Processeur MIPS

Départements : TIC

Unité d'enseignement ARO

Auteurs : **Bastian Chollet  
Kevin Ferati**

Professeur : **Marina Zapater**

Assistant : **Mike Meury**

Classe : **C**

Salle de labo : **A09**

Date : **01 mai 2022**

# Introduction

Contexte, processeur développé, instructions supportés, environnement de travail, méthodologie de travail, partage de travail

# FETCH

## Introduction

Rôle dans un processeur, fonctionnement général

## Analyse et conception

Explication des choix / solutions dem ise en place

## Réalisation

Problèmes rencontrés (section différente ?). Comment on s’est réparti les tâches.

## Simulation

Simulation avec un chronogramme ?

## Test

tests unitaires comme en prog ? Cf rapport FETCH (avec les screen et tout ça) (voir même pas de screen, trop chiant)

## Feedback

Retour de mike, pourquoi on a fait ces erreurs, comment on les a résolu, qu’on a-t-on tiré

# DECODE

## Introduction

## Analyse et conception

## Réalisation

## Simulation

## Test

## Feedback

# EXECUTE & MEMORY ACCESS

## Introduction

Cette section s’intéresse à l’implémentation du bloc EXECUTE ainsi que de la mémoire de données. Dans le laboratoire précédent, nous avons segmenté les différentes instructions pour qu’elle soit interprétée correctement par le processeur. Ici, nous allons donc mettre en œuvre l’exécution de ces instructions ainsi que l’accès en lecture/écriture à une mémoire externe au processeur.

## Analyse et conception

Dans la théorie, le bloc EXECUTE d’un processeur contient les éléments nécessaires à la réalisation d’opérations arithmétiques et logiques, ainsi qu’au décalage (shift) des données. Dans certains cas, il peut également contenir les éléments nécessaires à la représentation des nombres à virgules flottantes.

Ces différents éléments sont représentés respectivement par une unité arithmétique et logique (ALU) tentant à jour un registre d’état, et par une unité de décalage (SHIFTER). Toutefois les unités à virgules flottantes (FPU) et les unités multiplicatrices (MULTIPLIER) ne seront pas réalisées dans cette partie car non demandé dans la consigne.

La mémoire de donnée, quant à elle, est directement impliquée lors des instructions d’écriture et de lecture dans celle-ci, respectivement les commandes STR et LDR. Il s’agit d’une mémoire externe au processeur.

Afin de finaliser l’implémentation de notre processeur, il sera également nécessaire d’implémenter une stack qui pourra servir notamment lors d’interruptions.

La conception de cette partie peut se résumer ainsi :

* Un circuit EXECUTE composé de :
  + Un circuit ALU
  + Un circuit SHIFTER
  + Un circuit CPSR
* Connecter les entrées / sorties à la mémoire de donnée

## Réalisation

### EXECUTE

La conception de ce bloc consiste essentiellement à réaliser le décryptage d’un bus de contrôle construit lors de la partie DECODE. Dans la finalité, ce bus doit fournir les informations suivantes :

* L’opérande n° 1 : Soit le registre n, soit 0
* L’opérande n°2 : Soit le registre m, soit une valeur immédiate sur 3 ou 8 bits.
* Le n° de l’opération que l’ALU devra effectuer
* Le n° de l’opération que le SHIFTER devra effectuer
* Le nombre de décalage à effectuer : Soit une valeur dans le registre m soit une valeur immédiate sur 5 bits.
* L’activation ou non de la lecture du CPSR

Dans le fonctionnement, le premier opérande est passé dans le SHIFTER. La sortie de ce bloc est combinée à deuxième opérande dans l’ALU. Le résultat des deux opérandes est transmis sur la sortie **data \_out\_o** et les informations additionnelles de l’ALU (carry et overflow) sont transmis à un bloc **zcnv\_unit** qui se chargera de construire la valeur du registre d’état (CPSR) et de l’envoyer sur la sortie **cpsr\_o**.

### SHIFTER

Les instructions traitées sont les suivantes :

* ASR (décalage arithmétique vers la droite)
* LSL (décalage logique vers la gauche)
* LSR (décalage logique vers la droite)
* ROR (décalage rotatif à droite)

Logisim met déjà à disposition un bloc « shifter » que nous avons configuré pour qu’il réalise chacune des instructions ci-dessus. Nous avons connecté l’entrée **operand\_i** dans chacun de ces blocs. Une entrée additionnelle **sel\_op\_shift\_i** se chargera de sélectionner en sortie l’opération de décalage adéquate à l’aide d’un multiplexeur. Il est également important de noter que selon l’instruction, il sera nécessaire d’utiliser l’opérande sans pour autant lui faire subir de décalage. De ce fait, nous avons ajouté une entrée « bypass » au multiplexeur de façon que les données ne soient pas modifiées. Rappelons que cet opérande non modifié peut être nécessaire à l’ALU ultérieurement.

Nous avons donc 4 opérations + 1 bypass. L’entrée de sélection doit se faire sur 3 bits. Le multiplexeur a donc été implémenté en suivant la table de vérité suivante :

Une image contenant table

Description générée automatiquement

### ALU

Notre implémentation de ce bloc repose essentiellement sur la même qui avait été réalisée lors d’un laboratoire précédent dans notre cours de SYL. À la différence que cette fois-ci, aucune interdiction n’est posée sur l’usage des blocs arithmétiques, et qu’on ne distingue pas les opérations signées des opérations non signées. Ici, toutes les opérations arithmétiques seront signées.

Ainsi, notre implémentation est similaire à celle du SHIFTER. Nous disposons de deux opérandes 16 bits en entrée ainsi que d’un sélecteur d’opération sur 3 bits. Nous réalisons l’entier des opérations en simultanées, puis nous relions les différents résultats à un multiplexeur. La sortie de ce dernier est donc définie par le sélecteur en entrée susmentionné.

Notre ALU réalise les opérations suivantes :

* ADD
* SUB
* AND
* ORR
* MVN (équivalent à un NOT)
* NEG
* EOR (équivalent à un XOR)

Le schéma de notre multiplexeur suit la table de vérité suivante :

Une image contenant table

Description générée automatiquement

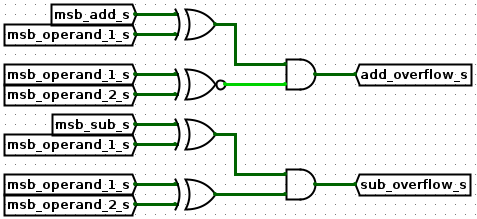
Le résultat de l’opération choisie sera ensuite transmis à la sortie **alu\_data\_out\_o**.

Le résultat de l’ALU va également être responsable de la mise à jour du registre d’état. Il convient donc de fournir les bits de carry et d’overflow correctement en sortie selon l’opération effectuée. Le laboratoire n’imposant aucune façon de faire particulière, nous avons opté pour une construction similaire au précédant laboratoire dans notre cours de SYL.

Une image contenant table

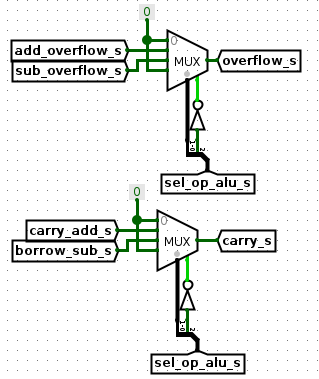
Description générée automatiquement

Pour faciliter le traitement des différents signes, nous avons pris soin de splitter le MSB de chaque opérande ainsi que ceux du résultat de l’addition et de la soustraction. En traduisant le tableau ci-dessus avec des portes logiques, nous obtenons le résultat suivant :



Le carry, quant à lui, se gère plus simplement puisque les blocs d’addition et de soustraction de Logisim mettent déjà à disposition des sorties carry et borrow qu’il nous suffit de relier à la sortie de notre ALU si l’opération choisie par cette dernière fait intervenir un des deux.

Comme évoque, les sorties **carry\_o** et **overflow\_o** doivent être mise à jour correctement selon l’opération effectuée. Puisque nous traitons l’ensemble des opérations de façon simultanées, il est important que le résultat choisi par l’entrée **sel\_op\_alu\_i** définisse également les valeurs de carry et/ou d’overflow. Nous avons donc organisé le choix de ces bits de la façon suivante :

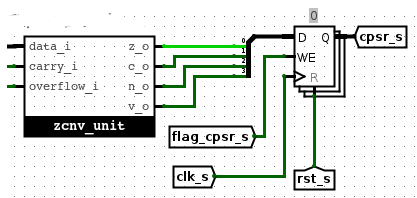


### CPSR

Comme son nom l’indique, le CPSR est un **registre** d’état. Il se base sur les valeurs de sortie de notre ALU.

Ce laboratoire met à notre disposition un bloc **zcnv\_unit** qui va se charger de construire les 4 bits significatifs du CPSR. Notre implémentation se limite donc à connecter correctement les valeurs de sortie de notre ALU aux entrées de cette unité, puis de combiner les différentes sorties de la **zcnv\_unit** afin de stocker la valeur finale dans un registre.

Dans la partie DECODE, nous avions implémenté un flag indiquant si oui ou non, le CPSR devait se mettre à jour. Nous utilisons donc ce flag dans EXECUTE que nous relions à l’entrée Write Enable du registre CPSR. Au final, notre implémentation ressemble à ceci :



### MEMORY ACCESS

Le bloc responsable de gérer la lecture et écriture en mémoire nous a déjà été fourni et implémenté. Il n’est donc pas pertinent d’en parler ici. Nous traiterons cependant cette partie dans les chapitres suivants, car nous avons bien évidemment dû la tester.

### Stack

Le processeur conçu dans ce laboratoire n’implémente aucune instruction de gestion de la pile. Il n’y a donc aucun circuit attitré aux gestions de ces instructions. Toutefois, les instructions POP et PUSH ont des fonctionnements qu’il est entièrement possible de construire à l’aide de différentes instructions supportée par notre processeur.

Ainsi la « réalisation » de cette partie se fera au travers de notre programme assembleur. Nous allons réaliser manuellement chaque étape des instructions POP et PUSH. Nous les avons implémentés de la façon suivante :

1. Réaliser un saut avec lien : Instruction BL
2. Stocker en mémoire la valeur du Link Register(LR) à l’adresse contenue dans le Stack Pointer(SP)
3. Incrémenter le SP de 2 (car architecture 16 bits)
4. En retour de saut, Décrémenter le SP de 2
5. Lire la mémoire de donnée à l’adresse du SP, et mettre la valeur lue dans LR
6. Mettre à jour le Program counter (PC) avec la valeur de LR.

## Simulation

## Test

## Feedback

Date : Date rendu du rapport

Noms des étudiants : Etudiant A Etudiant B

# Conclusion

Qu’a-t-on tiré du projet,quels problèmes rencontrés (organisationnels, méthodologiques, techniques, …) comment refaire certain point et remarques personnels, utilité des PDF des cours

# Annexe

Documents annexés avec numérotation et commentaires.