

Architecture : PARM

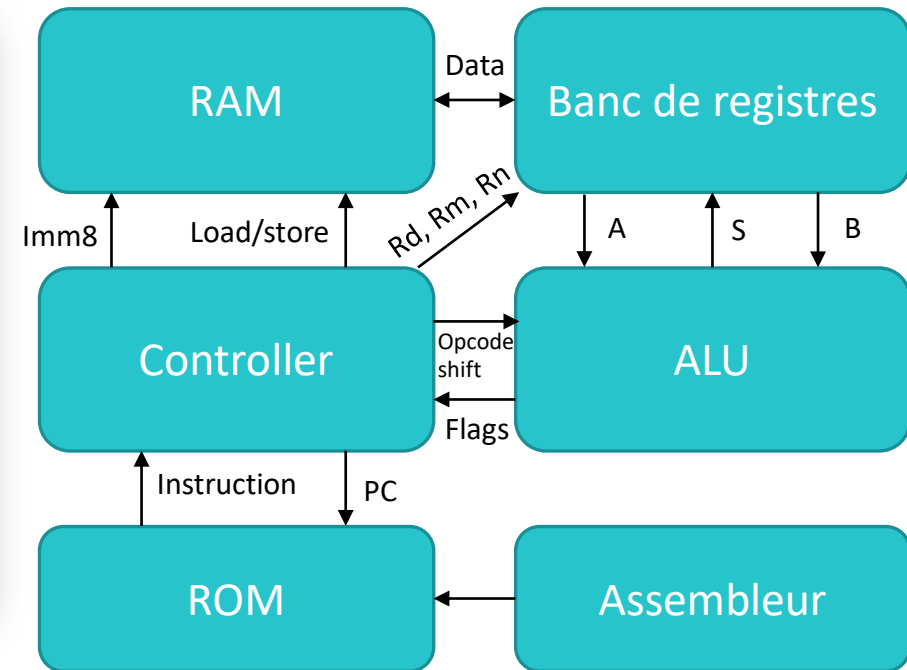
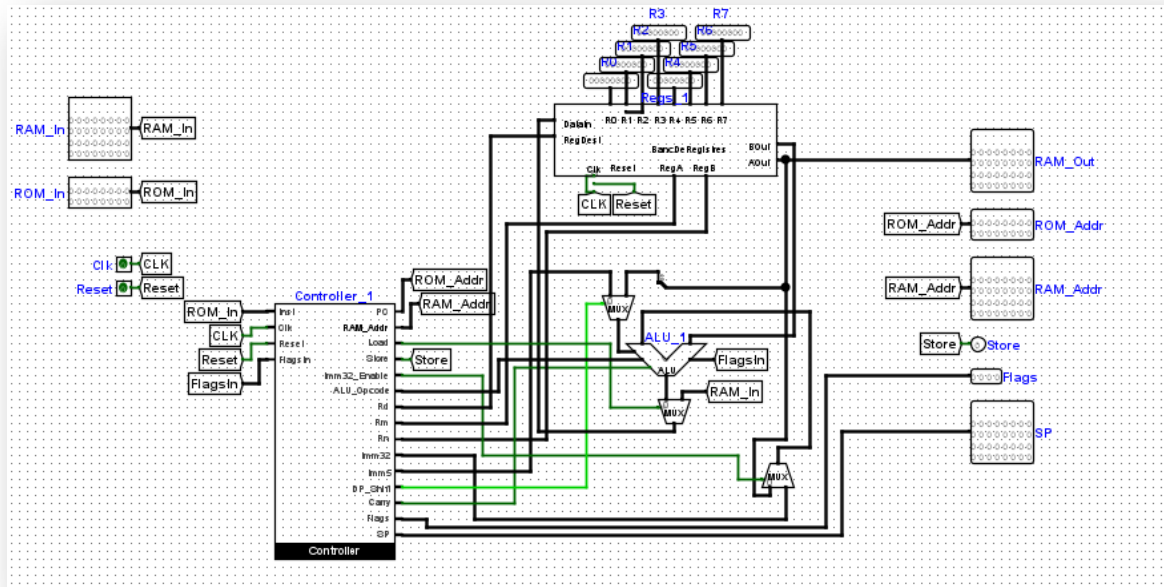
Groupe Polycrow

Marc Pinet – Arthur Rodriguez – Marcus Aas Jensen – Loïc Pantano

Sommaire

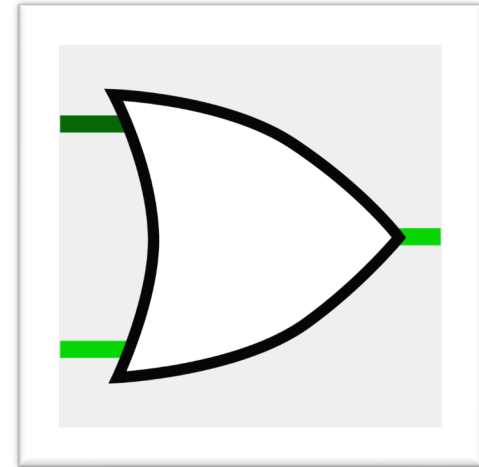
- Structure du projet (CPU)
- Composants
- Assembleur
- Passage des tests
- Simulateur logisim

Structure du projet (CPU)

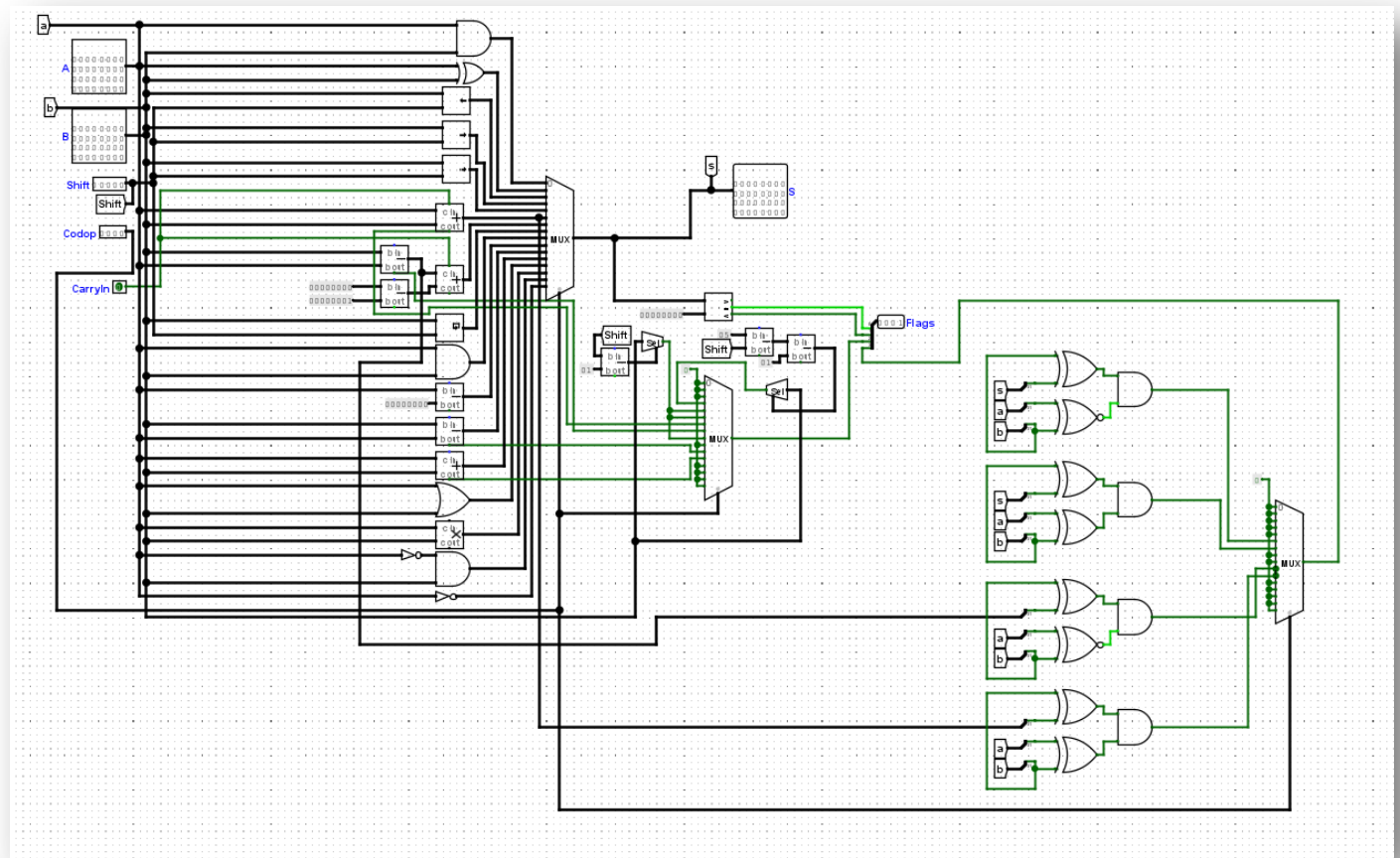


Composants Logisim

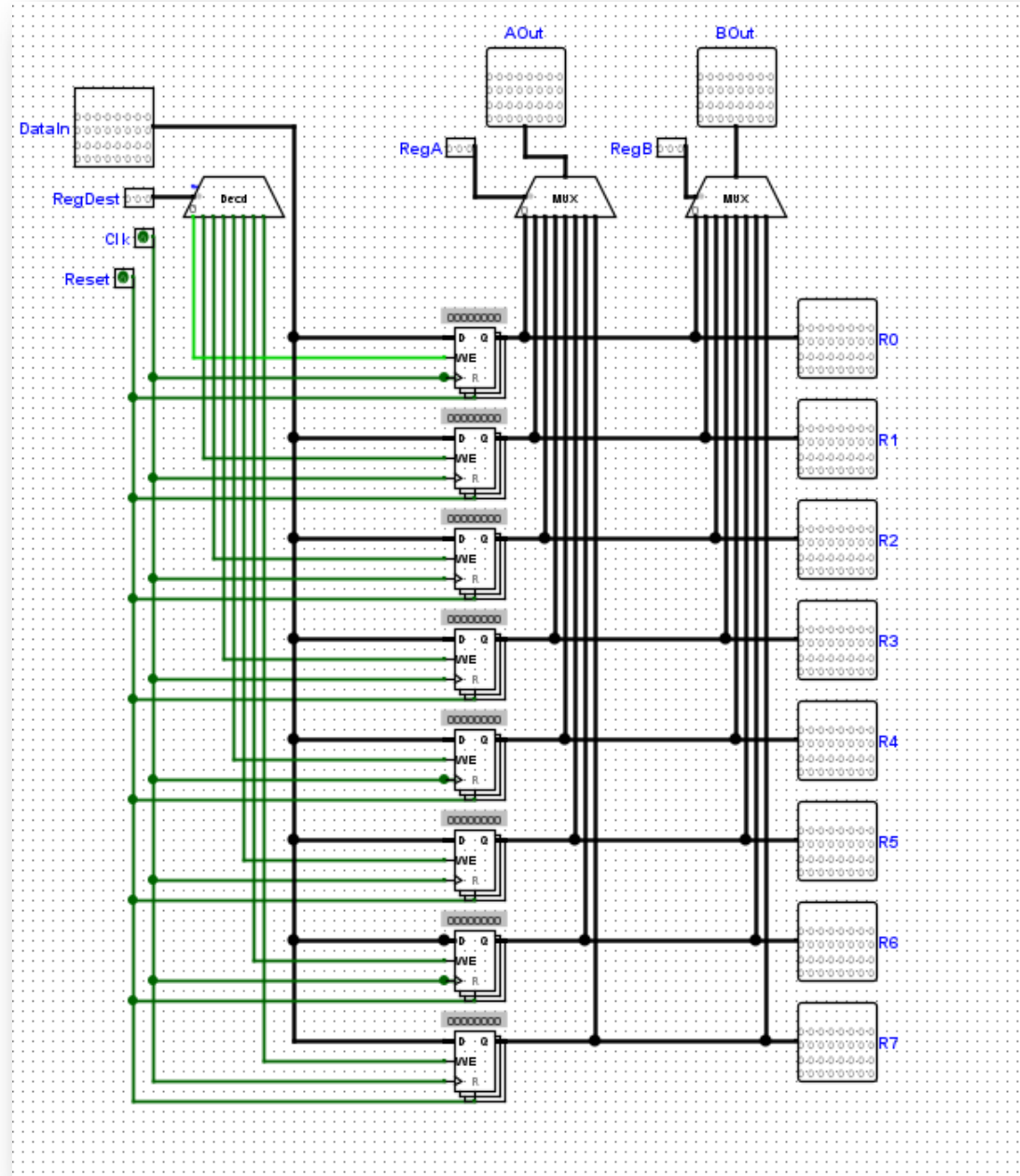
- ALU
- Banc de registres
- Opcode décodeur
- Shift, Add, Sub, Move
- Data processing
- Flag APSR
- Load store
- SP address
- Conditional
- Contrôleur (regroupement)



ALU



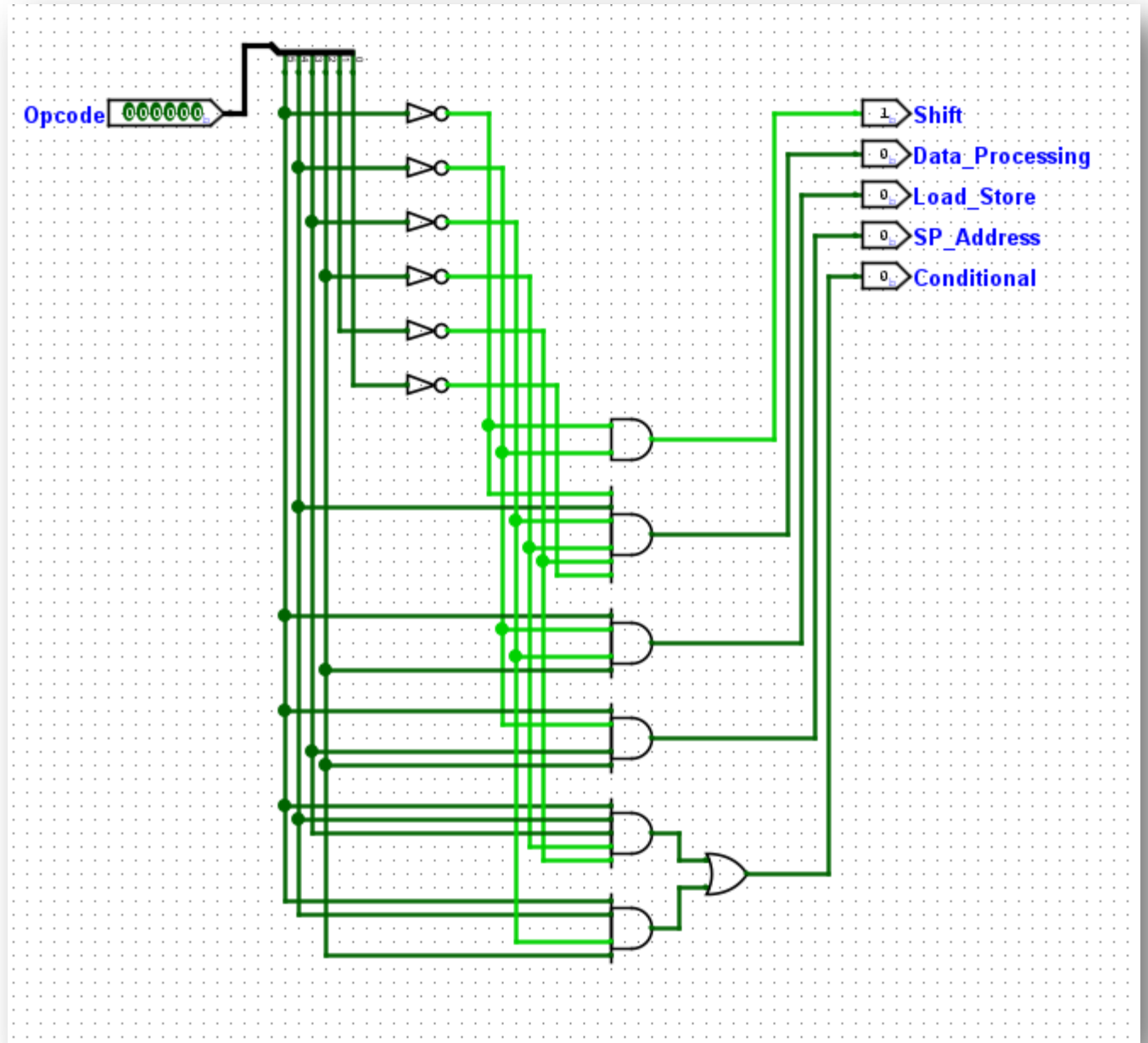
Banc de registres



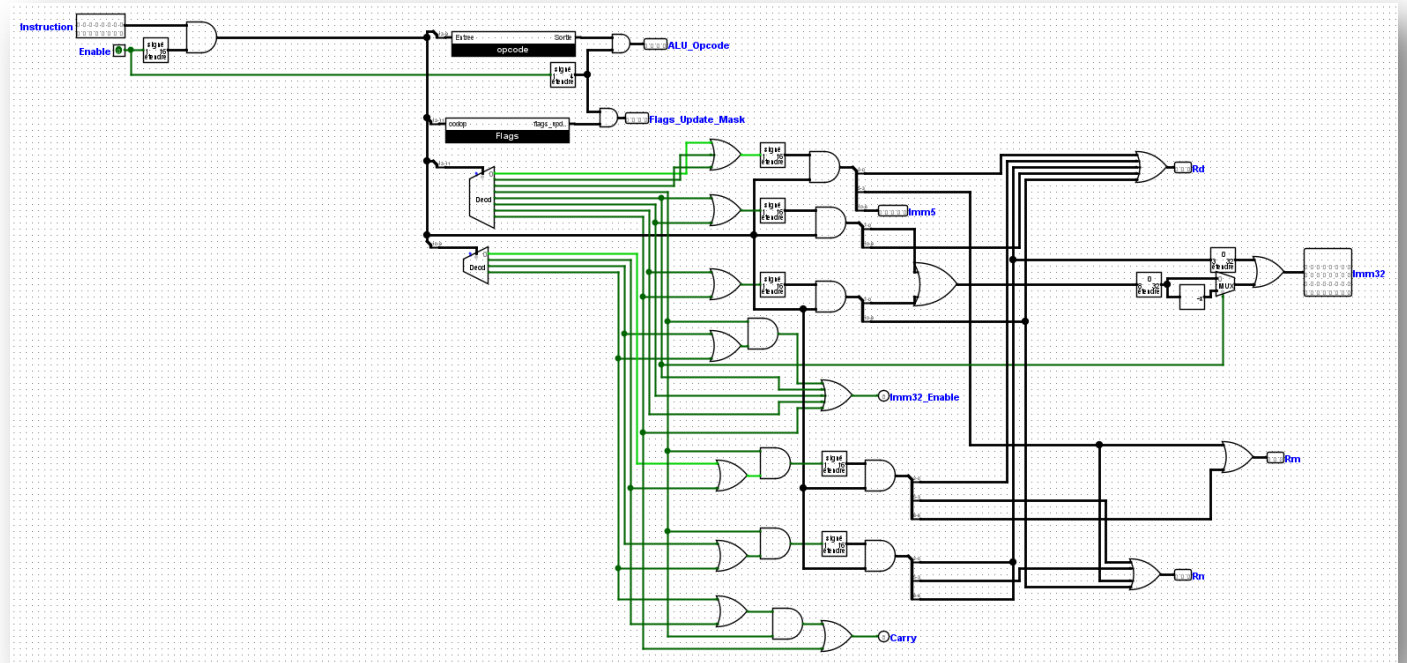
Opcode decoder

Opcode[5..0] | Shift Data_Processing Load_Store SP_Address Conditional

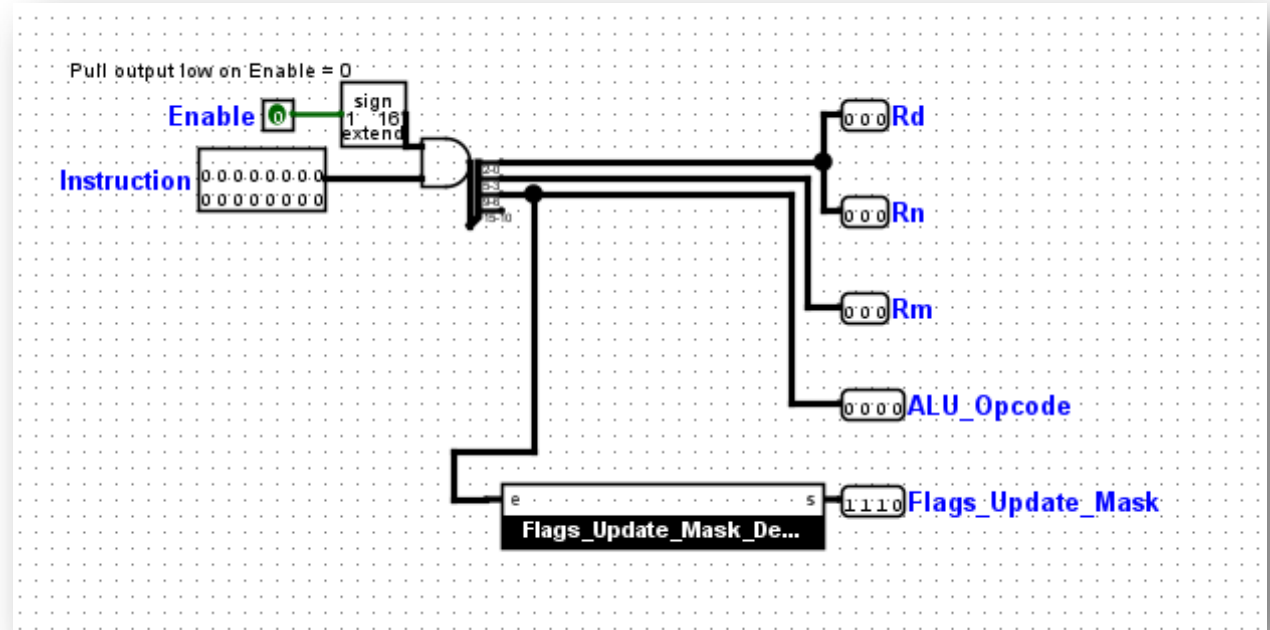
000000	1	0	0	0	0
000001	1	0	0	0	0
000010	1	0	0	0	0
000011	1	0	0	0	0
000100	1	0	0	0	0
000101	1	0	0	0	0
000110	1	0	0	0	0
000111	1	0	0	0	0
001000	1	0	0	0	0
001001	1	0	0	0	0
001010	1	0	0	0	0
001011	1	0	0	0	0
001100	1	0	0	0	0
001101	1	0	0	0	0
001110	1	0	0	0	0
001111	1	0	0	0	0
010000	0	1	0	0	0
010001	0	0	0	0	0
010010	0	0	0	0	0
010011	0	0	0	0	0
010100	0	0	0	0	0
010101	0	0	0	0	0
010110	0	0	0	0	0
010111	0	0	0	0	0
011000	0	0	0	0	0
011001	0	0	0	0	0
011010	0	0	0	0	0
011011	0	0	0	0	0
011100	0	0	0	0	0
011101	0	0	0	0	0
011110	0	0	0	0	0
011111	0	0	0	0	0
100000	0	0	0	0	0
100001	0	0	0	0	0
100010	0	0	0	0	0
100011	0	0	0	0	0
100100	0	0	1	0	0
100101	0	0	1	0	0
100110	0	0	1	0	0
100111	0	0	1	0	0
101000	0	0	0	0	0
101001	0	0	0	0	0
101010	0	0	0	0	0
101011	0	0	0	0	0
101100	0	0	0	1	0
101101	0	0	0	1	0
101110	0	0	0	1	0
101111	0	0	0	1	0
110000	0	0	0	0	0
110001	0	0	0	0	0
110010	0	0	0	0	0
110011	0	0	0	0	0
110100	0	0	0	0	1
110101	0	0	0	0	1
110110	0	0	0	0	1
110111	0	0	0	0	1
111000	0	0	0	0	1
111001	0	0	0	0	1
111010	0	0	0	0	0
111011	0	0	0	0	0
111100	0	0	0	0	0
111101	0	0	0	0	0
111110	0	0	0	0	0
111111	0	0	0	0	0



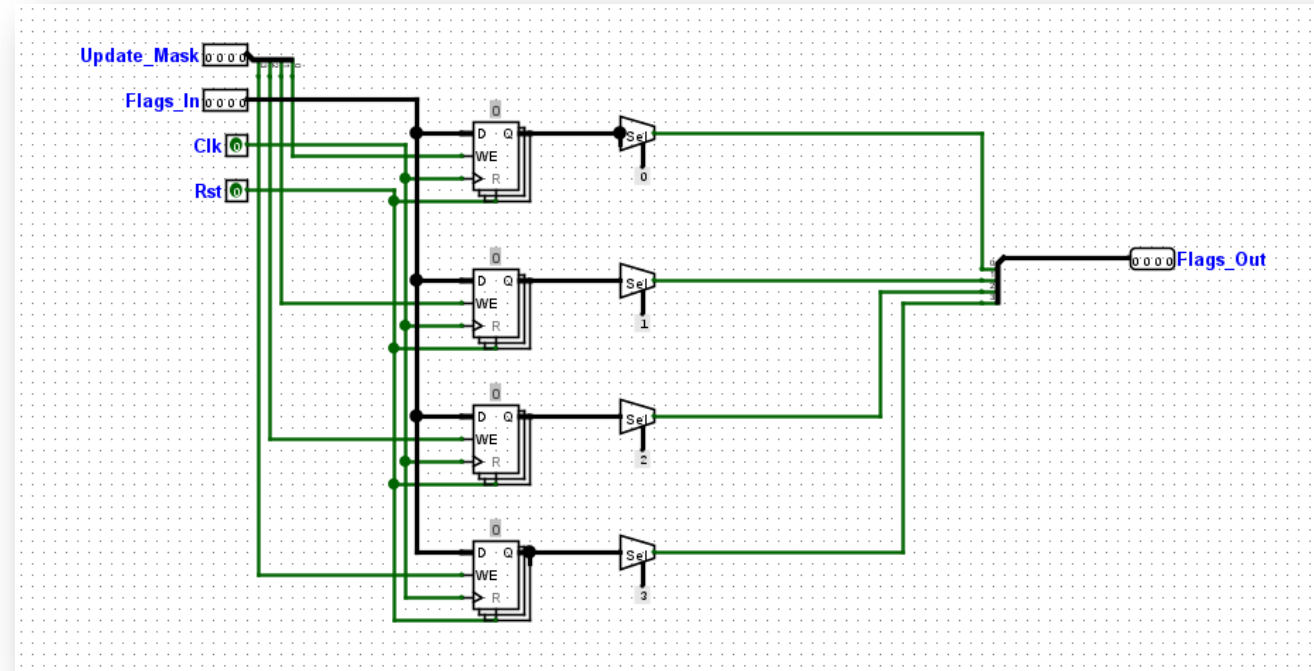
Shift
Add
Sub
Move



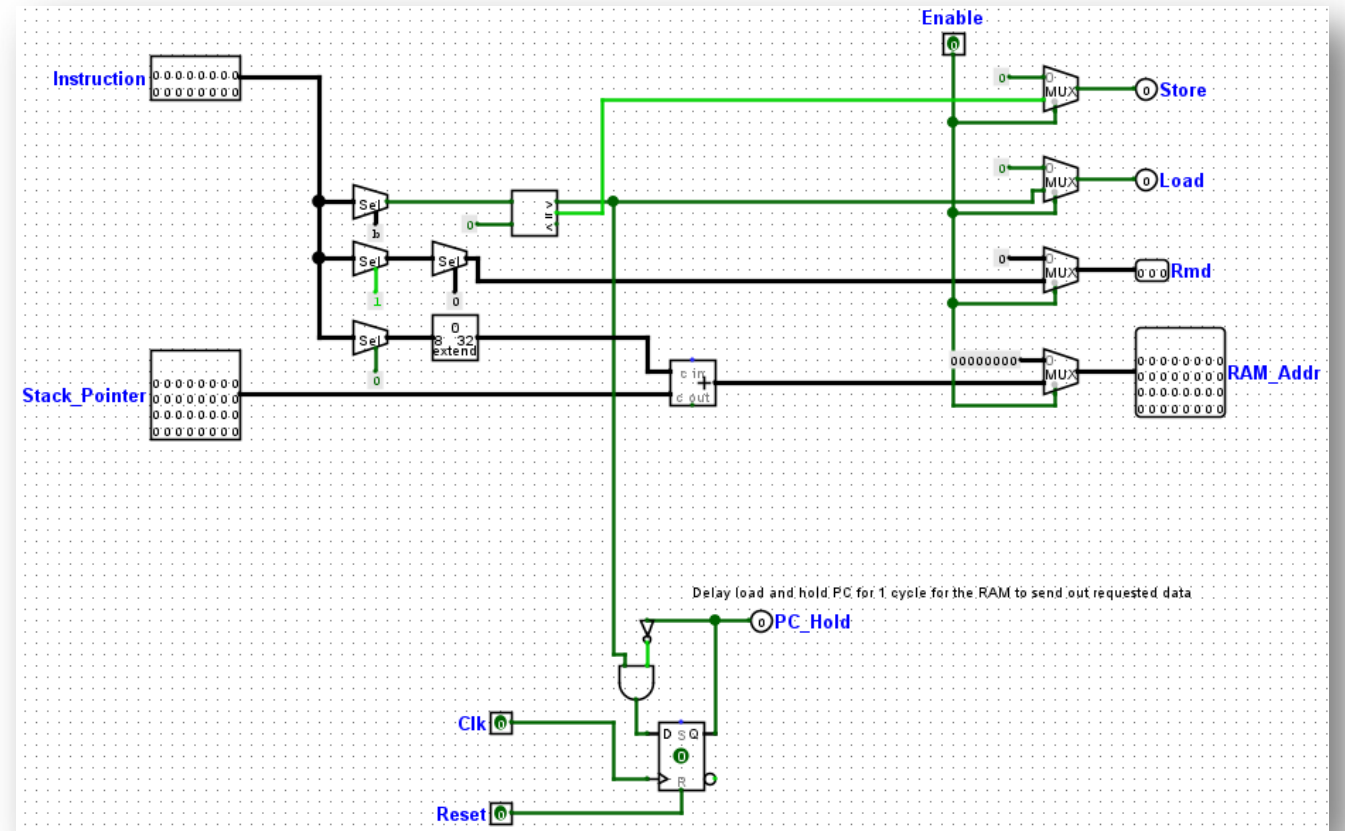
Data processing



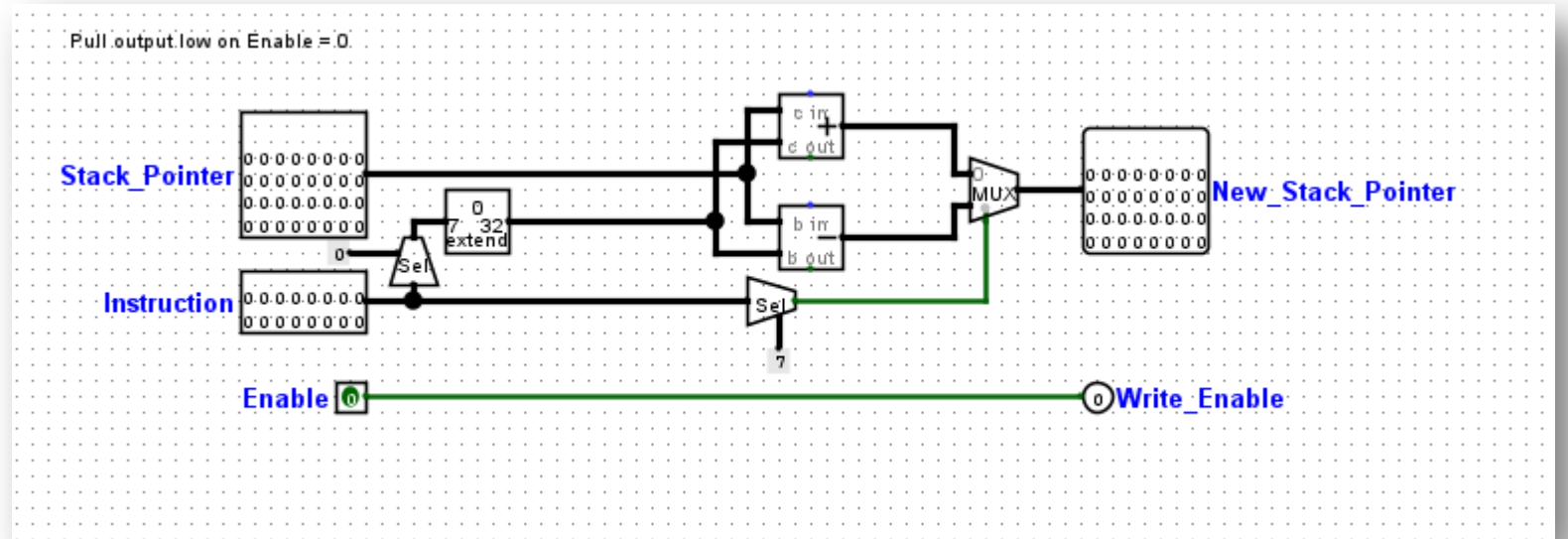
Flag APSR



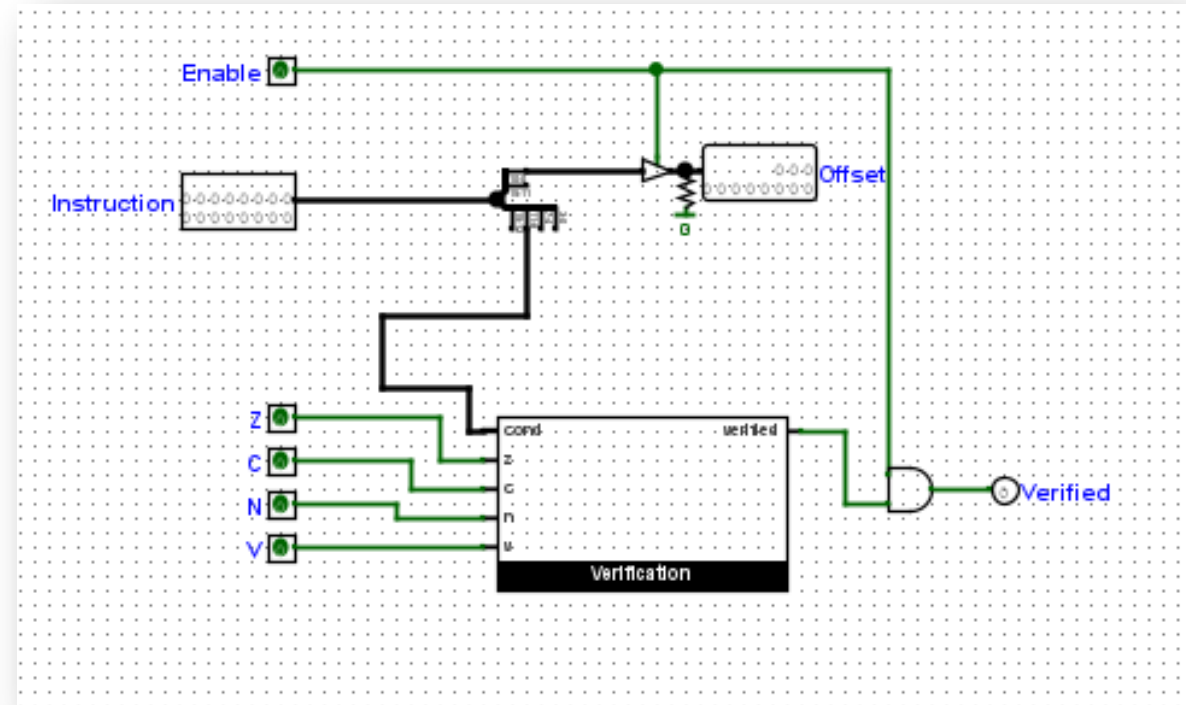
Load store



SP Address



Conditional



Controller

Composants regroupés :

Opcode decoder

Shift Add Sub Move

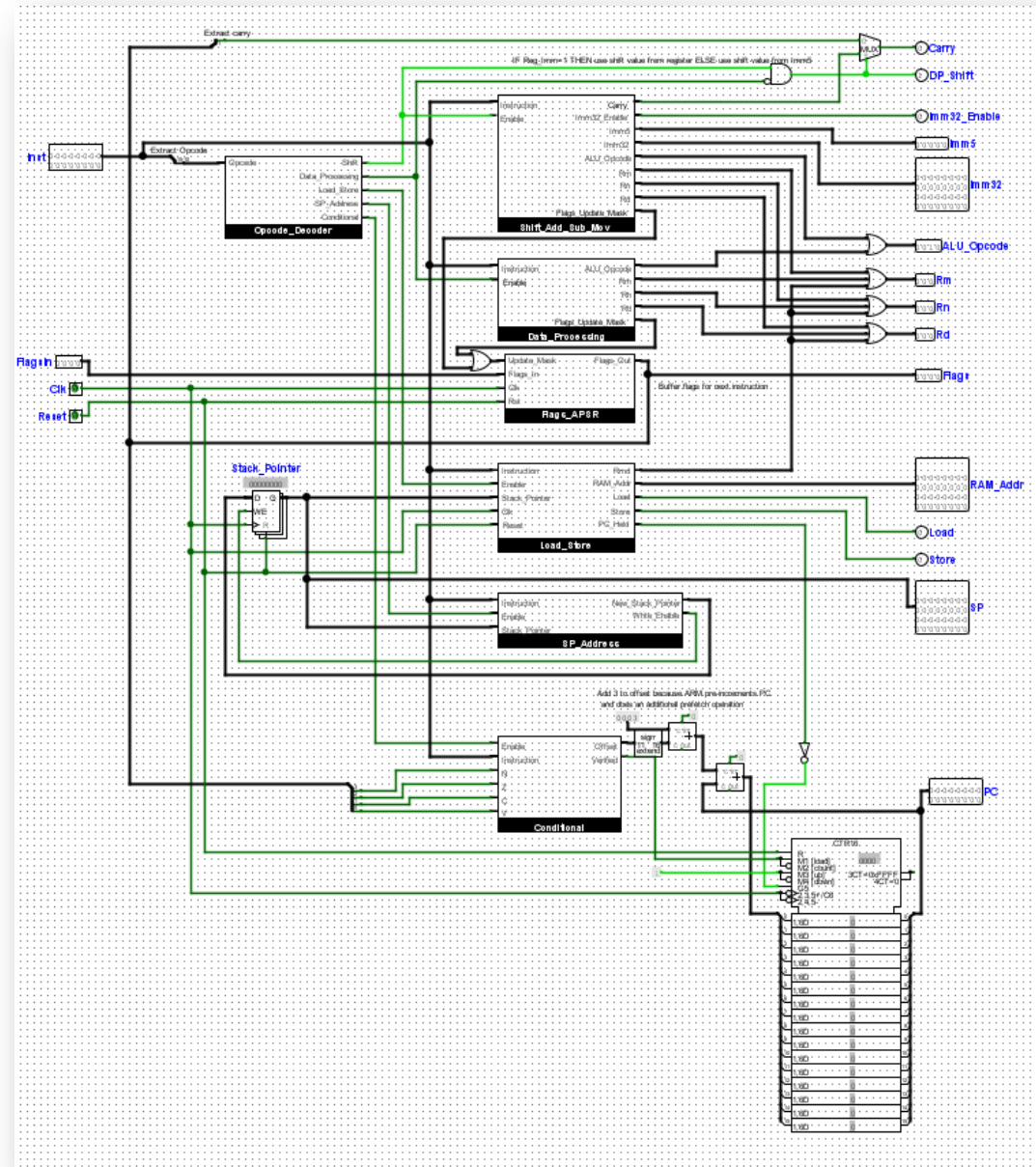
Data processing

Load store

SP address

Flag APSR

Conditional



Assembleur (python)

```
parsed_asm_code = [[l.strip().strip(',') for l in l.replace(
    ',', ', ').split()] for l in asm.splitlines() if len(l.split()) > 0]
machine_code = ""
```

... UNE FONCTION PAR INSTRUCTION ... PUIS ...

```
i = 0
for instruction in parsed_asm_code:

    keyword = instruction[0].lower()

    if keyword.endswith(':') or keyword.startswith('@', '.'):
        continue

    try:
        fun = locals()[f"_Parser_{keyword}"]
        binary = fun(instruction[1:], i)
    except:
        continue # ignoring unhandled instructions according to the project
    machine_code += Parser._convert(binary, 2,
                                     16).zfill(DEFAULT_HEXADEIMAL_SIZE) + " "

    i += 1

return machine_code[:-1]
```

```
@staticmethod
def __convert(number: str, base_src: int, base_dest: int) -> str:
    """Converts a number from one base to another (limit=16).

    Args:
        number (str): The number to convert.
        base_src (int): The base of the number to convert.
        base_dest (int): The base to convert the number to.

    Returns:
        str: The converted number.
    """
    base_map = "0123456789abcdef"
    base_10_number = int(number, base_src)
    if base_10_number == 0:
        return "0"
    result = ""
    while base_10_number > 0:
        result = base_map[base_10_number % base_dest] + result
        base_10_number //= base_dest
    return result
```

Assembleur (python)

Description			Instruction		Operands		Bits											Flags				
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	C	V	N	Z
Logical Shift Left			LSL	S	<Rd>	<Rn>	#imm5	0	0	0	0	0	imm5	Rn	Rd							
Logical Shift Right			LSR	S	<Rd>	<Rn>	#imm5	0	0	0	0	1	imm5	Rn	Rd							
Arithmetic Shift Right			ASR	S	<Rd>	<Rn>	#imm5	0	0	0	0	1	0	imm5	Rn	Rd						
Shift, add, sub, mov																						
Add register			ADD	S	<Rd>	<Rn>	<Rm>	0	0	0	0	1	1	0	0	Rn	Rd					
Subtract register			SUB	S	<Rd>	<Rn>	<Rm>	0	0	0	0	1	1	0	1	0	Rn	Rd				
Add 3-bit immediate			ADD	S	<Rd>	<Rn>	#imm3	0	0	0	0	1	1	1	0	imm3	Rn	Rd				
Subtract 3-bit immediate			SUB	S	<Rd>	<Rn>	#imm3	0	0	0	0	1	1	1	1	0	imm3	Rn	Rd			
Move			MOV	S	<Rd>	#imm8		0	0	0	1	0	0	0		Rd	imm8					
Data processing																						
Bitwise AND			AND	S	<Rd>	<Rn>		0	1	0	0	0	0	0	0	0	Rn	Rdn				
Exclusive OR			EOR	S	<Rd>	<Rn>		0	1	0	0	0	0	0	0	1	Rn	Rdn				
Logical Shift Left			LSL	S	<Rd>	<Rn>		0	1	0	0	0	0	0	0	1	0	Rn	Rdn			
Logical Shift Right			LSR	S	<Rd>	<Rn>		0	1	0	0	0	0	0	0	1	1	0	Rn	Rdn		
Arithmetic Shift Right			ASR	S	<Rd>	<Rn>		0	1	0	0	0	0	0	0	1	0	0	Rn	Rdn		
Add with Carry			ADC	S	<Rd>	<Rn>		0	1	0	0	0	0	0	0	1	0	1	Rn	Rdn		
Subtract with Carry			SBC	S	<Rd>	<Rn>		0	1	0	0	0	0	0	0	1	1	0	Rn	Rdn		
Rotate Right			ROR	S	<Rd>	<Rn>		0	1	0	0	0	0	0	0	1	1	1	Rn	Rdn		
Set flags on bitwise AND			TST		<Rn>	<Rn>		0	1	0	0	0	0	0	0	0	0	0	Rn			
Reverse Substruc from 0			RSB	S	<Rd>	<Rn>	#0	0	1	0	0	0	0	0	0	1	0	0	Rn -> Rm			
Compare Registers			CMP		<Rn>	<Rn>		0	1	0	0	0	0	0	0	1	0	0	Rn			
Compare Negative			CMN		<Rn>	<Rn>		0	1	0	0	0	0	0	0	1	0	1	Rn			
Logical OR			ORR	S	<Rd>	<Rn>		0	1	0	0	0	0	0	0	1	1	0	Rn	Rdn		
Multiply Two Registers			MUL	S	<Rd>	<Rn>	<Rdn>	0	1	0	0	0	0	0	0	1	1	0	Rn -> Rm	Rdm		
Bit Clear			BIC	S	<Rd>	<Rn>		0	1	0	0	0	0	0	0	1	1	1	0	Rn	Rdn	
Bitwise NOT			MVN	S	<Rd>	<Rn>		0	1	0	0	0	0	0	0	0	1	1	1	0	Rn	Rdn
Load / Store																						
Store Register			STR		<Rn>	[SP, #imm8]		1	0	0	0	1	0		Rn		imm8					
Load Register			LDR		<Rn>	[SP, #imm8]		1	0	0	0	1	0		Rn		imm8					
Miscellaneous 10-bit instructions																						
Add Immediate to SP			ADD	[SP]	SP	#imm7		0	0	1	1	0	0	0	0	0	0		imm7			
Subtract Immediate from SP			SUB	[SP]	SP	#imm7		0	0	1	1	0	0	0	0	0	1		imm7			
Conditional Branch			B		<label>			1	1	0	0	0	0	0	0	0	1		imm8			
égalité			BEQ		<label>			1	1	0	0	0	0	0	0	0	0		imm8			
différence			BNE		<label>			1	1	0	0	0	0	0	0	0	1		imm8			
retenue			BCS		<label>			1	1	0	0	1	0	0	0	0	0		imm8			
pas de retenue			BCC		<label>			1	1	0	0	1	0	0	0	0	1		imm8			
négatif			BMI		<label>			1	1	0	0	0	0	0	0	0	0		imm8			
positif ou nul			BPL		<label>			1	1	0	0	0	0	0	0	0	1		imm8			
dépassement de capacité			BVS		<label>			1	1	0	0	0	0	0	0	0	1		imm8			
pas de dépassement de capacité			BVC		<label>			1	1	0	0	0	0	0	0	0	1		imm8			
supérieur (non signé)			BHI		<label>			1	1	0	0	0	0	0	0	0	0		imm8			
inférieur ou égal (non signé)			BLS		<label>			1	1	0	0	0	0	0	0	0	1		imm8			
supérieur ou égal (signé)			BGE		<label>			1	1	0	0	0	0	0	0	0	0		imm8			
inférieur (signé)			BLT		<label>			1	1	0	0	0	0	0	0	0	1		imm8			
supérieur (signé)			BGT		<label>			1	1	0	0	0	0	0	0	0	1		imm8			
inférieur ou égal (signé)			BLE		<label>			1	1	0	0	0	0	0	0	0	1		imm8			
pourquoi pas			BAL		<label>			1	1	0	0	0	0	0	0	0	0		imm8			
continue execution			B		<label>			1	1	1	0	0	0	0	0	0	0		imm11			

```
def __lsls(args: list, instruction_number: int = None) -> str:
    """Parse the lsls instruction. (Logical Shift Left) LSL S Rd, Rm, #imm5 or LSL S Rdn, Rm"""
    if args[-1].startswith("#"):
        return "00000" + Parser.__get_immediate(args[-1], 5) + Parser.__get_register(args[1]) + Parser.__get_register(args[0])
    return "0100000010" + Parser.__get_register(args[1]) + Parser.__get_register(args[0])
```

```
@staticmethod
def __get_immediate(immediate: str, size: int, division_by_4: bool = False) -> str:
    """Get the binary representation of an immediate.

    Args:
        immediate (str): The immediate to get the binary representation of.

    Returns:
        str: The binary representation of the immediate.
    """
    if immediate[1:] == 'sp':
        return '0' * size
    return bin(int(immediate[1:]))[2:].zfill(size) if not division_by_4 else bin(int(immediate[1:]) // 4)[2:].zfill(size)
```

```
@staticmethod
def __get_register(register: str) -> str:
    """Get the binary representation of a register.

    Args:
        register (str): The register to get the binary representation of.

    Returns:
        str: The binary representation of the register.
    """
    return bin(int(register[1:]))[2:].zfill(DEFAULT_REGISTER_SIZE)
```


Assembleur (python)

```
def __get_label_offset(instruction_number: int, label: str, size: int) -> str:
    """Get the the binary representation of a label offset.

    Args:
        label (str): The label to get the offset as binary representation of.

    Returns:
        str: The binary representation of the label offset.
    """
    i = 0
    for instruction in parsed_asm_code:
        if instruction[0] == label + ":":
            return bin(2**size + i - instruction_number - 3)[-size:]

        elif instruction[0].endswith(':'):
            continue

        i += 1
    raise Exception(f"Error: Label {label} not defined.")
```

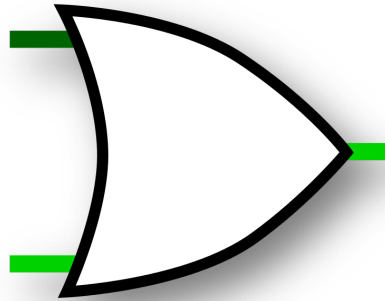
```
def __ble(args: list, instruction_number: int = None) -> str:
    """Parse the ble instruction. (Branch if less or equal) BLE <label> (with label written on 8 bits as imm8)"""
    return "11011101" + __get_label_offset(instruction_number, args[0], 8)

def __bal(args: list, instruction_number: int = None) -> str:
    """Parse the bal instruction. (Branch and link) BAL <label> (with label written on 8 bits as imm8)"""
    return "11011110" + __get_label_offset(instruction_number, args[0], 8)

def __b(args: list, instruction_number: int = None) -> str:
    """Parse the b instruction. (Branch) B <label> (with label written on 11 bits as imm11)"""
    return "111100" + __get_label_offset(instruction_number, args[0], 11)
```

<- Fonction interne à la fonction principale de parsing
(ayant accès à la variable `parsed_asm_code`)

Démonstration du simulateur Logisim



Passage des tests (Parser)

```
C:\Users\marcp\Documents\Programming\Python\Projet PARM\asm>asm2mc.py
Testing or compiling? (t/c)
t
Testing tests\conditional\branch.s...
Test passed for tests\conditional\branch.s
Testing tests\data_processing\1-4_instructions.s...
Test passed for tests\data_processing\1-4_instructions.s
Testing tests\data_processing\11-12_instructions.s...
Test passed for tests\data_processing\11-12_instructions.s
Testing tests\data_processing\13-16_instructions.s...
Test passed for tests\data_processing\13-16_instructions.s
Testing tests\data_processing\5-10_instructions.s...
Test passed for tests\data_processing\5-10_instructions.s
Testing tests\load_store\load_store.s...
Test passed for tests\load_store\load_store.s
Testing tests\miscellaneous\sp.s...
Test passed for tests\miscellaneous\sp.s
Testing tests\more\calckeyb.s...
Test passed for tests\more\calckeyb.s
Testing tests\more\calculator.s...
Test passed for tests\more\calculator.s
Testing tests\more\simple_add.s...
Test passed for tests\more\simple_add.s
Testing tests\more\testfp.s...
Test passed for tests\more\testfp.s
Testing tests\more\tty.s...
Test passed for tests\more\tty.s
Testing tests\shift_add_sub_mov\1-4_instructions.s...
Test passed for tests\shift_add_sub_mov\1-4_instructions.s
Testing tests\shift_add_sub_mov\5-8_instructions.s...
Test passed for tests\shift_add_sub_mov\5-8_instructions.s
```

```
298 def main() → None:
299     test_folder = TESTS_PATH if TESTS_PATH.endswith('/') else TESTS_PATH + '/' # Ensure we have a well-formed path
300     asm_files = glob(f"{test_folder}/**/*.s", recursive=True)
301
302     answer = input("Testing or compiling? (t/c)\n")
303     test = True if answer.lower() == 't' else False
304
305     for asm_file in asm_files:
306         if test:
307             print(f"Testing {asm_file}...")
308
309             # Getting the machine code from the parser
310             with open(asm_file, "r") as f:
311                 asm_code = f.read()
312                 machine_code = Parser.parse_asm_into_machine_code(asm_code)
313
314             bin_file = ''.join(asm_file.split('.')[:-1]) + ".bin"
315
316             if test:
317                 # Checking if the content of the machine code is the same as the content of the .bin file in the same directory
318                 with open(bin_file, "r") as f:
319                     expected_machine_code = ''.join(f.readlines()[1:])
320                     if machine_code.strip() != expected_machine_code.strip():
321                         print(f"Error in: {asm_file}:")
322                         print(f"1: {expected_machine_code}", end='')
323                         print(f"2: {machine_code}")
324                         exit(1)
325                     else:
326                         print(f"Test passed for {asm_file}")
327             else:
328                 # Writing to .bin file
329                 with open(bin_file, "w") as f:
330                     f.write(BINARY_HEADER + '\n' + machine_code.strip())
331
```

Fin de la présentation

Questions