UNIVERSITÉ CÔTE D'AZUR

POLYTECH NICE-SOPHIA

# Architecture : PARM

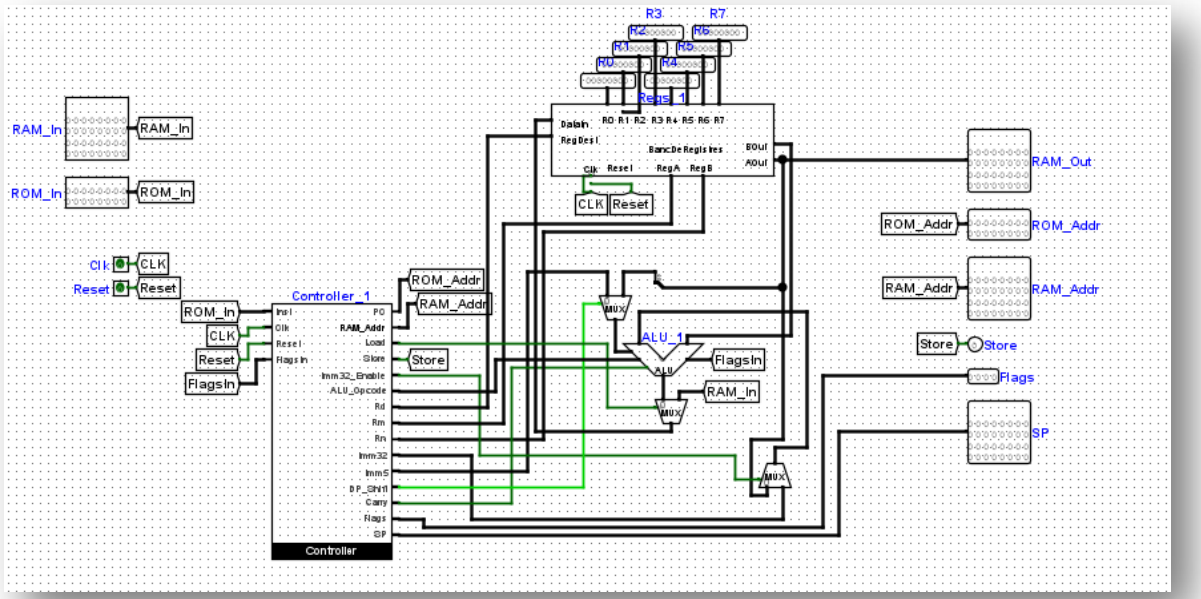
## Groupe Polycrew

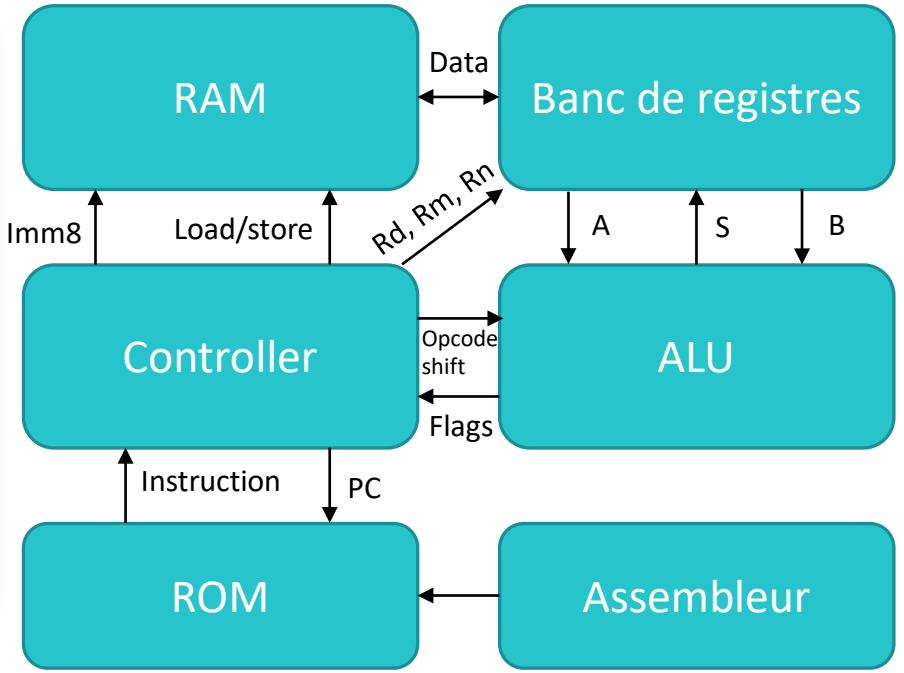
Marc Pinet – Arthur Rodriguez – Marcus Aas Jensen – Loïc Pantano

# Sommaire

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

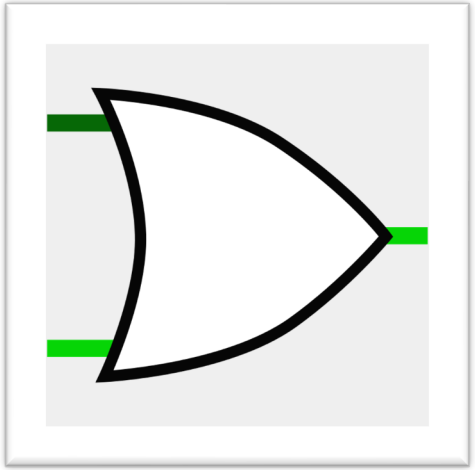
# Structure du projet (CPU)



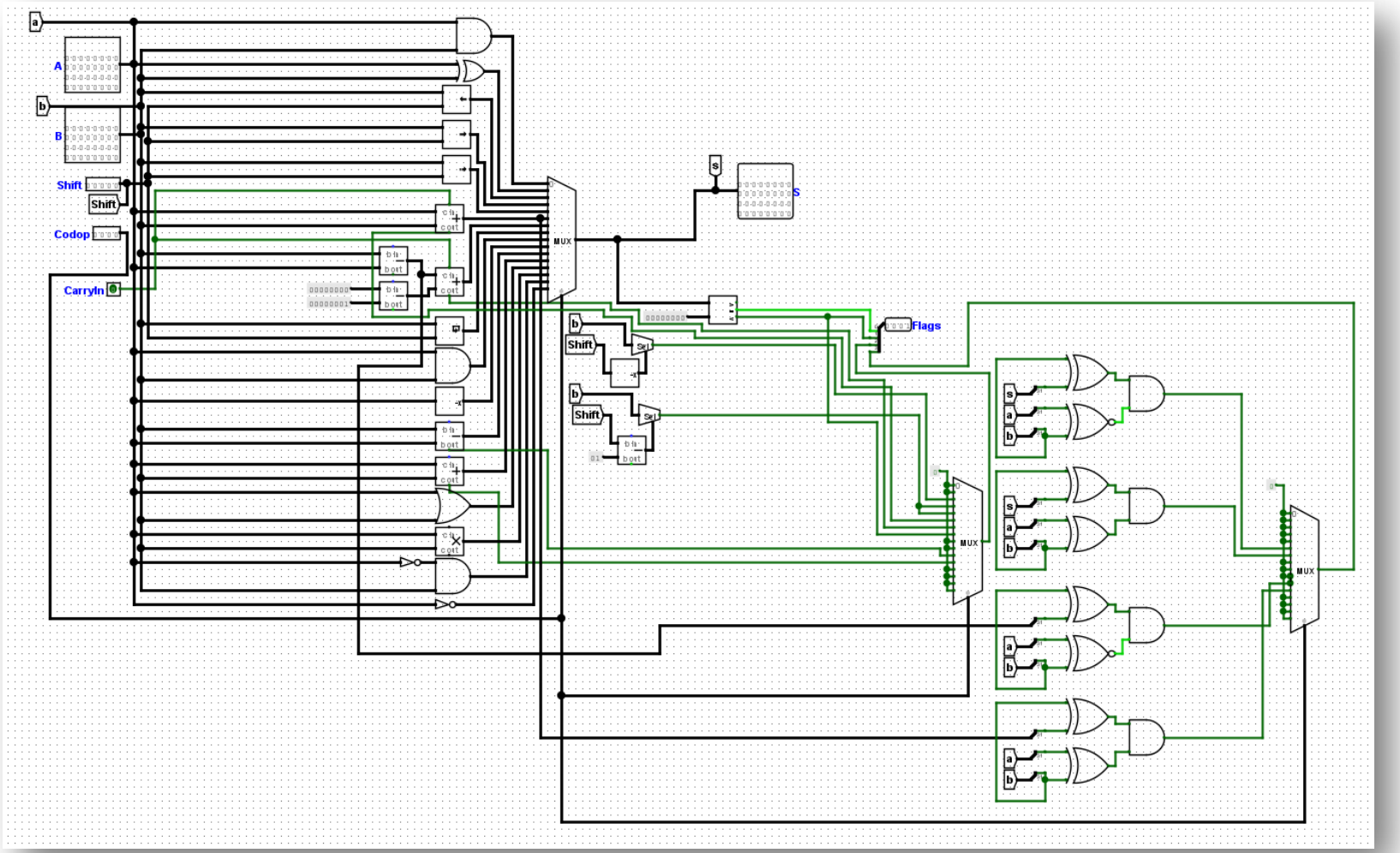



# Composants Logisim

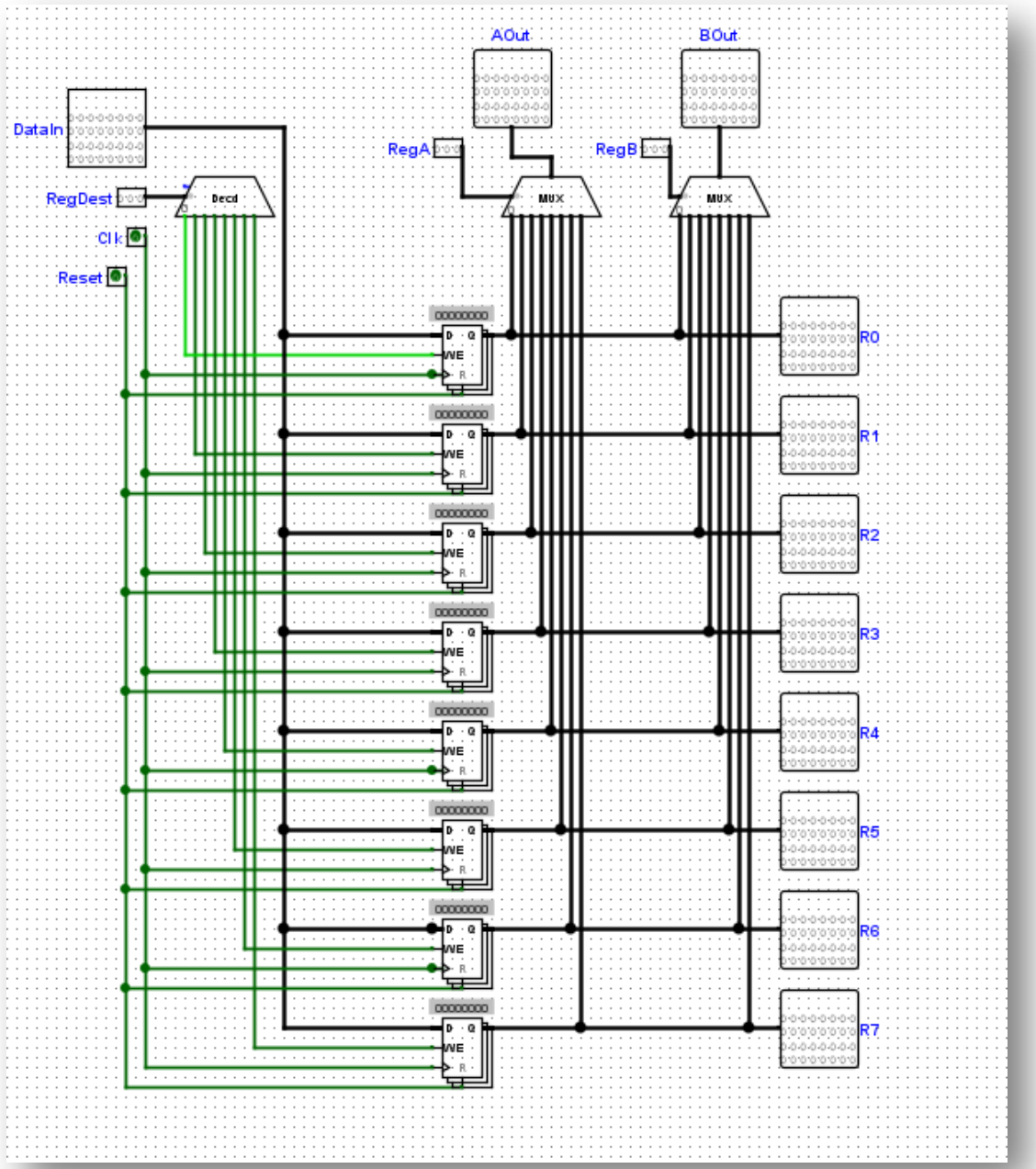
- ALU
- Banc de registres
- Opcode décodeur
- Shift, Add, Sub, Move
- Data processing
- Flag APSR
- Load store
- SP address
- Conditional
- Contrôleur (regroupement)



# ALU

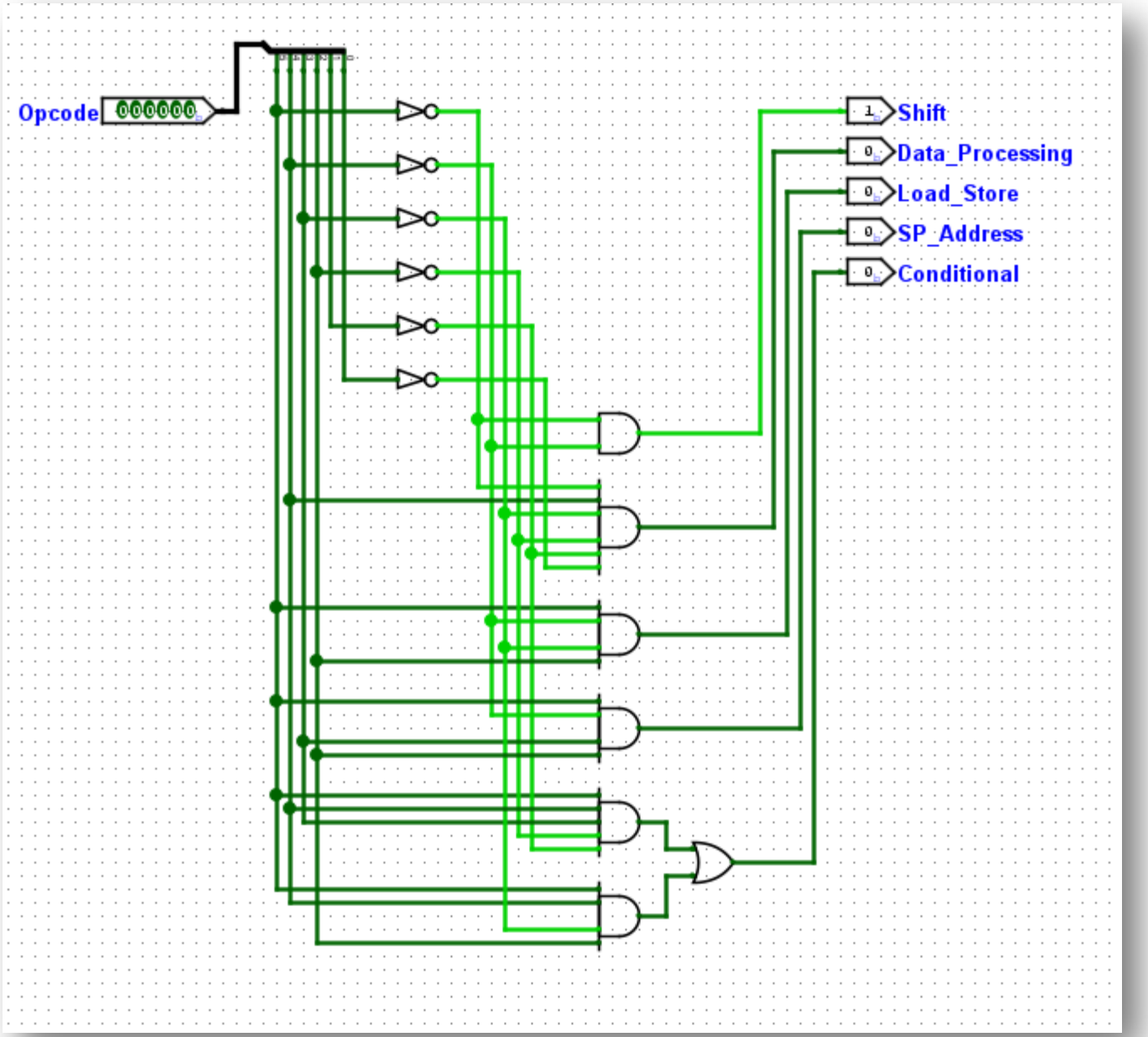

a
A
b
B
Shift
Shift
Codop
CarryIn
MUX
s
S
Flags
b
Shift
b
Shift
MUX
MUX

# Banc de registres

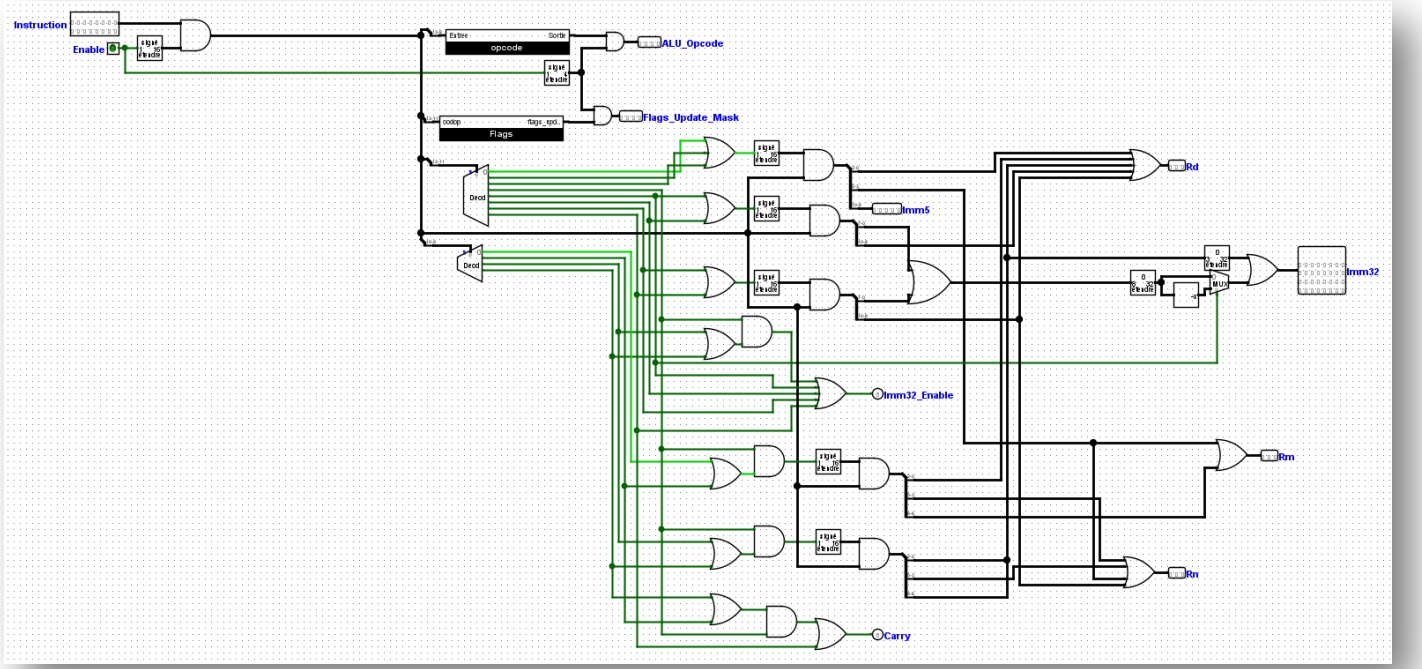


# Opcode decoder

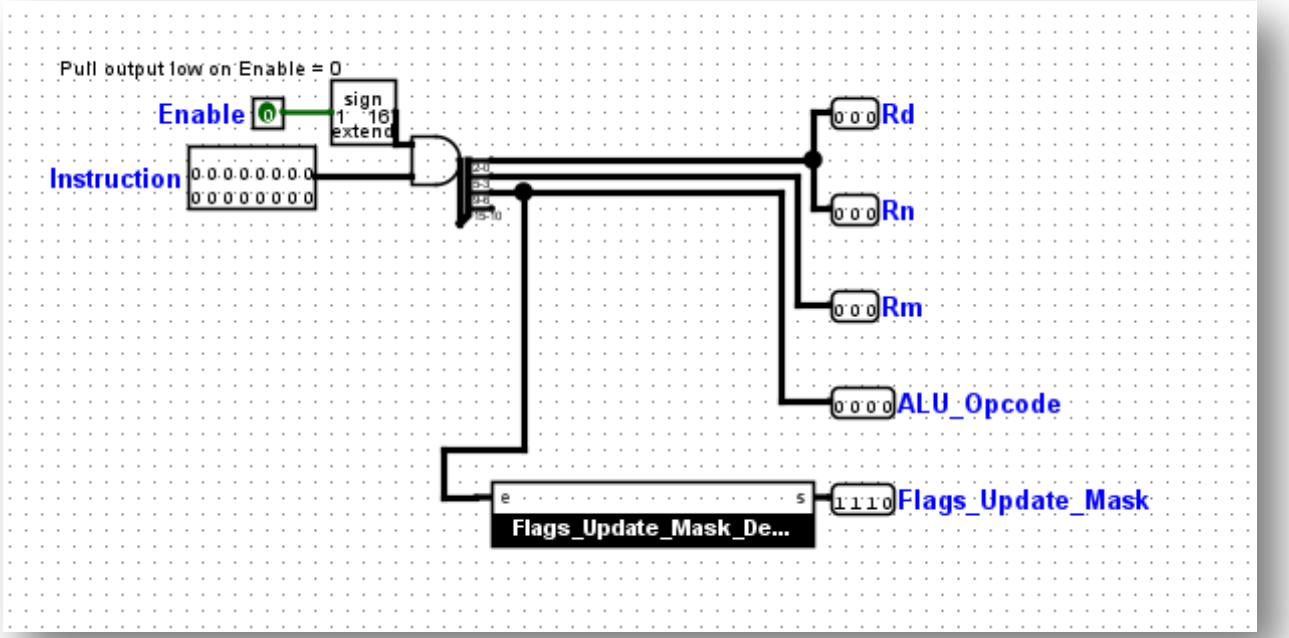
| Opcode[5..0] | Shift | Data_Processing | Load_Store | SP_Address | Conditional |
|---|---|---|---|---|---|
| 000000 | 1 | 0 | 0 | 0 | 0 |
| 000001 | 1 | 0 | 0 | 0 | 0 |
| 000010 | 1 | 0 | 0 | 0 | 0 |
| 000011 | 1 | 0 | 0 | 0 | 0 |
| 000100 | 1 | 0 | 0 | 0 | 0 |
| 000101 | 1 | 0 | 0 | 0 | 0 |
| 000110 | 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 |



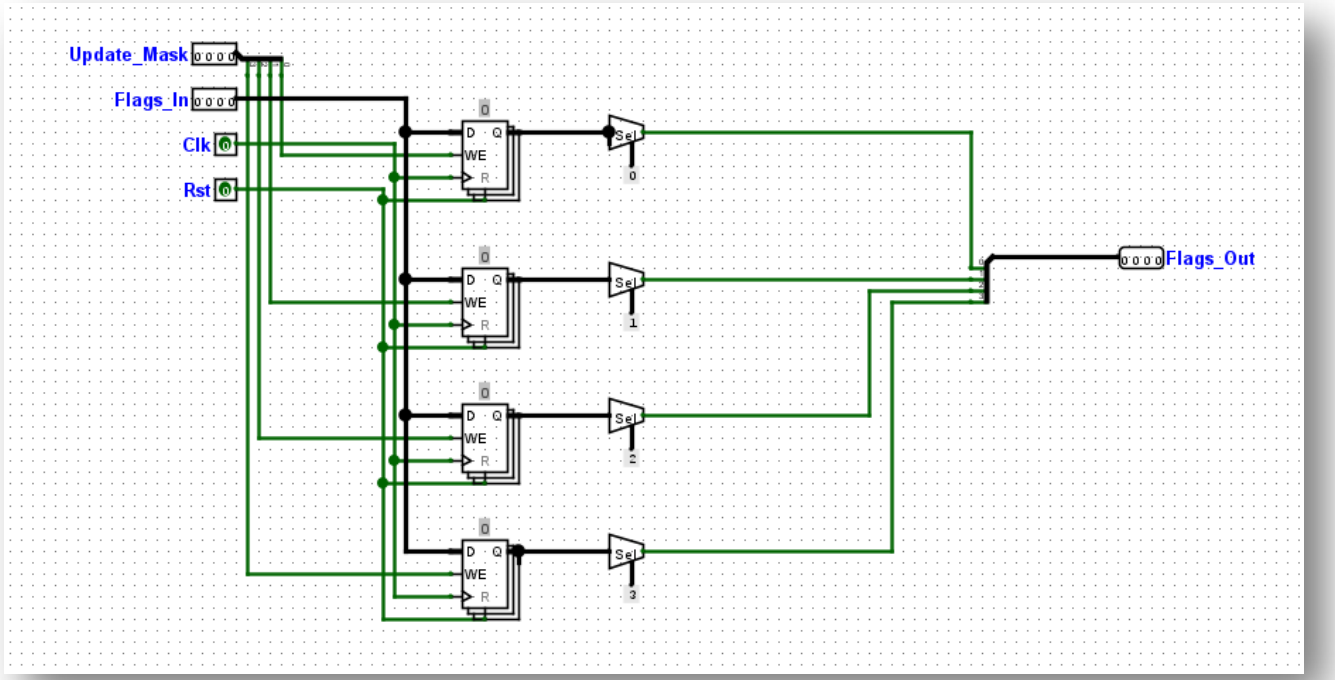
# Shift
# Add
# Sub
# Move



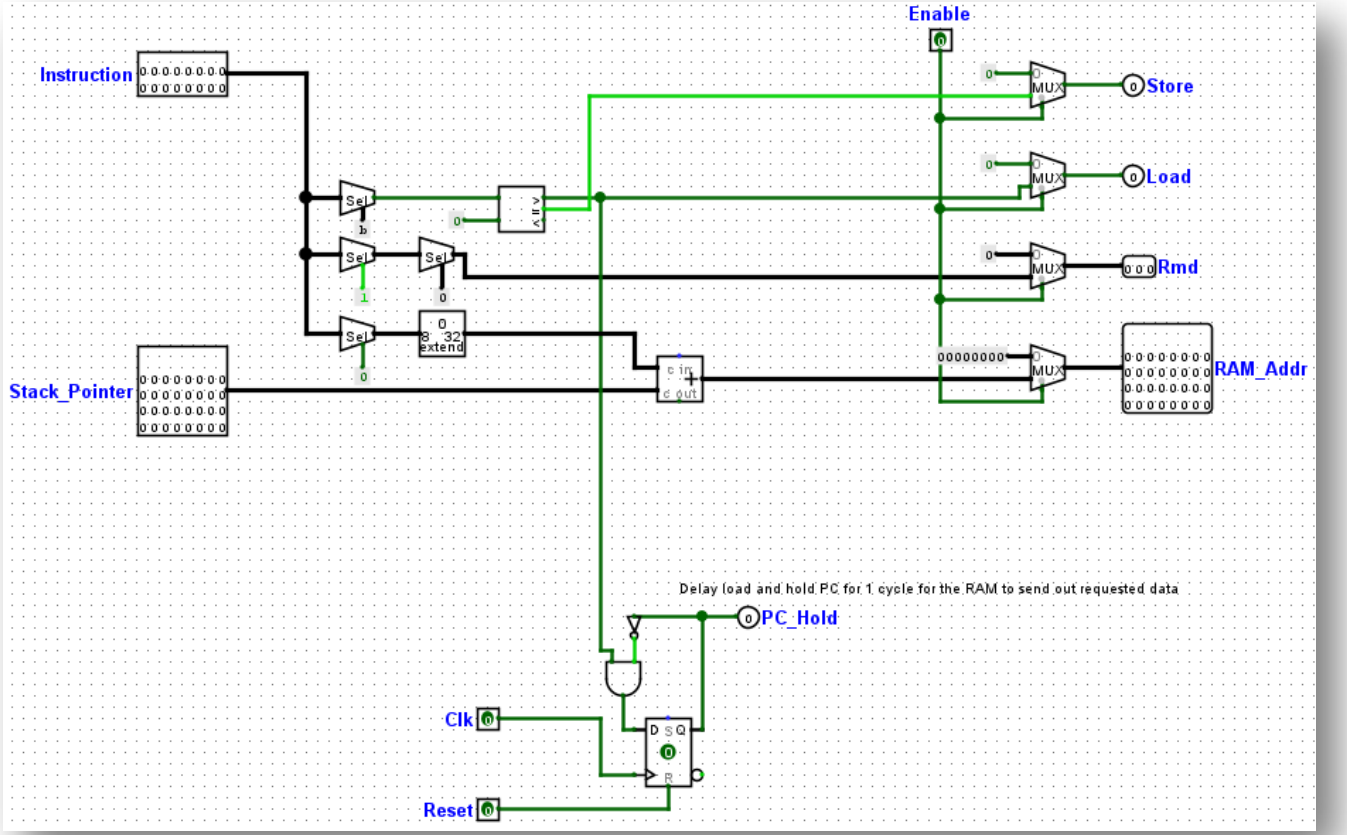
# Data processing



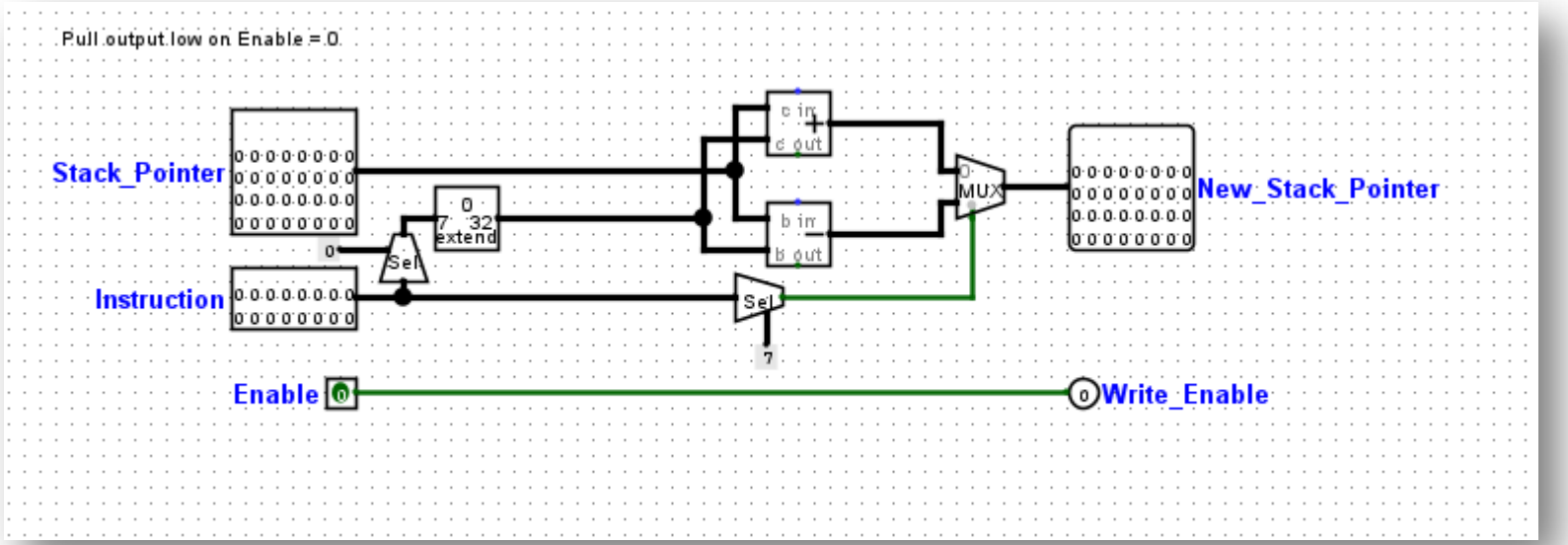
# Flag APSR



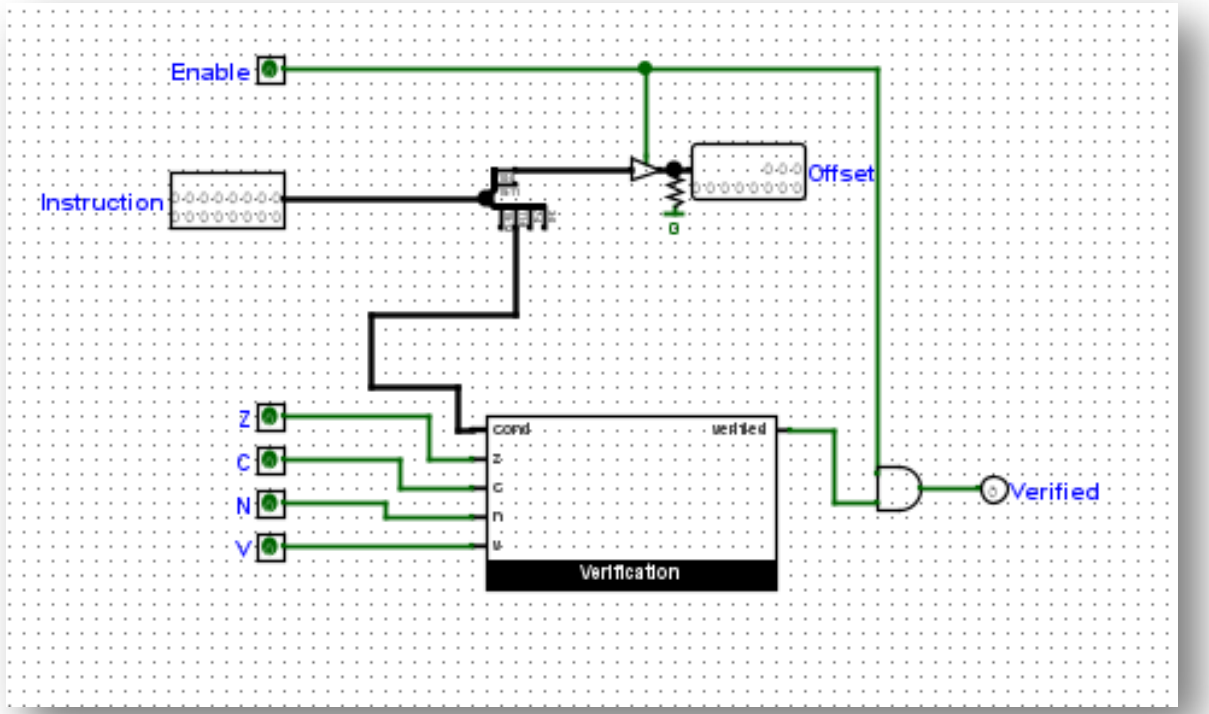
# Load store


Enable
Instruction
Store
Load
Rmd
RAM_Addr
Stack_Pointer
Delay load and hold PC for 1 cycle for the RAM to send out requested data
PC_Hold
Clk
Reset
MUX
Sel
ALU
extend
D S Q
R

# SP Address


Pull output low on Enable = 0
Stack_Pointer
Instruction
Enable
0
7 32
extend
Sel
c in
c out
b in
b out
MUX
New_Stack_Pointer
Write_Enable

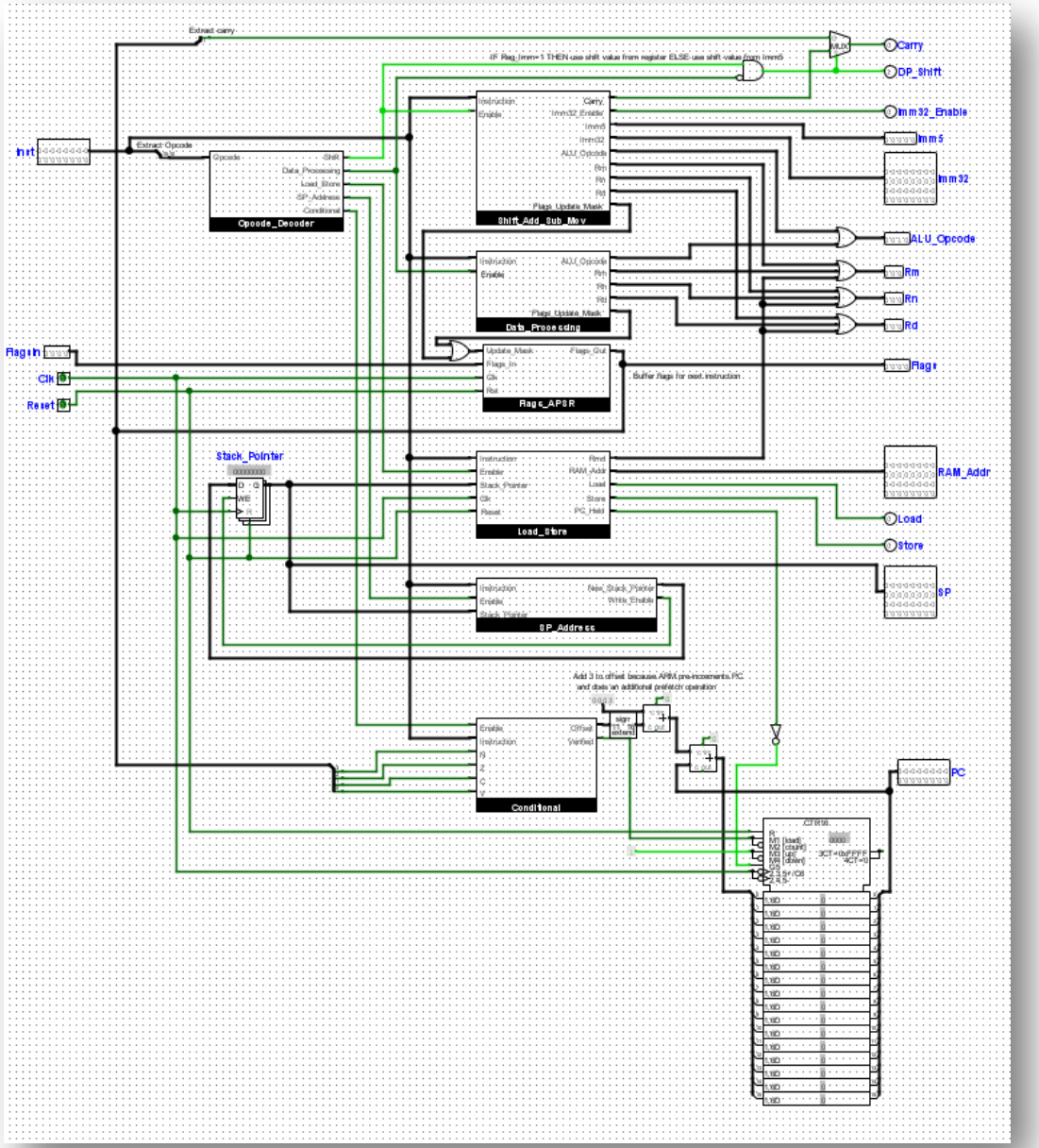
# Conditional



# Controller

Composants regroupés :

- Opcode decoder
- Shift Add Sub Move
- Data processing
- Load store
- SP address
- Flag APSR
- Conditional



# Assembleur (python)

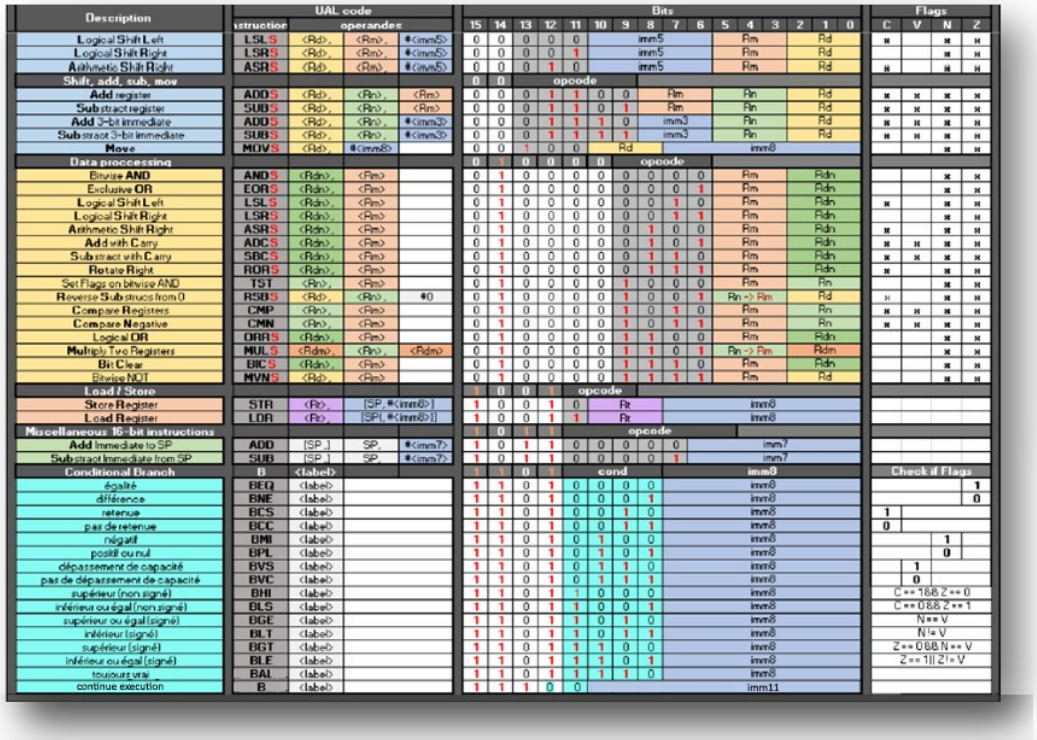
```
parsed_asm_code = [[l.strip().strip(',') for l in l.replace(
    ',', ", ").split()] for l in asm.splitlines() if len(l.split()) > 0]
machine_code = ""
```

… 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()["_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_HEXADECIMAL_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 | 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>, | <Rm>, | #<imm5> | 0 | 0 | 0 | 0 | 0 | imm5 | | | | | Rm | | | Rd | | | x | | x | x |
| Logical Shift Right | LSRS | <Rd>, | <Rm>, | #<imm5> | 0 | 0 | 0 | 0 | 1 | imm5 | | | | | Rm | | | Rd | | | | | x | x |
| Arithmetic Shift Right | ASRS | <Rd>, | <Rm>, | #<imm5> | 0 | 0 | 0 | 1 | 0 | imm5 | | | | | Rm | | | Rd | | | x | | 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 | | | | | | | | x | x |
| **Data processing** | | | | | 0 | 1 | 0 | 0 | 0 | 0 | opcode | | | | | | | | | | | | | |
| Bitwise AND | ANDS | <Rdn>, | <Rm> | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rm | | | Rdn | | | | | x | x |
| Exclusive OR | EORS | <Rdn>, | <Rm> | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Rm | | | Rdn | | | | | x | x |
| Logical Shift Left | LSLS | <Rdn>, | <Rm> | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | Rm | | | Rdn | | | x | | x | x |
| Logical Shift Right | LSRS | <Rdn>, | <Rm> | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Rm | | | Rdn | | | | | x | x |
| Arithmetic Shift Right | ASRS | <Rdn>, | <Rm> | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | Rm | | | Rdn | | | x | | 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 | | | x | x | x | x |
| Rotate Right | RORS | <Rdn>, | <Rm> | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | Rm | | | Rdn | | | x | | x | x |
| Set Flags on bitwise AND | TST | <Rn>, | <Rm> | | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | Rm | | | Rn | | | | | x | x |
| Reverse Substruct from 0 | RSBS | <Rd>, | <Rn>, | #0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Rn -> Rm | | | Rd | | | 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 | | | | | x | x |
| Multiply Two Registers | MULS | <Rdm>, | <Rn>, | <Rdm> | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | Rn -> Rm | | | Rdm | | | | | x | x |
| Bit Clear | BICS | <Rdn>, | <Rm> | | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | Rm | | | Rdn | | | | | x | x |
| Bitwise NOT | MVNS | <Rd>, | <Rm> | | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Rm | | | Rd | | | | | x | x |
| **Load / Store** | | | | | 1 | 0 | 0 | 1 | opcode | | | | | | | | | | | | | | | |
| Store Register | STR | <Rt>, | [SP, #<imm8>] | | 1 | 0 | 0 | 1 | 0 | Rt | | | | | imm8 | | | | | | | | | |
| Load Register | LDR | <Rt>, | [SP{, #<imm8>}] | | 1 | 0 | 0 | 1 | 1 | Rt | | | | | imm8 | | | | | | | | | |
| **Miscellaneous 16-bit instructions** | | | | | 1 | 0 | 1 | 1 | opcode | | | | | | | | | | | | | | | |
| Add Immediate to SP | ADD | [SP,] | SP, | #<imm7> | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | imm7 | | | | | | | | | |
| Substract Immediate from SP | SUB | [SP,] | SP, | #<imm7> | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | imm7 | | | | | | | | | |
| **Conditional Branch** | B | <label> | | | 1 | 1 | 0 | 1 | cond | | | | | | imm8 | | | | | | Check if Flags | | | |
| égalité | BEQ | <label> | | | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | imm8 | | | | | | | | | | | 1 |
| différence | BNE | <label> | | | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | imm8 | | | | | | | | | | | 0 |
| retenue | BCS | <label> | | | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | imm8 | | | | | | | | 1 | | | |
| pas de retenue | BCC | <label> | | | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | imm8 | | | | | | | | 0 | | | |
| négatif | BMI | <label> | | | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | imm8 | | | | | | | | | | 1 | |
| positif ou nul | BPL | <label> | | | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | imm8 | | | | | | | | | | 0 | |
| dépassement de capacité | BVS | <label> | | | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | imm8 | | | | | | | | | 1 | | |
| pas de dépassement de capacité | BVC | <label> | | | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | imm8 | | | | | | | | | 0 | | |
| supérieur (non signé) | BHI | <label> | | | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | imm8 | | | | | | | | C == 1&& Z == 0 | | | |
| inférieur ou égal (non signé) | BLS | <label> | | | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | imm8 | | | | | | | | C == 0&& Z == 1 | | | |
| supérieur ou égal (signé) | BGE | <label> | | | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | imm8 | | | | | | | | N == V | | | |
| inférieur (signé) | BLT | <label> | | | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | imm8 | | | | | | | | N != V | | | |
| supérieur (signé) | BGT | <label> | | | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | imm8 | | | | | | | | Z == 0&& N == V | | | |
| inférieur ou égal (signé) | BLE | <label> | | | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | imm8 | | | | | | | | Z == 1\|\| Z != V | | | |
| toujours vrai | BAL | <label> | | | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | imm8 | | | | | | | | | | | |
| continue execution | B | <label> | | | 1 | 1 | 1 | 0 | 0 | imm11 | | | | | | | | | | | | | | |

```python
@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)
```

```python
@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)
```

```python
def __lsls(args: list, instruction_number: int = None) -> str:
    """Parse the lsls instruction. (Logical Shift Left) LSLS Rd, Rm, #<imm5> or LSLS 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])
```

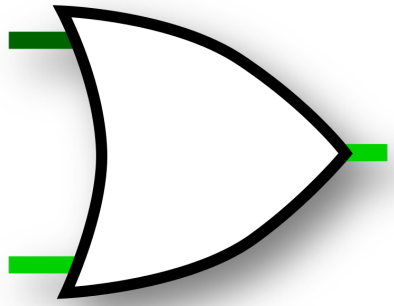
# Assembleur (python)

```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.")
```

```python
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 "11100" + __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
```

```python
def main() -> None:
    test_folder = TESTS_PATH if TESTS_PATH.endswith('/') else TESTS_PATH + '/'  # Ensure we have a well-formed path
    asm_files = glob(f"{test_folder}**/*.s", recursive=True)

    answer = input("Testing or compiling? (t/c)\n")
    test = True if answer.lower() == 't' else False

    for asm_file in asm_files:
        if test:
            print(f"Testing {asm_file} ... ")

        # Getting the machine code from the parser
        with open(asm_file, "r") as f:
            asm_code = f.read()
            machine_code = Parser.parse_asm_into_machine_code(asm_code)

        bin_file = ''.join(asm_file.split('.')[:-1]) + ".bin"

        if test:
            # Checking if the content of the machine code is the same as the content of the .bin file in the same directory
            with open(bin_file, "r") as f:
                expected_machine_code = ''.join(f.readlines()[1:])
                if machine_code.strip() != expected_machine_code.strip():
                    print(f"Error in: {asm_file}:")
                    print(f"1: {expected_machine_code}", end='')
                    print(f"2: {machine_code}")
                    exit(1)
                else:
                    print(f"Test passed for {asm_file}")
        else:
            # Writing to .bin file
            with open(bin_file, "w") as f:
                f.write(BINARY_HEADER + '\n' + machine_code.strip())

```

Fin de la présentation

Questions