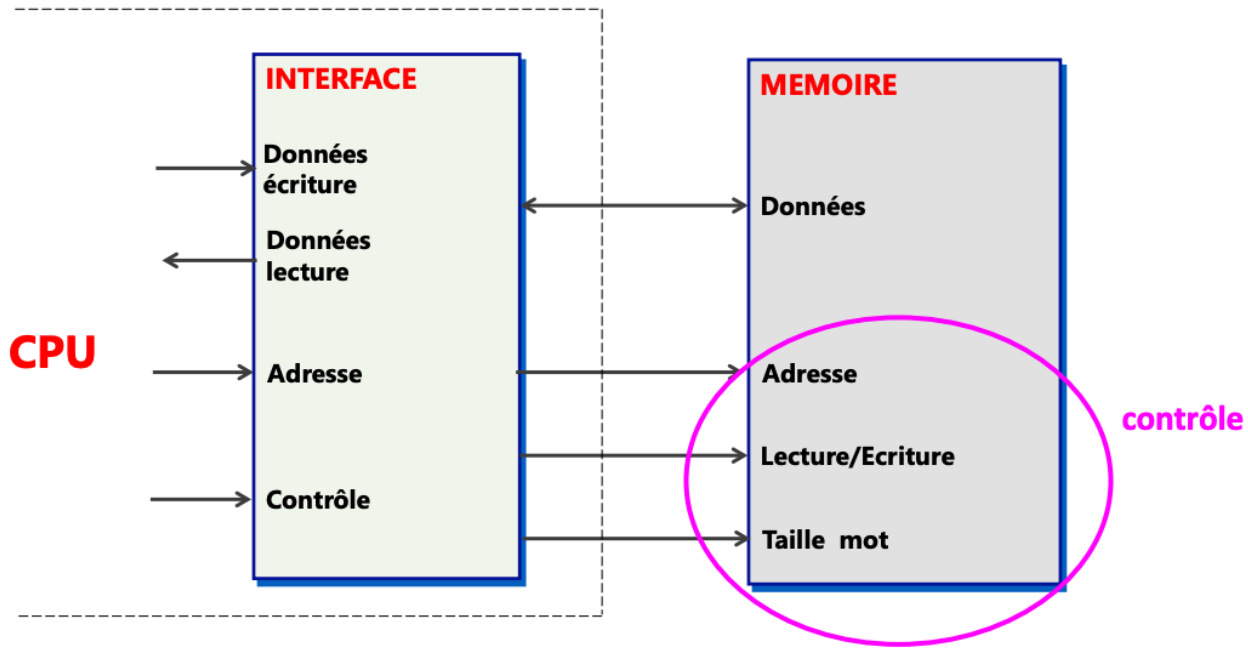
Résumé du document

Le texte aborde l'importance de l'accès à la mémoire dans un processeur pour charger des instructions, sauvegarder les données et garantir un fonctionnement efficace des programmes. Il explique également le concept de memory mapping pour définir les emplacements de la mémoire et des périphériques, ainsi que l'utilisation de la pile pour stocker des données selon le principe Last In First Out (LIFO), telles que les adresses de retour des fonctions et les variables locales.

Table des matières

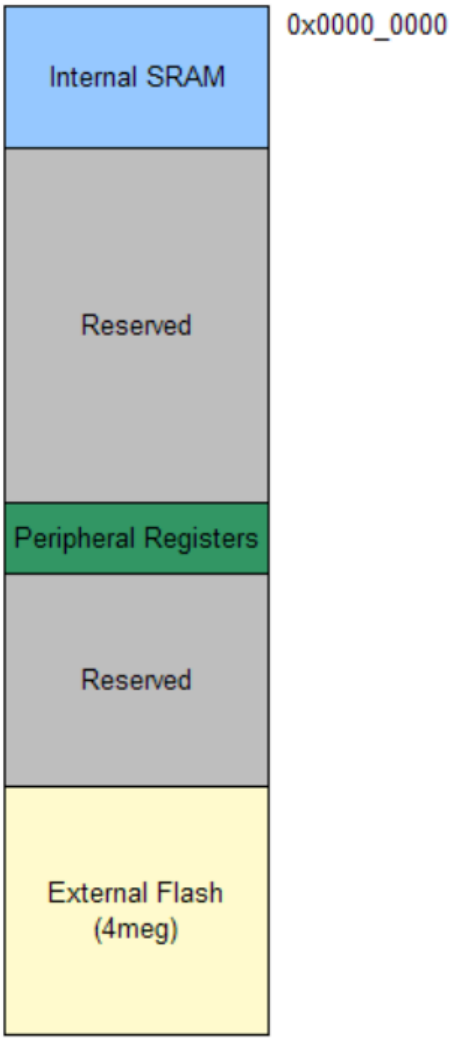
1. MEMORY ACCESS

L'accès à la mémoire dans un processeur permet au CPU d'interagir avec la mémoire système pour lire ou écrire des données. Cette étape est essentielle pour charger des instructions et des données depuis la mémoire principale, ainsi que pour sauvegarder les résultats des calculs. Le memory access garantit que le processeur a un accès rapide et efficace à la mémoire, ce qui est crucial pour le bon fonctionnement des programmes informatiques en permettant le stockage et la récupération de données à des emplacements spécifiques en mémoire.



2. Memory map

- L'espace mémoire total adressable par un processeur avec un bus d'adresse de n bits est 2^n bytes
- Les mémoire et les périphériques occupent des espaces définis à des adresses précises : ceci s'appelle le memory mapping
- L'adresse de début et la taille des espaces mémoire sont définis par un décodage d'adresse effectué par comparaison des bits de poids fort



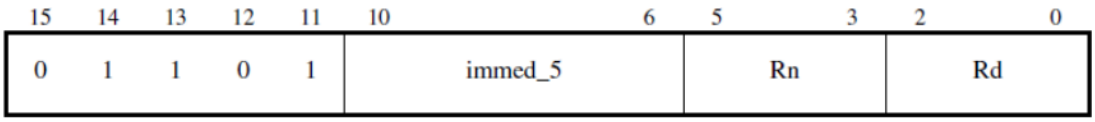
3. Décodage d’une adresse mémoire

À faire

4. Lecture et écriture

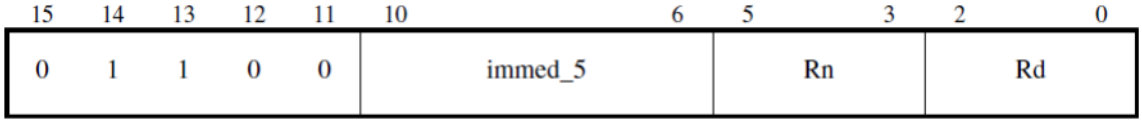
4.1. Lecture mémoire

- LDR → Load to Register
- syntaxe : `LDR <Rd>, [<Rn>, #<immed_5> * 4]`
- charge dans le registre Rd la donnée (32 bits) en mémoire (lecture mémoire)
- l'adresse mémoire est calculée en ajoutant la valeur dans `Rn` à l'offset `immed_5` (5 bits non signé) multiplié par 4
- `Rd ← M[Rn + immed_5 * 4]`



4.2. Écriture mémoire

- STR signifie Store from Register
- syntaxe : `STR<Rd>, [<Rn>, #<immed_5> * 4]`
- écrit en mémoire la donnée contenue dans le registre Rd
- l'adresse mémoire est calculée en ajoutant la valeur dans Rn à l'offset immed_5 (5 bits non signé) multiplié par 4
- `M[Rn+immed_5*4] ← Rd`



4.3. Lecture & ecriture autres formats

Ecriture /Lecture mots de 16 bits

- `LDRH <Rd>, [<Rn>, #<immed_5> * 2]`
- `STRH <Rd>, [<Rn>, #<immed_5> * 2]`

Ecriture /Lecture octets (8 bits)

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

5. Pile (Stack)

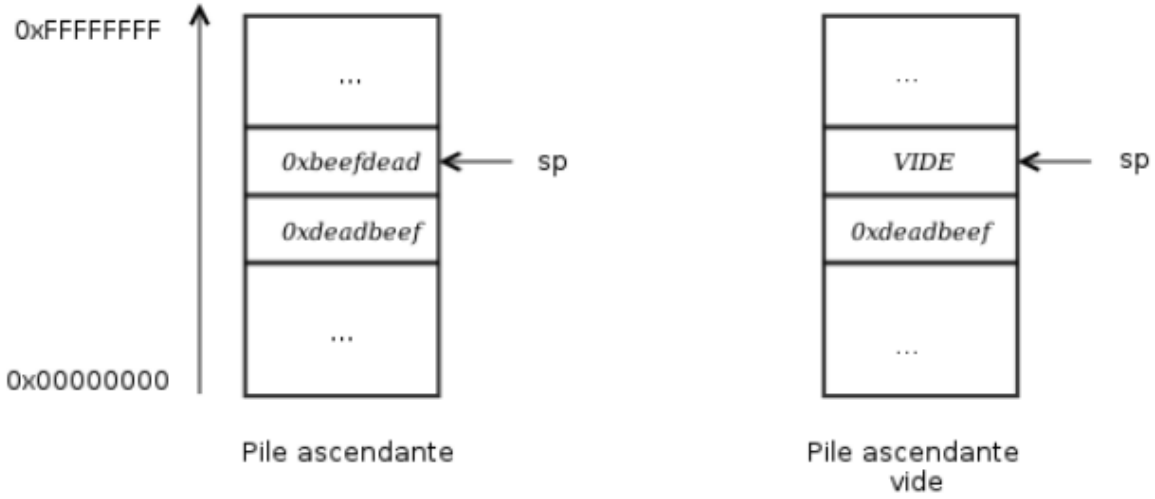
La pile est une zone mémoire réservée dans laquelle les données sont stockées et récupérées selon un principe de LIFO (Last In First Out). La pile est utilisée pour stocker les adresses de retour des fonctions, les variables locales, les paramètres de fonctions et d'autres données temporaires.

Rappel : SP → Stack Pointer

5.1. Pile ascendante

les données sont écrites en incrémentant les adresses

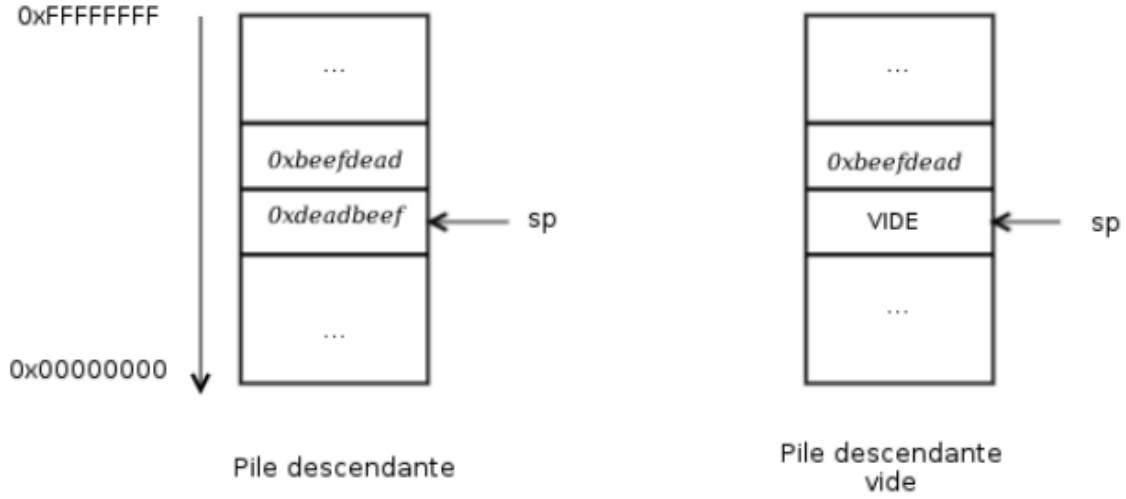
- SP initialisé avec l'adresse du bas de la pile



5.2. Pile descendante

les données sont écrites en décrémentant les adresses

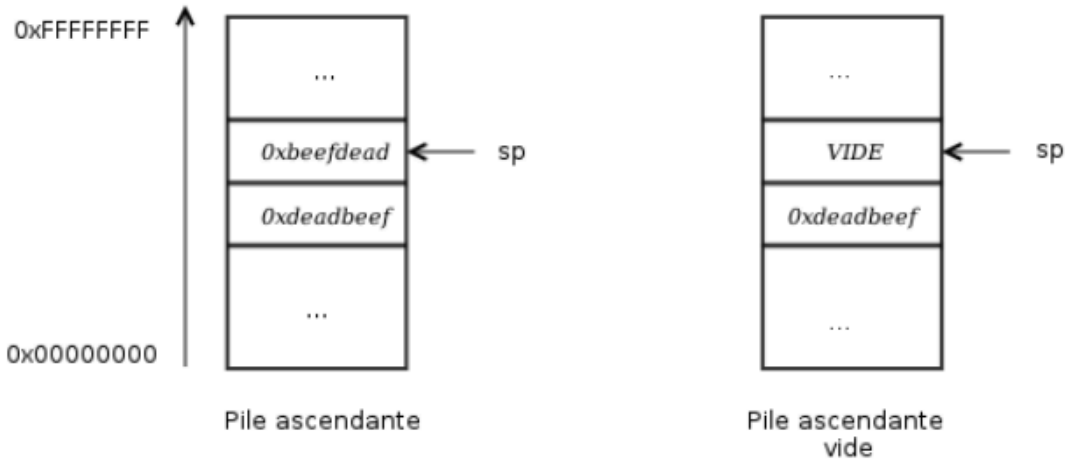
- SP initialisé avec l'adresse du haut de la pile



5.3. Pré/post-incrémentation

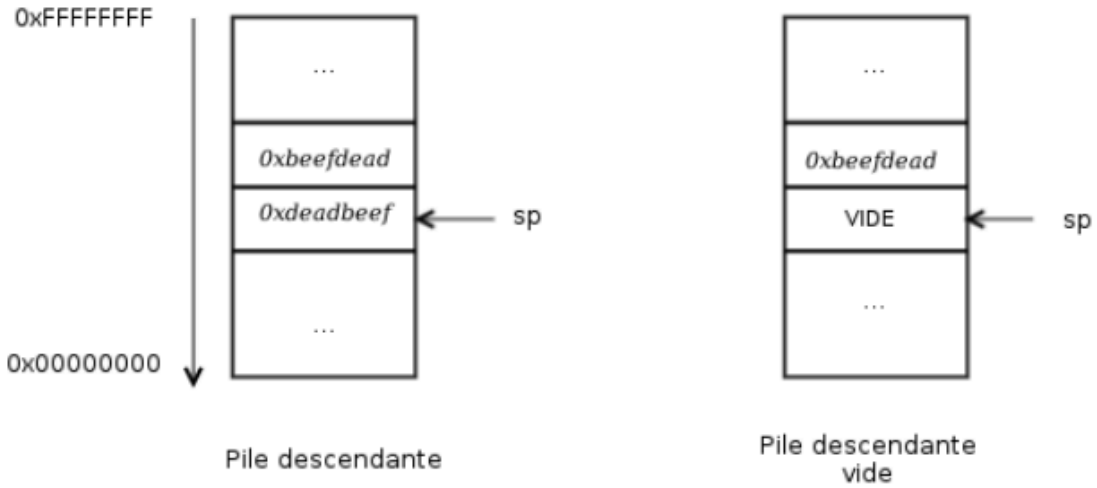
Pré-incrémentation: ++i

- Incrémentation de SP, puis écriture dans le registre à l'adresse SP



Post-incrémentation : i++

- Ecriture dans le registre à l'adresse SP, puis incrémentation de SP

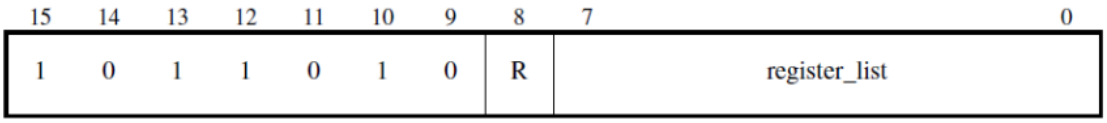


5.4. Utilisation de la pile

- Stockage des adresses de retour pour :
  - appels de fonctions
  - interruptions
- Sauvegarde de registres
  - changement de contexte (fonctions, interruptions)
- Stockage de variables locales
  - en C : variables déclarées dans le corps d'une fonction

5.5. Ecriture dans la pile (ARM)

- PUSH écrit les valeurs de plusieurs registres dans la pile
- Liste de registres de R0 à R7 (un bit par registre) et R pour le Link Register


15 14 13 12 11 10 9 8 7 0
1 0 1 1 0 1 0 R register_list