Rapport projet VLSI : Conception logique d'un processeur ARM

Thaïs MILLERET 21101028 - Guillaume REGNAULT 21107756

Responsable de l'UE : Jean-Lou DESBARBIEUX
Sorbonne Université Master Informatique Semestre 1 09/2024 - 01/2025

Introduction

Dans le cadre de l'UE Conception de circuits intégrés numériques, nous avons réalisé en partie un processeur capable de décode et exécuter le jeu d'instructions ARM. Le processeur est décrit à l'aide du langage de description matérielle VHDL.

Architecture du processeur

(Synthèse du Cours 5 : Architecture générale du processeur ARM, détail étage EXE)

L'objectif principal est que le processeur puisse exécuter les instructions du jeu d'instructions ARM v2.3 conformément à leur spécification.

Comme pour n'importe quel processeur pipeliné, nous essayons d'obtenir un CPI (nombre de Cycles Par Instruction) de 1, c'est-à-dire que le processeur exécute en moyenne une instruction par cycle d'horloge.

Le jeu d'instructions ARM comprend des instructions de complexité très variable. Le cours propose donc de concevoir un pipeline asynchrone. On évite ainsi le gel simultané de tous les étages.

Un pipeline doit suivre trois règles :

1. les étages sont équilibrés (le temps de propagation de l'instruction dans un étage est similaire pour tous les étages)
2. les étages sont séparés par un matériel permettant la transmission des données à travers le pipeline (par exemple un banc de registres)
3. chaque matériel appartient à un seul étage

Une architecture asynchrone

La progression des instructions dans les étages du pipeline peut être découplée. Cela implique des contraintes dans la gestion de la propagation d'une instruction dans le pipeline.

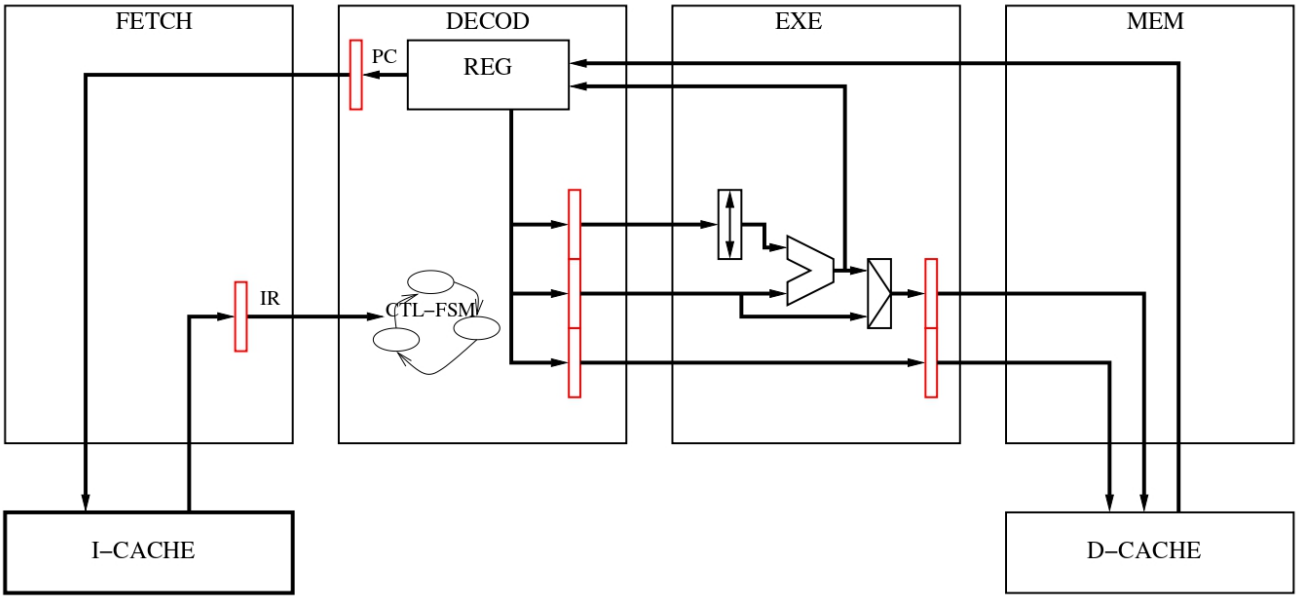
Par rapport à la règle 2 du pipeline, on ne peut pas se contenter d'un banc de registres simples. À la place, les étages du pipeline sont séparés par des fifos (structure qui respecte l'ordonnancement First In First Out). La progression d'une instruction d'un étage au suivant dépend donc de l'état de la fifo qui sépare les deux étages.
Dès que la fifo qui alimente un étage n'est pas vide, l'exécution de l'instruction peut commencer.

4 étages

L'architecture comporte 4 étages :

1. IFETCH : récupération de l'instruction
2. DECOD : décodage de l'instruction et write-back (écriture du résultat de l'instruction dans le banc de registres visible du logiciel)
3. EXE : calcul arithmétique ou logique
4. MEM : accès mémoire (lecture ou écriture)

Schéma du processeur :



La réalisation de IFETCH et MEM nous est fournie. Nous devons concevoir les étages EXE et DEC à l'aide des cours et des sujets de TP.

Registres

Il y a 16 registres User, en particulier :

- 13 : SP = Stack Pointer, pointeur vers le sommet de la pile
- 14 : LR = Link Register, adresse de retour de fonction
- 15 : PC = Program Counter, pointeur d'instruction dans le code
- 16 : CPSR = Current Program Status Register

Le registre CPSR contient des flags générés par l'ALU et mémorisés si l'instruction est suffixée S :

- N : le résultat de l'ALU est négatif
- Z : le résultat de l'ALU est égal à 0
- C : retenue générée par l'ALU dans le cas des instructions arithmétiques et par le shifter dans le cas des instructions logiques
- V : dépassement de capacité dans le cas d'une opération arithmétique signée

Jeu d'instructions ARM

Une instruction ARM peut être exécutée ou non selon un prédicat.

Schéma des prédicats (Cours 3 : Présentation architecture et jeu d'instructions ARM) :

0000 **EQ** - $Z = 1$
0001 **NE** - $Z = 0$
0010 **HS**/**CS** - $C = 1$
0011 **LO**/**CC** - $C = 0$
0100 **MI** - $N = 1$
0101 **PL** - $N = 0$
0110 **VS** - $V = 1$
0111 **VC** - $V = 0$
1000 **HI** - $C = 1$ et $Z = 0$
1001 **LS** - $C = 0$ ou $Z = 1$
1010 **GE** - supérieur ou égal
1011 **LT** - strictement inférieur
1100 **GT** - strictement supérieur
1101 **LE** - inférieur ou égal
1110 **AL** - toujours
1111 **NV** - réservé.

Cette partie présente rapidement les types d'instruction que le processeur doit pouvoir exécuter.

Traitement de données

Ces instructions utilisent l'ALU, et parfois le shifter, pour faire des calculs arithmétiques et logiques.

Multiplications

La multiplication est une opération arithmétique complexe qui implique plusieurs additions.

Branchements

Un branchement est une rupture de séquence : au lieu d'exécuter l'instruction suivante du code assembleur, on saute vers une autre instruction qui peut être située avant comme après l'instruction séquentielle.

Accès mémoire simples

Ce type d'instruction correspond à un accès mémoire unique : on lit ou écrit un mot (ou un octet).

Accès mémoire multiples

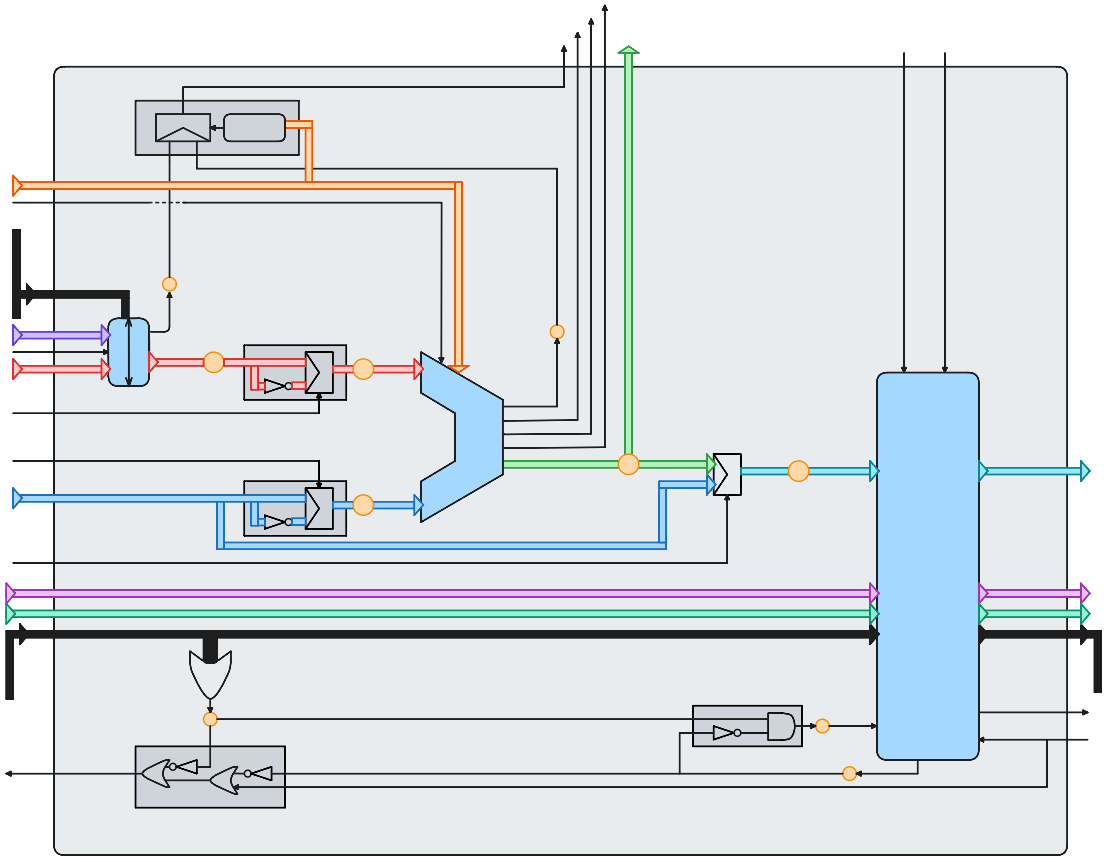
Ce type d'instruction correspond à le lecture ou l'écriture de plusieurs registres du banc de registres User à la fois. On les utilise pour gérer la pile lors d'appel de fonctions.

L'étage EXE

EXE est principalement constitué d'un shifter (décaleur) et d'une ALU. Il s'occupe des calculs arithmétiques et logiques. Cela permet le traitement des instructions arithmétiques et logiques, mais aussi du calcul d'adresse pour les transferts mémoire et les branchements.

Il reçoit ses instructions de l'étage DECOD via une FIFO (profondeur d'une instruction codée sur 32 bits) et renvoie ses résultats au banc de registres REG situé dans DECOD. Les instructions mémoire sont directement transmises vers l'étage MEM via une FIFO.

Schéma global de l'étage EXE (cf. fichier SVG pour zoomer) :

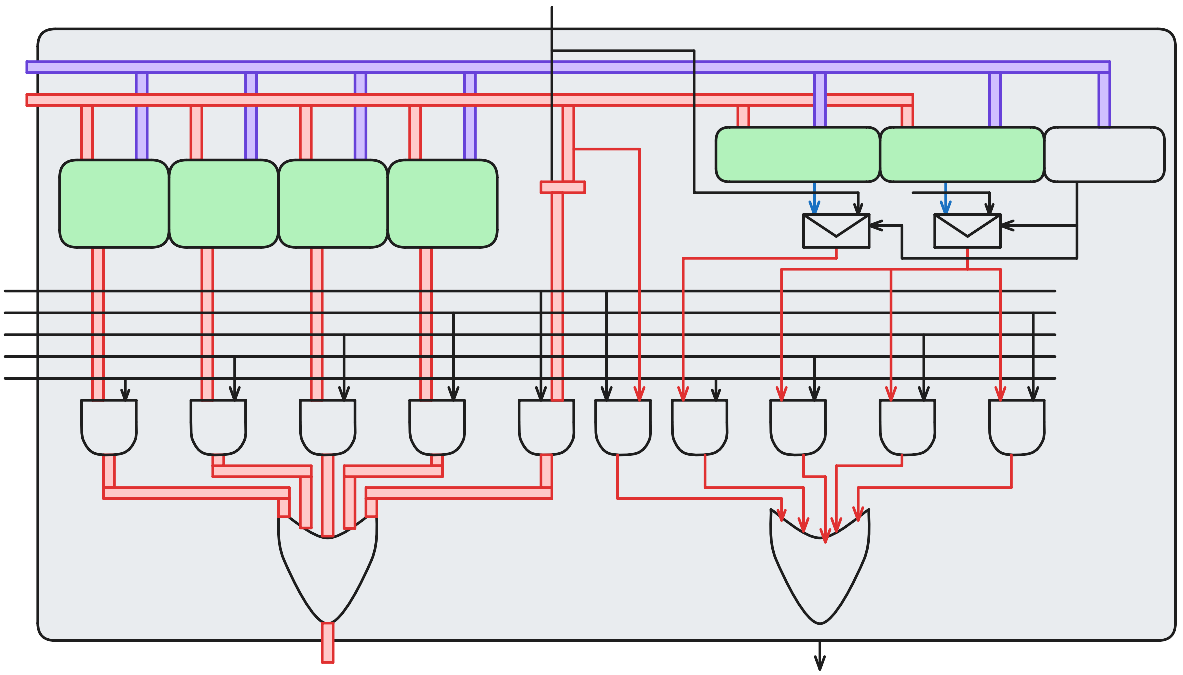


Les instructions arithmétiques et logiques peuvent avoir jusqu'à 3 opérandes sources. Les deux premières opérandes (dec_op1 et dec_op2) peuvent être inversées, d'où la présence d'un inverseur et d'un multiplexeur pour traiter chaque opérande avant son entrée dans l'ALU. La 2e opérande passe par le shifter de l'ALU car elle peut être décalée, parfois grâce à la 3e opérande source.

Shifter

Le shifter s'occupe des décalages et des rotations (sur Op2 ou pour les instructions de traitement de données). Il effectue un décalage logique à gauche ou à droite, un décalage arithmétique à droite, une rotation à droite ou une rotation à droite de 1 bit avec retenue (si RRX) en fonction des indications transmises par DECOD (via les entrées shift_lsl, shift_lsr, shift_asr, shift_ror et shift_rrx).

Schéma du shifter :



cin est la valeur à shifter sur 32 bits. dout est le résultat de l'opération. cin et cout sont les retenues respectivement entrante et sortante. shift_val est un entier sur 4 bits qui correspond à la valeur du shift (0 à 15).

ALU

L'Unité Arithmétique et Logique peut réaliser 4 opérations de base : addition, ET logique, OU logique et OU exclusif. Elle reçoit 2 opérandes (sur 32 bits) A et B et une retenue entrante cin. Elle fournit le résultat res (sur 32 bits) et 4 flags Z, N, V et Cout (ce sont les flags enregistrés dans le registre CPSR).

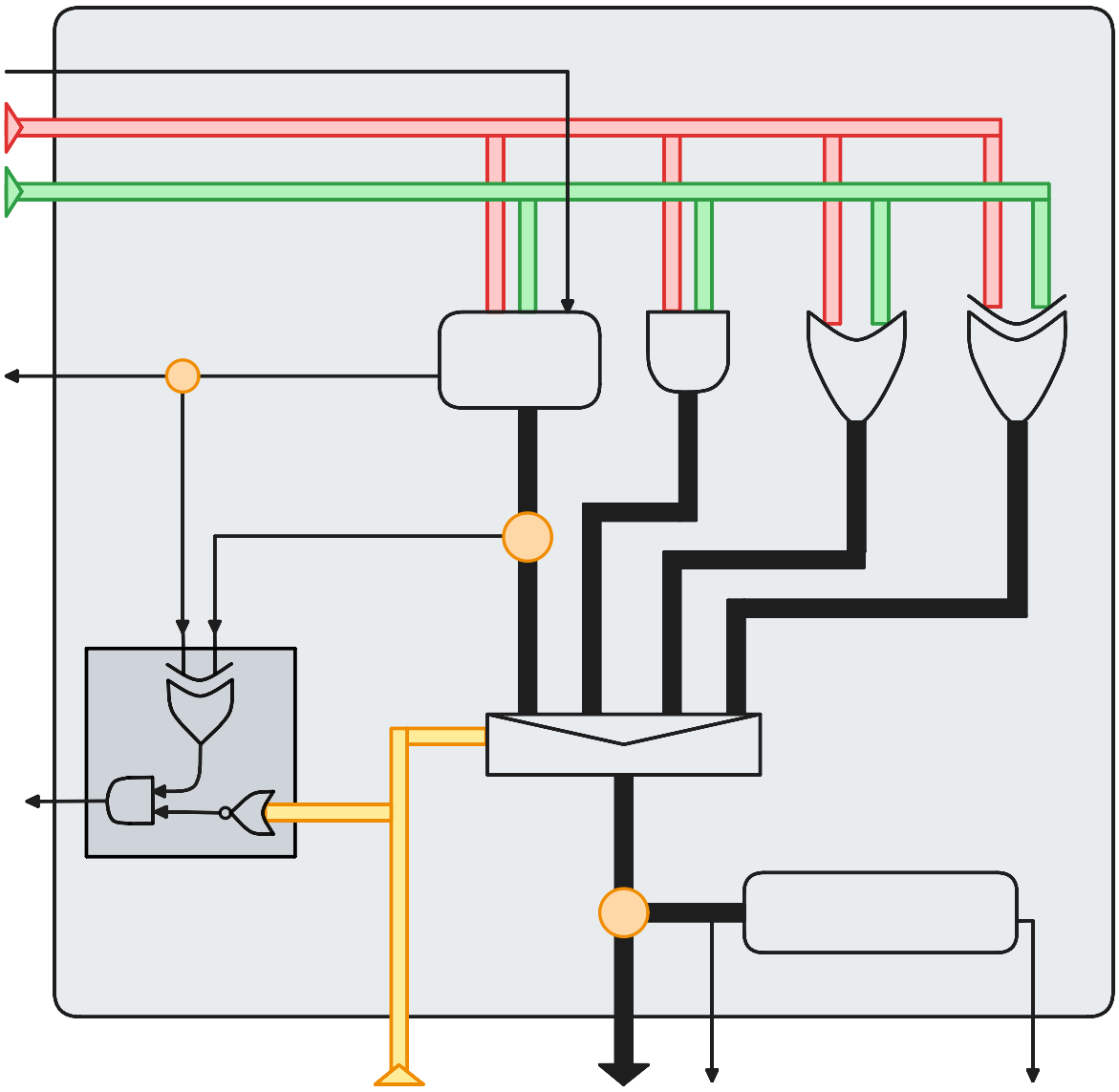
L'entrée cmd spécifie l'opération à effectuer :

- 00 : addition ADD avec la retenue cin,
- 01 : ET logique AND,
- 10 : OU logique OR,
- 11 : OU exclusif XOR.

Nous avons utilisé les opérateurs AND, OR et XOR fournis par VHDL.
Pour l'addition, on aurait pu utiliser l'opérateur + de VHDL sur les vecteurs A et B. Nous avons choisi d'implémenter un additionneur à propagation de retenue (Ripple Carry Adder) en branchant en série des Full Adder.

L'ALU réalise les 4 opérations mais une seule est envoyée en sortie via un multiplexeur dont le sélecteur est cmd.

Schéma de l'ALU :



Test bench de EXE

Le shifter et l'ALU ont été testé séparément pour limiter les tests à faire sur EXE dans sa globalité.

Le test bench du shifter contient des tests pour les 5 opérations (LSL, LSR? ASR, ROR et RRX) pour des valeurs aléatoires.

Le test bench de l'ALU contient des tests pour les 4 opérations (ADD, AND, OR et XOR) pour des valeurs aléatoires et des cas extrêmes. Nous avons également

testé le bon fonctionnement des flags.

Pour tester EXE, nous avons envoyé artificiellement des instructions décodées à EXE (comme si elles venaient de DECOD). Le test est cadencé par une horloge. On vérifie la sortie de EXE vers REG (valeurs des signaux, par exemple par rapport à write back) ou MEM (adresse, data, indications pour MEM). On vérifie également l'état des FIFO qui relient EXE à DECOD et MEM (contenu, entrées et sorties) avant et après avoir pop/push dessus.

L'étage DECOD

DECOD contient un banc de registres REG et une machine à états. Il doit assurer deux fonctionnalités :

- décoder les instructions pour permettre leur exécution par les étages EXE et MEM,
- assurer le séquencement du pipeline, gérer les aléas et traiter les instructions multi-cycles (comme les transferts multiples).

L'instruction est reçue sous la forme d'un mot de 32 bits. DECOD la convertit en un mot de 127 bits interprétable par EXE et/ou MEM. C'est pourquoi DECOD contient un très grand nombre de signaux qui sont mis à jour selon la machine à états. Les signaux sont regroupés en 4 catégories : décodage des instructions, commande de EXE, commande MEM et séquencement/contrôle du pipeline (interaction avec les FIFO).

Comme l'architecture du processeur est asynchrone, il faut gérer la synchronisation des étages pour une instruction. Une instruction n'est lancée que si toutes ses opérandes sources sont valides et que le registre de destination est valide.

Le registre de destination de l'instruction est marqué comme non valide lors du lancement de l'instruction. Le registre de destination repasse à valide lorsque le résultat de l'instruction est disponible. Avec la contrainte qu'une instruction ne peut être lancée que si son registre de destination est valide, ce mécanisme permet de maintenir l'ordre des affectations (instructions) pour un registre donné.

Schéma global de l'étage DECOD (cf. fichier SVG pour zoomer) : ???

Banc de registres REG

Le banc de registres contient les 16 registres décrits dans la partie "Architecture du processeur" et les 4 flags fournis par l'ALU (C, Z, N et V).

Un registre contient une entrée din, une sortie dout, une horloge et un signal de reset (le reset peut être asynchrone par rapport à l'horloge). Si reset est à 1 on remet le registre à 0. Sur le front montant de l'horloge, on copie la valeur de din dans dout. La synchronisation des étages du pipeline nécessite que les registres soient associés à des bits de validité : 1 bit pour la valeur du registre, 1 pour les flags C, N et Z et 1 pour le flag V.

Au reset, tous les registres sont considérés comme valide pour qu'on puisse lancer une instruction (le registre de destination doit être valide). Quand un registre est identifié par DECOD comme registre de destination d'une instruction, il est invalidé. Quand un résultat produit par EXE ou MEM est écrit dans REG, le registre de destination est validé. On ne peut écrire dans un registre que s'il est marqué comme invalide.
De même, on ne peut mettre à jour les flags que si leur bit de validité respectif est à 0. Les instructions logiques écrivent C, N et Z. Seules les instructions arithmétiques affectent V.

MEM et EXE peuvent produire simultanément un résultat. En cas de conflit (les résultats de MEM et EXE vont dans le même registre de destination), on ignore l'écriture de MEM car elle est nécessairement plus ancienne. En effet, une instruction exécutée par EXE ne peut être lancée que si ses opérandes sources sont à jour (registres sources valides) donc le résultat produit par EXE est effectivement plus récent que n'importe quelle donnée chargée par MEM.

Le registre 15 (PC) est un registre particulier. On lui associe un opérateur réalisant l'opération +4 et son contenu et sa validité sont accessibles directement via l'interface de REG. On remarque ici que le calcul de l'instruction suivante se fait dans DECOD.

Schéma de REG (cf. fichier SVG pour zoomer) : ???

Machine à états

DECOD repose sur une machine à états cadencé par l'horloge globale du processeur.

??? TP 7

Schéma de l'automate de DECOD : ???

Etat FETCH

???

Etat RUN

???

Etat LINK

???

Etat BRANCH

???

Etat MTRANS

???

Idées pour les transferts multiples

???

Test bench de DECOD

Comme pour EXE, nous avons testé séparément REG pour simplifier le test bench de DECOD.

Le test bench de REG ???

??? test bench de DECOD

Test d'instructions arithmétiques et logiques

???

Test de branchement

???

Test d'accès mémoire simple

???

Test de programme : somme d'un vecteur

???

Plateforme globale

??? tests réalisés par Guillaume

Conclusion

??? voir ce qu'a fait Danaël