**EXPT 1**

from sys import exit  
  
MOT = {  
 "STOP": (0, 1), "ADD": (1, 1), "SUB": (2, 1), "MULT": (3, 1),  
 "MOVER": (4, 1), "MOVEM": (5, 1), "COMP": (6, 1), "BC": (7, 1),  
 "DIV": (8, 1), "READ": (9, 1), "PRINT": (10, 1), "START": (1, 1),  
 "END": (2, 1), "EQU": (3, 1), "ORIGIN": (4, 1), "LTORG": (5, 1),  
 "DS": (1, 1), "DC": (2, 1), "AREG": (1, 1), "BREG": (2, 1),  
 "CREG": (3, 1), "DREG": (4, 1), "A": (1, 1), "B": (2, 1)  
}  
  
l = []  
relativeAddress = []  
machineCode = []  
RA = 0  
  
n = int(input("Enter the no of instruction lines : "))  
for i in range(n):  
 instructions = input(f"Enter instruction line {i+1}: ").upper()  
 l.append(instructions)  
  
for i, x in enumerate(l):  
 tokens = x.split()  
  
 if tokens[0] not in MOT:  
 print("Instruction is not in Op Code Table.")  
 exit(0)  
  
 op\_code, size = MOT[tokens[0]]  
 RA += size  
  
 if len(tokens) == 1:  
 machineCode.append(str(op\_code))  
 elif tokens[1].isalpha():  
 machineCode.append(f"{op\_code}")  
 else:  
 b = tokens[1]  
 b = ' '.join([b[j:j+2] for j in range(0, len(b), 2)])  
 machineCode.append(f"{op\_code} {b}")  
  
 relativeAddress.append(RA - size)  
  
print("Relative Address Instruction OpCode")  
for i in range(n):  
 print(f"{relativeAddress[i]} {l[i]} {machineCode[i]}")

**EXPT 4**

KEYWORDS = ["for", "while", "if", "else", "def", "return", "in", "not", "and", "or", "print", "range", "input"]  
FUNCTIONS = ["len", "int", "str", "float", "bool", "list", "tuple", "dict", "set", "sorted", "max", "min"]  
OPERATORS = ["+", "-", "\*", "/", "%", "//", "\*\*", "+=", "-=", "=", "/=", "==", "!=", "<", ">", "<=", ">=", "not", "in", "and", "or"]  
  
def parse\_code(code):  
 for line in code:  
 parts = line.split(" ")  
 for part in parts:  
 if part in KEYWORDS:  
 print("Keyword: " + part)  
 elif part in FUNCTIONS:  
 print("Function: " + part)  
 elif part in OPERATORS:  
 print("Operator: " + part)  
 elif part.isnumeric():  
 print("Number: " + part)  
 else:  
 print("Identifier: " + part)  
  
code = [  
 "def main():",  
 "a = 5",  
 "b = 7",  
 "if (a > b):",  
 " print('a is greater')",  
 "else:",  
 " print('b is greater')",  
 "print(a + b)",  
 "return 0"  
]  
  
parse\_code(code)

**EXPT 5**

import re  
  
class Token:  
 def \_\_init\_\_(self, token\_type, value):  
 self.token\_type = token\_type  
 self.value = value  
  
class Parser:  
 def \_\_init\_\_(self, text):  
 self.tokens = self.tokenize(text)  
 self.pos = 0  
  
 def parse(self):  
 return self.expr()  
  
 def tokenize(self, text):  
 token\_exprs = [  
 (r'\d+', 'INT'),  
 (r'\+', 'PLUS'),  
 (r'-', 'MINUS'),  
 (r'\\*', 'MULTIPLY'),  
 (r'/', 'DIVIDE'),  
 (r'\(', 'LPAREN'),  
 (r'\)', 'RPAREN'),  
 (r'\s+', None) # skip whitespace  
 ]  
 tokens = []  
 pos = 0  
 while pos < len(text):  
 match = None  
 for token\_expr in token\_exprs:  
 pattern, token\_type = token\_expr  
 regex = re.compile(pattern)  
 match = regex.match(text, pos)  
 if match:  
 value = match.group(0)  
 if token\_type:  
 token = Token(token\_type, value)  
 tokens.append(token)  
 break  
 if not match:  
 raise ValueError(f'Invalid input at position {pos}')  
 else:  
 pos = match.end(0)  
 return tokens  
  
 def consume(self, token\_type):  
 if self.pos < len(self.tokens) and self.tokens[self.pos].token\_type == token\_type:  
 self.pos += 1  
 else:  
 raise ValueError(f'Expected token type {token\_type} at position {self.pos}')  
  
 def factor(self):  
 token = self.tokens[self.pos]  
 if token.token\_type == 'INT':  
 self.consume('INT')  
 return int(token.value)  
 elif token.token\_type == 'LPAREN':  
 self.consume('LPAREN')  
 value = self.expr()  
 self.consume('RPAREN')  
 return value  
  
 def term(self):  
 value = self.factor()  
 while self.pos < len(self.tokens):  
 token = self.tokens[self.pos]  
 if token.token\_type == 'MULTIPLY':  
 self.consume('MULTIPLY')  
 value \*= self.factor()  
 elif token.token\_type == 'DIVIDE':  
 self.consume('DIVIDE')  
 value /= self.factor()  
 else:  
 break  
 return value  
  
 def expr(self):  
 value = self.term()  
 while self.pos < len(self.tokens):  
 token = self.tokens[self.pos]  
 if token.token\_type == 'PLUS':  
 self.consume('PLUS')  
 value += self.term()  
 elif token.token\_type == 'MINUS':  
 self.consume('MINUS')  
 value -= self.term()  
 else:  
 break  
 return value  
  
text = '2 \* (3 + 4) - 5 / 2'  
parser = Parser(text)  
result = parser.parse()  
print(result) # Output: 12.5

**EXPT 6**

import ast  
  
  
class IntermediateCodeGenerator(ast.NodeVisitor):  
 def \_\_init\_\_(self):  
 self.instructions = []  
 self.temp\_count = 0  
  
 def new\_temp(self):  
 temp = f"t{self.temp\_count}"  
 self.temp\_count += 1  
 return temp  
  
 def visit\_Assign(self, node):  
 target = node.targets[0].id  
 value = self.visit(node.value)  
 self.instructions.append(('ASSIGN', target, value))  
  
 def visit\_BinOp(self, node):  
 left = self.visit(node.left)  
 right = self.visit(node.right)  
 op = type(node.op).\_\_name\_\_  
 temp = self.new\_temp()  
 self.instructions.append((op, temp, left, right))  
 return temp  
  
 def visit\_Num(self, node):  
 return node.n  
  
 def visit\_Name(self, node):  
 return node.id  
  
 def visit\_Print(self, node):  
 value = self.visit(node.values[0])  
 self.instructions.append(('PRINT', value))  
  
  
def generate\_intermediate\_code(source\_code):  
 ast\_tree = ast.parse(source\_code)  
 icg = IntermediateCodeGenerator()  
 icg.visit(ast\_tree)  
 return icg.instructions  
  
  
# Example usage  
source\_code = """   
a = 10  
b = 20  
c = a + b  
print(c)  
"""  
instructions = generate\_intermediate\_code(source\_code)  
for instruction in instructions:  
 print(instruction)

**EXPT 7**

class CodeGenerator:  
 def \_\_init\_\_(self, intermediate\_code):  
 self.intermediate\_code = intermediate\_code  
 self.generated\_code = []  
  
 def generate\_code(self):  
 self.generated\_code.append('\_start:')  
 for instruction in self.intermediate\_code:  
 opcode = instruction['opcode']  
 operands = instruction['operands']  
  
 if opcode == 'ADD':  
 self.add(operands)  
 elif opcode == 'SUB':  
 self.sub(operands)  
 elif opcode == 'MULT':  
 self.mult(operands)  
 elif opcode == 'DIV':  
 self.div(operands)  
 else:  
 raise ValueError(f"Invalid opcode '{opcode}'")  
  
 self.generated\_code.append('MOVER R0, %ebx')  
 self.generated\_code.append('MOVER R1, %REGA')  
 self.generated\_code.append('int $0x80')  
  
 def add(self, operands):  
 op1, op2, result = operands  
 self.generated\_code.append(f"MOVER {op1}, %REGA")  
 self.generated\_code.append(f"ADD {op2}, %REGA")  
 self.generated\_code.append(f"MOVER %REGA, {result}")  
  
 def sub(self, operands):  
 op1, op2, result = operands  
 self.generated\_code.append(f"MOVER {op1}, %REGA")  
 self.generated\_code.append(f"SUB {op2}, %REGA")  
 self.generated\_code.append(f"MOVER %REGA, {result}")  
  
 def mult(self, operands):  
 op1, op2, result = operands  
 self.generated\_code.append(f"MOVER {op1}, %REGA")  
 self.generated\_code.append(f"MULT {op2}, %REGA")  
 self.generated\_code.append(f"MOVER %REGA, {result}")  
  
 def div(self, operands):  
 op1, op2, result = operands  
 self.generated\_code.append(f"MOVER {op1}, %eax")  
 self.generated\_code.append(f"cdq")  
 self.generated\_code.append(f"DIV {op2}")  
 self.generated\_code.append(f"MOVER %eax, {result}")  
  
  
intermediate\_code = [  
 {'opcode': 'ADD', 'operands': [2, 3, 'result']},  
 {'opcode': 'SUB', 'operands': [10, 5, 'result']},  
 {'opcode': 'MULT', 'operands': [4, 6, 'result']},  
 {'opcode': 'DIV', 'operands': [12, 3, 'result']}  
]  
# Generate x86 assembly code  
code\_generator = CodeGenerator(intermediate\_code)  
code\_generator.generate\_code()  
assembly\_code = '\n'.join(code\_generator.generated\_code)  
# Print generated x86 assembly code  
print(assembly\_code)

**EXPT 2**