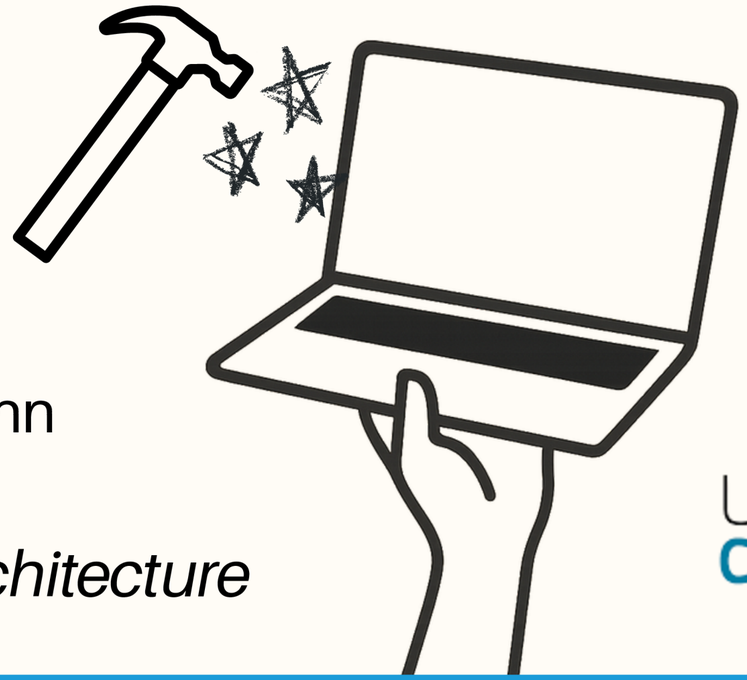
SOUTENANCE:
PROJET PARM

Présenté par l'équipe: PARMis nous



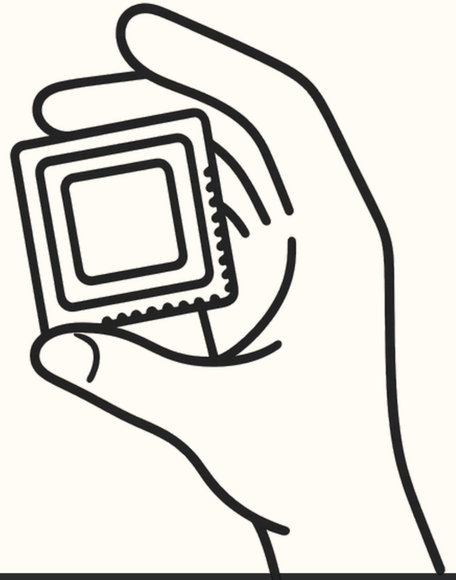
GOURE Raphaël
WAELES--DEVAUX Marie
WILLIAME-NEURANTER Swann
BERNACKI Luc

29 Janvier 2026 - SI3 S5 Architecture

Simuler un processeur 32 bits inspiré
de l'architecture
ARM Cortex-M0 sur le logiciel Logisim

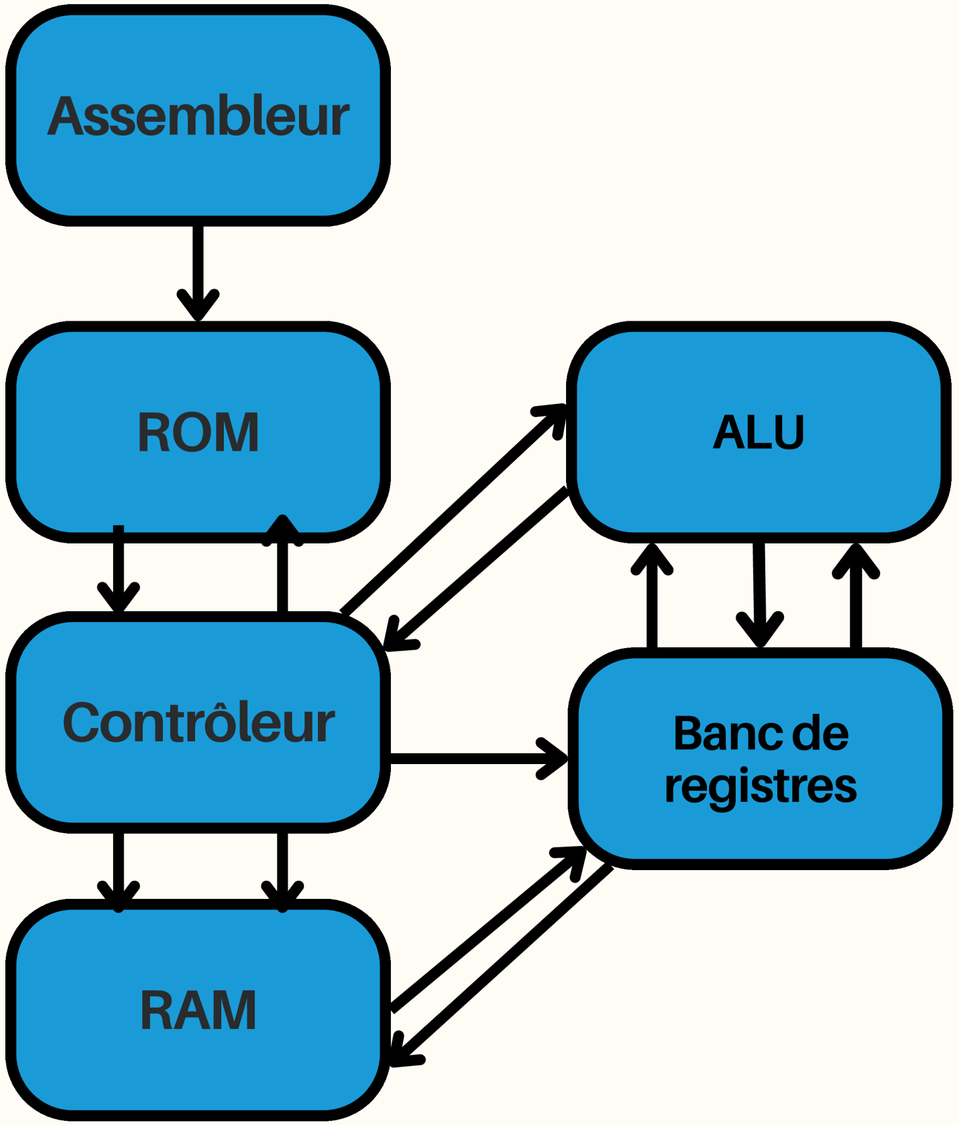
UNIVERSITÉ CÔTE D'AZUR | POLYTECH NICE SOPHIA

LE DÉFI PARM, C'EST QUOI ?

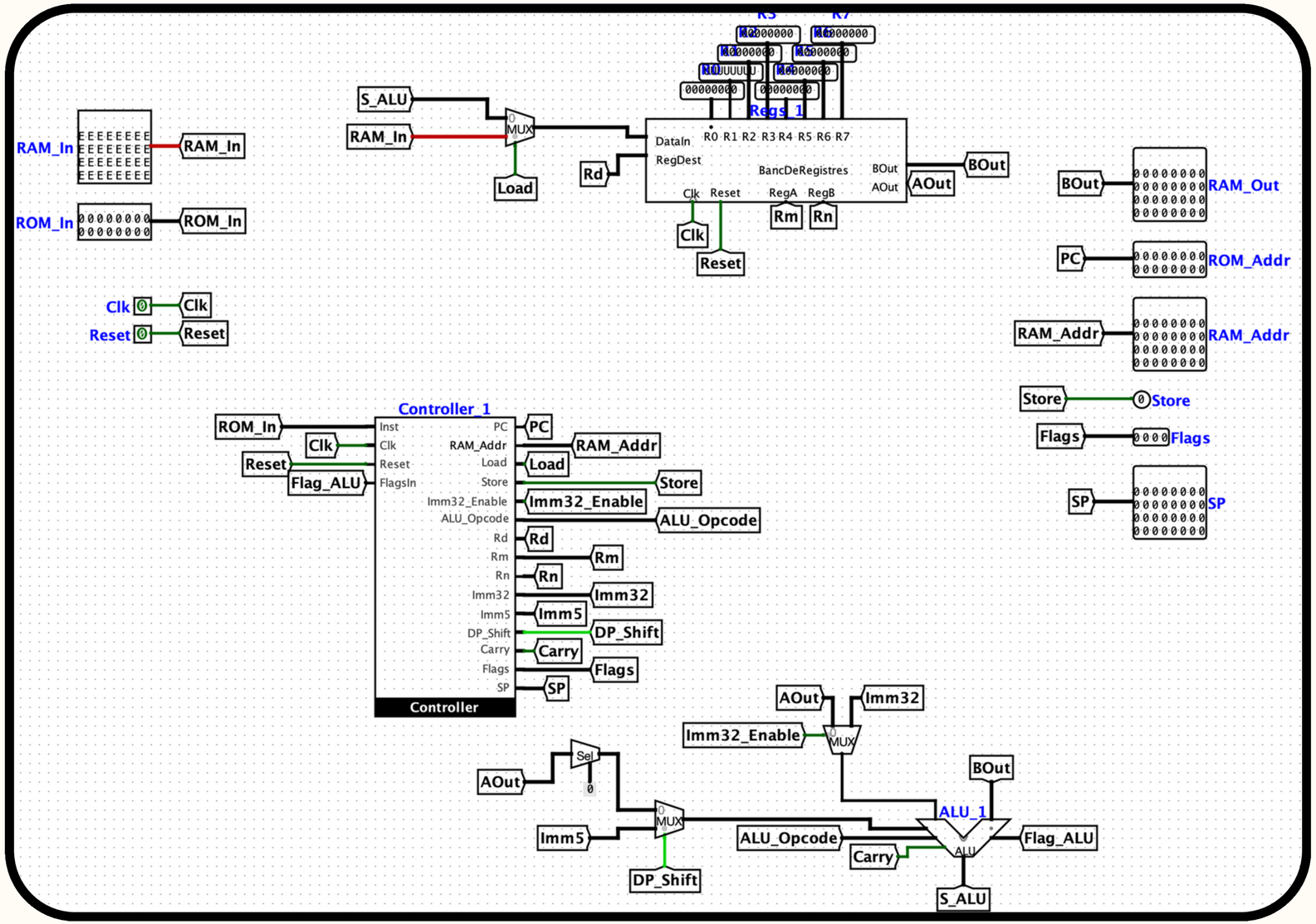


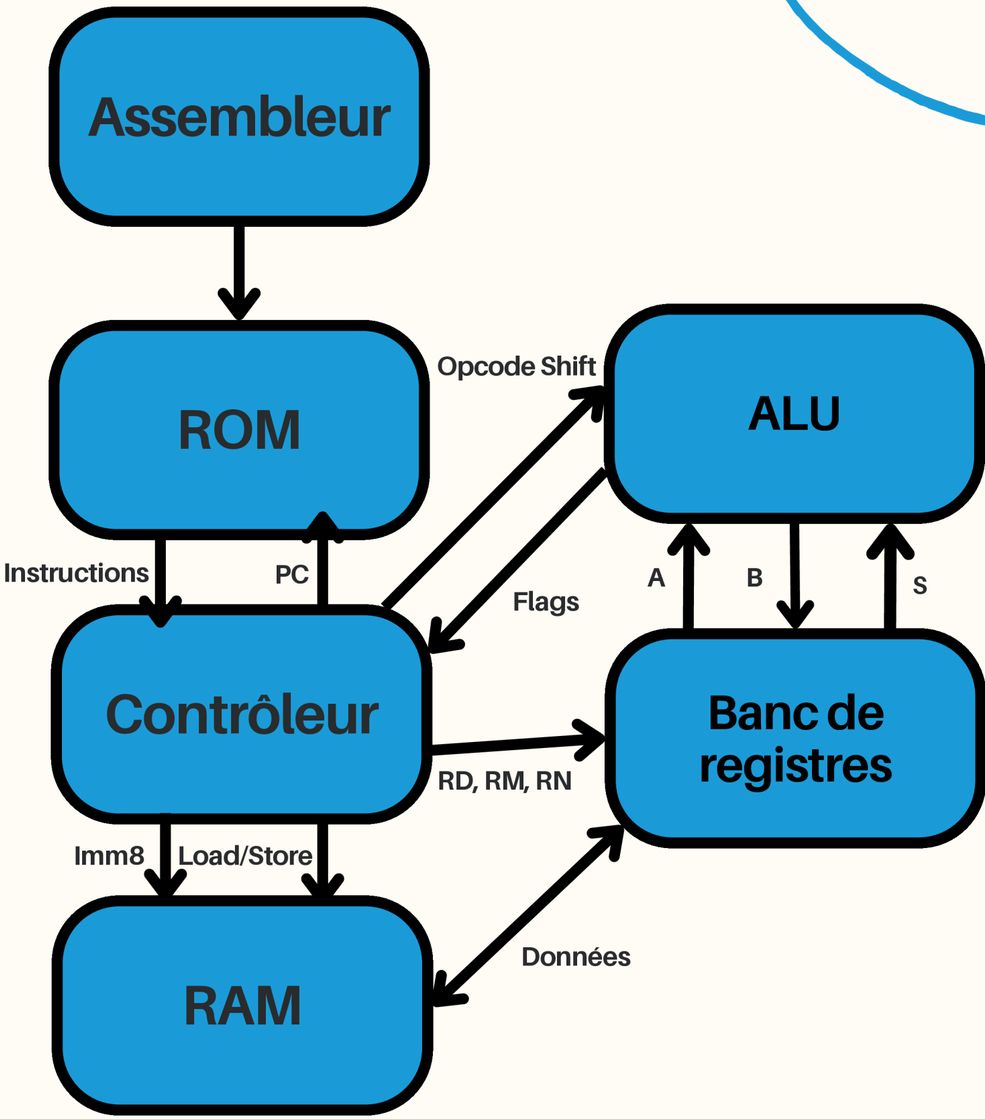
Le projet PARM :
Ou, Polytech ARM-based embedded processor,

consiste à concevoir et simuler un microprocesseur 32 bits basé sur l'architecture ARM Cortex-M0 en utilisant le logiciel de simulation électronique Logisim.



Structure du CPU:





ARCHITECTURE: LISTE DES COMPOSANTS RÉALISÉS

ALU

Shift, Add, Sub, Mov

Flags APSR

SP Address

Load/Store

Conditionnal

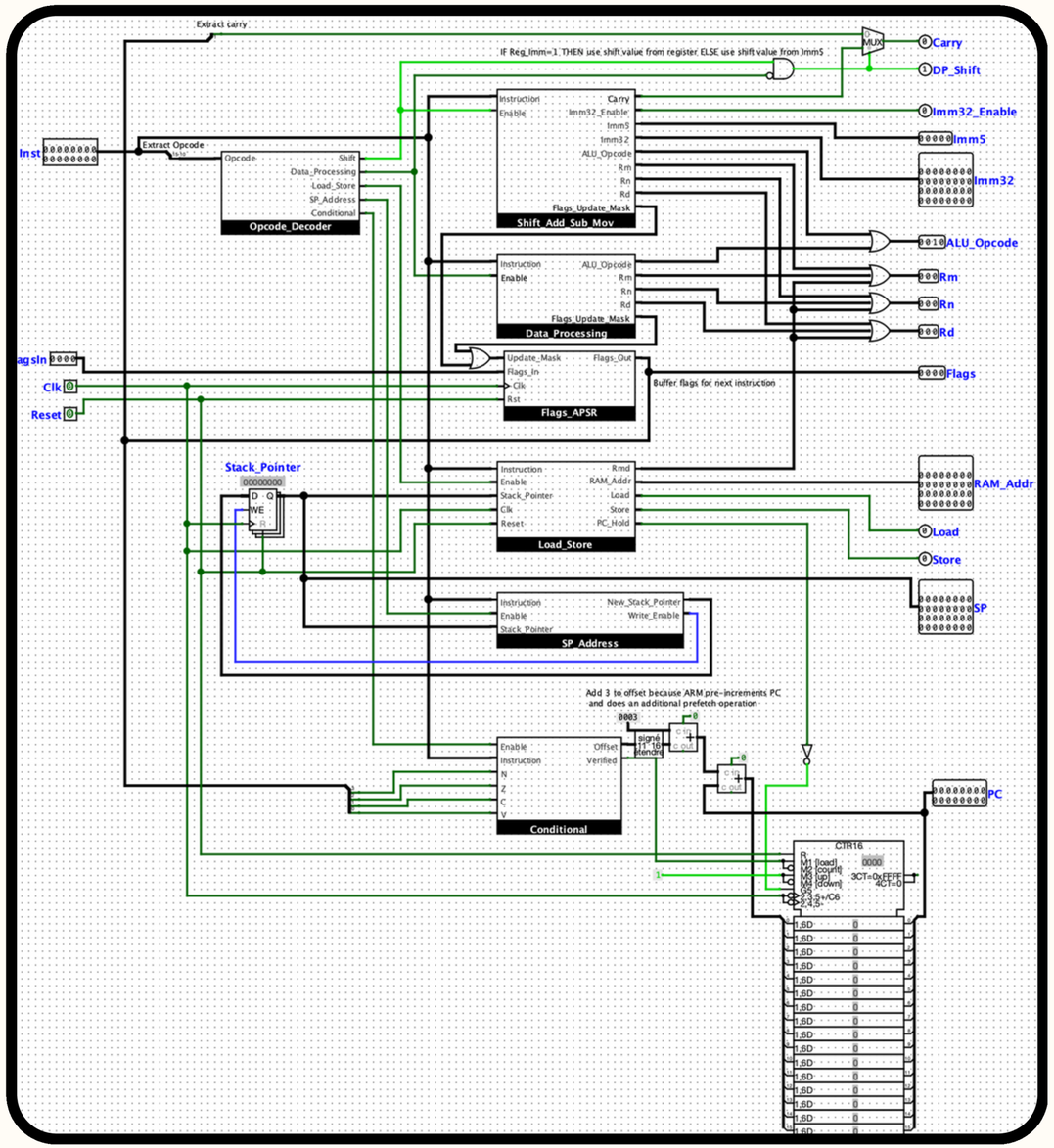
Data Processing

Banc de registre

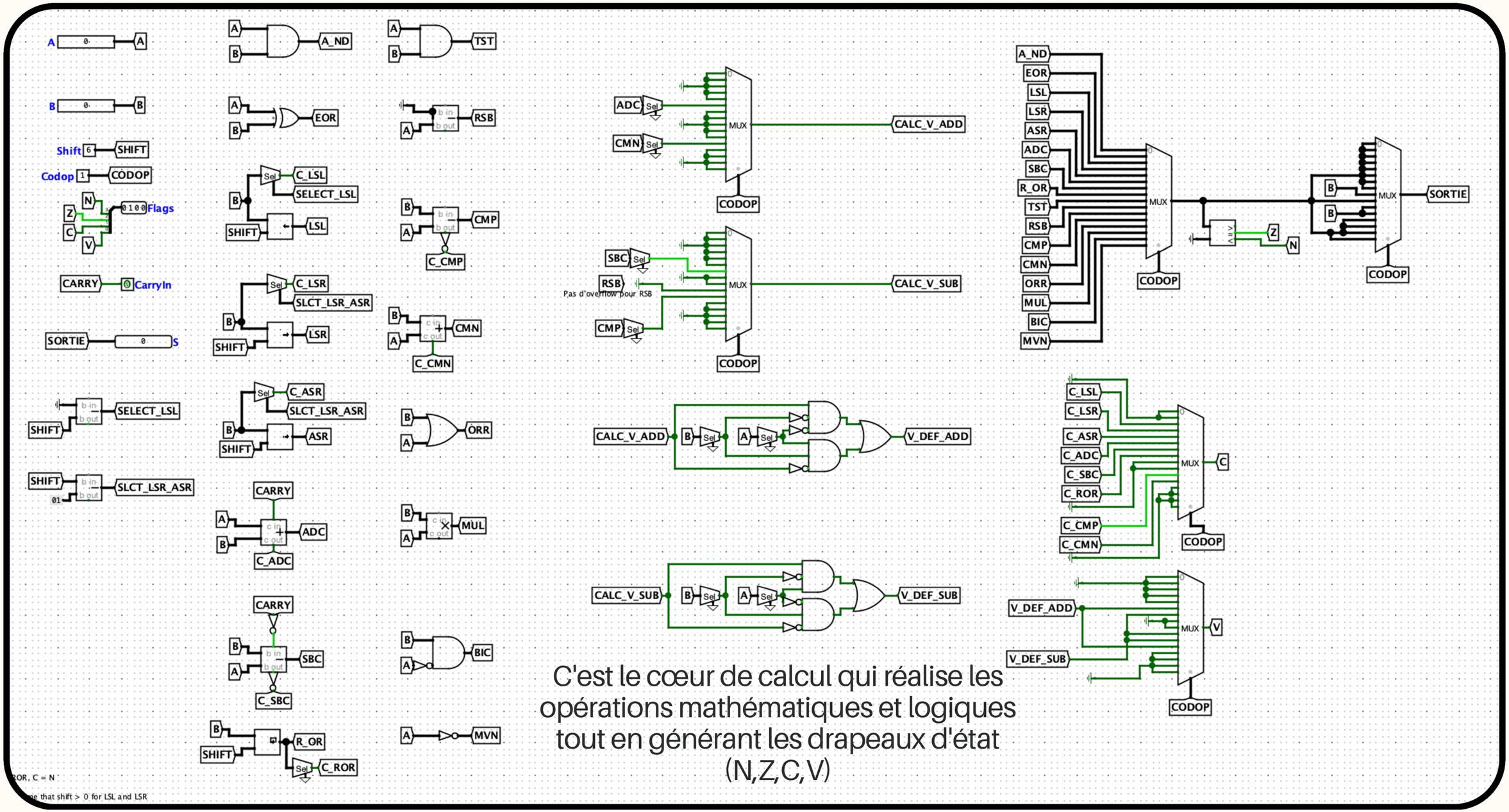
Opcode Décodeur

Contrôleur

CONTRÔLEUR :



ALU



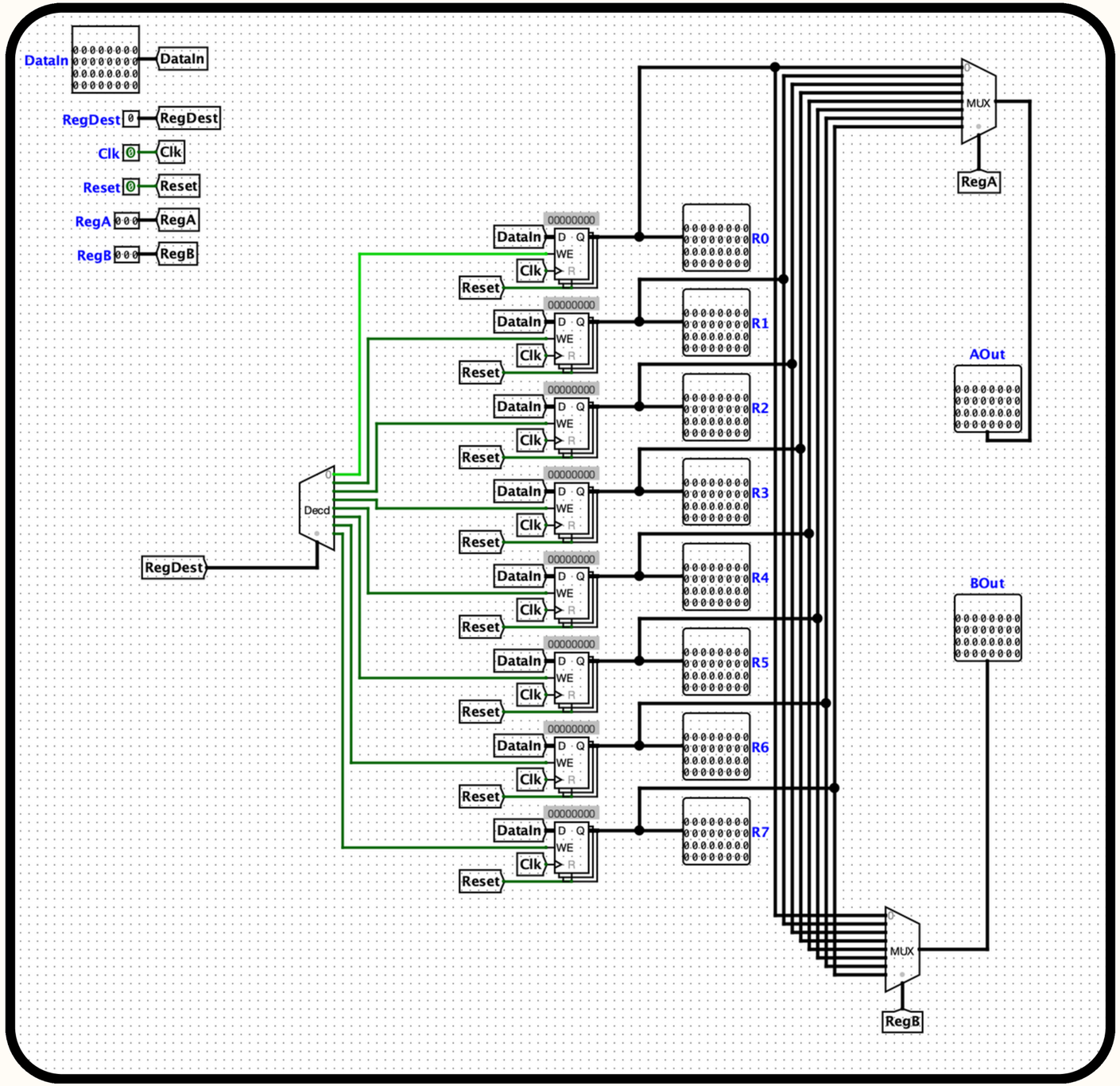
C'est le cœur de calcul qui réalise les opérations mathématiques et logiques tout en générant les drapeaux d'état (N,Z,C,V)

ROR, C = N

e that shift > 0 for LSL and LSR

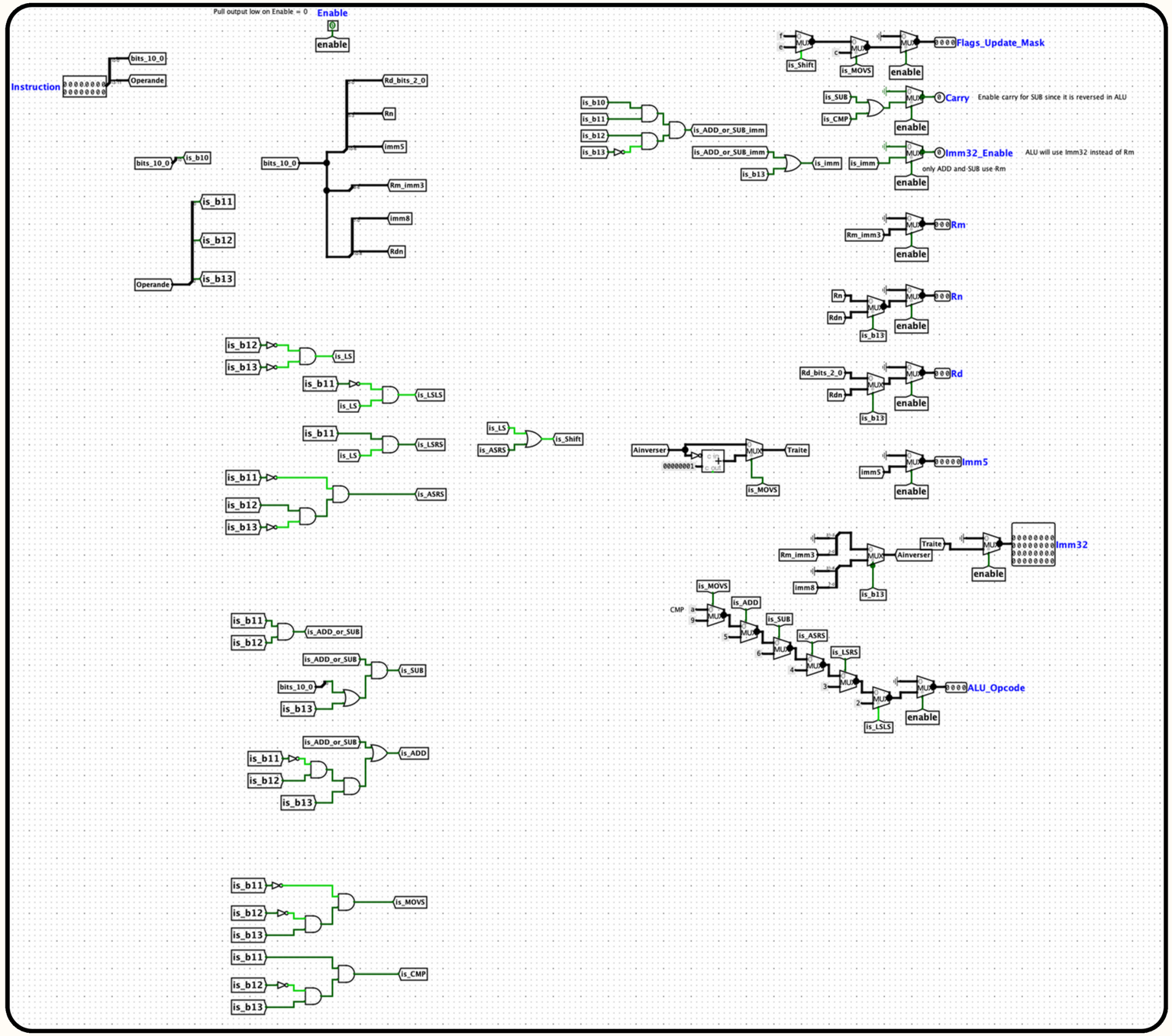
BANC DE REGISTRE

Il s'agit d'une mémoire interne rapide stockant les valeurs des registres (R0 à R7) utilisées pour les calculs du processeur.



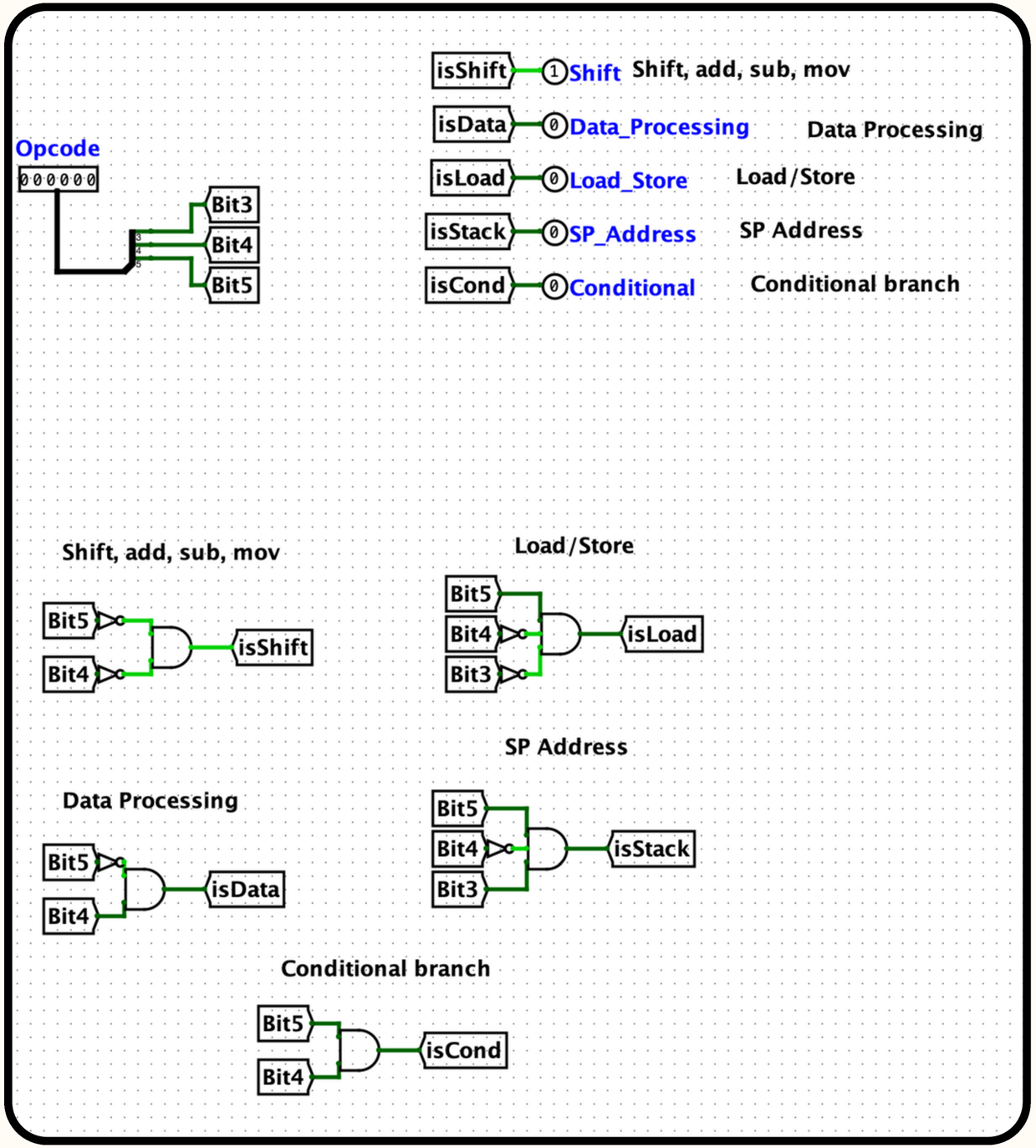
SHIFT, ADD, SUB, MOV

Ce bloc est dédié au décodage et au traitement des instructions de décalage, d'addition, de soustraction et de déplacement de données.



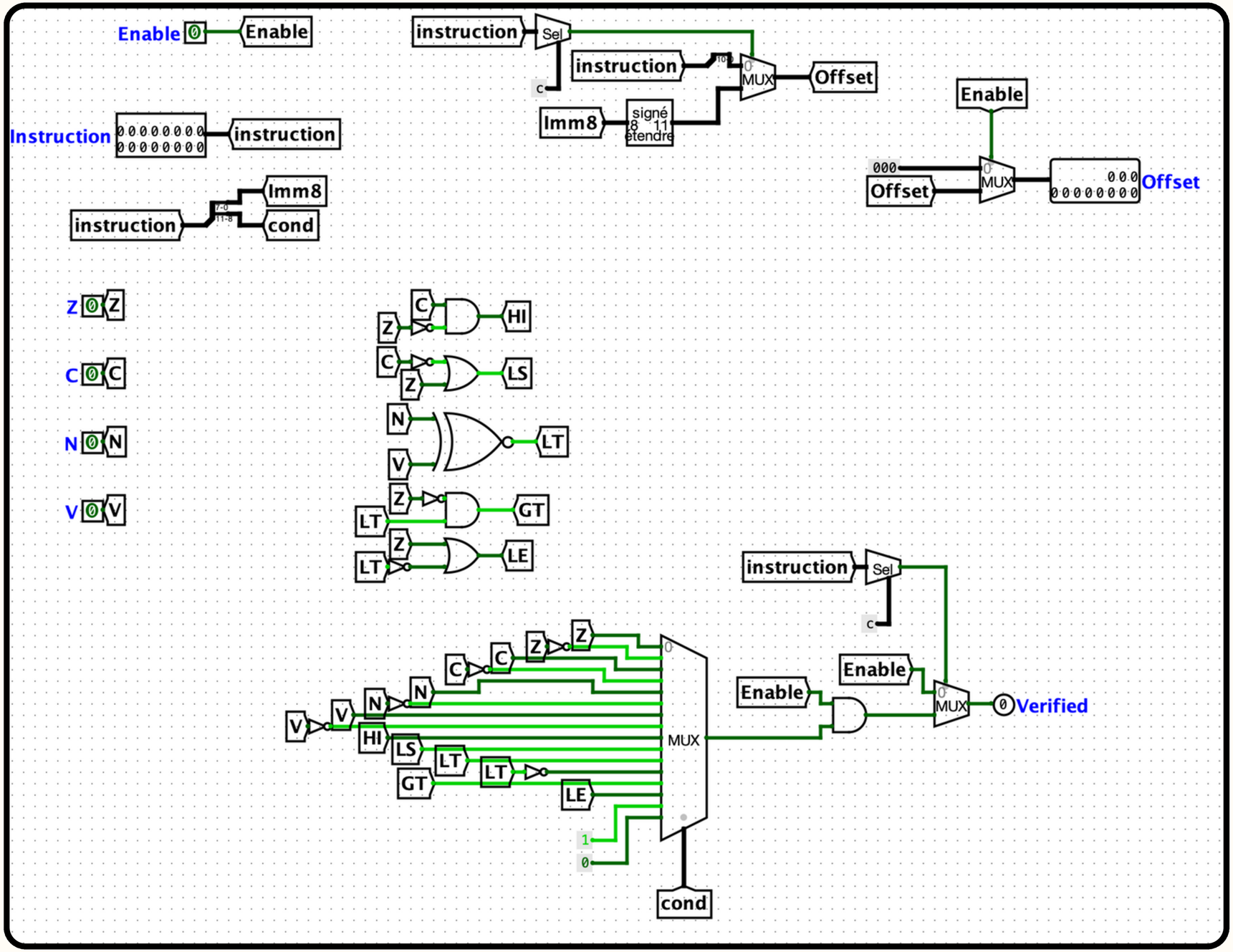
OPCODE, DECODEUR

Ce composant analyse les bits de poids fort de l'instruction pour identifier sa catégorie et activer les blocs correspondants du contrôleur.



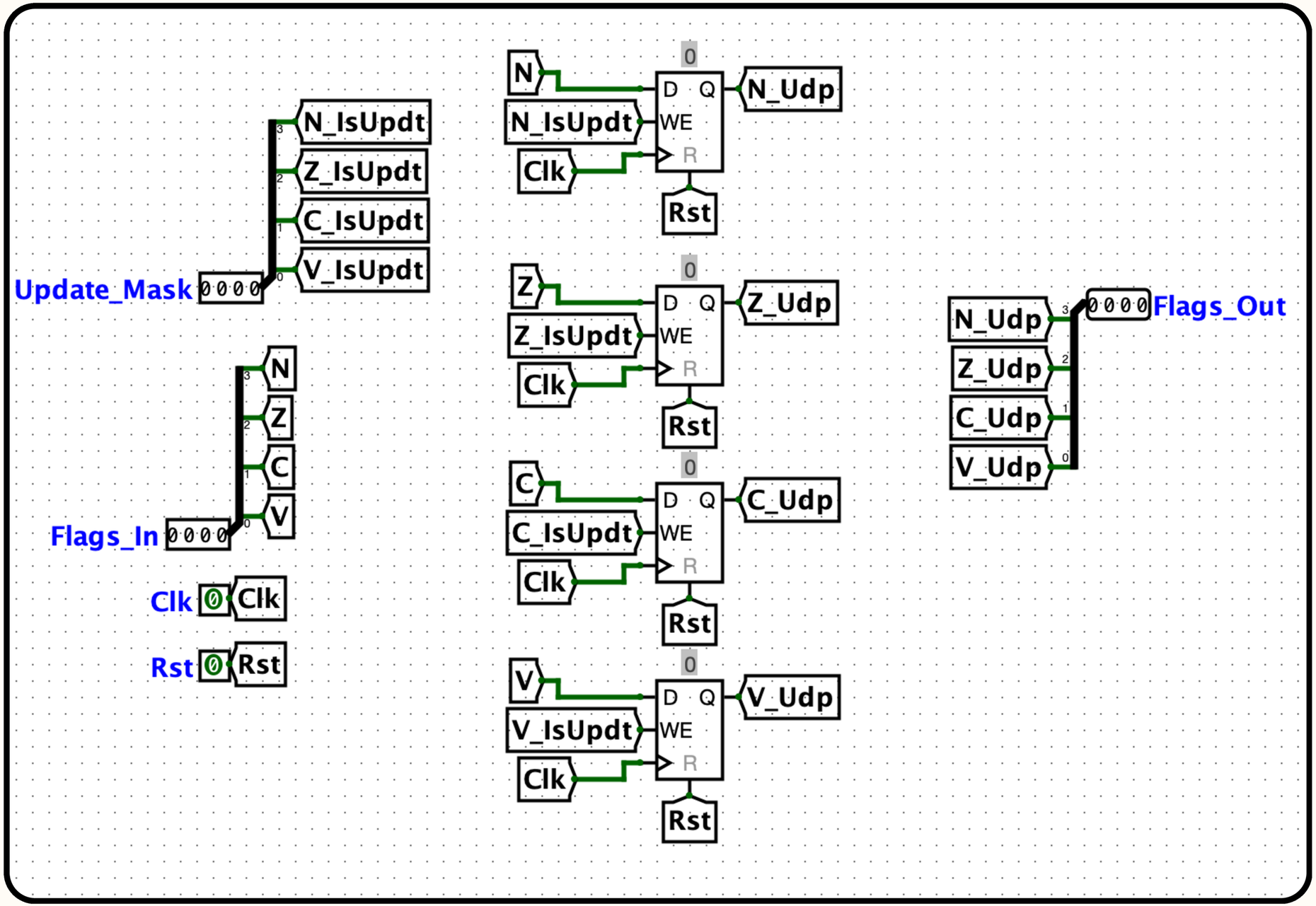
CONDITIONAL

Ce module évalue si une condition de branchement est remplie en fonction des drapeaux pour décider d'un saut dans le programme.

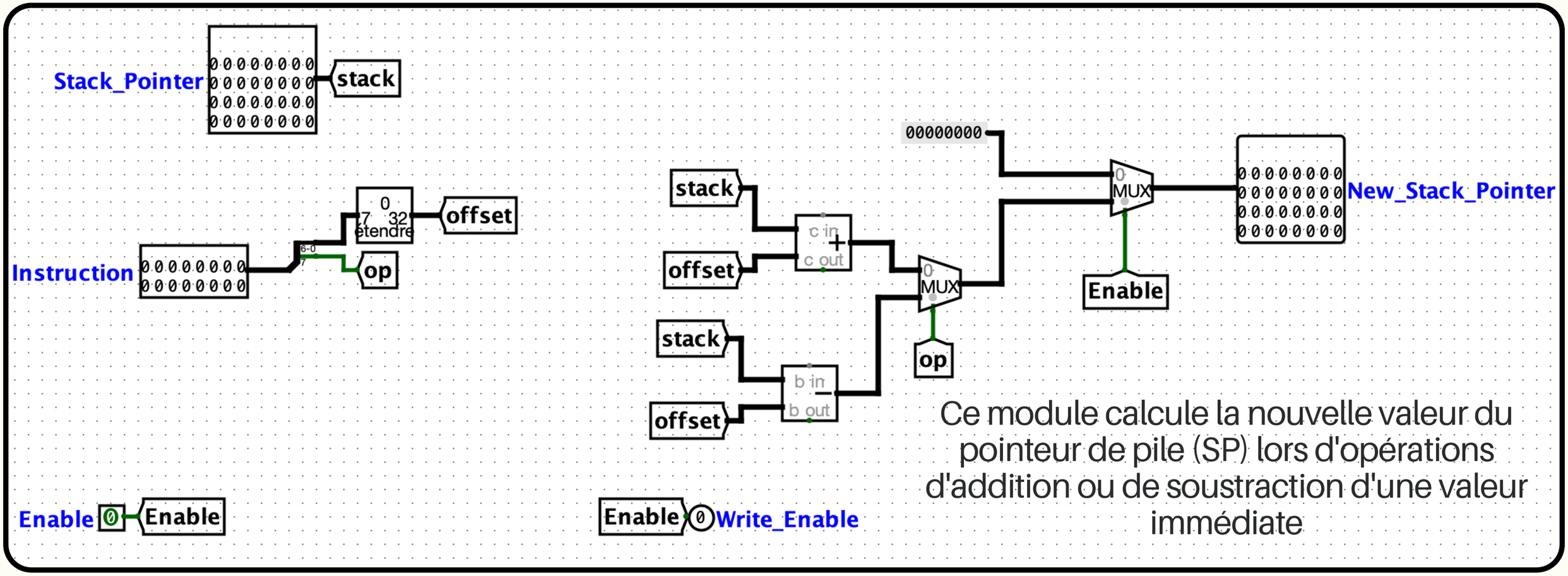


FLAG APSR

Ce bloc conserve les drapeaux d'état (N,Z,C,V) générés par la dernière instruction pour les rendre disponibles pour les instructions suivantes

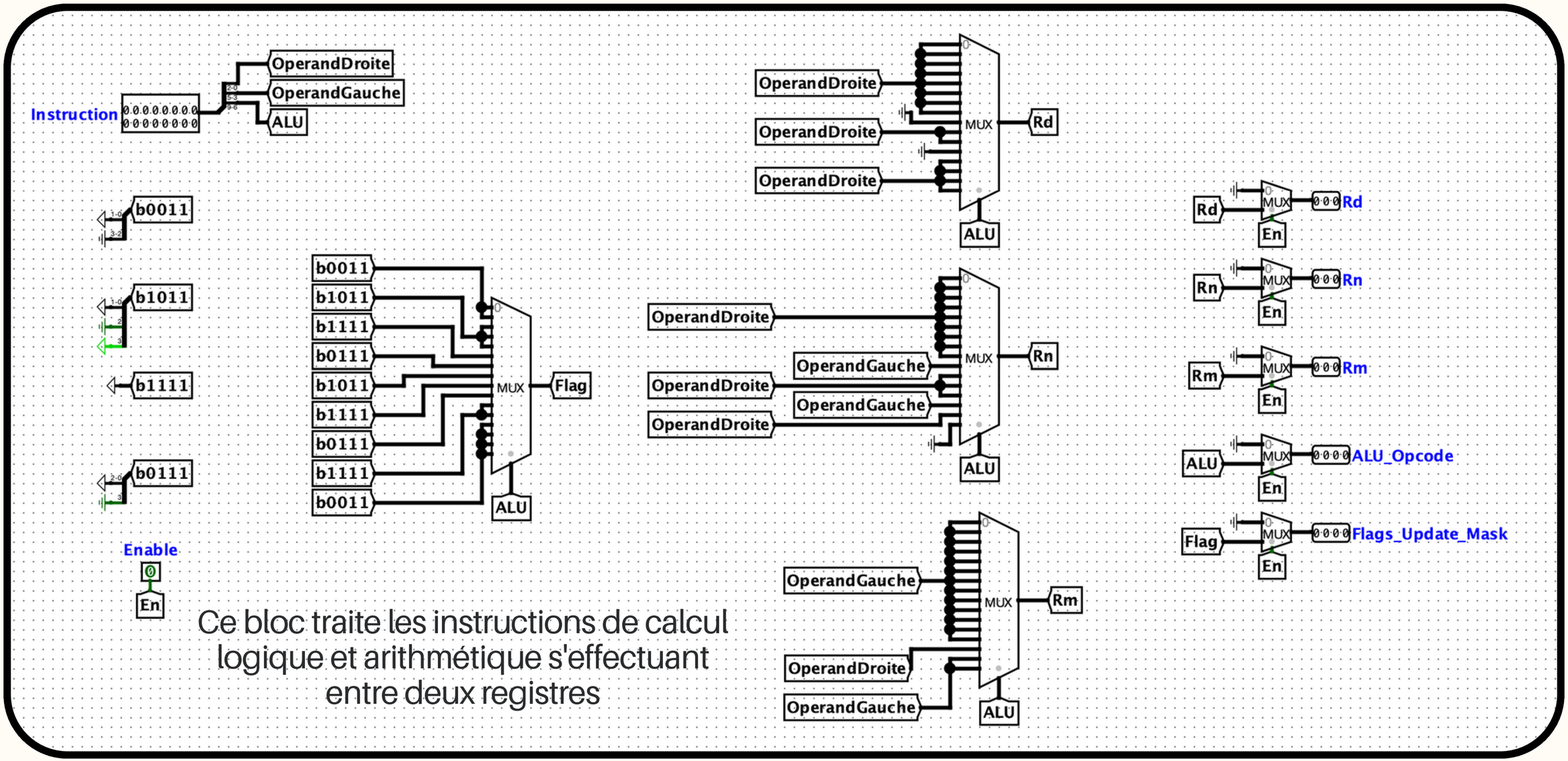


SP ADDRESS



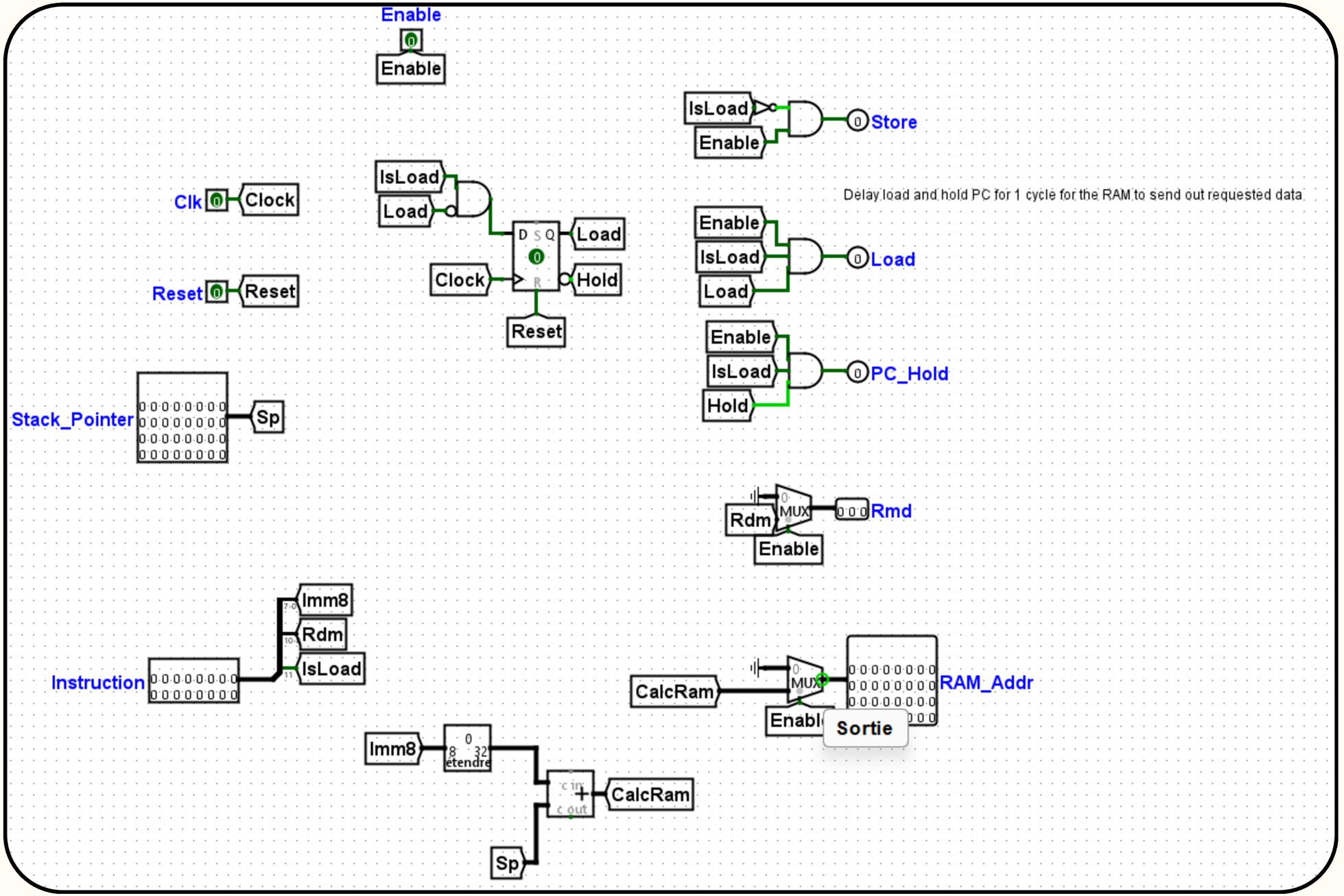
Ce module calcule la nouvelle valeur du pointeur de pile (SP) lors d'opérations d'addition ou de soustraction d'une valeur immédiate

DATA PROCESSING



LOAD/STORE

Il gère les transferts de données (lecture et écriture) entre les registres du processeur et la mémoire RAM



NOTRE TABLEAU DE SYNTHÈSE REMANIÉ

Concrètement, il sert à:

- **l'Identification des Opcodes** : Il donne les bits de poids fort uniques pour chaque instruction (ex: 00100 pour MOVS).
- **la structure des arguments** : Il indique la position exacte des bits pour les registres (Rd,Rn,Rm) et les valeurs immédiates dans le code de 16 bits.
- **règles de traduction** : Il sert de dictionnaire de référence pour transformer chaque ligne de texte assembleur en sa valeur hexadécimale correspondante comprise par Logisim

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
Logical Shift Left	LSLS	<Rd>	<Rn>	<imm5>	0	0	0	0	0	imm5					Rn			Rd			x	0	x	x
Logical Shift Right	LSRS	<Rd>	<Rn>	<imm5>	0	0	0	0	1	imm5					Rn			Rd			x	0	x	x
Arithmetic Shift Right	ASRS	<Rd>	<Rn>	<imm5>	0	0	0	1	0	imm5					Rn			Rd			x	0	x	x
Shift, add, sub, mov					0	0	opcode																	
Add register	ADDS	<Rd>	<Rn>	<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd			x	x	x	x
Substract register	SUBS	<Rd>	<Rn>	<Rm>	0	0	0	1	1	0	1	Rm			Rn			Rd			x	x	x	x
Add 3-bit immediate	ADDS	<Rd>	<Rn>	<imm3>	0	0	0	1	1	1	0	imm3			Rn			Rd			x	x	x	x
Substract 3-bit immediate	SUBS	<Rd>	<Rn>	<imm3>	0	0	0	1	1	1	1	imm3			Rn			Rd			x	x	x	x
Move	MOVS	<Rd>	<imm8>		0	0	1	0	0	Rd					imm8						0	0	x	x
Compare	CMP	<Rd>	<imm8>		0	0	1	0	1	Rd					imm8						x	x	x	x
Add 8-bit immediate	ADDS	<Rdn>	<imm8>		0	0	1	1	0	Rdn					imm8						x	x	x	x
Substract 8-bit immediate	SUBS	<Rdn>	<imm8>		0	0	1	1	1	Rdn					imm8						x	x	x	x
Data proccessing					0	1	0	0	0	0	opcode													
Bitwise AND	ANDS	<Rdn>	<Rm>		0	1	0	0	0	0	0	0	0	0	Rm			Rdn			0	0	x	x
Exclusive OR	EORS	<Rdn>	<Rm>		0	1	0	0	0	0	0	0	0	1	Rm			Rdn			0	0	x	x
Logical Shift Left	LSLS	<Rdn >	<Rm >		0	1	0	0	0	0	0	0	1	0	Rm			Rdn			x	0	x	x
Logical Shift Right	LSRS	<Rdn >	<Rm >		0	1	0	0	0	0	0	0	1	1	Rm			Rdn			x	0	x	x
Arithmetic Shift Right	ASRS	<Rdn >	<Rm >		0	1	0	0	0	0	0	1	0	0	Rm			Rdn			x	0	x	x
Add with Carry	ADCS	<Rdn >	<Rm >		0	1	0	0	0	0	0	1	0	1	Rm			Rdn			x	x	x	x
Substract with Carry	SBCS	<Rdn >	<Rm>		0	1	0	0	0	0	0	1	1	0	Rm			Rdn			0	x	x	x
Rotate Right	RORS	<Rdn >	<Rm >		0	1	0	0	0	0	0	1	1	1	Rm			Rdn			x	0	x	x
Set Flags on bitwise AND	TST	<Rn >	<Rm >		0	1	0	0	0	0	1	0	0	0	Rm			Rn			x	x	x	x
Reverse Substract from O	RSBS	<Rd >	<Rn >	0	0	1	0	0	0	0	1	0	0	1	Rn			Rd			0	x	x	x
Compare Registers	CMP	<Rn >	<Rm >		0	1	0	0	0	0	1	0	1	0	Rm			Rn			x	x	x	x
Compare Negative	CMN	<Rn >	<Rm >		0	1	0	0	0	0	1	0	1	1	Rm			Rn			x	x	x	x
Logical OR	ORRS	<Rdn >	<Rm >		0	1	0	0	0	0	1	1	0	0	Rm			Rdn			0	0	x	x
Multiply Two Registers	MULS	<Rdm >	<Rn >	<Rdm >	0	1	0	0	0	0	1	1	0	1	Rn			Rdm			0	0	x	x
Bit Clear	BICS	<Rdn >	<Rm >		0	1	0	0	0	0	1	1	1	0	Rm			Rdn			0	0	x	x
Bitwise NOT	MVNS	<Rd >	<Rm >		0	1	0	0	0	0	1	1	1	1	Rm			Rd			0	0	x	x
Load / Store					opcode																			
Store Register	STR	<Rt>	<imm8>		1	0	0	1	0	Rt					imm8						0	0	0	0
Load Register	LDR	<Rt>	<imm8>		1	0	0	1	1	Rt					imm8						0	0	0	0
Miscellaneous 16-bit instructions					opcode																			
Add Immediate to SP	ADD	<imm7/4>			1	0	1	1	0	0	0	0	0	imm7							0	0	0	0
Substract Immediate from SP	SUB	<imm7/4>			1	0	1	1	0	0	0	0	1	imm7							0	0	0	0
Conditional Branch	B	<c>	<label>		1	1	0	1	cond					imm8										
égalité	BEQ	<im8>			1	1	0	1	0	0	0	0	imm8								0	0	0	x
difference	BNE	<im8>			1	1	0	1	0	0	0	1	imm8								0	0	0	x
retenue	BCS	<im8>			1	1	0	1	0	0	1	0	imm8								x	0	0	0
pas de retenue	BCC	<im8>			1	1	0	1	0	0	1	1	imm8								x	0	0	0
negatif	BMI	<im8>			1	1	0	1	0	1	0	0	imm8								0	0	x	0
positif ou nul	BPL	<im8>			1	1	0	1	0	1	0	1	imm8								0	0	x	0
dépassement de capacité	BVS	<im8>			1	1	0	1	0	1	1	0	imm8								0	x	0	0
pas de dépassement de capacité	BVC	<im8>			1	1	0	1	0	1	1	1	imm8								0	x	0	0
supérieur (non signé]	BHI	<im8>			1	1	0	1	1	0	0	0	imm8								x	0	0	x
inférieur ou égal (non signé)	BLS	<im8>			1	1	0	1	1	0	0	1	imm8								x	0	0	x
supérieur ou égal (signé)	BGE	<im8>			1	1	0	1	1	0	1	0	imm8								0	x	x	0
inférieur (signé)	BLT	<im8>			1	1	0	1	1	0	1	1	imm8								0	x	x	0
supérieur (signé)	BGT	<im8>			1	1	0	1	1	1	0	0	imm8								0	x	x	x
inférieur ou égal (signé)	BLE	<im8>			1	1	0	1	1	1	0	1	imm8								0	x	x	x
toujours vrai	B ou BAL	<im8>			1	1	0	1	1	1	1	0	imm8								0	0	0	0
toujours faux		<im8>			1	1	0	1	1	1	1	1	imm8								0	0	0	0
Unconditional Branch	B	<label>			1	1	1	0	0	imm11											0	0	0	0

NOS TESTS

Scénario d'Intégratio	Statut
Conditionnal	OK
Data Processing	80%
Load/Store	50%
Shift Add Sub Mov	OK
Miscallaneous	OK

L'ASSEMBLEUR PYTHON

Dictionnaires et utilitaire

```
##########################################################################
# la définition des constantes (Tableau de synthèse)
REGISTRES = {
    'r0': 0, 'r1': 1, 'r2': 2, 'r3': 3,
    'r4': 4, 'r5': 5, 'r6': 6, 'r7': 7,
    'sp': 13, 'lr': 14, 'pc': 15
}
##########################################################################
```

```
##########################################################################
# codes de conditions pr les branchements
CONDITIONS = {
    'eq': 0b0000, 'ne': 0b0001, 'cs': 0b0010, 'cc': 0b0011,
    'mi': 0b0100, 'pl': 0b0101, 'vs': 0b0110, 'vc': 0b0111,
    'hi': 0b1000, 'ls': 0b1001, 'ge': 0b1010, 'lt': 0b1011,
    'gt': 0b1100, 'le': 0b1101, 'al': 0b1110 # Always
}
##########################################################################

# Ajout pour la gestion des sauts (donc le Double Passage)
LABELS = {}
```

```
##########################################################################
def parse_val(arg, current_line=0):
    arg = arg.strip().lower()

    if arg in REGISTRES:
        return REGISTRES[arg]
    elif arg.startswith('#'):
        return int(arg.replace('#', ''), 0)
    elif arg in LABELS:
        # CALCUL DU SAUT (Offset)
        target_line = LABELS[arg]
        offset = target_line - current_line - 3
        return offset
    else:
        try:
            return int(arg, 0)
        except ValueError:
            return 0
##########################################################################
```

L'ASSEMBLEUR PYTHON

Nettoyage et Cartographie des Labels

```python
# boucle qui mange les labels (ex: TRUC1: TRUC2: movs...)
# Tant que le premier mot finit par ':', c'est un label
while len(parts) > 0 and parts[0].endswith(':'):
    label_name = parts[0][:-1].lower() # On enlève le ':'
    LABELS[label_name] = instruction_count
    parts.pop(0) # On retire le label de la ligne et on continue
```

```python
# --- FILTRES C ---
mnemonic = parts[0].lower()
if mnemonic in ['push', 'pop', 'bx', 'blx', 'bl']: continue
if mnemonic == 'add' and 'r7' in parts and 'sp' in parts: continue

# On reconstruit la ligne propre sans les labels pour la passe 2
clean_instruction = " ".join(parts)      You, last week • correction
lines_to_process.append(clean_instruction)
instruction_count += 1
```

L'ASSEMBLEUR PYTHON

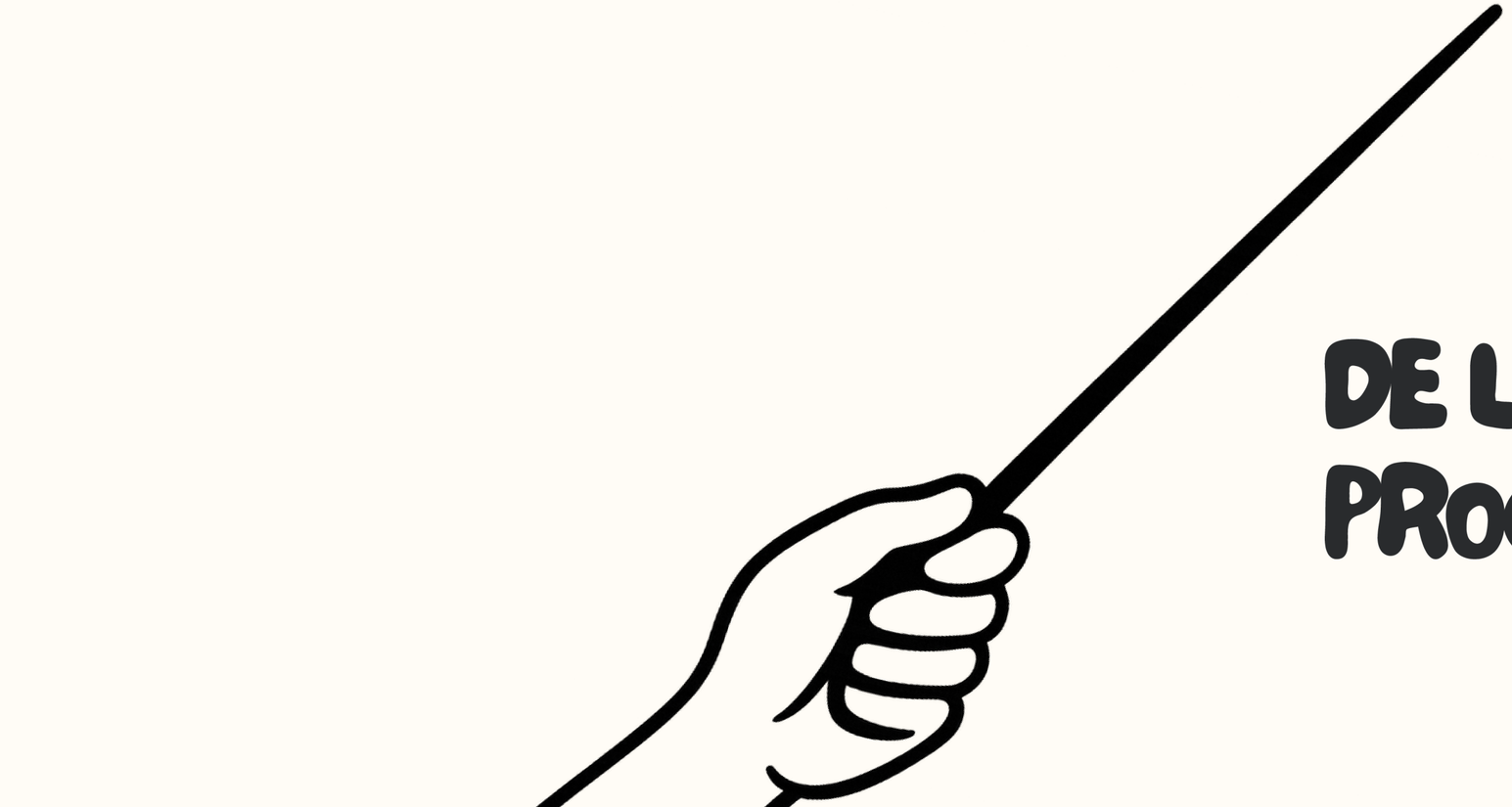
Encodage binaire et génération

```
# ------------------------------------------------------------------------------
# GROUPE 1 : Décalages immédiats (LSLS, LSRS, ASRS)
# ------------------------------------------------------------------------------
elif mnemonic in ['lsls', 'lsrs', 'asrs'] and len(args) == 3 and args[2].startswith('#'):
    if mnemonic == 'lsls': base = 0b00000
    elif mnemonic == 'lsrs': base = 0b00001
    elif mnemonic == 'asrs': base = 0b00010
    rd = parse_val(args[0])
    rn = parse_val(args[1])
    imm5 = parse_val(args[2])
    instr_bin = (base << 11) | (imm5 << 6) | (rn << 3) | rd
```

```
# ------------------------------------------------------------------------------
# GROUPE 6 & 7 : BRANCHEMENTS
# ------------------------------------------------------------------------------
elif mnemonic.startswith('b') and mnemonic != 'b':
    cond_str = mnemonic[1:]
    if cond_str in CONDITIONS:
        cond_bits = CONDITIONS[cond_str]
        offset = parse_val(args[0], current_line)
        instr_bin = (0b1101 << 12) | (cond_bits << 8) | (offset & 0xFF)

elif mnemonic == 'b':
    offset = parse_val(args[0], current_line)
    instr_bin = (0b11100 << 11) | (offset & 0x7FF)

return instr_bin
```

```
# --- PASSE 2 : GÉNÉRATION DU CODE ---
with open(output_file, 'w') as f_out:
    f_out.write("v2.0 raw\n")

    for idx, line in enumerate(lines_to_process):
        parts = line.replace(',', ' ').replace('[', ' ').replace(']', ' ').split()
        if not parts: continue

        mnemonic = parts[0].lower()

        # Clang met 'mov' au lieu de 'movs'
        if mnemonic == 'mov': mnemonic = 'movs'

        args = parts[1:]

        try:
            code = assembler_instruction(mnemonic, args, idx)
            f_out.write(f"{code:04x} ")
        except Exception as e:
            print(f"Erreur instruction {idx} ({line}): {e}")
            f_out.write("0000 ")

print(f"Succès ! Fichier généré : {output_file}")
```

PLACE AUX DÉMOS



DE L'ASSEMBLEUR AU PROCESSEUR

DÉMO DE L'ASSEMBLEUR : TESTS AUTOMATIQUES

```
 Voulez-vous tester, ou compiler un .s? (t/c)
t
--- Lancement des tests (14 fichiers) ---
Testing ./code_asm/test_integration/conditional/branch.s... PASSED
Testing ./code_asm/test_integration/data_processing/1-4_instructions.s... PASSED
Testing ./code_asm/test_integration/data_processing/11-12_instructions.s... PASSED
Testing ./code_asm/test_integration/data_processing/13-16_instructions.s... PASSED
Testing ./code_asm/test_integration/data_processing/5-10_instructions.s... PASSED
Testing ./code_asm/test_integration/load_store/load_store.s... PASSED
Testing ./code_asm/test_integration/miscellaneous/sp.s... PASSED
Testing ./code_asm/test_integration/shift_add_sub_mov/1-4_instructions.s... PASSED
Testing ./code_asm/test_integration/shift_add_sub_mov/5-8_instructions.s... PASSED
Testing ./code_c/calckeyb.s... PASSED
Testing ./code_c/calculator.s... PASSED
Testing ./code_c/simple_add.s... PASSED
Testing ./code_c/testfp.s... PASSED
Testing ./code_c/tty.s... PASSED
```

```python
# --- FONCTION DE TEST AUTOMATIQUE ------------------------------------------
def run_automated_tests(base_folder):
    asm_files = glob.glob(os.path.join(base_folder, "**/*.s"), recursive=True)
    print(f"\nTesting or compiling? (t/c)\nt")
    print(f"--- Lancement des tests ({len(asm_files)} fichiers) ---")
    for asm_path in sorted(asm_files):
        ref_bin = asm_path.replace(".s", ".bin")
        temp_bin = "temp_test_output.bin"
        print(f"Testing {asm_path}...", end=" ")
        if not os.path.exists(ref_bin):
            print("Pas de .bin de référence")
            continue
        try:
            main(asm_path, temp_bin, silent=True)
            with open(temp_bin, 'r') as f1, open(ref_bin, 'r') as f2:
                if f1.read().strip().replace('\n', ' ') == f2.read().strip().replace('\n', ' '):
                    print("PASSED")
                else:
                    print("Différence binaire")
        except Exception:
            print("Crash")
    if os.path.exists("temp_test_output.bin"): os.remove("temp_test_output.bin")

#-------------------------------------------------------------------------------
```

MERCI POUR VOTRE ATTENTION !

PLACE AUX QUESTIONS