Processeur MIPS

Départements : TIC

Unité d'enseignement ARO

Auteurs : **Bastian Chollet**

**Kevin Ferati**

Professeur : **Marina Sancho Zapater**

Assistant : **Mike Meury**

Classe : **C**

Salle de labo : **A09**

Date : **03 mai 2022**

Table des matières

[1 Introduction 1](#_Toc102493682)

[2 FETCH 3](#_Toc102493683)

[2.1 Introduction 3](#_Toc102493684)

[2.2 Analyse et conception 3](#_Toc102493685)

[2.3 Réalisation 4](#_Toc102493686)

[2.4 Test 5](#_Toc102493687)

[2.5 Feedback 6](#_Toc102493688)

[3 DECODE & BANK REGISTERS 7](#_Toc102493689)

[3.1 Introduction 7](#_Toc102493690)

[3.2 Analyse et conception 7](#_Toc102493691)

[3.2.1 Banque de registre 7](#_Toc102493692)

[3.2.2 Decode 8](#_Toc102493693)

[3.3 Réalisation 8](#_Toc102493694)

[3.3.1 Banque de registre 8](#_Toc102493695)

[3.3.2 DECODE 10](#_Toc102493696)

[3.4 Test 11](#_Toc102493697)

[3.4.1 Banque de registres 11](#_Toc102493698)

[3.4.2 DECODE 12](#_Toc102493699)

[3.5 Feedback 15](#_Toc102493700)

[4 EXECUTE & MEMORY ACCESS 16](#_Toc102493701)

[4.1 Introduction 16](#_Toc102493702)

[4.2 Analyse et conception 16](#_Toc102493703)

[4.3 Réalisation 16](#_Toc102493704)

[4.3.1 EXECUTE 16](#_Toc102493705)

[4.3.2 SHIFTER 17](#_Toc102493706)

[4.3.3 ALU 18](#_Toc102493707)

[4.3.4 CPSR 20](#_Toc102493708)

[4.3.5 MEMORY ACCESS 20](#_Toc102493709)

[4.3.6 Stack 20](#_Toc102493710)

[4.4 Test 21](#_Toc102493711)

[4.4.1 ALU et SHIFTER 21](#_Toc102493712)

[4.4.2 Memory Access 21](#_Toc102493713)

[4.5 Feedback 22](#_Toc102493714)

[5 Conclusion 22](#_Toc102493715)

# Introduction

Dans le cadre du cours d’ARO à l’HEIG, nous devons concevoir un processeur à jeu d’instructions réduit (RISC) simplifié. Les instructions supportées seront celles du kit d’instruction ARM THUMB (16b).  
  
Les composants développés sont :

1. FETCH
2. DECODE & REGISTER BANK
3. EXECUTE & MEMORY ACCESS

Ce rapport est segmenté selon chaque partie du processeur. Dans chacune de ces dernières, nous tenterons de présenter une analyse du travail à effectuer, l’implémentation choisie, ainsi que divers tests validant notre réalisation. Finalement, chaque partie comportera une section commentant les divers feedbacks reçus tout au long de ce projet.

# FETCH

## Introduction

Dans un processeur, le composant FETCH est celui qui va déterminer, en fonction du Program Counter (un registre sauvegardant l’instruction à traiter) et de l’instruction de saut, l’adresse de l’instruction qui doit être exécutée. Plus précisément, la sortie du FETCH est une adresse transmise à la mémoire d’instructions, laquelle restitue l’instruction correspondante.

La mémoire d’instruction est une Read-Only Memory qui garde les instructions d’un programme.

FETCH prend en entrée (principalement) :

* L’instruction en cours afin de déterminer s’il doit y avoir un saut (conditionnel, inconditionnel, …). S’il n’y a pas de saut, la sortie sera le PC incrémenté par le nombre d’octets dans une instruction (soit, 2) ;
* L’adresse contenue dans le Program Counter (afin de pouvoir l’incrémenter si nécessaire) ;
* L’adresse de saut s’il y a un saut ;
* Le bus de contrôle qui indique si une instruction est un saut et est conditionnelle ainsi que la condition à tester. Ceci est normalement fait dans le composant DECODE, mais pour des questions d’essais nous devons l’implémenter déjà ici.

FETCH renverra :

* L’adresse pour la prochaine instruction ;
* Les données de l’instruction courante ;
* La valeur future du Program Counter.

## Analyse et conception

Le Program Counter (PC) doit se situer normalement dans une banque de registre. À notre stade, ayant besoin uniquement de ce registre, nous l’implémentons à part. De là, nous pourrons tester des simples instructions pour voir si le PC a été correctement implémenté.

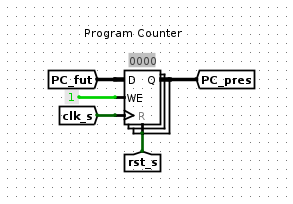
Pour la gestion du saut, normalement la détermination du saut (c’est-à-dire s’il y en a un et s’il est conditionnel) se fait dans la partie EXECUTE. Vu qu’on en besoin, on implémente un mini-déterminateur. Ceci se fait depuis les données de l’instruction. Ce déterminateur va analyser l’instruction et comparer les 4 premiers bits pour savoir le type de saut.

La gestion des sauts conditionnels (c’est-à-dire le calcul des conditions et l’enregistrement des résultats de la comparaison dans un registre) se faisant dans l’EXECUTE, nous testons les sauts conditionnels en mettant « à la main » en spécifiant les constantes de résultats de comparaison à la main. Par exemple, dans le cas d’un BEQ, il faut que le bit à la position 0 (le LSB) soit à 1. C’est la situation que nous allons prendre ce qui permettrait de faciliter les tests.

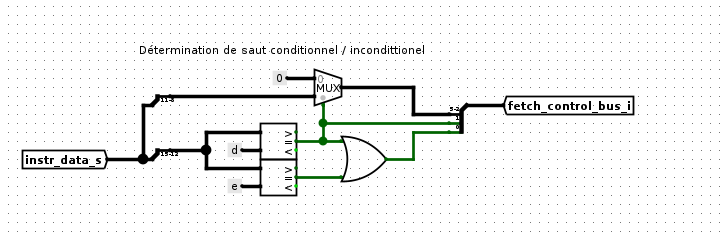
## Réalisation

Nous n’avons eu aucun problème à réaliser les différentes parties du composant.

Résultat du Program Counter : Ici, rien de spécial.

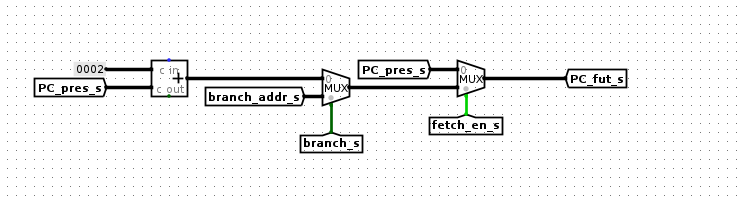


Détermination du saut. Ici, rien de difficile : on vérifie les premiers bits pour savoir si c’est un saut et on récupère les conditions si c’est le cas.



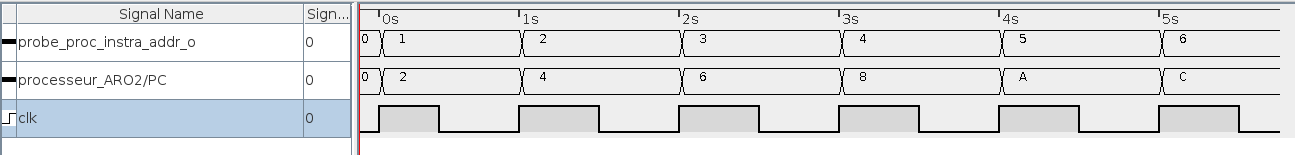
Pour le calcul de l’adresse de saut, nous avons respecté le manuel et les cours : Une multiplication par 2, suivi d’une extension à 16b et d’une addition de 4 et du PC.

Sélection de l’adresse d’instruction finale : S’il y a un saut, le futur PC est l’adresse de saut. Sinon, ce sera l’adresse courante incrémentée de 2. Dans tous les cas, si le FETCH est désactivé, l’adresse ne change pas.



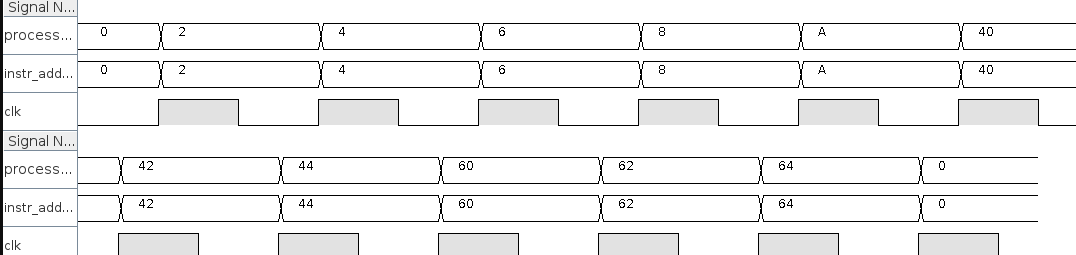
## Test

Comme première simulation, nous avons essayé, après implémentation du PC, son incrémentation. On peut voir le résultat ci-dessous. La première ligne correspond à l’adresse de sortie du FETCH (sur 15 bits) et la deuxième au PC. On observe qu’à chaque horloge, le PC est incrémenté de deux et l’adresse de 1, ce qui est correct. Il ne faut pas oublier que pour cette dernière les bits 15-1 seulement sont prises et on ignore le dernier bit.



Une image contenant texte

Description générée automatiquementIl faut ensuite simuler des sauts. Pour cela, nous avons préparé un programme assembleur avec deux sauts inconditionnels et un conditionnel. Le programme commençant à 0x0, après avoir traité à l’instruction 0xA le programme devrait vers un saut sur 0x40. Après l’instruction à 0x44, le programme saute à 0x60. Ensuite, à 0x64, il revient à 0x0.

Résultat :  
  


La première ligne est le PC et la deuxième est l’adresse d’instruction (cette fois à 16b pour plus de clarté). Nous observons que ces deux éléments suivent le chemin précédemment énoncé : il y a 3 sauts (A => 40, 44 => 60 et 64 => 0). Ainsi, le FETCH semble fonctionner.

## Feedback

Nous avons reçu un simple retour, le reste fonctionnant très bien. En effet, nous avions utilisé des extenseurs qui complexifiait les schémas. Nous les avons simplement remplacés par des splitters.

# DECODE & BANK REGISTERS

## Introduction

La banque de registre (BR) est un composant regroupant les registres disponibles pour lecture et écriture. C’est le composant mémoire le plus proche du CPU et donc le plus rapide mais aussi le plus coûteux.

Dans notre cas, il est composé de 8 registres de 16b. La notation est R<numéro de registre> à partir de 0.

* Les registres 0 à 4 sont n’ont pas de fonctions particulières autre que le stockage.
* Le registre 5 est nommé le Stack Pointer et contient l’adresse vers l’instruction de la stack courante.
* Le registre 6 est nommé le Link Register et contient l’adresse de retour vers l’instruction à exécuter à la fin d’une interruption. Sa gestion se fait au moyen du composant « LR\_Manager », déjà fourni.
  + LR\_Manager indique les données à écrire dans le LR et si l’écriture est activée.
  + L’écriture dans le LR est activée s’il y a un lien, si l’écriture est activée « manuellement » avec sélection du registre 6.
* Le registre 7 est le Program Counter, mentionné lors du FETCH. La possibilité d’écrire dedans dépend de l’existence d’un BL.

Les entrées principales du composant sont :

* Trois entrées de sélection de registre pour lecture, tous en 3b (car 8 registres). Le contenu des registres sélectionnés par deux de ces sélecteurs (M, N) ira dans le composant EXECUTE, et le dernier (MEM) servira dans les cas de calcul d’adresse pour écrire en mémoire ;
* Une entrée de sélection de registre pour écriture (D), sur 3b ;
* Les données à écrire dans ledit registre ;
* Le bus de contrôle sur 2b (le LSB n’est pas utilisé et le MSB active ou non l’écriture) ;
* Une entrée spéciale pour le PC qui va définir ses données.

Le composant DECODE fait, comme son nom l’indique, un décodage de l’instruction transmise par FETCH. Plus précisément, ce composant :

* Sépare une instruction par ses différentes opérandes et valeurs immédiates ;
* Sélectionne l’instruction à exécuter. Ceci se fait au moyen du composant Main Control Unit.
  + Ce composant compare simplement les premiers bits d’une instruction et active le flag correspond à l’opération. Il gère aussi les contrôles bus de différents composants.

## Analyse et conception

### Banque de registre

Pour l’écriture, il nous faut activer le registre sélectionné par l’opérande d’écriture (reg\_d\_sel\_i). Ceci peut se faire au moyen d’un décodeur. L’écriture est activée seulement si le MSB du bus de contrôle est activé et que la banque est activée.

Pour la lecture, un simple MUX de 8 choix par sélecteur de registre en lecture, donc 3 MUX suffit.

### Decode

Nous distinguons plusieurs cas :

* Sélection des registres ;
  + Pour ceux-là (sélecteur N, M) un MUX suffit amplement car il faut sélectionner différentes valeurs en fonction d’un sélecteur.
* Ajout de décodage de certaines instructions ;
  + Pour cela, dans le composant opcode\_supported\_unit, il suffit d’ajouter les comparaisons avec les premiers bits des instructions manquantes.
* Les bus de contrôles des différents composants. Pour ceci, il suffit de mettre ne porte OU pour les différentes sorties des bus.

Plusieurs composants nous sont déjà fournis :

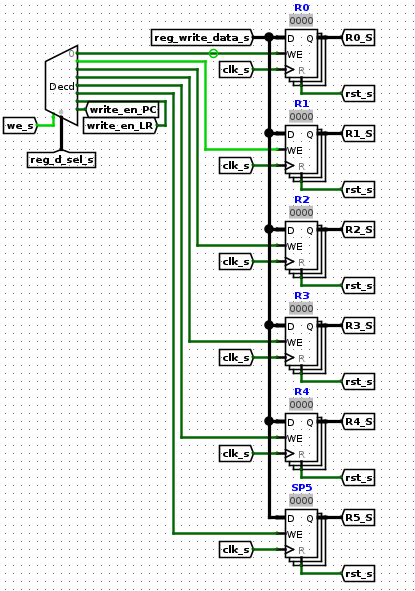
* Decode\_instr\_splitter
  + Ce composant fourni, depuis une instruction de 16b, différentes zones de l’instruction (opérande, valeurs imédiates), nous facilitant un peu la vie.
* Main\_controle\_unit
  + Ce composant est le lieu où les différents composants ont leur bus calculé et activent ou non différents éléments.
* Bus\_constructor
  + Ce composant permet de créer les bus des autres composants

## Réalisation

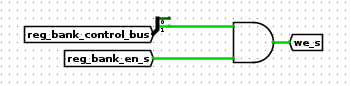
### Banque de registre

Nous n’avons pas eu de problème à la réalisation de cette partie.

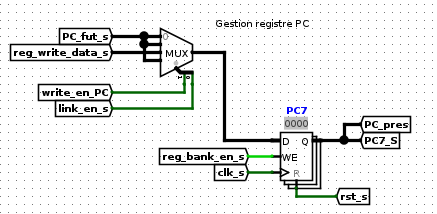
Pour l’écriture, nous avons, comme convenu, implémenté un simple décodeur qui active le registre correspond au sélecteur « D » : avec le sélecteur D à 001 (1) :



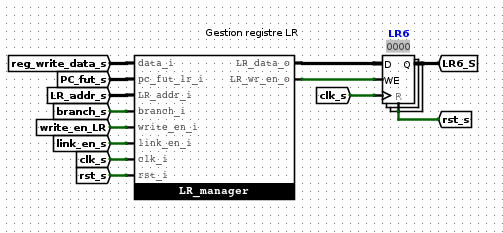
« we\_s » (signal qui active le décodeur et donc l’écriture) correspond à :

  
Il faut donc que la banque ET que le MSB du bus de contrôle soit activés pour que l’écriture le soit.

Pour respecter le cahier des charges concernant l’écriture dans le registre PC, nous avons créé un MUX avec 2b de sélection :



Pour le LR, nous avons correctement implémenté le LR Manager :



Pour la lecture, comme convenu durant l’analyse, 3 MUX avec les 3 différents sélecteurs ont suffi. Chaque entrée correspondait à un registre.

### DECODE

Cette partie a surtout consisté en une série de tirage de câble. Le plus compliqué était la différence entre le manuel et le laboratoire entre les notations de registre. Tout d’abord, il s’agissait de créer les sélecteurs pour N, M, D. Pour cela, nous avons simplement suivi le cahier des charges en implémentant plusieurs MUX et en décortiquant le bus de contrôle du decode.

Le decode\_controle\_bus vient du main\_control\_unit, déjà fournie. Cependant, il fallait y ajouter des instructions manquantes. Ceci s’est fait assez facilement, à l’exception d’un court problème : dans certain cas, une instruction SUB R R R était confondue avec une autre instruction et il a fallu tester les instructions une par une pour savoir où était le problème et il s’agissait juste d’une mauvaise vérification sur le nombre de bits. Finalement, nous nous sommes calqués sur le même système que les autres instructions : des comparaisons des x premiers bits.

La réalisation des bus de contrôle s’est aussi réalisée sans accroc, le plus difficile étant de ne pas se perdre dans la réalisation. Nous avons aussi remplis et vérifiés les tableaux du cahier des charges.

La réalisation du signal sel\_op\_shift\_o s’est aussi faite sans problème. Pour le réaliser, nous avions observé les cas où le bit [0] était actif, puis le [1] et le [2]. Nous avions donc :

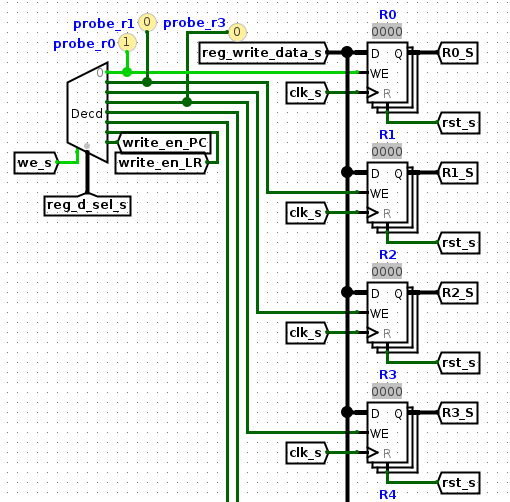
[0] = asr\_r\_r\_s + lsr\_r\_r\_imm\_s + lsr\_r\_r\_s + asr\_r\_r\_imm\_s   
[1] = lsl\_r\_r\_s + lsr\_r\_r\_imm\_s + lsr\_r\_r\_s + lsl\_r\_r\_imm\_s  
[2] = ror\_r\_r\_s

## Test

### Banque de registres

Pour tous les tests de lecture, nous observons les sélecteurs.

Pour l’écriture, nous observons la sortie du décodeur d’activation d’écriture de R0, R1 et R3 :

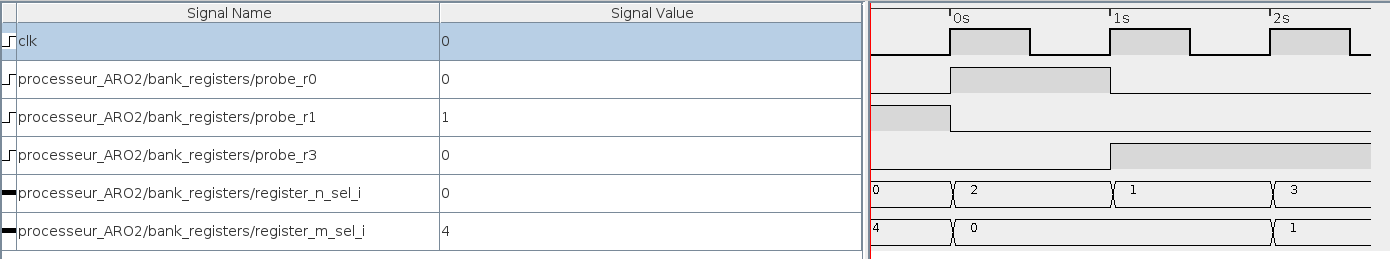


Programme de test :

Une image contenant texte

Description générée automatiquement

Résultat :



|  |  |  |
| --- | --- | --- |
| Instruction | Comportement attendu | Comportement effectif |
| MOV r1, #5 | R1 activé pour écriture | Probe\_r1 = 1 => OK |
| MOV r0, #21 | R0 activé pour écriture | Probe\_r0 = 0 => OK |
| ADD r3, r1, r0 | R3 activé pour écriture  R1, r0 sélectionné pour le cture | Probe\_r3 vaut 1  N\_sel\_i = 1 (registre 1) et m\_sel\_i = 0 (registre 0) => OK |
| AND r3, r1 | R3 activé pour écriture et lecture, r1 pour lecture | Probe\_r3 = 1, n\_sel\_i = 3, m\_sel\_i = 1 => OK |

### DECODE

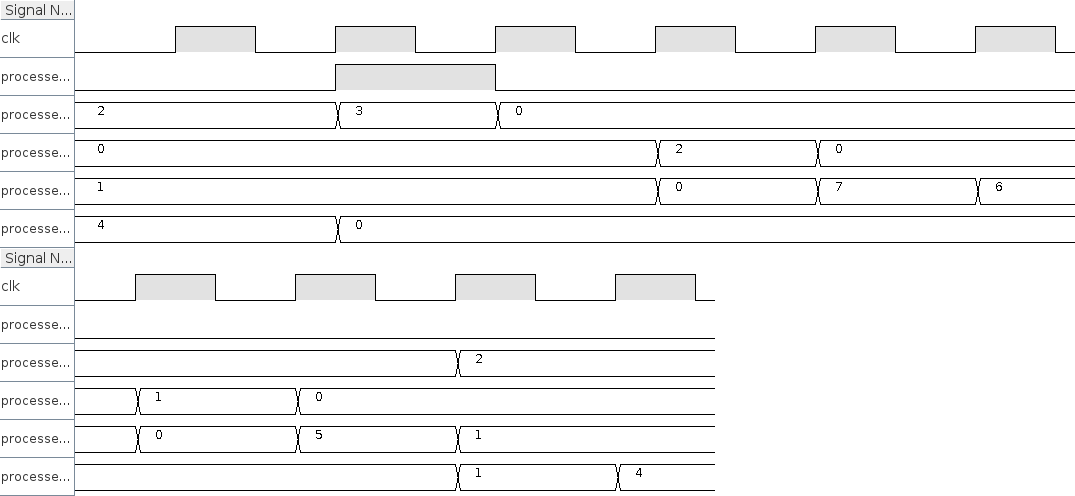
Cette partie part du principe que les composants fournis sont fonctionnels.

Les prochaines captures représentent certains bus de contrôle.

Le programme de test commun est :

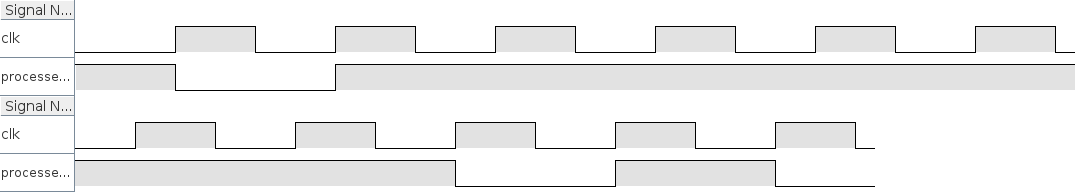
Une image contenant texte, table

Description générée automatiquement  
  
Nous exécutons le programme ci-dessus puis nous générons les signaux pour les différents bus que nous avons implémenté. Nous comparons ensuite les bus au cahier des charges. Pour des raisons de clartés et de brièveté, tous les bus de contrôles ne sont pas testés avec un chronogramme. Tous les tests sont fonctionnels.

Bus de contrôle EXECUTE

Ordre des signaux :



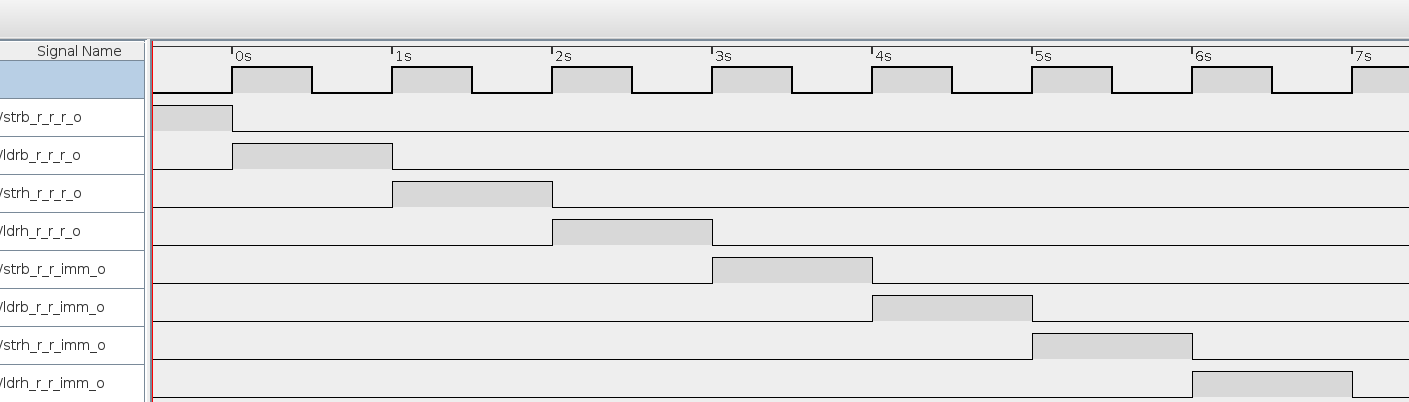
Bus REGBANK  
  
Signal : reg\_bank\_wr

Il s’agit ici de tester les nouvelles instructions placées dans opcode\_supported\_unit .

Le programme est :

Une image contenant texte

Description générée automatiquement

Voici le résultat. Toutes les nouvelles instructions sont correctement activées.

## Feedback

Le retour qu’on a eu portait sur opcode\_supported\_unit qui manquait que nous avons tout de suite rajoutée. Il manquait aussi une instruction dans le bus de contrôle de DECODE.

# 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

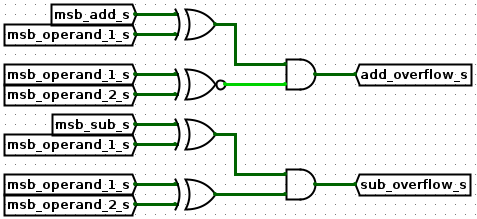
Le résultat de l’opération choisie sera ensuite transmis à la sortie **alu\_data\_out\_o**.

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

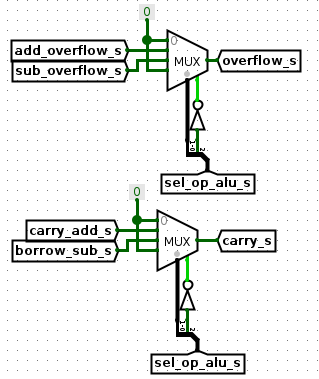
Description générée automatiquement

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 :

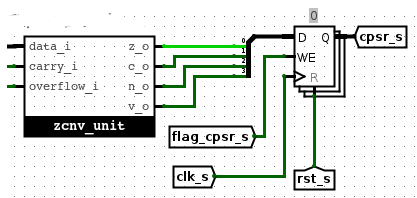


### CPSR

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.

## Test

### ALU et SHIFTER

Le but sera de tester les différents composants de cette partie. Pour des raisons de simplifications, nous avons simplement représenté sur un chronogramme les valeurs des opérandes ainsi que du résultat obtenu. Nous n’avons pas également représenté l’ensemble des instructions ici.

Soit le programme suivant ;

Une image contenant table

Description générée automatiquement

Le chronogramme qui en résulte est le suivant. Nous pouvons constater qu’on y retrouve bien les résultats attendus.

Une image contenant table

Description générée automatiquement

### Memory Access

Étant donné que cette partie n’a pas été implémenté par nous, nous n’avons pas jugé pertinent de tester le fonctionnement détaillé de ce circuit car nous supposons que ce dernier a été testé en amont. Toutefois, la partie qu’il est important de vérifier est que le DECODE des instructions faisant intervenir la mémoire de donnée soient testées correctement. Ce test est effectué dans la partie précédente

## Feedback

Le principal feedback reçu pour cette partie venait de notre gestion de l’ALU. La première était une erreur de notre part sur la gestion du borrow lors d’une soustraction. En effet, nous avions inversé cette valeur en sortie pour le carry alors que le bloc logisim inverse déjà cette valeur pour nous. Cette erreur a donc été corrigée.

Nous sélectionnons à l’aide de multiplexeurs la valeur de sortie du carry et de l’overflow. Toutefois, l’implémentation actuelle aurait pu être simplifiée. Nous n’avons toutefois engagé de temps à cette optimisation, car malgré une légère augmentation de la complexité, le composant fonctionne correctement.

# Conclusion

Au travers de ce rapport, nous réalisons le défi complexe qu’est la réalisation d’une architecture de processeur, et à quel point chaque partie est critique pour le bon fonctionnement de l’ensemble. Nous comprenons désormais mieux le rôle de chaque section d’un processeur ainsi que l’imbrication entres elles. Notons toutefois que ce rapport est une exposition simplifiée et, avant tout, pédagogique d’un processeur, et que la réalité est parfois bien lien d’une simple représentation Logisim…

Date : 03 mai 2022

Noms des étudiants : Bastian Chollet Kevin Ferati