Polytech 2024-2025 SI3 FISA

PROJET PARM

Compte rendu du travail réalisé

Jana Saad, Marion Valls, Maud Marconcini, Salma Idmansour

Table des matières

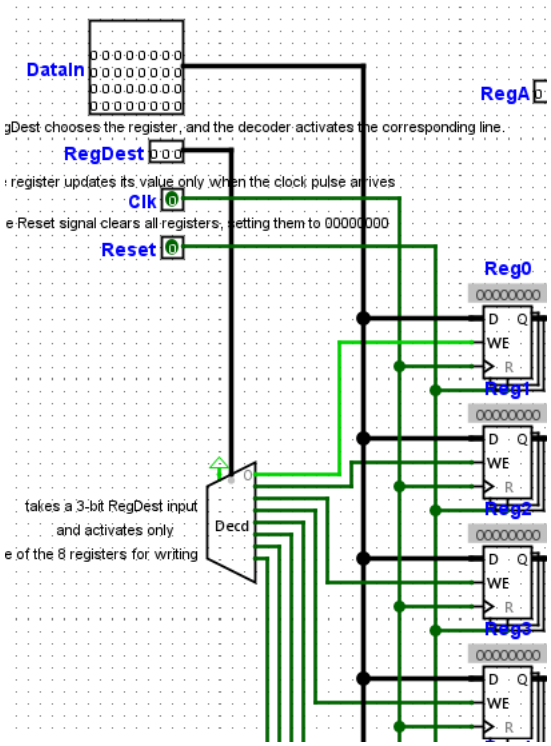
Banc de Registre

Ecriture

Ce circuit utilise un **décodage d'adresse** pour activer l'écriture dans un registre spécifique.

- Le signal **RegDest (3 bits)** est envoyé à un **décodage binaire à 8 sorties**.
- Chaque sortie du décodeur est connectée au **signal Write Enable (WE) d'un registre**.
- Lorsqu'un registre est sélectionné, et que le signal d'horloge (Clk) est actif, la valeur de **DataIn** est stockée dans ce registre.
- Si le signal **Reset** est activé, tous les registres sont remis à zéro.

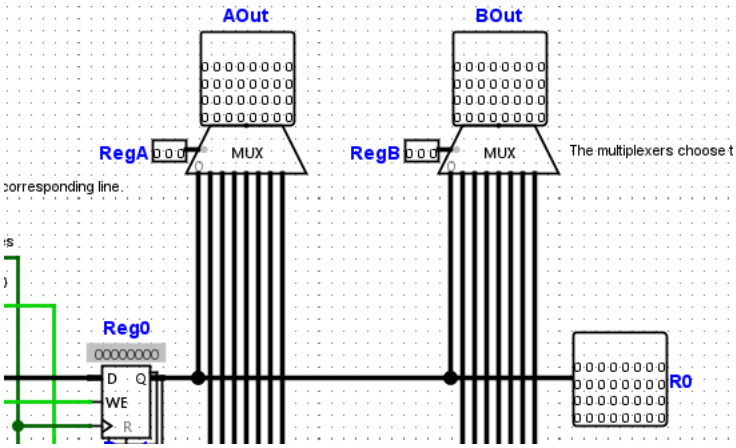
Ce mécanisme garantit que **seul le registre sélectionné** est modifié, évitant toute corruption de données.



Lecture

Ce circuit permet également la lecture des registres via les **multiplexeurs (MUX)** connectés aux sorties **RegA et RegB**.

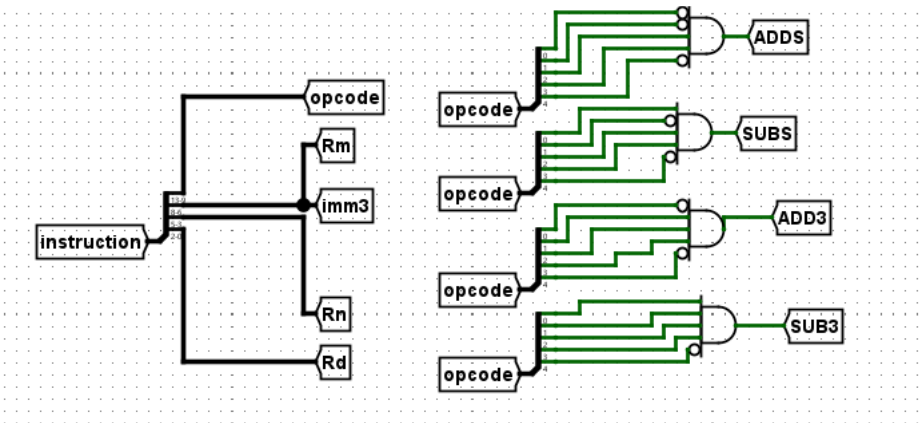
- Les signaux **RegA et RegB (3 bits chacun)** sélectionnent **deux registres** à lire.
- Les multiplexeurs transmettent les valeurs des registres sélectionnés aux sorties **AOut et BOut**.
- Cela permet au processeur d'accéder rapidement aux données stockées pour les opérations arithmétiques ou logiques.



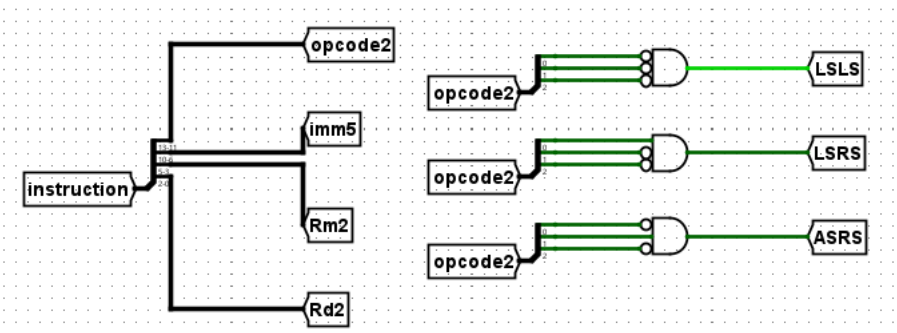
Shift Add Sub Move

On a regroupé nos instructions selon 3 catégories :

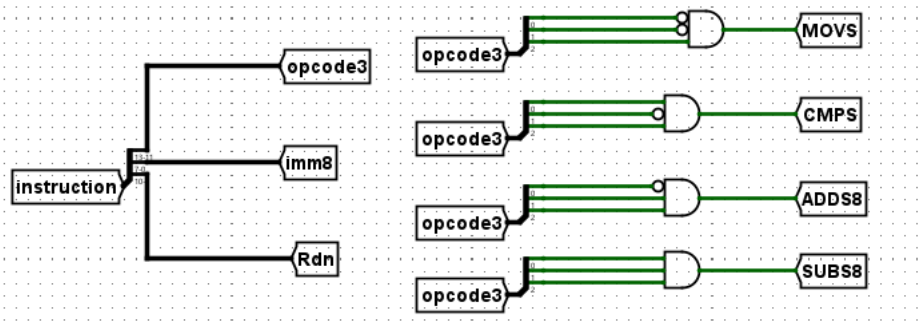
1. Instructions dont l'opcode est codé sur 5 bits (ex : Add register, substract register, add 3 bit immediate et substract 3 bit immediate)



2. Instructions dont l'opcode est codé sur 3 bits ne dépendant que de 2 opérandes (ex : Move, compare, add 8bit immediate et substract 8 bit immediate)

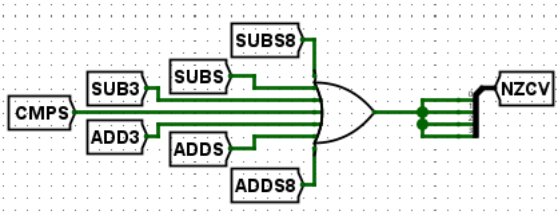


3. Instructions dont l'opcode est codé sur 3 bits et dépendant de 3 opérandes (ex : Logical Shift Left, Logical Shift Right et Arithmetic Shift Right)

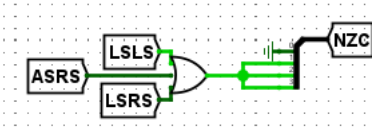


De même, pour mettre à jour les flags, on les a regroupé selon 3 catégories :

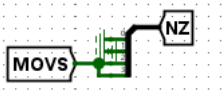
1. **Flags complets (NZCV)** : Certaines instructions affectent tous les flags (Negative, Zero, Carry, Overflow).



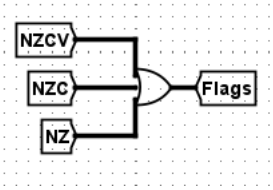
2. **3 Flags (NZC)** : D'autres n'affectent que les flags Negative, Zero et Carry.



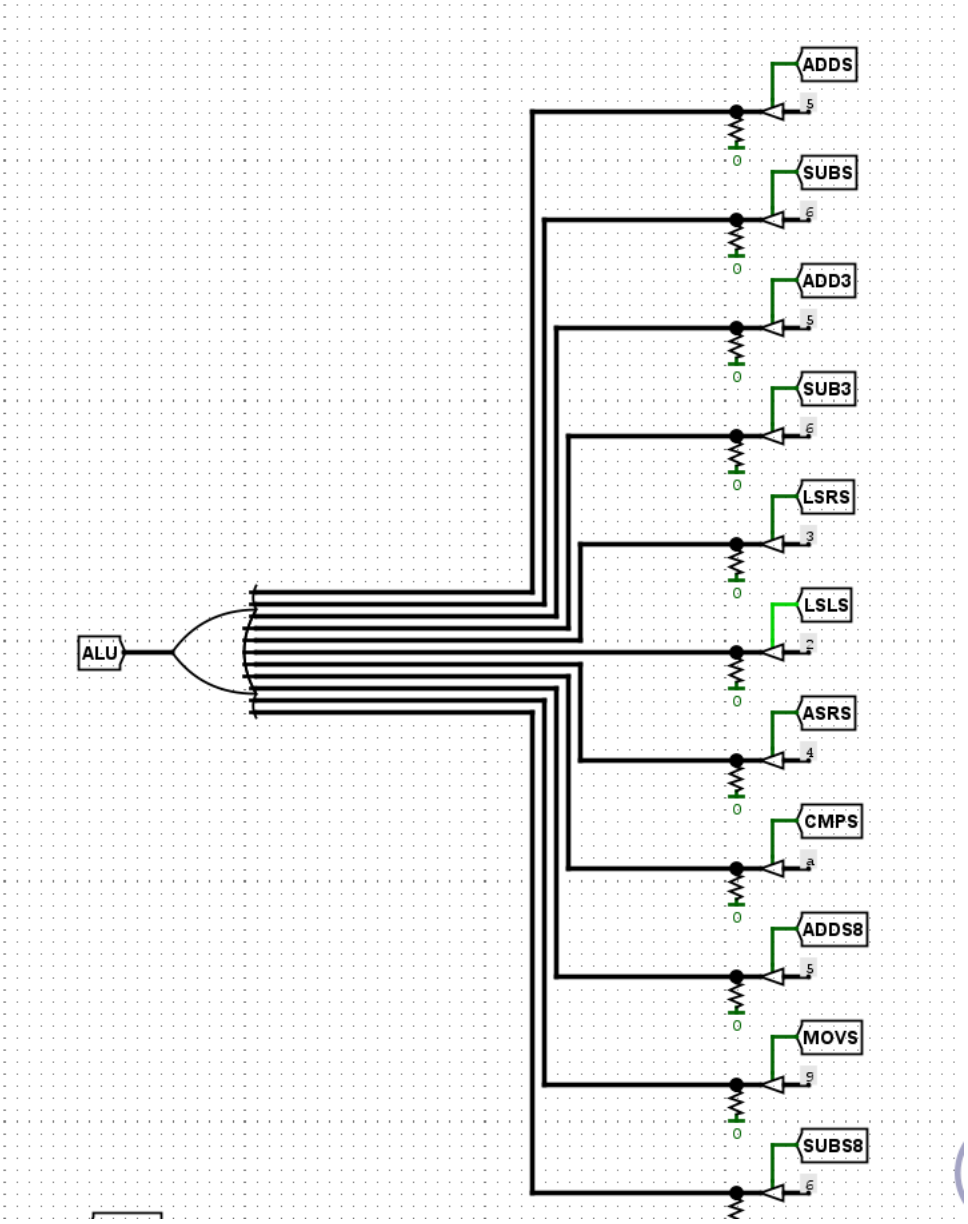
3. **2 Flags (NZ)** : Certaines instructions ne modifient que 2 flags.



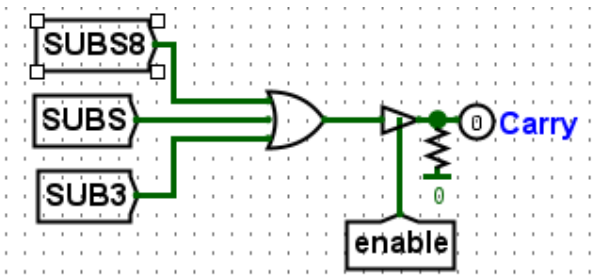
Selon l'instruction séléctionné les bon falgs sont mis à jour :



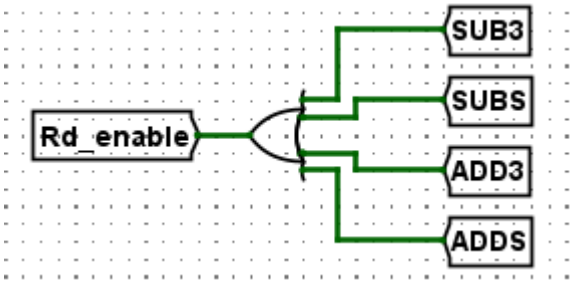
Les constantes associées aux instructions (ADDS, SUBS, LSLS, etc.) correspondent à la conversion en décimal de l'opcode binaire de chaque instruction. Cela signifie que chaque sortie de l'ALU est activée en fonction d'un code binaire spécifique qui, lorsqu'il est converti en décimal, correspond à une des instructions listées.



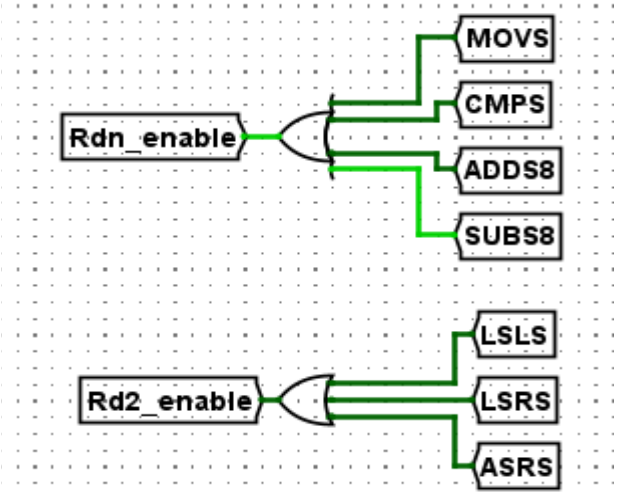
La sortie Carry force la valeur de la retenue entrante pour l'ALU. Carry est à 1 si l'opcode de l'instruction correspond à celui de SUBS,SUB3 ou SUBS8 et si Enable est à 1. On utilise donc un **Buffer Contrôlé** qui permet le signal si enable est à 1, le bloque sinon.



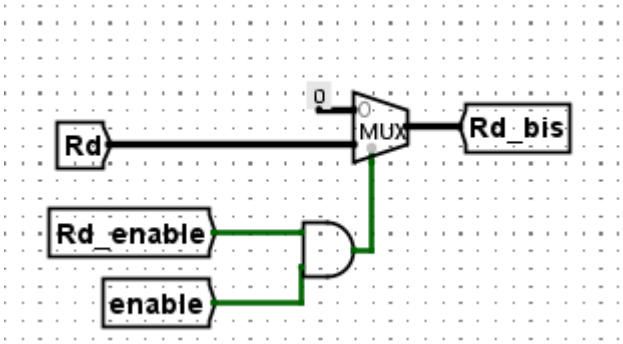
Sortie Rd (même logique pour Rn) :



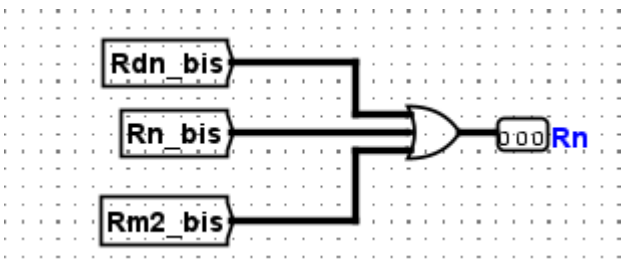
Rd est un opérande de SUBS/3 et ADDS/3, on crée donc une étiquette Rd_enable qui permet de savoir si on manipule une des instructions qui utilise Rd.
Rd_enable est la sortie de la disjonction entre SUB3, SUBS, ADD3, ADDS.
On applique la même logique à Rdn et Rd2 et on obtient donc Rdn_enable et Rd2_enable.



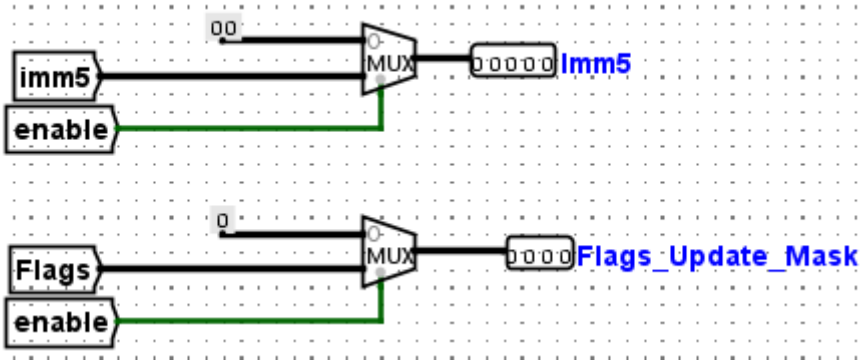
Ensuite, on ne veut utiliser le registre Rd (Rd2/Rdn) que si Enable est à 1.



Rd_bis ne prend la valeur de Rd que si enable et Rd_enable sont vrais, c'est-à-dire que si on a une des quatre instructions ADDS, ADD3, SUBS, SUB3 et que enable est à 1.
La même logique s'applique à Rd2_bis et Rdn_bis.
Enfin, la sortie Rd est la disjonction entre Rd, bis, Rd2_bis et Rdn_bis.

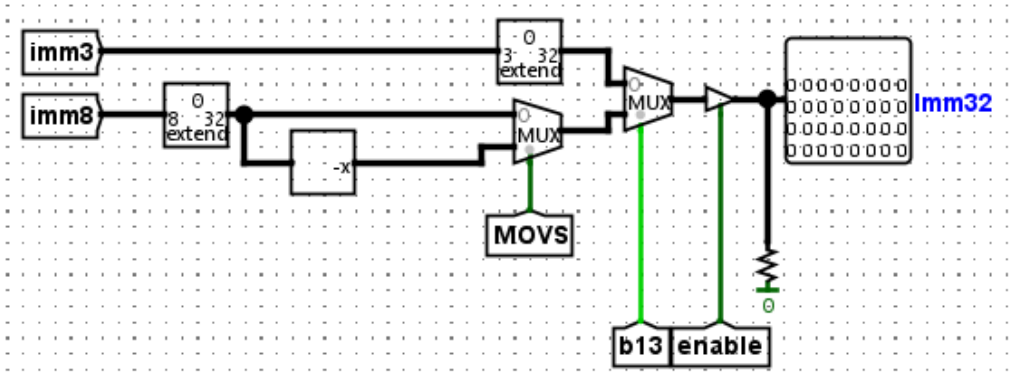


Imm5 et Flags_Update_Mask récupèrent les valeurs de imm5 et Flags respectivement que si enable est à 1. On utilise donc un multiplexeur qui renvoie Imm5 (resp. Flags) à la sortie Imm5(resp. Flags_Update_Mask) en considérant enable comme bit de sélection.



Imm32 :

La sortie Imm32 est utilisée pour communiquer les valeurs des immédiats de MOV, CMP, ADD3 et SUB3 à l'ALU, donc imm3 et imm8.



On utilise un étendeur de bits pour étendre les bits de imm3 et imm8 de 3 et respectivement 8 bits à 32.

On remarque que le bit 13 permet de différencier entre CMP ou MOV et ADD3 ou SUB3 Si le bit 13 de l'instruction est à 1, on est donc dans une instruction MOVS ou CMP, sinon on manipule ADD3 ou SUB3. Le bit 13 est donc le bit de sélection entre imm3 et imm8 dans le multiplexeur.

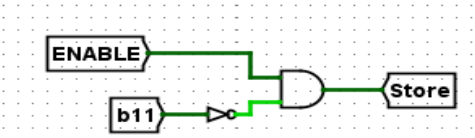
Si l'instruction est MOVS, imm8 doit être inversé, on utilise pour cela un inverseur.

Enfin, après avoir identifié quel immédiat on manipule, on utilise un buffer contrôlé pour ajouter la contrainte de enable avant d'envoyer les valeurs à la sortie Imm32.

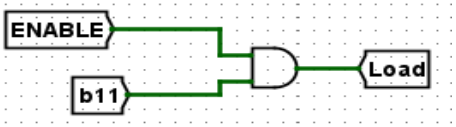
LOAD AND STORE

Load and store assure la gestion des accès mémoire (lecture/écriture), du pointeur de pile (StackPointer) et du maintien du compteur de programme. Il utilise des portes logiques et des multiplexeurs pour gérer les instructions de manière efficace et garantir une synchronisation correcte avec la mémoire.

Le 11ème bit (b11 dans notre circuit) décide s'il s'agira d'un écriture ou lecture en mémoire.

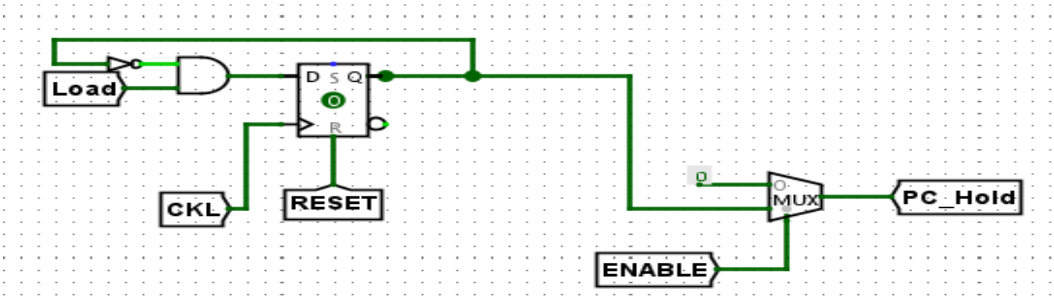


Si **b11 = 0**, le signal **Store** est activé (écriture en mémoire).

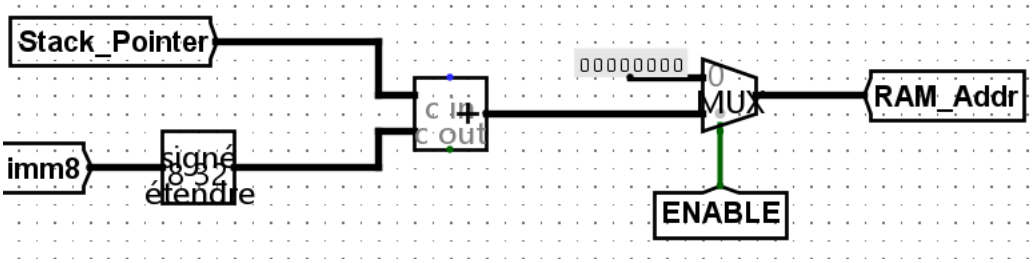


Si **b11 = 1**, le signal **Load** est activé (lecture en mémoire).

Notre circuit utilise une **bascule D (D Flip-Flop)** pour gérer le signal **PC_Hold**. La bascule D stocke et maintient l'état du signal en fonction de l'horloge (**CLK**). Lorsque **Load** est actif et qu'un front montant de l'horloge se produit, la valeur en entrée **D** est transférée à la sortie **Q**, maintenant ainsi l'état de **PC_Hold**. Ce mécanisme permet de retarder l'exécution d'une instruction afin de laisser à la mémoire RAM le temps de fournir les données demandées. De plus, un signal **RESET** permet de forcer la sortie à 0, assurant ainsi une remise à zéro du système lorsque nécessaire.



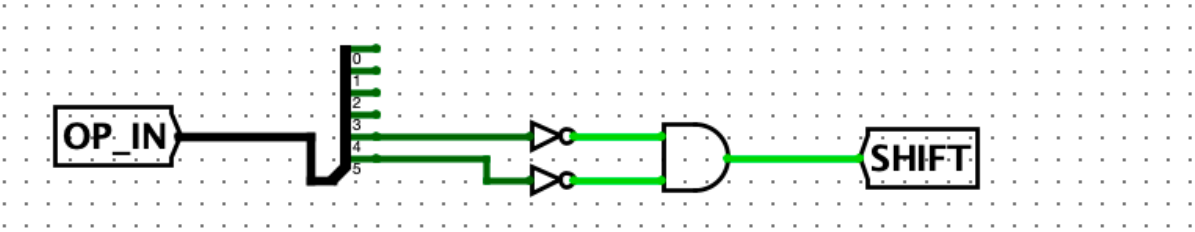
La RAM-Addr est calculée en fonction du *Stack Pointer* et de l'*imm8*. L'immédiat est étendu en 32 bits avant d'être additionné au Stack Pointer. Un multiplexeur *MUX* sélectionne ensuite entre le résultat de cette addition et une adresse fixe (00000000), selon l'état du signal d'activation (*ENABLE*).



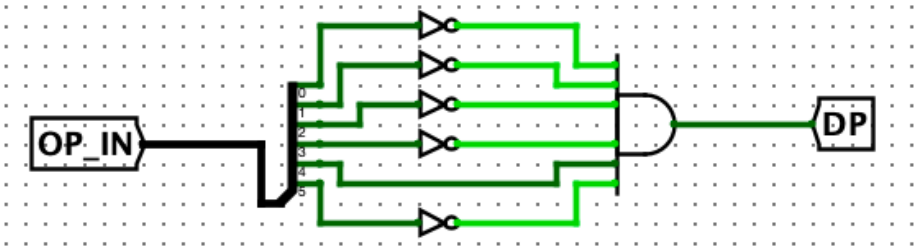
Opcode :

Le décodeur d'instructions analyse l'Opcode (6 bits) pour activer un bloc fonctionnel : Shift, Data Processing, Load/Store, SP Address ou Conditional. Chaque sortie correspond à une catégorie d'instruction, comme les décalages pour Shift ou les branches conditionnelles pour Conditional. Les portes logiques (AND, OR, NOT) décodent les bits spécifiques pour activer la sortie correspondante. Ce mécanisme garantit une exécution précise et simplifie la gestion des instructions dans les architectures de processeurs.

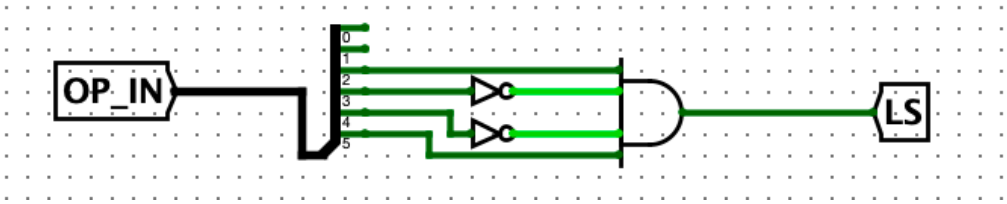
Ce circuit logique active la sortie SHIFT lorsqu'une instruction de type "SHIFT, ADD, SUB, MOV" est détectée. Il analyse les bits 4 et 5 de l'entrée OP_IN, en exigeant que les bits 4 et 5 soient inactifs (0). Les portes NOT inversent les bits 4 et 5, et une porte AND combine ces conditions.



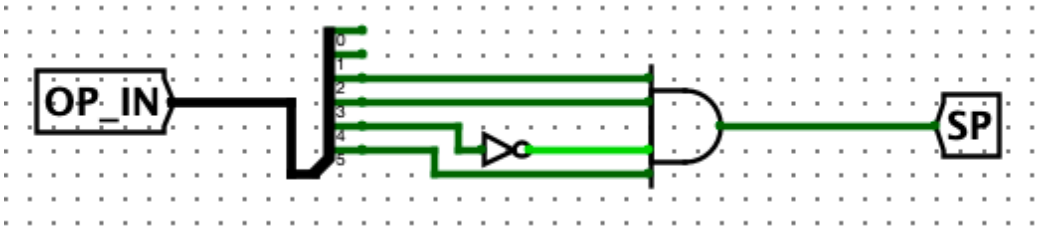
Ce circuit active la sortie DP (Data Processing) lorsque l'Opcode correspond à une instruction de traitement de données. Les bits de OP_IN sont inversés pour vérifier qu'ils sont à 0 sauf le bit 4 qui doit être à 1. Une porte AND combine ces conditions pour détecter précisément cette configuration, ça permet d'activer uniquement le bloc fonctionnel dédié aux instructions Data Processing.



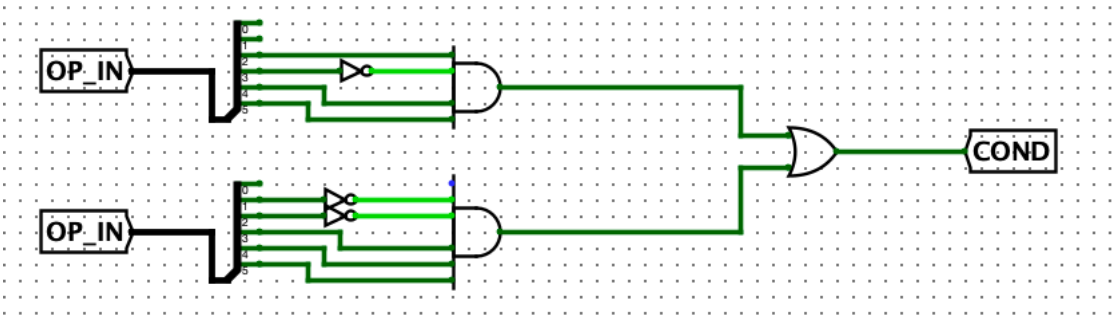
Ce circuit active la sortie LS (Load/Store) pour les instructions de transfert de données. Les bits 2 et 5 de OP_IN doivent être actifs (1), tandis que les bits 3 et 4 doivent être inactifs (0), vérifiés à l'aide de portes NOT. Une porte AND combine ces conditions pour détecter la configuration spécifique. Si toutes les conditions sont remplies, le bloc fonctionnel LS est activé.



Ce circuit active la sortie SP pour les instructions liées au Stack Pointer. Les bits 2, 3 et 5 de OP_IN doivent être actifs (1), tandis que le bit 4 doit être inactif (0), vérifié par une porte NOT. Une porte AND combine ces conditions pour détecter cette configuration spécifique. Si toutes les conditions sont remplies, le bloc fonctionnel SP est activé.



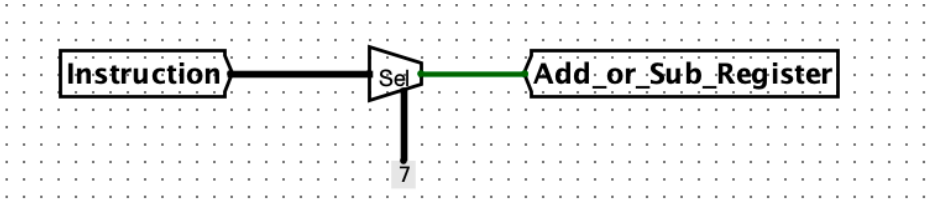
Ce circuit active la sortie COND (Conditional) pour des instructions de branchement conditionnel en combinant deux ensembles de conditions. Le premier ensemble utilise une porte AND pour vérifier que les bits 2, 4 et 5 sont actifs (1) et que le bit 3 est inactif (0). Le second ensemble vérifie une configuration différente en utilisant les bits 1, 2, 3, 4 et 5. Les deux ensembles sont reliés à une porte OR, ce qui permet d'activer COND si l'une ou l'autre des configurations est remplie. Cette structure permet de gérer plusieurs types d'instructions conditionnelles.



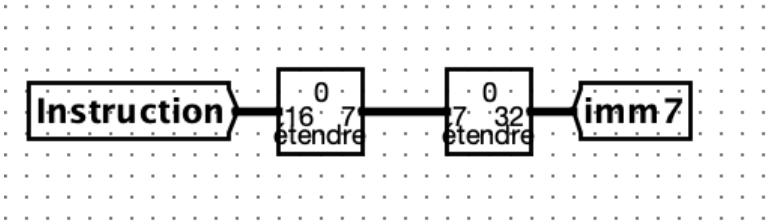
Sp-Address :

Le controller SP Address met à jour le pointeur de pile (Stack_Pointer) selon l'instruction reçue. L'opération est déterminée par le décodage de l'Instruction. Si Enable est actif, le nouveau pointeur calculé est envoyé à la sortie New_Stack_Pointer, et l'écriture est validée via Write_Enable. Si Enable est désactivé, les sorties restent à 0. Ce mécanisme garantit une mise à jour précise et conditionnelle du pointeur de pile.

Ici ça sélectionne une opération (addition ou soustraction) en fonction du bit 7 de l'Instruction. Le bit 7 agit comme signal de sélection pour le multiplexeur. Si le bit 7 est à 0, l'opération choisie est une addition, et s'il est à 1, une soustraction. La sortie Add_or_Sub_Register reflète l'opération sélectionnée. Ce mécanisme permet un choix rapide et conditionnel entre les deux opérations.



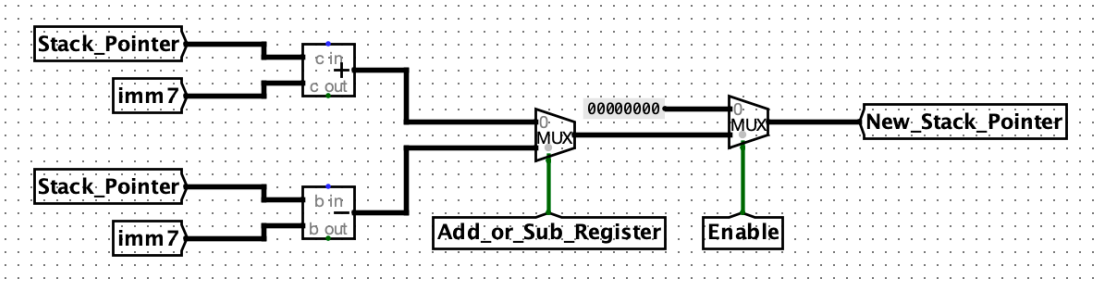
Ici ça extrait un immédiat de 7 bits (imm7) à partir d'une instruction de 16 bits. Les bits correspondants sont d'abord étendus de 7 à 16 bits, puis de 16 à 32 bits pour garantir la compatibilité avec les opérations de calcul. Chaque étape utilise une extension de signe ou de zéro selon les besoins. La sortie imm7 est ainsi un immédiat de 32 bits, prêt à être utilisé dans des calculs ou opérations.



Aussi :

Diviser l'extension en deux étapes (7→16 puis 16→32 bits) garantit modularité et compatibilité avec des blocs logiques standards. Cela facilite le débogage, la réutilisation des étapes intermédiaires et rend le schéma plus clair. Certaines architectures ne permettent pas une extension directe de 7 à 32 bits. Si ces contraintes n'existent pas, un seul bloc pourrait être utilisé. Cette configuration est souvent un choix pratique ou lié aux outils disponibles.

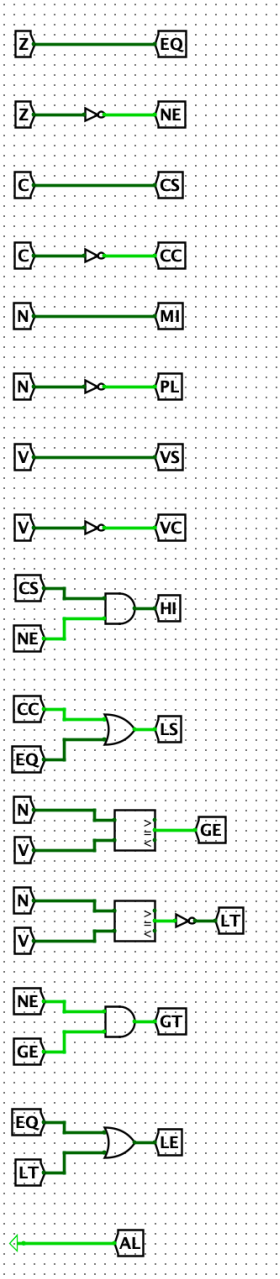
Ce circuit calcule la nouvelle valeur du Stack_Pointer (New_Stack_Pointer) en fonction d'une opération d'addition ou de soustraction avec imm7. Les blocs + et - effectuent respectivement l'addition et la soustraction entre Stack_Pointer et imm7. Un multiplexeur utilise Add_or_Sub_Register pour choisir l'opération. Un second multiplexeur contrôle l'écriture de la nouvelle valeur en fonction du signal Enable. Si Enable est désactivé, la sortie reste à 0, garantissant une gestion conditionnelle du pointeur de pile.



Conditional :

Le **contrôleur Conditional** gère l'**exécution conditionnelle des instructions** en fonction des **flags**(Zéro, Négatif, Overflow, Carry). Il vérifie si une condition est remplie (ex : égalité, supérieur, inférieur) avant d'autoriser l'exécution d'une instruction. Cela permet d'**optimiser le flux d'instructions** en évitant des branchements inutiles. En résumé, il décide **si une instruction doit être exécutée ou ignorée** selon l'état du processeur.

Ces curcuits génèrent les conditions de branchement en fonction des flags.



code	symbole	signification	drapeaux
0000	EQ	égalité	Z == 1
0001	NE	différence	Z == 0
0010	CS ou HS	retenue	C == 1
0011	CC ou LO	pas de retenue	C == 0
0100	MI	négatif	N == 1
0101	PL	positif ou nul	N == 0
0110	VS	dépassement de capacité	V == 1
0111	VC	pas de dépassement de capacité	V == 0
1000	HI	supérieur (non signé)	C == 1 et Z == 0
1001	LS	inférieur ou égal (non signé)	C == 0 ou Z == 1
1010	GE	supérieur ou égal (signé)	N == V
1011	LT	inférieur (signé)	N != V
1100	GT	supérieur (signé)	Z == 0 et N == V
1101	LE	inférieur ou égal (signé)	Z == 1 ou N != V
1110	AL	toujours vrai	

Ce circuit gère l'**exécution conditionnelle des branchements** en fonction des **drapeaux du processeur**.

Multiplexeur principal (MUX) :

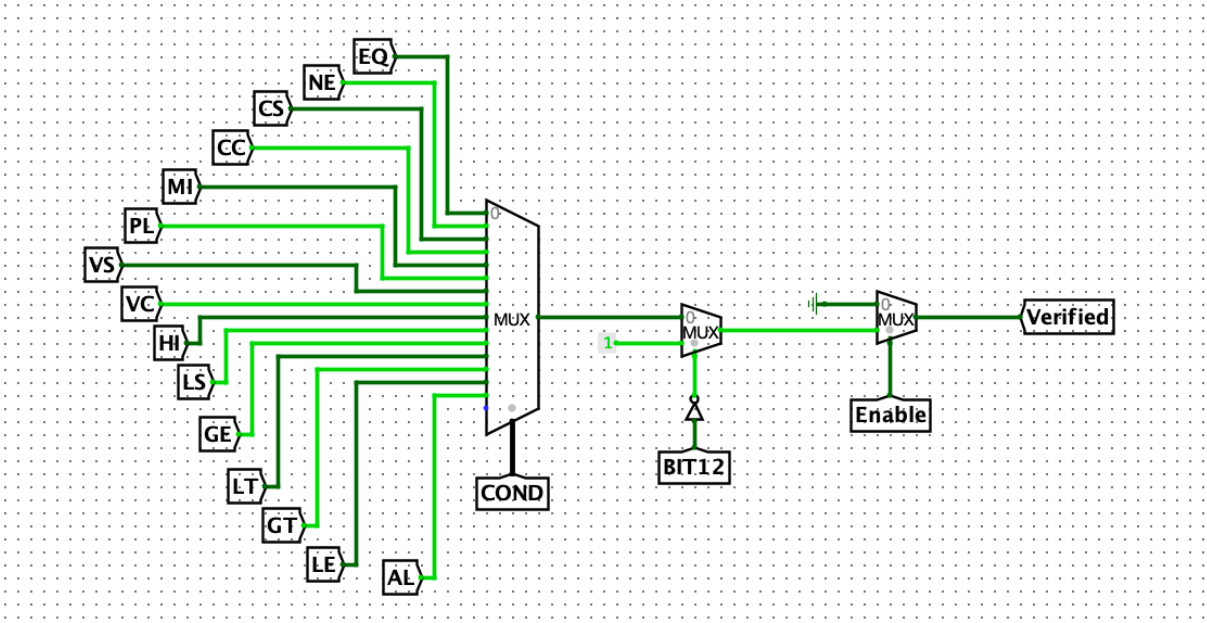
- Il prend plusieurs entrées (EQ, NE, CS, CC...), qui représentent les conditions de branchement (égalité, supérieur, inférieur, etc.).
- Le **signal COND** détermine **quelle condition est sélectionnée** pour être évaluée.

Signal BIT12 :

- Si BIT12 est actif, un second **MUX** force la sélection de la condition **"toujours vrai" (AL)**, permettant un branchement inconditionnel.

Multiplexeur de validation :

- Une fois qu'une condition est validée, **un dernier MUX vérifie Enable**.
- Si Enable = 0, la sortie **Verified est forcée à 0**, empêchant le branchement.
- Si Enable = 1, la condition sélectionnée est transmise pour exécution.

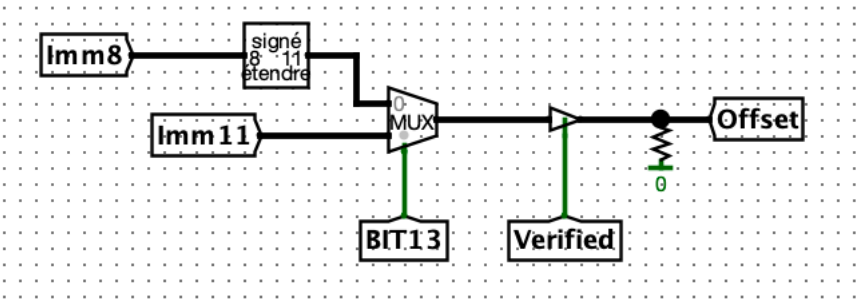


Multiplexeur (MUX) :

- **Rôle :** Choisir entre deux valeurs d'immédiats (Imm8 et Imm11).
- **Contrôle :** Le signal **BIT13** détermine le choix :
    - Si BIT13 = 0 : le MUX sélectionne Imm8 comme offset, étendu à 11 bits (pour correspondre au format attendu).
    - Si BIT13 = 1 : le MUX sélectionne Imm11.

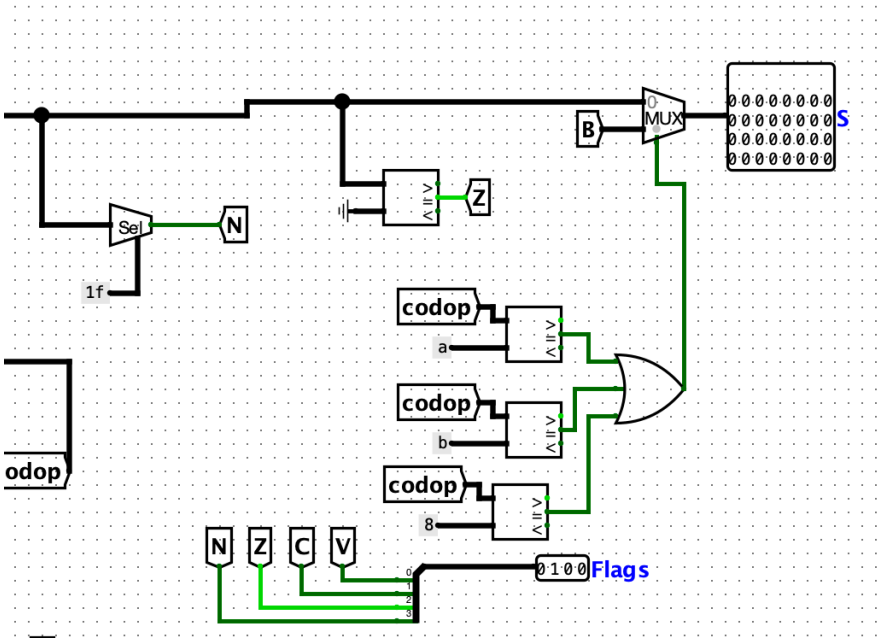
Signal Verified :

- **Rôle :** Vérifier si le branchement doit être effectué.
- **Fonctionnement :**
    - Si Verified = 1 : l'**offset** choisi est appliqué, modifiant l'adresse du **Program**.
    - Si Verified = 0 : une **résistance de pull-down** force l'offset à 0, annulant tout branchement.

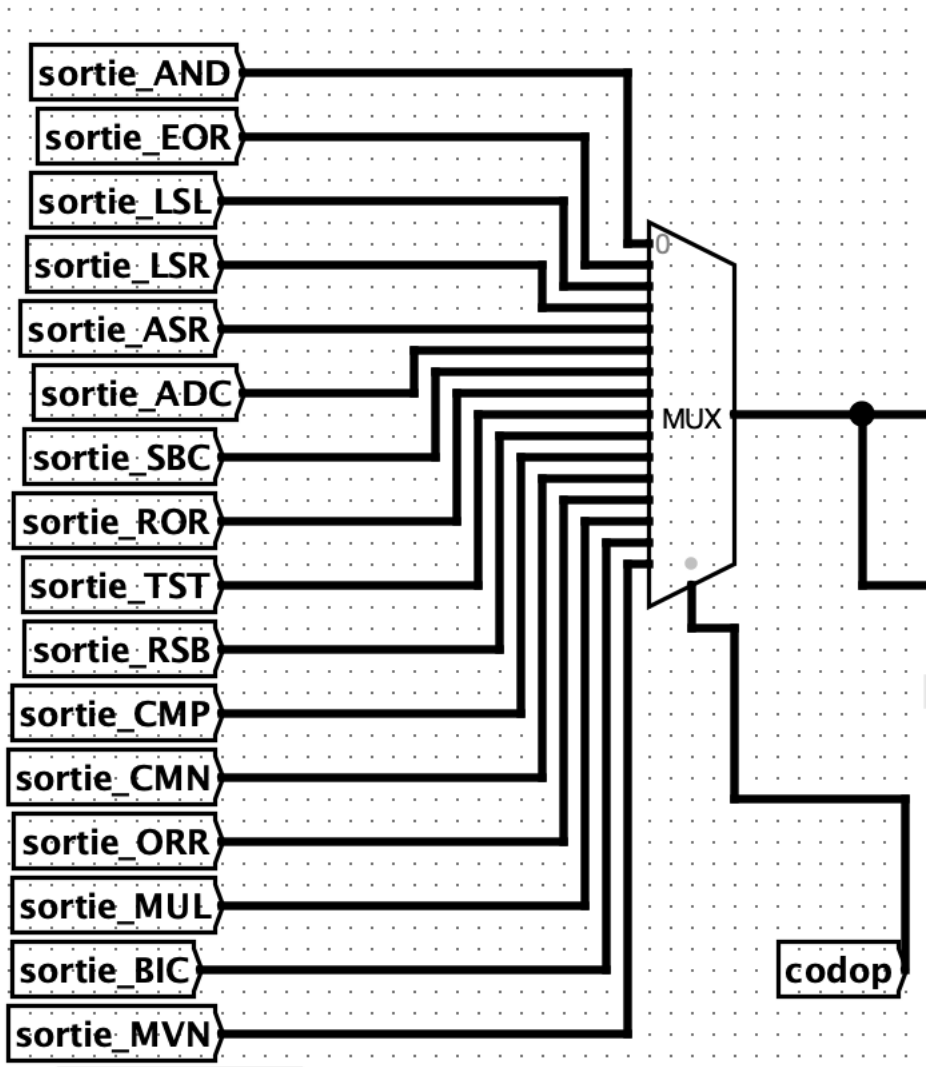


ALU

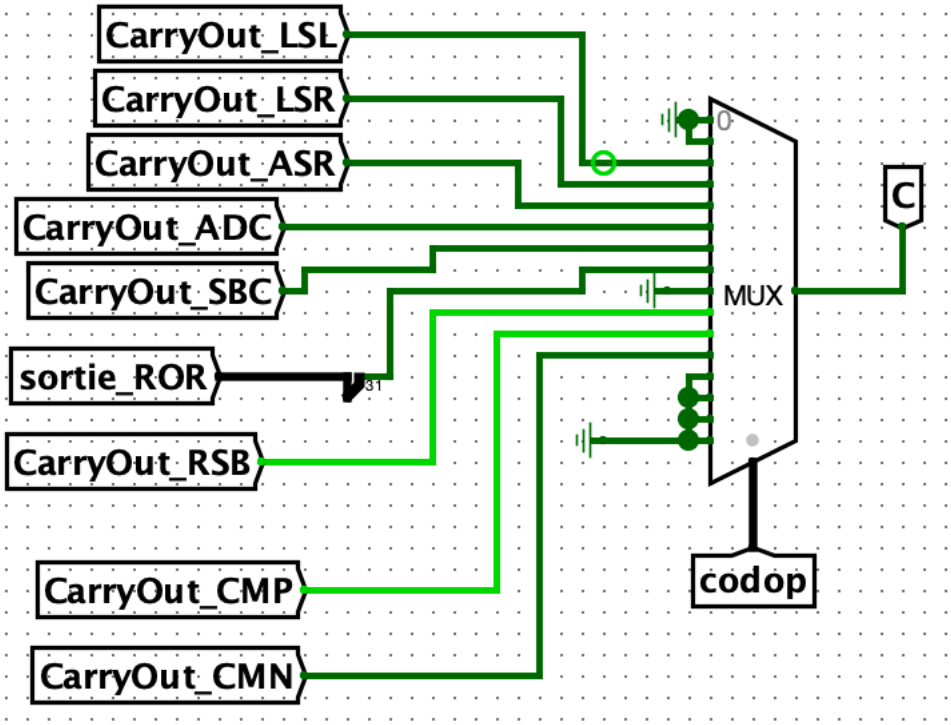
L'ALU ou l'Arithmetic Logic Unit gère les calculs dans un processeur. Il est responsable d'un certain nombre d'opérations mathématiques. Il doit également gérer les effets de bord comme la retenu, le dépassement de capacité, les négatifs et le zéro.



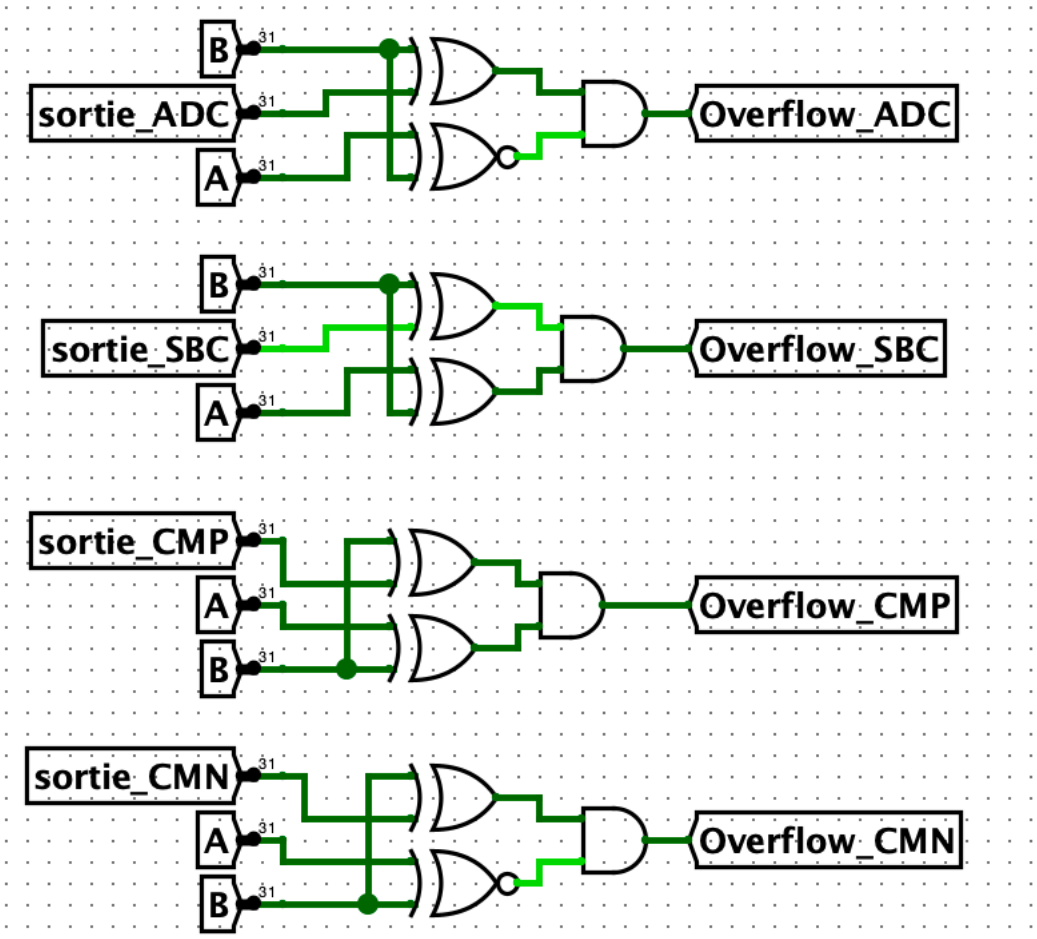
Commençons par la gestion de la sortie. Ici nous pouvons d'abord voir que nous effectuons une vérification sur 3 opcodes. En effet c'est dernier ne renvoie pas de résultat, nous devons renvoyer l'entrée B. Nous voyons également la gestion du zéro et du négatif.



Ici nous voyons rapidement que nous avons choisi une logique ou nous effectuons toutes les actions mais nous choisissons le résultat en fonction de l'opération désirée.

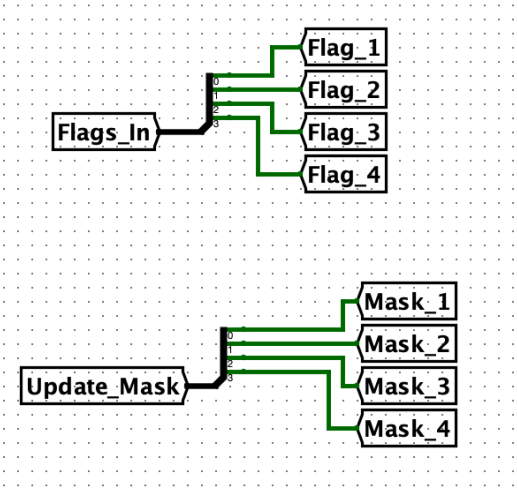


Ici nous appliquons la même logique que précédemment mais pour les retenus sortantes. Les opérations n'ayant pas de sortie de retenu sont mise à zéro.



Enfin pour ce qui est de la gestion du dépassement de capacité, nous regardons les derniers des nombres d'entrées et celui de la sortie est vérifions de quoi il résulte. Par exemple pour l'addition si le dernier bit de A et B faut 1 et que celui de la sortie vaut 0 alors il a eu un dépassement.

Flags APSR

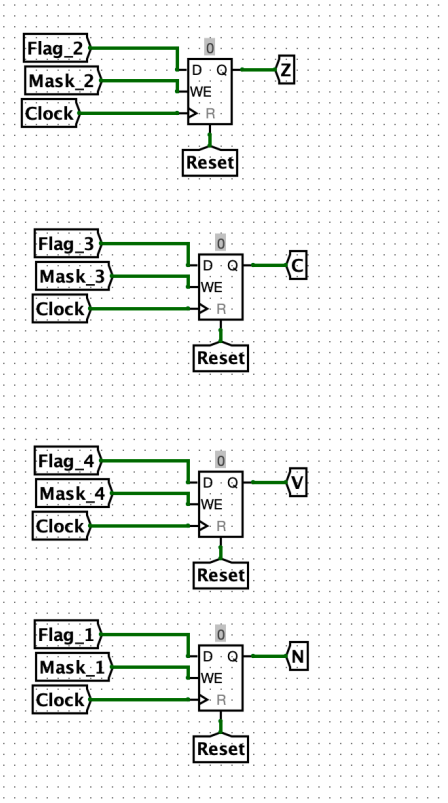


Le bloc *Flags APSR* correspond aux 4 bits de poids fort du registre **Application Program Status Register (APSR)** de l'architecture ARM. Il conserve les drapeaux générés par la dernière instruction afin qu'ils soient disponibles pour la prochaine instruction et puissent influencer l'exécution du programme.

L'entrée **Update_Mask** permet de réinjecter l'ancien état d'un drapeau si le bit correspondant est à **0**. Si le bit est à **1**, le drapeau est mis à jour. Cette approche permet de ne modifier que les bits nécessaires tout en conservant les valeurs précédentes des autres drapeaux.

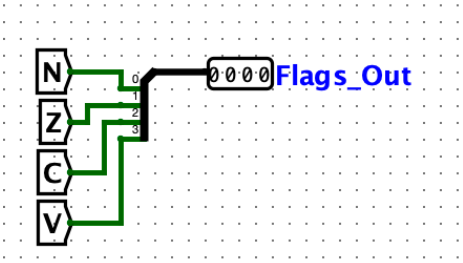
J'ai commencé par séparer les bits des drapeaux et des masques correspondants (cf. schéma ci-contre).

Ensuite, j'ai utilisé des registres pour constituer chaque flag séparément :



- **N (Negative)** : Indique si le résultat d'une opération est négatif.
- **Z (Zero)** : Indique si le résultat est nul.
- **C (Carry)** : Utilisé pour indiquer un dépassement lors d'une addition ou une retenue lors d'une soustraction.
- **V (Overflow)** : Détecte un dépassement de capacité dans les opérations arithmétiques signées.

Enfin, j'ai réuni les drapeaux obtenus ensemble pour former la sortie **Flags_Out**, qui pourra être exploitée par les instructions suivantes.

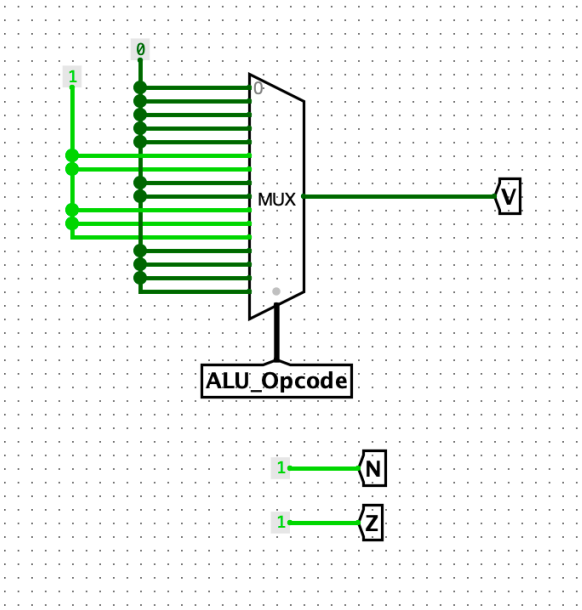


Data processing

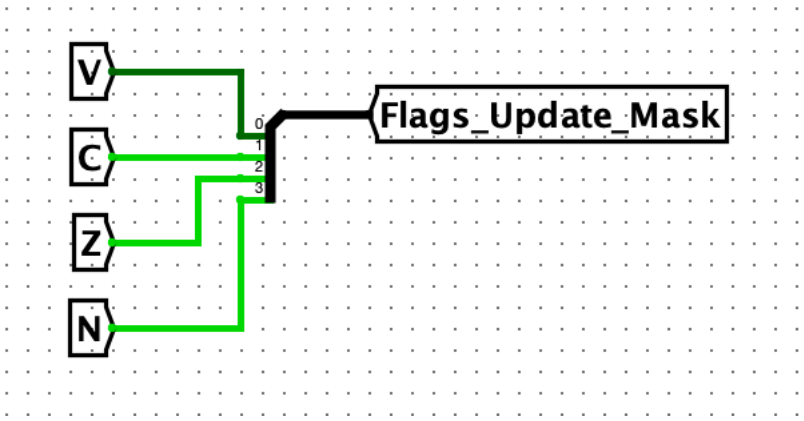
Le but principal de ce bloc est de gérer les calculs et manipulations de données en fonction des instructions qui lui sont envoyées.

Ici, le masque de certains flags est forcé à 1. De plus, parmi les instructions, seules 5 d'entre elles vont modifier V (elles sont donc forcées à 1 au niveau du multiplexeur à l'opcode correspondant). J'ai déduit cela du tableau suivant :

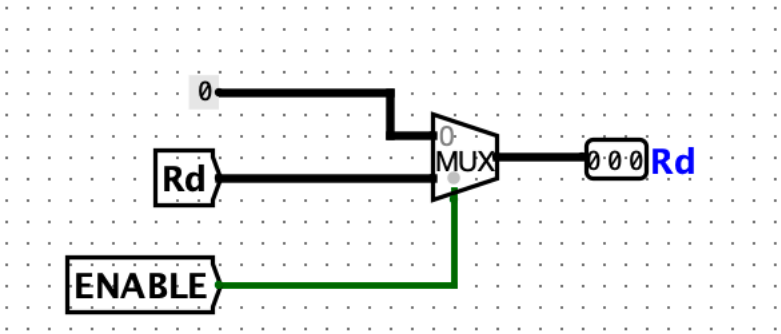
Ref Doc		Description	UAL Code			Bits														Flags			
			Instruction	Operandes		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	C	V	N	Z
		Data Processing										opcode													
p.35 11.1.2	1	Bitwise AND	ANDS	Rdn	Rm	0	1	0	0	0	0	0	0	0	0	Rm				Rdn		0		x	x
	2	Exclusive OR	EORS	Rdn	Rm	0	1	0	0	0	0	0	0	0	1	Rm				Rdn		1		x	x
	3	Logical Shift Left	LSLS	Rdn	Rm	0	1	0	0	0	0	0	0	1	0	Rm				Rdn		x		x	x
	4	Logical Shift Right	LSRS	Rdn	Rm	0	1	0	0	0	0	0	0	1	1	Rm				Rdn		x		x	x
	5	Arithmetic Shift Right	ASRS	Rdn	Rm	0	1	0	0	0	0	0	1	0	0	Rm				Rdn		x		x	x
	6	Add with Carry	ADCS	Rdn	Rm	0	1	0	0	0	0	0	1	0	1	Rm				Rdn		x	x	x	x
	7	Substract with Carry	SBCS	Rdn	Rm	0	1	0	0	0	0	0	1	1	0	Rm				Rdn		x	x	x	x
	8	Rotate Right	RORS	Rdn	Rm	0	1	0	0	0	0	0	1	1	1	Rm				Rdn		x		x	x
	9	Set Flags on bitwise AND	TSTS	Rdn	Rm	0	1	0	0	0	0	0	1	0	0	Rm				Rn		0		x	x
	10	Reverse Substract from 0	RSBS	Rd	Rn	0	1	0	0	0	0	1	0	0	1	Rn				Rd		0	x	x	x
	11	Compare Registers	CMPS	Rn	Rm	0	1	0	0	0	0	1	0	1	0	Rm				Rn		x	x	x	x
	12	Compare Negative	CMNS	Rn	Rm	0	1	0	0	0	0	1	0	1	1	Rm				Rn		x	x	x	x
	13	Logical OR	ORRS	Rdn	Rm	0	1	0	0	0	0	1	1	0	0	Rm				Rdn		0		x	x
	14	Multiply Two Registers	MULS	Rdm	Rn	Rdm	0	1	0	0	0	0	1	1	0	1	Rn				Rdm			x	x
	15	Bit Clear	BICS	Rdn	Rm	0	1	0	0	0	0	1	1	1	0	Rm				Rdn		0		x	x
	16	Bitwise NOT	MVNS	Rd	Rm	0	1	0	0	0	0	1	1	1	1	Rm				Rd		1		x	x



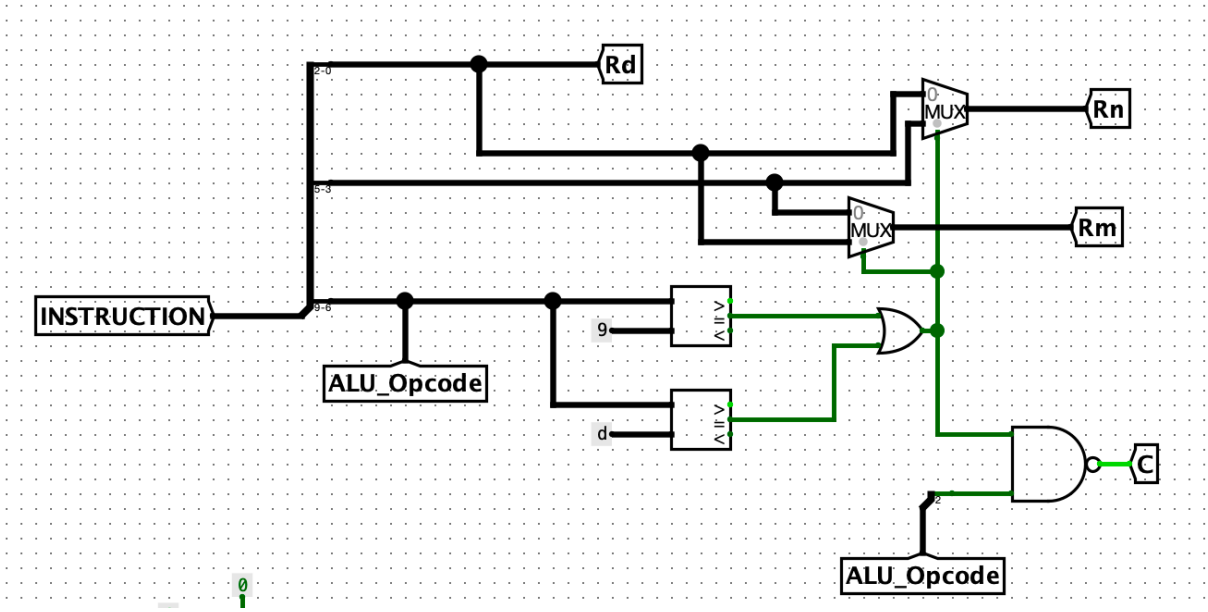
Les flags sont ensuite reconstitués à la fin. Il s'agit en fait ici d'un masque, c'est-à-dire que si le bit du masque est mis à 0 alors le flag correspondant ne changera pas. En revanche s'il vaut 1, il faudra mettre à jour sa valeur.



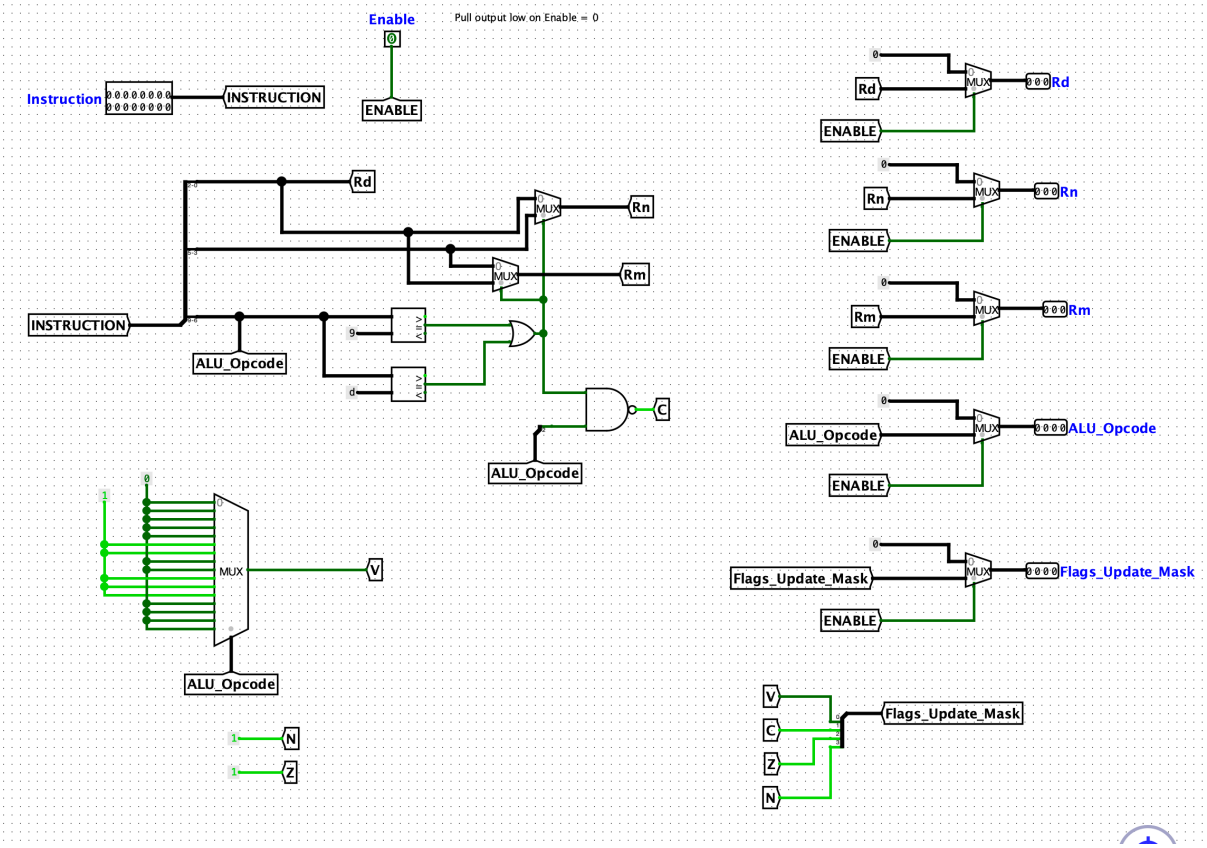
Pour chaque sortie, on utilise un multiplexeur pour gérer le fait que si Enable est à 0, on force l'ensemble des sorties à 0.



On implémente ici l'assignation des données dans les registres selon l'instruction. On en extrait l'opcode et on teste si l'opcode est égal à 9 ou d ce qui correspond aux instructions RSB et MUL.



Voici le composant en entier :



Programme Python

Dans le dossier **python-parsing**, il y a plusieurs fichiers :

- constants.py : définit les constantes du programme (les opcodes et les conditionnal) et prend en compte le decorator pour associer l'instruction test à son code binaire.
- opcodes_binary.py : convertit les instructions en binaire
- utils.py : contient les fonctions qui servent à analyser le fichier assembleur pour en extraire les instructions, les paramètres…
- main.py : qui comprend le code final qui affiche au fur et à mesure qu'il traite le document et appelle les fonctions pour coder en binaire puis en hexadécimal.

```
ldr r1, [sp, #12]
Instruction:  ldr
Parameters:  ['r1', '[sp', '#12]']
    Type of param r1:  register
    Type of param [sp:  label
    Type of param #12]:  immediate number
Binary:  1001100100000011
Hex:  0x9903


adds    r0, r0, r1
Instruction:  adds
Parameters:  ['r0', 'r0', 'r1']
    Type of param r0:  register
    Type of param r0:  register
    Type of param r1:  register
Binary:  0001100001000000
Hex:  0x1840
```

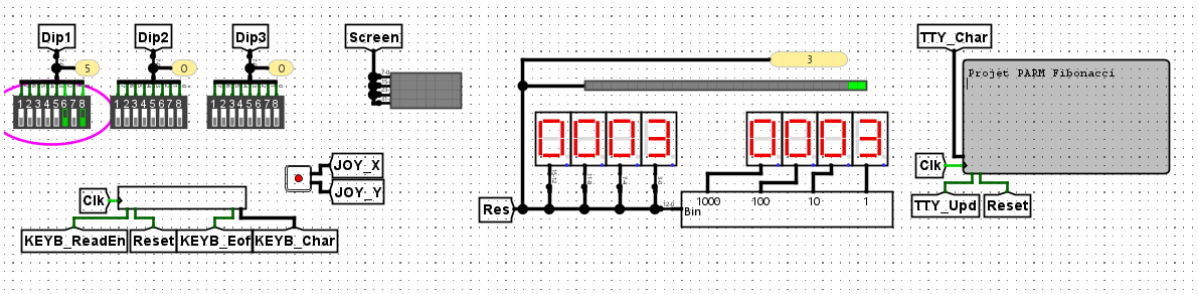
Affichage du programme pendant l'encodage.

Programme personnalisé en C (compilé puis traduit en .bin)

Nous avons choisi d'ajouter un programme qui nous permettait de tester un maximum des fonctionnalités proposées par notre processeur. Nous ne pouvions pas implémenter de tableaux ou de fonctions. Nous avons décidé de coder la suite de Fibonacci. Elle n'est pas codée en récursive car nous ne pouvons pas ajouter de fonctions. Voici le code :

```c
#include "parm.h"
#include "stdio.h"

#define N   DIP1

void run()
{
    BEGIN();

    int a = 0;      // Premier terme de Fibonacci
    int b = 1;      // Deuxième terme de Fibonacci
    int temp;
    int count = 0;  // Compteur pour afficher les termes
    int max = N;    // Nombre de termes à afficher (réglé par N)

    RES = N;

    PUTCHAR('P','r','o','j','e','t',' ','P','A','R','M', ' ', 'F','i','b','o','n','a','c','c','i', '\n');

    while (count < max)
    {
            RES = a;     // Affiche le terme courant
            count++;     // Incrémente le compteur
            temp = a;
            a = b;
            b = temp + b; // Mise à jour de a et b pour le prochain terme

    }

    END();
```

Il prend en entrée le x-ième nombre de fibonacci que l'on souhaite obtenir et utilise l'affichage numérique pour l'afficher et l'écran pour afficher un message.



Le fichier utilisé se trouve dans python_parsing/fibo.bin.