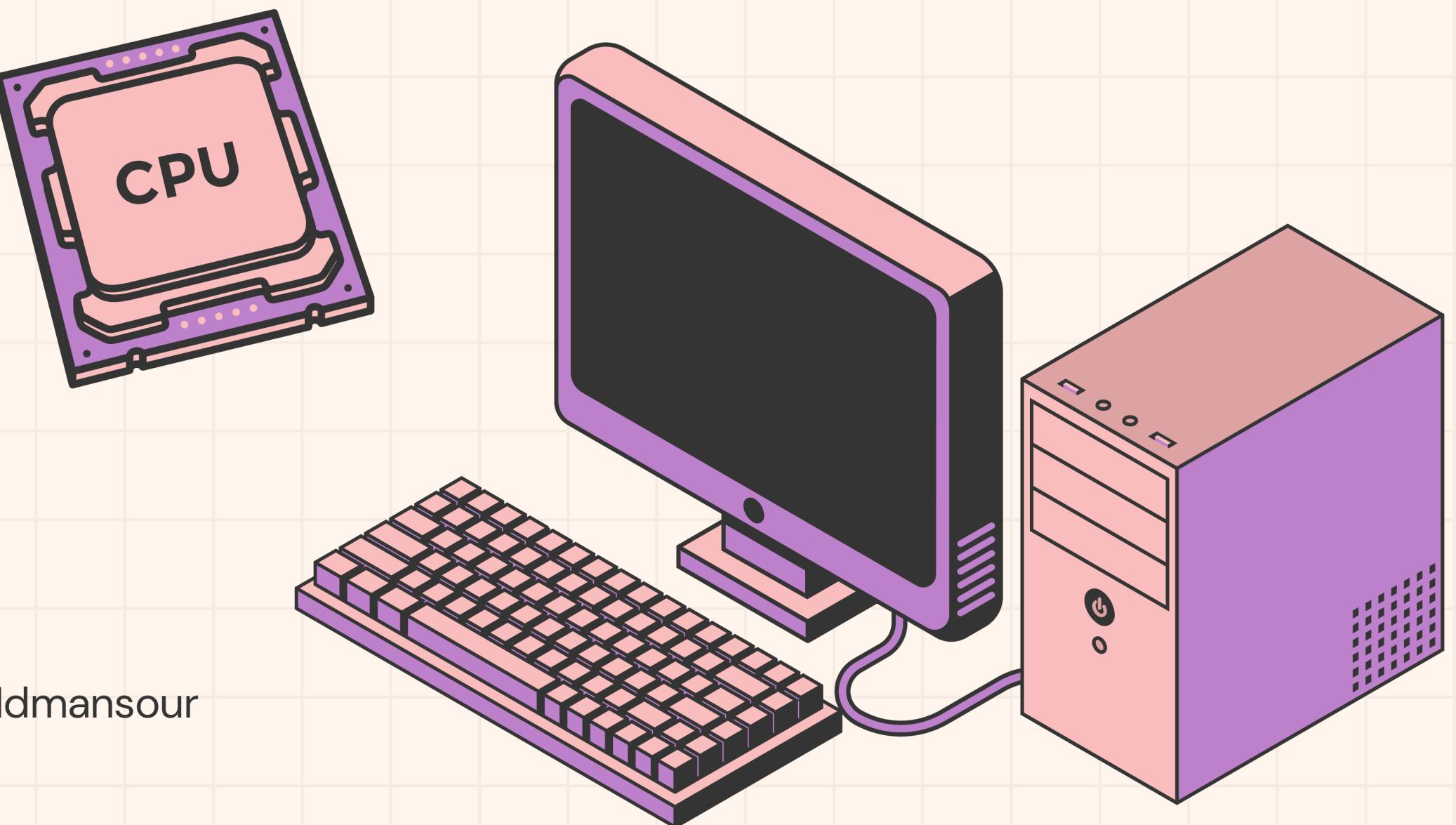


# PROJET PARM

Groupe ShePARM:

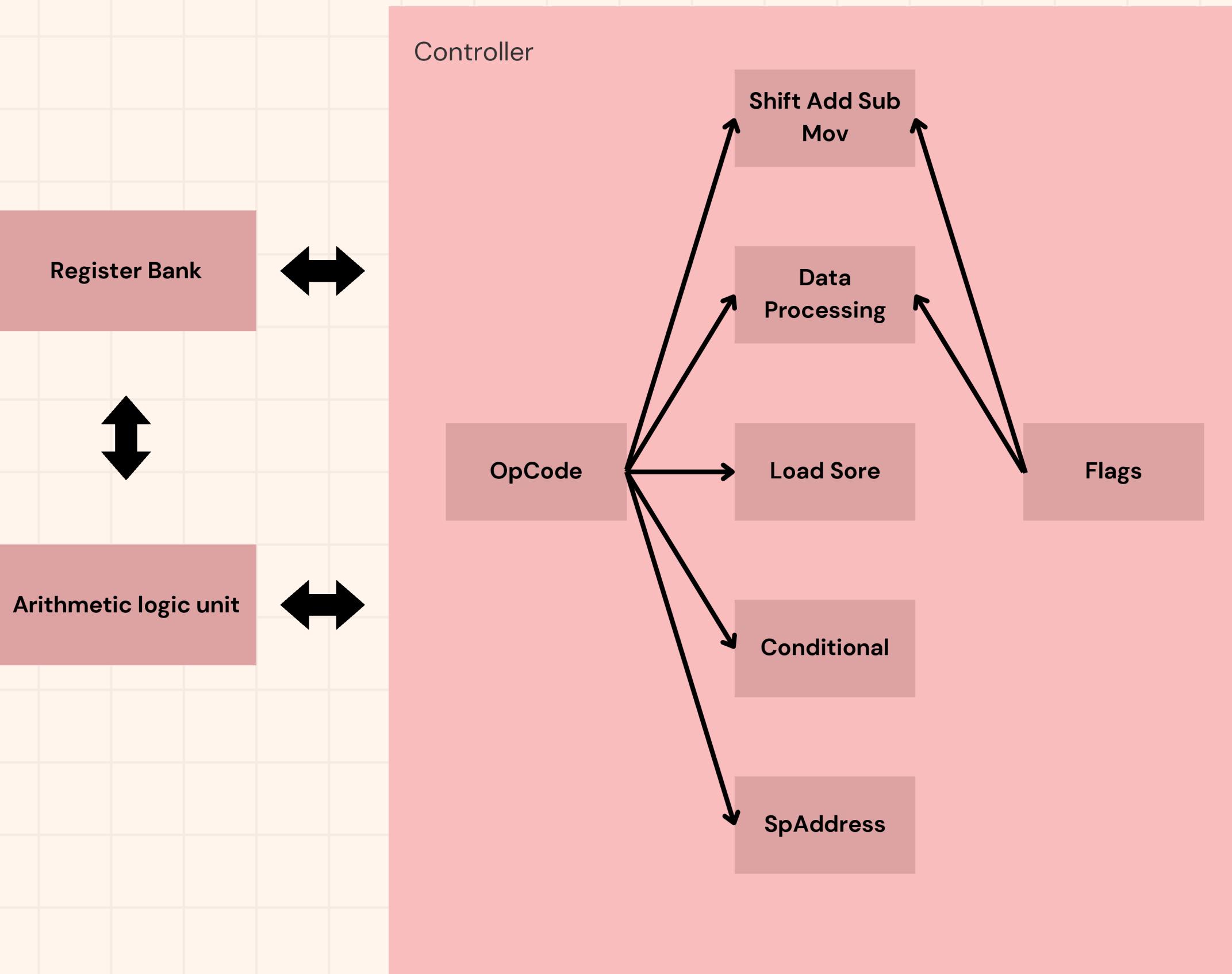
Jana Saad, Marion Valls, Maud Marconcini, Salma Idmansour

SI3 – FISA

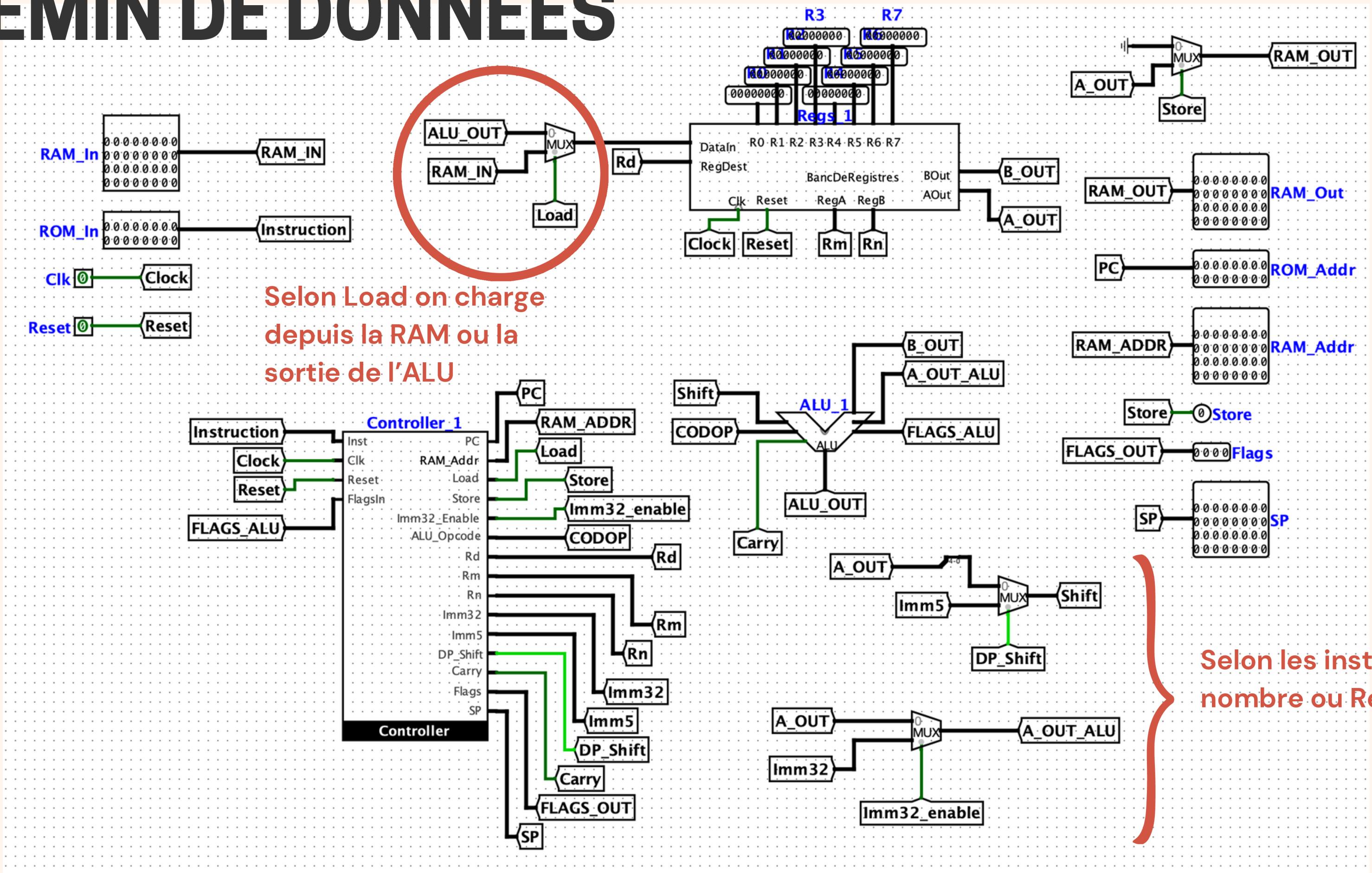


# ARCHITECTURE

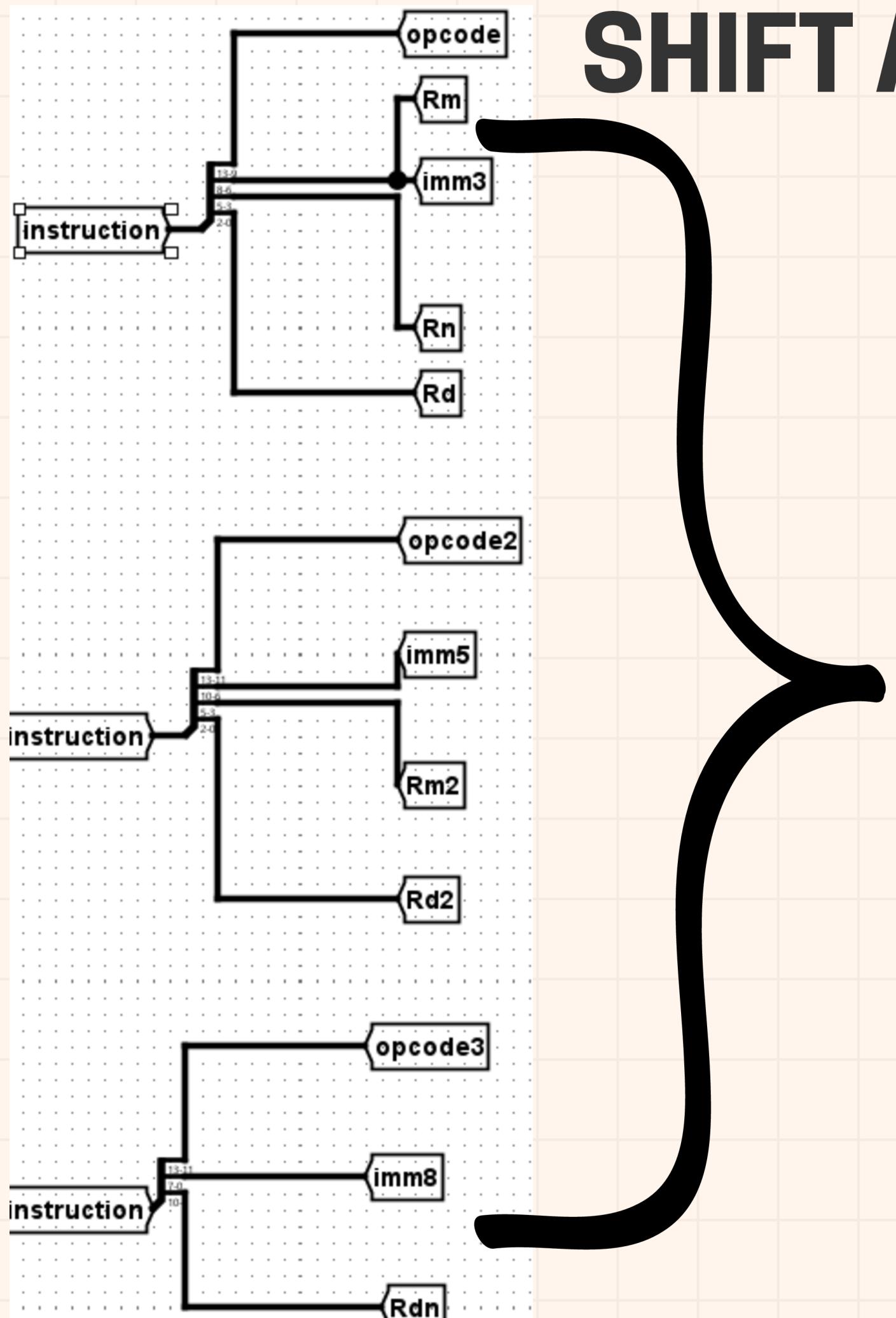
Cette présentation explore l'architecture ARM v7, en mettant en avant son organisation, son jeu d'instructions (ISA) et l'assembleur. Nous verrons le rôle de l'ALU pour les calculs et du contrôleur pour l'exécution des instructions. L'objectif est de comprendre le fonctionnement des drapeaux, registres et modes d'adressage.



# CHEMIN DE DONNÉES

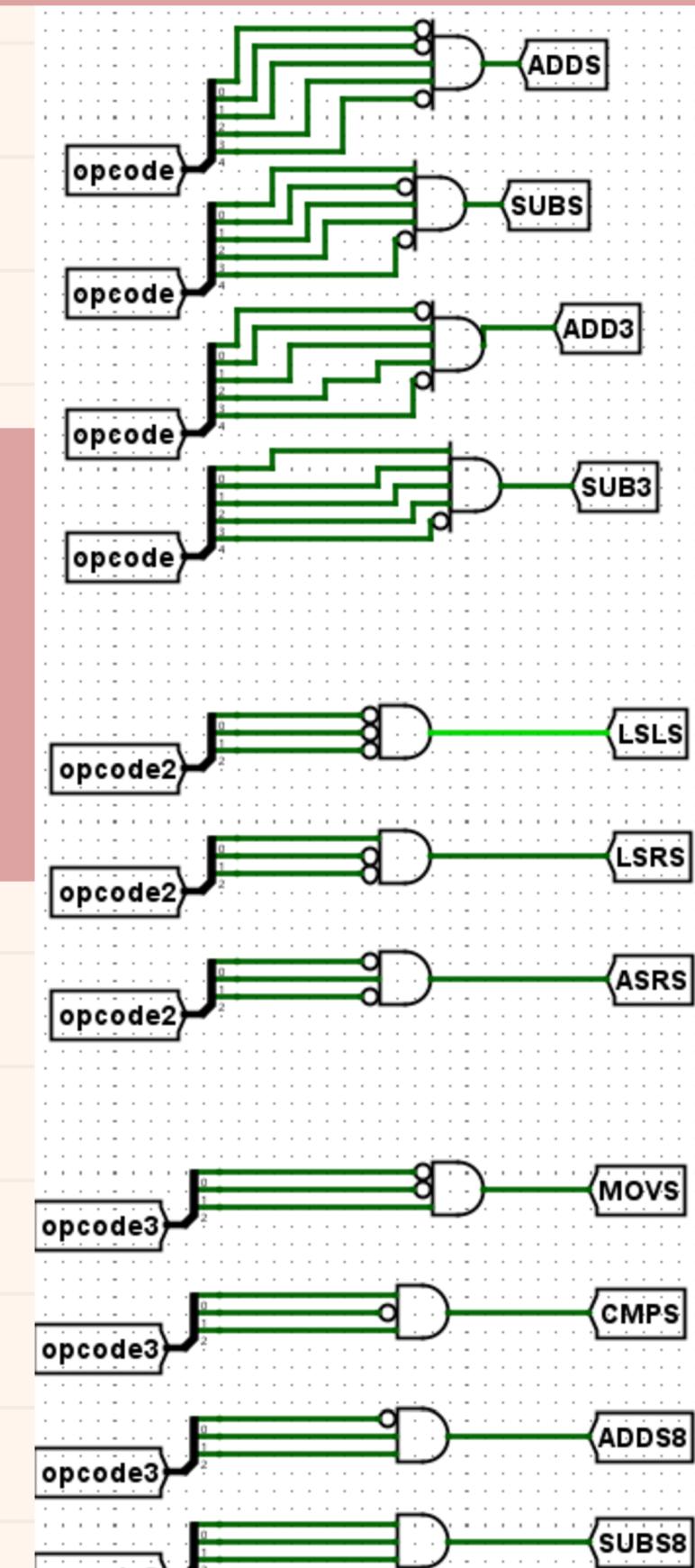


# SHIFT ADD SUB MOV

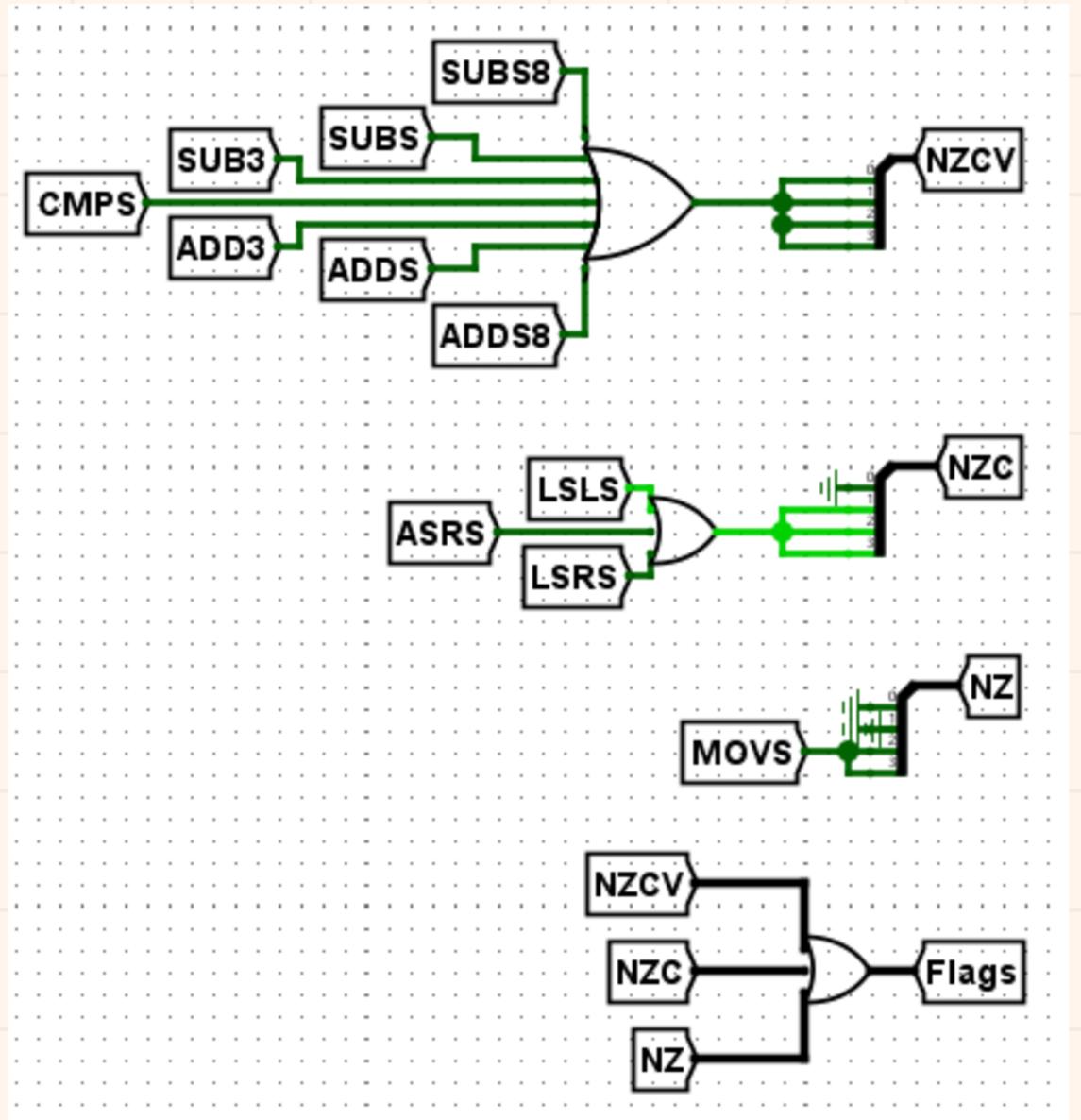


A l'aide du répartiteur on a réparti les bits correspondant aux différentes opérande et à l'opcode

L'opcode obtenu par le répartiteur nous a permis de définir les instructions

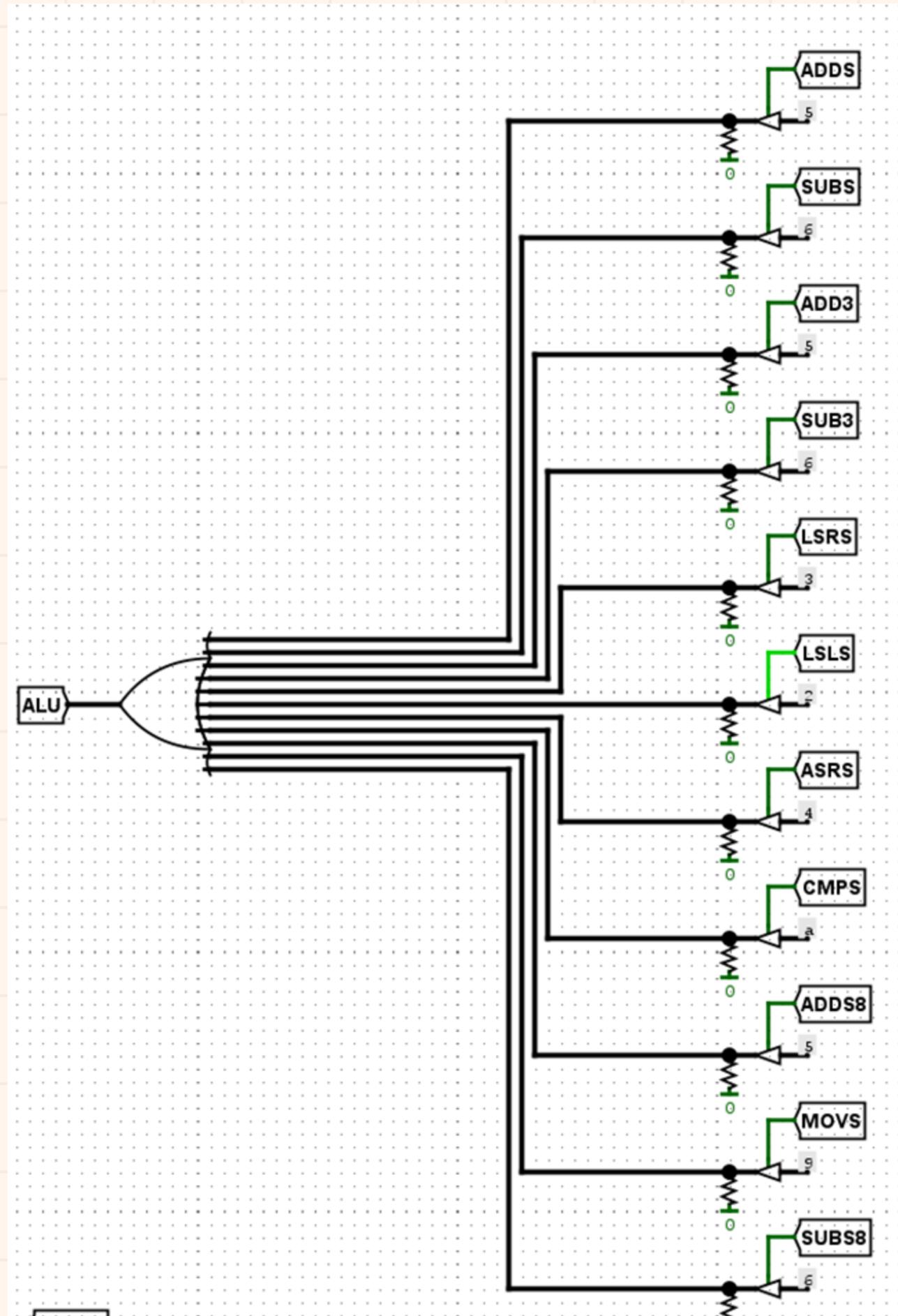
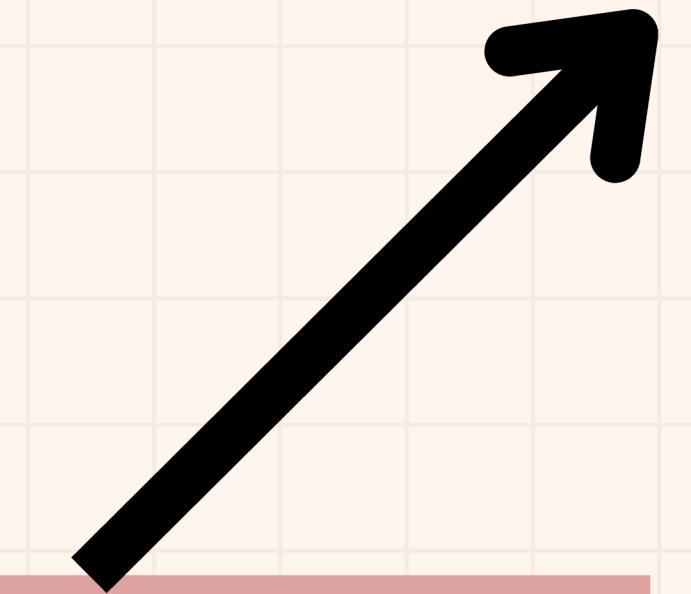


# SHIFT ADD SUB MOV



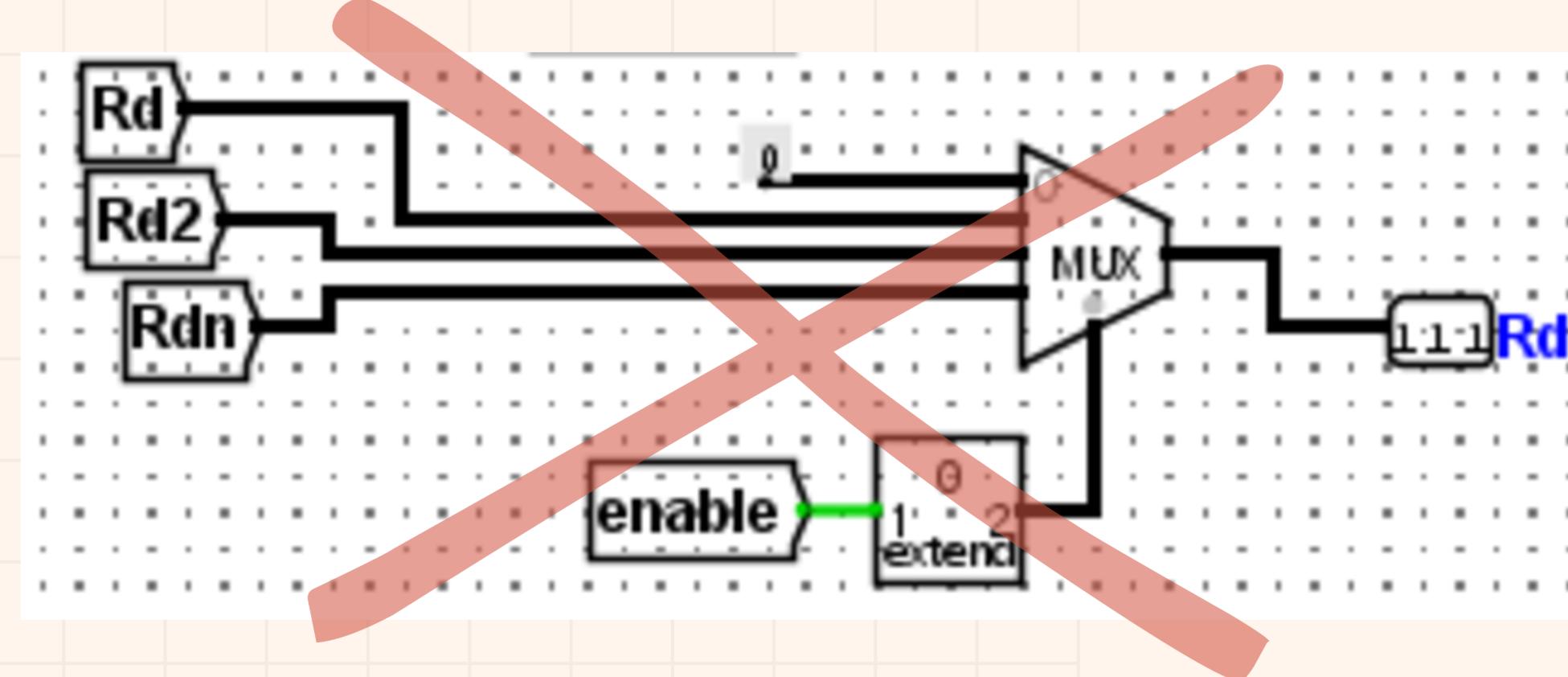
Les flags sont mis à jour selon l'instruction

chaque sortie de l'ALU est activée en fonction de l'opcode des instructions en binaire qui, lorsqu'il est converti en decimal, correspond à une des instructions listées

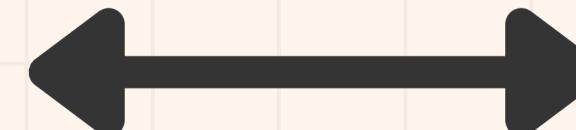


# SHIFT ADD SUB MOV

Comment obtenir la sortie Rd à partir de Rd, Rd2, et Rdn?



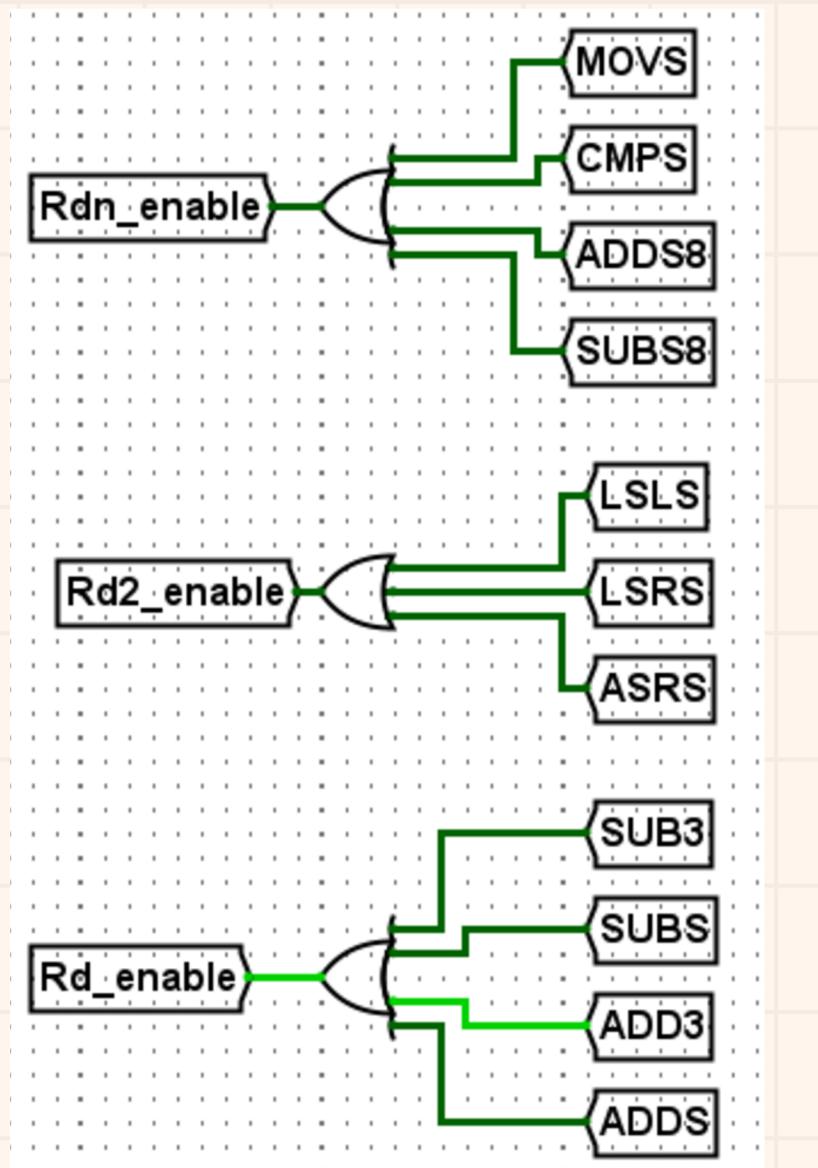
La sortie de l'étendeur de bits sera soit 00 soit 01.  
On a que 2 combinaisons différentes



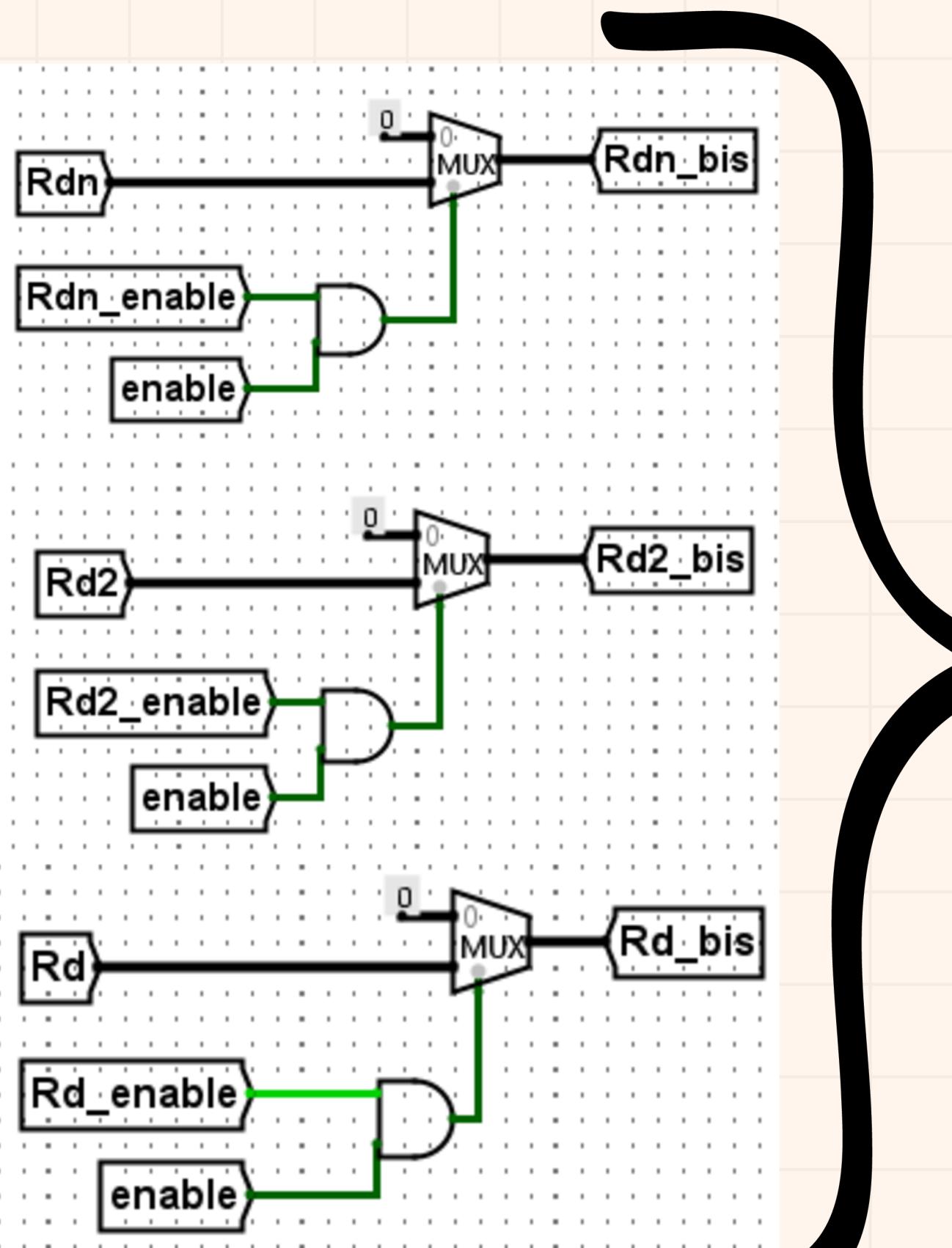
Des entrées ne seront jamais sélectionnées!

# SHIFT ADD SUB MOV

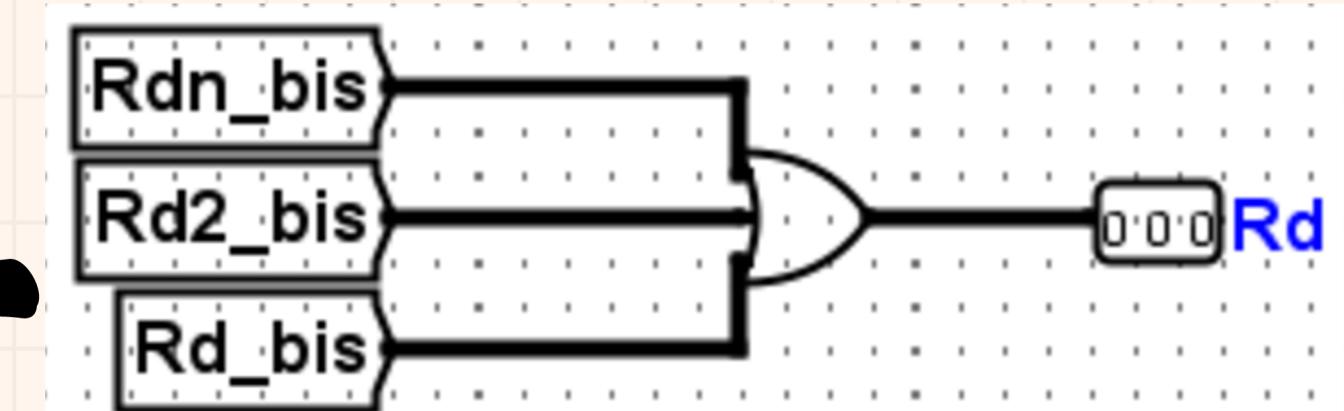
Comment obtenir la sortie Rd à partir de Rd, Rd2, et Rdn?



Un signal \_enable qui permet d'identifier le groupe auquel appartient l'instruction en question

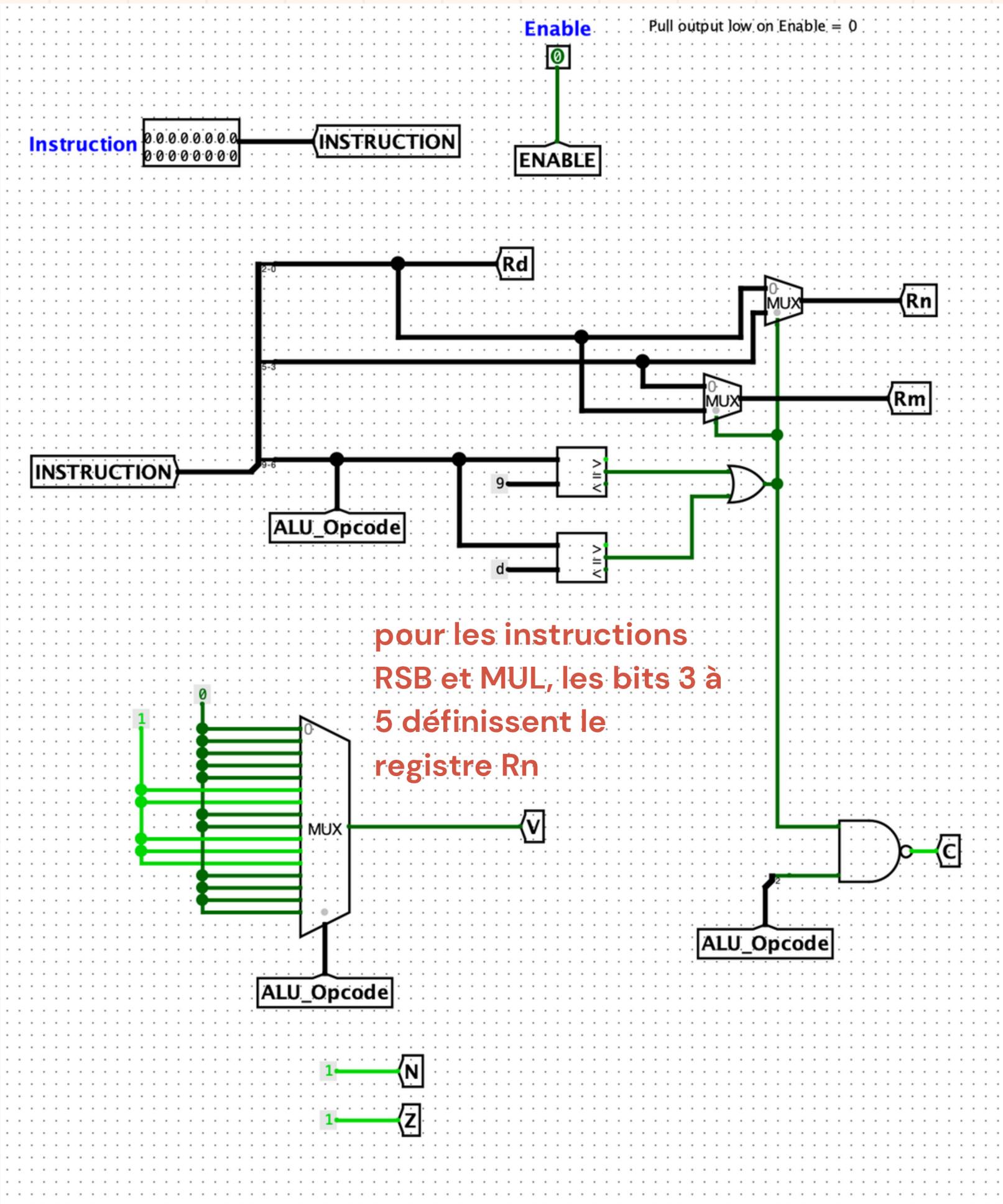


On aura toujours au maximum un des 3 signaux qui est différent de 0

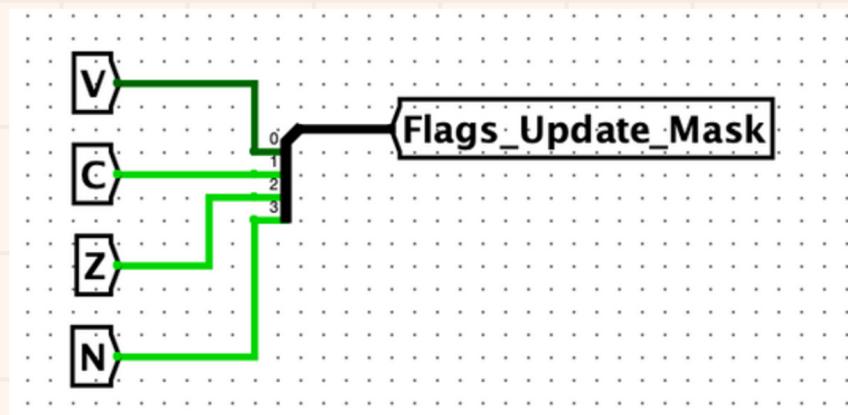


Rd sera égal à 000 ou à la valeur de l'un des 3 signaux (Rdn\_bis, Rd\_bis ou Rd2\_bis) qui eux sont égaux à Rdn, Rd ou Rd2 respectivement

# DATA PROCESSING COMPONENTS



Pour chacune des sorties on check enable



Reconstitution  
des flags

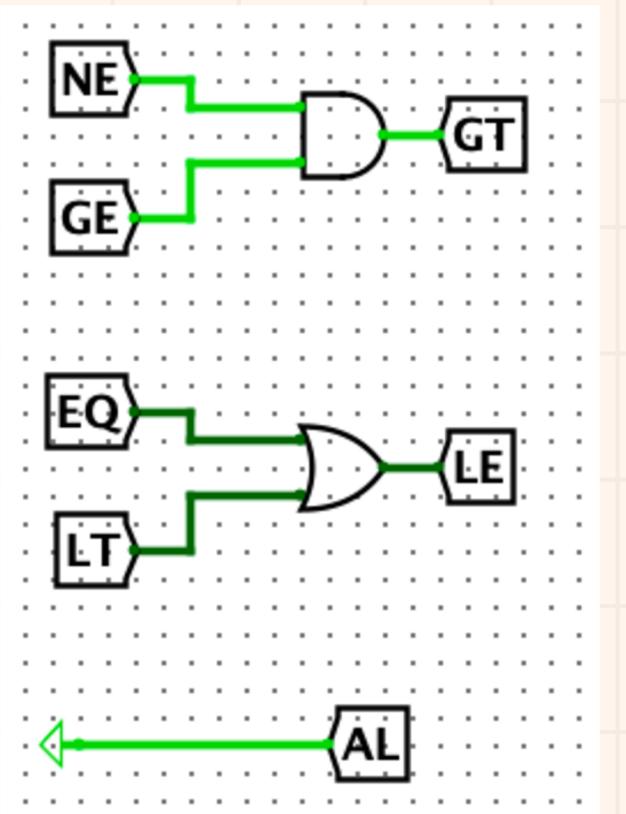
# CONDITIONAL COMPONENTS

## FONCTIONNEMENT

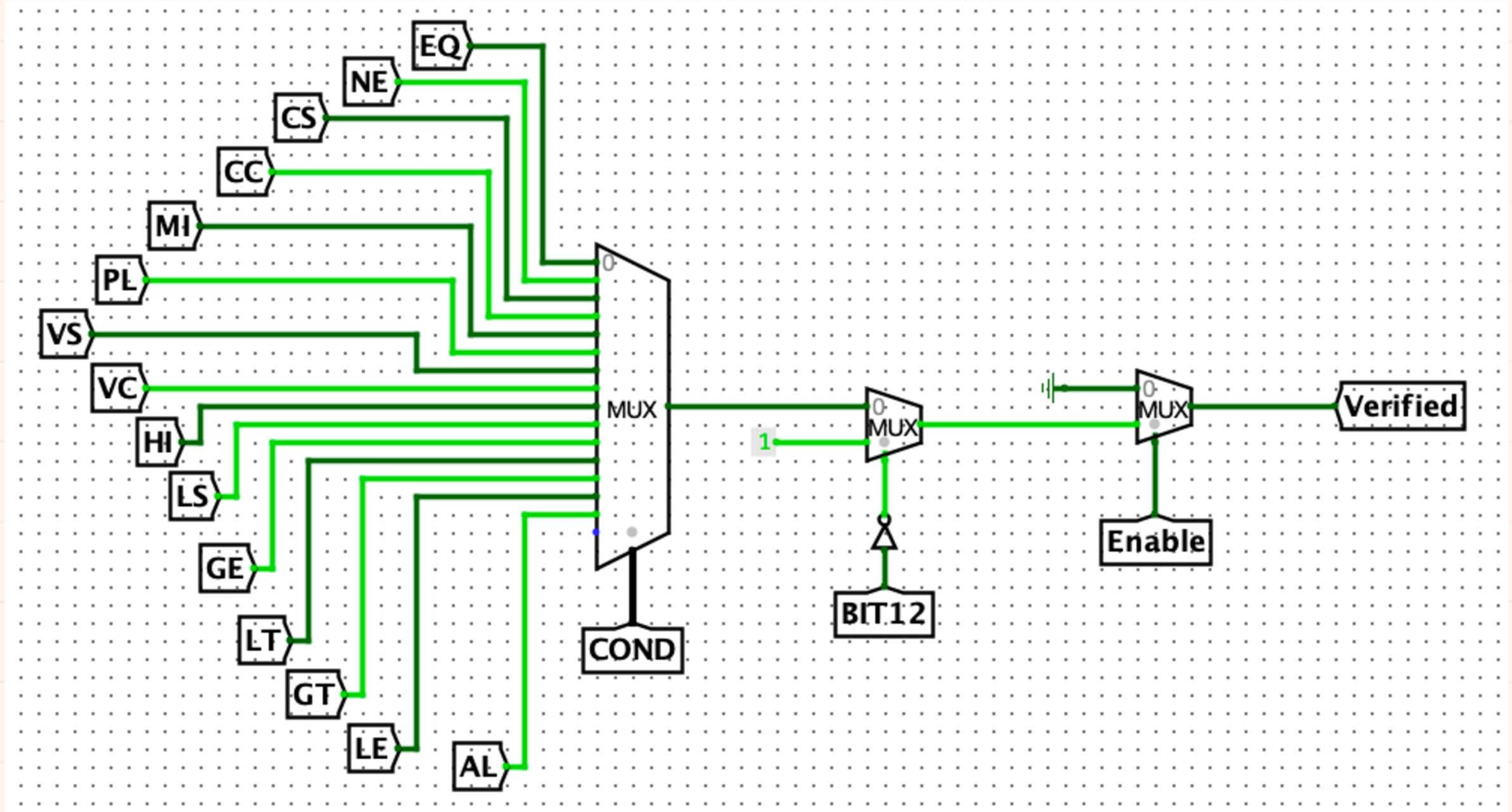
Le contrôleur Conditional -> l'exécution conditionnelle des instructions en fonction des Flags.

- Si une condition est remplie avant d'autoriser l'exécution d'une instruction.

Décide si une instruction doit être exécutée ou ignorée selon l'état du processeur.

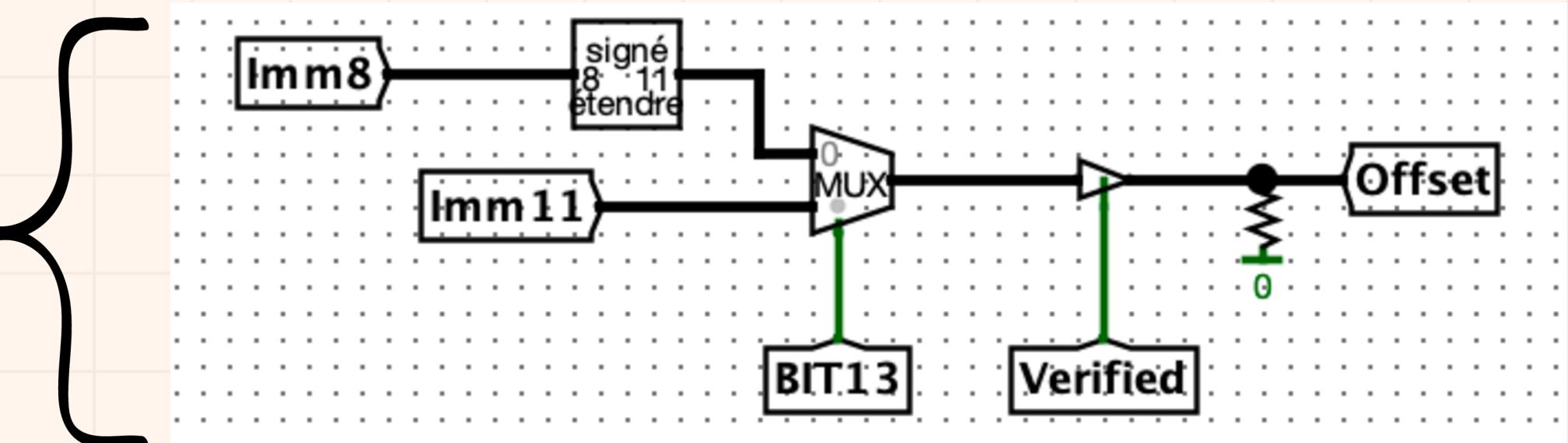


code	symbole	signification	drapeaux
0000	EQ	égalité	Z == 1
0001	NE	différence	Z == 0
0010	CS ou HS	retenue	C == 1
0011	CC ou LO	pas de retenue	C == 0
0100	MI	négatif	N == 1
0101	PL	positif ou nul	N == 0
0110	VS	dépassement de capacité	V == 1
0111	VC	pas de dépassement de capacité	V == 0
1000	HI	supérieur (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	



Gère l'exécution conditionnelle des branchements en fonction des flags.

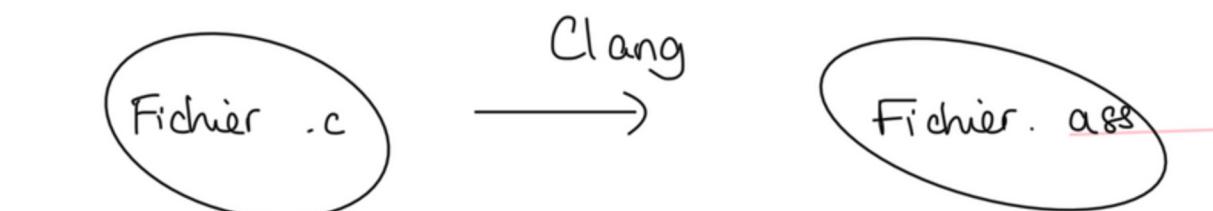
Contrôle la génération d'un offset pour le branchement conditionnel → modifier l'adresse.



# ASSEMBLEUR TO BINARY

```
ldr r0, [sp, #4]
Instruction: ldr
Parameters: ['r0', '[sp', '#4']
Type of param r0: register
Type of param [sp: label
Type of param #4]: immediate number
Binary: 1001100000000001
Hex: 0x9801
```

```
adds r0, r0, #1
Instruction: adds
Parameters: ['r0', 'r0', '#1']
Type of param r0: register
Type of param r0: register
Type of param #1: immediate number
Binary: 0001110001000000
Hex: 0x1c40
```



- ① Nettoyer le fichier
- ② Récupérer les étiquettes et les stocker.  
Vides le fichier.  
Si étiquette : ligne actuelle  
- index étiquette - 3
- ③ Récuperer les instructions et les stocker
  - ↳ Récuperer liste param
  - ↳ Récuperer l'instruction
  - ↳ Convertir (binaire → hex)

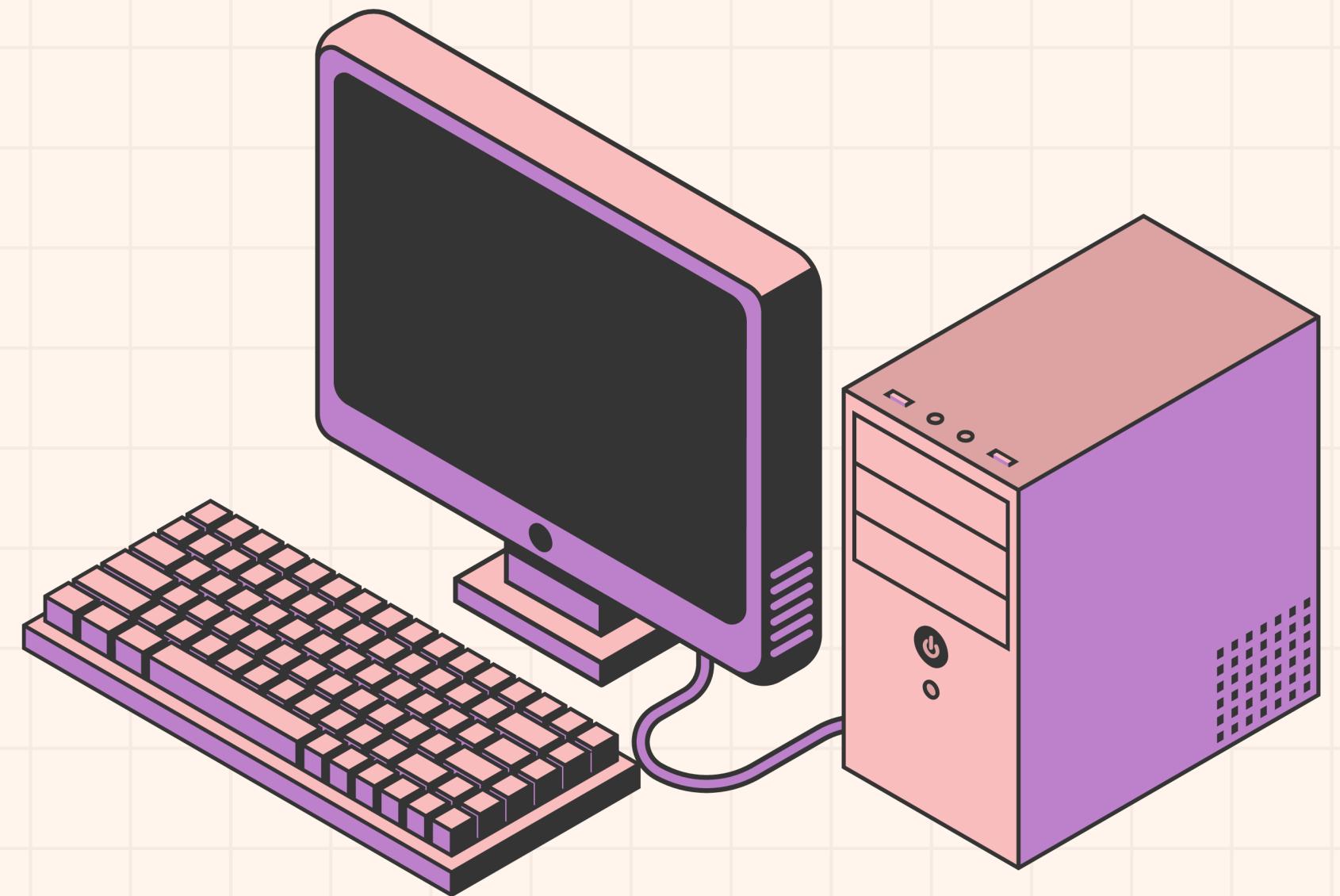
```
.file "calckeyb.c"
.globl run
.p2align 1
.type run,%function
.code 16
.thumb_func
run:
.fnstart
```



v2.0 raw  
b09a b0ff b0f1 2000 9004 2101 9103 9001

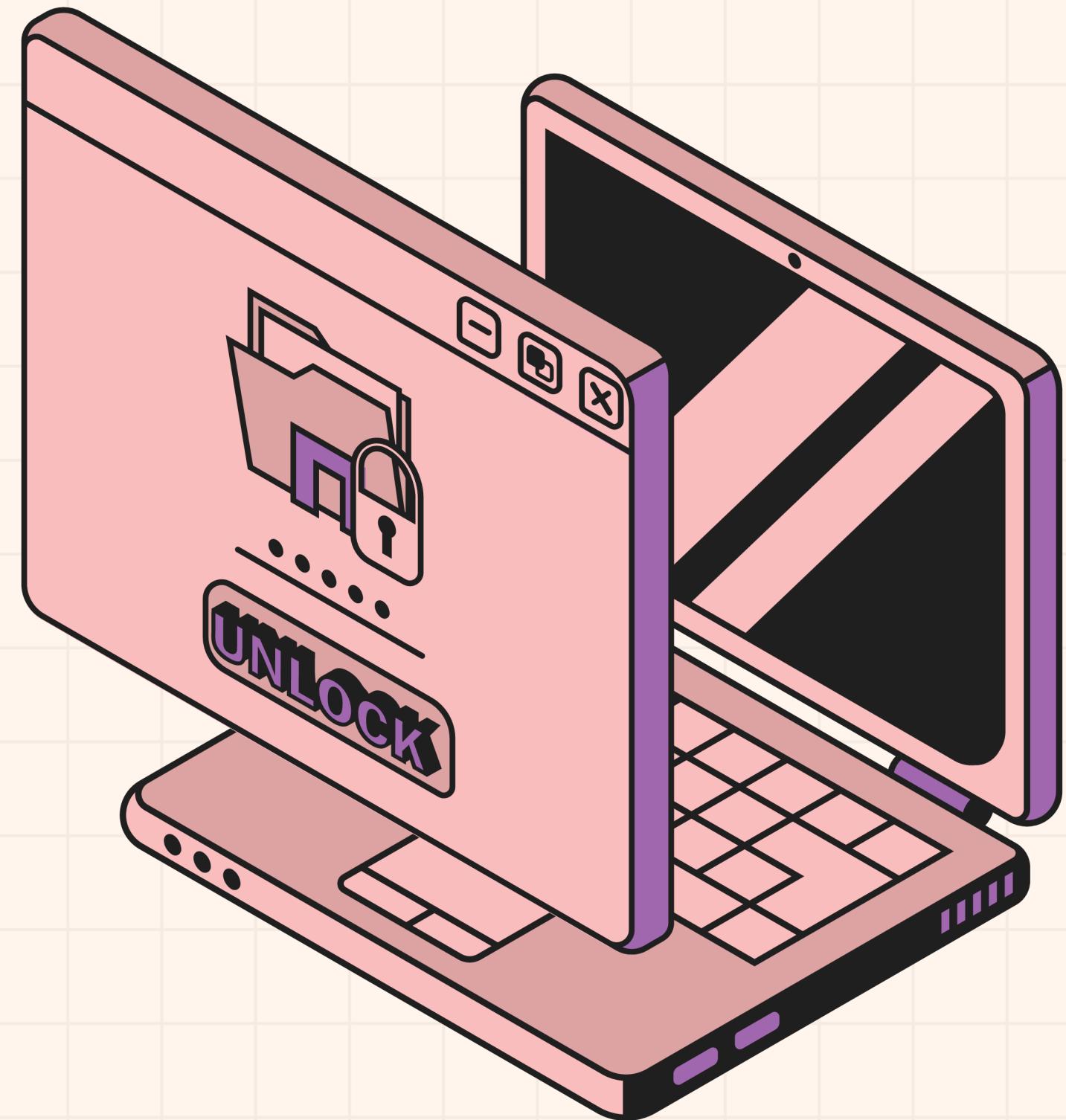


# DEMO



# PROBLÈMES RENCONTRÉS

- Problème de sauvegarde sur Logisim et d'utilisation des différents composants au début du projet
- Le nombre de bits de chaque registres (du langage assembleur au binaire)
- Sélection du registre de sortie sur le Shift Add Sub Move
- La ligne .goto ne s'exécute pas lors du test branch
- Codage sur un nombre de bits différent de la branche non conditionnelle

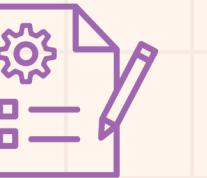


# DOCUMENTS COMPLÉMENTAIRES



## TABLEAU JEU D'INSTRUCTION

Nous avons créé un tableau du jeu d'instructions répertoriant les différentes instructions utilisées. Il inclut leur nom, description, effet sur les registres et drapeaux. Ce tableau permet de mieux comprendre et structurer l'exécution des instructions.



## DOCUMENT TEXTE

Nous avons chacun réalisé un document Word pour expliquer notre travail, en détaillant le fonctionnement des composants et leur rôle. Chaque document permet de comprendre les circuits et les tests effectués. Cela facilite le partage des connaissances entre nous.



## EXCEL VALIDATION TESTS

Nous avons réalisé 3 tableaux Excel : un pour les tests unitaires, un pour les tests en C et un pour les tests en assembleur. Chaque tableau contient la description des tests et leur état de validation. Cela permet un suivi clair et organisé de nos vérifications.

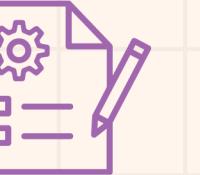
Ref Doc		Description	UAL Code				Bits												Flags				
		Instruction	Operandes																				
		Shift , add, sub, mov																					
							15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
11.1.1	p.30	1 Logical Shift Left	LSLS	Rd	Rm	imm5	0	0	0	0	0		imm5		Rm		Rd	x	x	x	x		
		2 Logical Shift Right	LSRS	Rd	Rm	imm5	0	0	0	0	1		imm5		Rm		Rd	x	x	x	x		
		3 Arithmetic Shift Right	ASRS	Rd	Rm	imm5	0	0	0	1	0		imm5		Rm		Rd	x	x	x	x		
		4 Add register	ADDS	Rd	Rn	Rm	0	0	0	1	1	0	0	Rm		Rn		Rd	x	x	x	x	
		5 Subtract register	SUBS	Rd	Rn	Rm	0	0	0	1	1	0	1	Rm		Rn		Rd	x	x	x	x	
		6 Add 3-bit immediate	ADD3	Rd	Rn	imm3	0	0	0	1	1	1	0	imm3		Rn		Rd	x	x	x	x	
		7 Subtract 3-bit immediate	SUB3	Rd	Rn	imm3	0	0	0	1	1	1	1	imm3		Rn		Rd	x	x	x	x	
		8 Move	MOVS	Rd	imm8		0	0	1	0	0		Rd			imm8				x	x	x	x
		9 Compare	CMPS	Rd	imm8		0	0	1	0	1		Rd			imm8				x	x	x	x
		10 Add 8-bit immediate	ADDS8	Rdn	imm8		0	0	1	1	0		Rdn			imm8				x	x	x	x
		11 Subtract 8-bit immediate	SUBS8	Rdn	imm8		0	0	1	1	1		Rdn			imm8				x	x	x	x
		Data Processing																					
11.1.2	p.35	1 Bitwise AND	ANDS	Rdn	Rm		0	1	0	0	0	0	0	0	0	0	Rm		Rdn	0	x	x	x
		2 Exclusive OR	EORS	Rdn	Rm		0	1	0	0	0	0	0	0	0	1	Rm		Rdn	1	x	x	x
		3 Logical Shift Left	LSLS	Rdn	Rm		0	1	0	0	0	0	0	0	1	0	Rm		Rdn	x	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	x
		5 Arithmetic Shift Right	ASRS	Rdn	Rm		0	1	0	0	0	0	0	1	0	0	Rm		Rdn	x	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 Subtract 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	x
		9 Set Flags on bitwise AND	TSTS	Rdn	Rm		0	1	0	0	0	0	1	0	0	0	Rm		Rn	0	x	x	x
		10 Reverse Subtract 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	x
		14 Multiply Two Registers	MULS	Rdm	Rn	Rdm	0	1	0	0	0	0	1	1	0	1	Rn		Rdm	x	x	x	x
		15 Bit Clear	BICS	Rdn	Rm		0	1	0	0	0	0	1	1	1	0	Rm		Rdn	0	x	x	x
		16 Bitwise NOT	MVNS	Rd	Rm		0	1	0	0	0	0	1	1	1	1	Rm		Rd	1	x	x	x
		Load/Store																					
11.1.3	p.41	1 Store Register	STR	Rt	imm8	SP	1	0	0	1	0		Rt			imm8							
		2 Load Register	LDR	Rt	imm8	SP	1	0	0	1	1		Rt			imm8							
		Miscellaneous 16-bit instructions																					
11.1.4	p.42	1 Add Immediate to SP	ADD	SP	imm7	SP	1	0	1	1	0	0	0	0	0		imm7						
		2 Subtract Immediate from SP	SUB	SP	imm7	SP	1	0	1	1	0	0	0	0	1		imm7						
		Branche conditionnelle																					
11.1.5	p.42	Equal	EQ				1	1	0	1	0	0	0	0	0		imm8						
		Less than	NE				1	1	0	1	0	0	0	1			imm8						
		Carry Set	CS ou HS				1	1	0	1	0	0	1	0			imm8						
		Carry clear	CC ou LO				1	1	0	1	0	0	1	1			imm8						
		Minus, negative	MI				1	1	0	1	0	1	0	0	0		imm8						
		Plus, positive or zero	PL				1	1	0	1	0	1	0	1	0		imm8						
		Overflow	VS				1	1	0	1	0	1	1	0	0		imm8						
		No overflow	VC				1	1	0	1	0	1	1	1	1		imm8						
		Unsigned higher	HI				1	1	0														

# DOCUMENTS COMPLÉMENTAIRES



## TABLEAU JEU D'INSTRUCTION

Nous avons créé un tableau du jeu d'instructions répertoriant les différentes instructions utilisées. Il inclut leur nom, description, effet sur les registres et drapeaux. Ce tableau permet de mieux comprendre et structurer l'exécution des instructions.



## DOCUMENT TEXTE

Nous avons chacun réalisé un document Word pour expliquer notre travail, en détaillant le fonctionnement des composants et leur rôle. Chaque document permet de comprendre les circuits et les tests effectués. Cela facilite le partage des connaissances entre nous.



## EXCEL VALIDATION TESTS

Nous avons réalisé 3 tableaux Excel : un pour les tests unitaires, un pour les tests en C et un pour les tests en assembleur. Chaque tableau contient la description des tests et leur état de validation. Cela permet un suivi clair et organisé de nos vérifications.

Nom test	Validé	Description
Test shift add sub mov	OK	test unitaire qui couvre toutes les possibilité pour le composant shift add sub mov
Test conditional	OK	test unitaire qui couvre toutes les possibilité pour le composant conditional
Test sp address	OK	test unitaire qui couvre toutes les possibilité pour le composant sp address
Test op code	OK	test unitaire qui couvre toutes les possibilité pour le composant op code
Test data processing	OK	test unitaire qui couvre toutes les possibilité pour le composant data processing
Test load store	OK	test unitaire qui couvre toutes les possibilité pour le composant load store
Test alu	OK	test unitaire qui couvre toutes les possibilité pour le composant alu

Nom test	Validé	Description
calckeyb.c	OK	calckeyb.c implémente une calculatrice interactive prenant deux entiers en entrée (a et b). L'utilisateur choisit une opération (+ - * / % &   ^) via READKEY(), et le programme exécute le calcul correspondant. Les résultats sont affichés avec PRINTRES_SIGN(), et une attente (WAITKEY()) permet de visualiser le résultat avant réinitialisation (RESET()). Il vérifie la bonne gestion des entrées, des opérations mathématiques/logiques et de l'affichage.
calculator.c	OK	calculator.c implémente une calculatrice matérielle utilisant des entrées (OPa, OPb, CMD) issues de commutateurs (DIP1, DIP2, DIP3). Selon la valeur de CMD, il effectue une addition, une soustraction, une multiplication ou un décalage à gauche (LSL). Le résultat est stocké dans RES et mis à jour en boucle. Il permet de tester la bonne interprétation des entrées physiques et l'exécution correcte des opérations.
simple_add.c	OK	simple_add.c vérifie le bon fonctionnement d'une addition simple en initialisant deux variables (a = 1, b = 2), puis en stockant leur somme dans c. Le résultat est assigné à RES pour validation. L'exécution est encadrée par BEGIN() et END(), garantissant une structure correcte. Il sert à tester l'exécution basique des opérations arithmétiques et l'affectation des résultats.
testfp.c	OK	testfp.c vérifie les calculs en virgule fixe (fixed_t). Il initialise deux valeurs (3.5 et 7.5), effectue leur multiplication (MULTFP), puis affiche le résultat. Ensuite, il calcule la racine carrée (SQRTFP) de ce produit et l'affiche. Enfin, il divise cette valeur par 1.5 (DIVFP) et affiche le résultat. Il valide l'implémentation des opérations mathématiques en virgule fixe et leur affichage.
tty.c	OK	tty.c vérifie l'affichage de texte en imprimant "Projet PARM" via PUTCHAR(). Il s'exécute entre BEGIN() et END(), garantissant une exécution correcte. Il permet de tester la sortie standard et l'affichage des caractères sur l'interface cible.

Nom test	Validé	Description
Test conditional	OK	branch.s teste les branches conditionnelles en boucle. Il initialise r0 = 0, r1 = 1, et r2 = 20, puis compare r0 et r1 pour inverser r2 si r0 < r1. Ensuite, il vérifie si r2 < r1 et, si vrai, assigne 50 à r0 et recommence la boucle. La somme finale r3 = r0 + r2 doit être 70 ou 46, validant le comportement des sauts conditionnels.
Test data processing	OK	1-4_instructions.s teste les opérations de traitement de données (ANDS, EORS, LSLS, LSRS).  11-12_instructions.s teste les instructions de comparaison (CMP, CMN) et leur effet sur les drapeaux du registre d'état (NZCV).  13-16_instructions.s teste les opérations logiques et arithmétiques tel que OU logique, ET logique, etc.
Test load store	OK	5-10_instructions.s teste les opérations arithmétiques, logiques et de rotation suivantes tel que le décalage vers la droite ...  load_store.s teste le stockage (STR) et le chargement (LDR) avec le pointeur de pile (sp). Il initialise r0 = 170 (0xAA), r1 = 255 (0xFF), et ajuste sp. r0 est stocké à sp + 4 et r1 à sp + 0, puis sp est décrémenté. Ensuite, r2 recharge la valeur depuis sp + 4, confirmant la bonne gestion mémoire. Il valide l'implémentation des opérations sur sp et l'accès en RAM.
Test miscellaneous	OK	sp.s teste la modification du pointeur de pile (sp). Il commence par ajouter 16 à sp, puis soustrait 4, modifiant ainsi sa valeur. Lors du premier état haut, sp est fixé à 4, et lors du second, il est ajusté à 3. Ce test vérifie la gestion des opérations arithmétiques sur le registre sp, essentiel pour la gestion de la pile en assembleur.
Test shift add sub mov	OK	1-4_instructions.s teste les instructions de décalage, addition et soustraction (LSLS, LSRS, SUBS, ASRS, ADDS).  5-8_instructions.s teste les opérations arithmétiques de base tel que subb add mov ...

# THANK YOU

JANA SAAD, MARION VALLS, MAUD MARCONCINI, SALMA IDMANSOUR

