Universidade Federal de Lavras
Arquitetura de Computadores I

Uma implementação de um simulador funcional para o processador μRisc

Rafael Mancini Santos

Resumo:

A implementação do simulador funcional de um processador µRisc foi feita por meio da linguagem “python”. Este simulador possui as seguintes características:(Determinadas pelo documento de instrução do trabalho).

- 16 bits;
- 8 registradores de uso geral de 16 bits de largura;
- 42 instruções;
- instruções de 3 operandos;
- endereçamento *big endian*;
- memória endereçada a nível de palavra; cada endereço de memória deve se referir a dois bytes. No total, a memória é de 128KB.

Também foi simulado suas 4 unidades funcionais - IF, ID, EX/MEM e WB – com algumas diferenças por questões de implementação.

Decisões de implementação:

Como dito anteriormente, *python* foi a linguagem usada na implementação deste simulador por questões de familiaridade com a linguagem, e rapidez na prototipagem e desenvolvimento do código em si.

As unidades do simulador foram divididas em classes, com exceção dos registros, que foram implementados, simplesmente, como um módulo separado do simulador, provendo funções de escrita e leitura.

A unidade ALU simulada na implementação, diferente de um processador real como o MIPS - em que se trata de uma unidade de circuito lógico combinatória - comporta-se como uma unidade de circuito lógico sequencial, devida à implementação de todo o simulador ter usado orientação a objetos. Foi, assim, mais fácil tratar suas flags como estados da unidade.

Também em consequência do uso de orientação a objetos, foi criado o tipo *Instruction*, que representa, o que seriam os campos da instrução, por atributos distintos, de forma que todas as instruções possuam todos os campos. Porém, aquelas que não usam alguns campos os tem, como padrão, definidos como *None*, o que indica que estão vazios e não devem ser usados por aquele tipo de instrução.

Diferente do proposto pelo documento de instruções sobre o trabalho, foi usado uma lista, em alternativa à memória, para armazenar as instruções do programa, dado que elas são representadas por objetos do tipo *Instruction*.

Tutorial:

A princípio, é importante notar que, para a execução do simulador, é necessário ter o *python* instalado. Acredito que você já o tenha, mas, se não tiver, é possível instalá-lo pelo site oficial: https://www.python.org/downloads/

A execução do programa é dada pelo uso de um parâmetro obrigatório e dois opcionais; o parâmetro obrigatório é o nome do arquivo com o código fonte µRisc a ser executado pelo simulador; um opcional é o de *dump* de memória, dado pela posição inicial na memória, seguido de quantas palavras devem ser impressas, na saída padrão, a partir desta posição inicial; o outro parâmetro opcional é o *screen*, que imprime o estado do processador, e aguarda um *enter* do teclado para prosseguir, após cada instrução executada.

É possível executar o simulador na linha de comando da seguinte forma:

[user@*localhost]$ python uRisc.py [nomedoarquivo] -d [posinicial npalavras] -s*

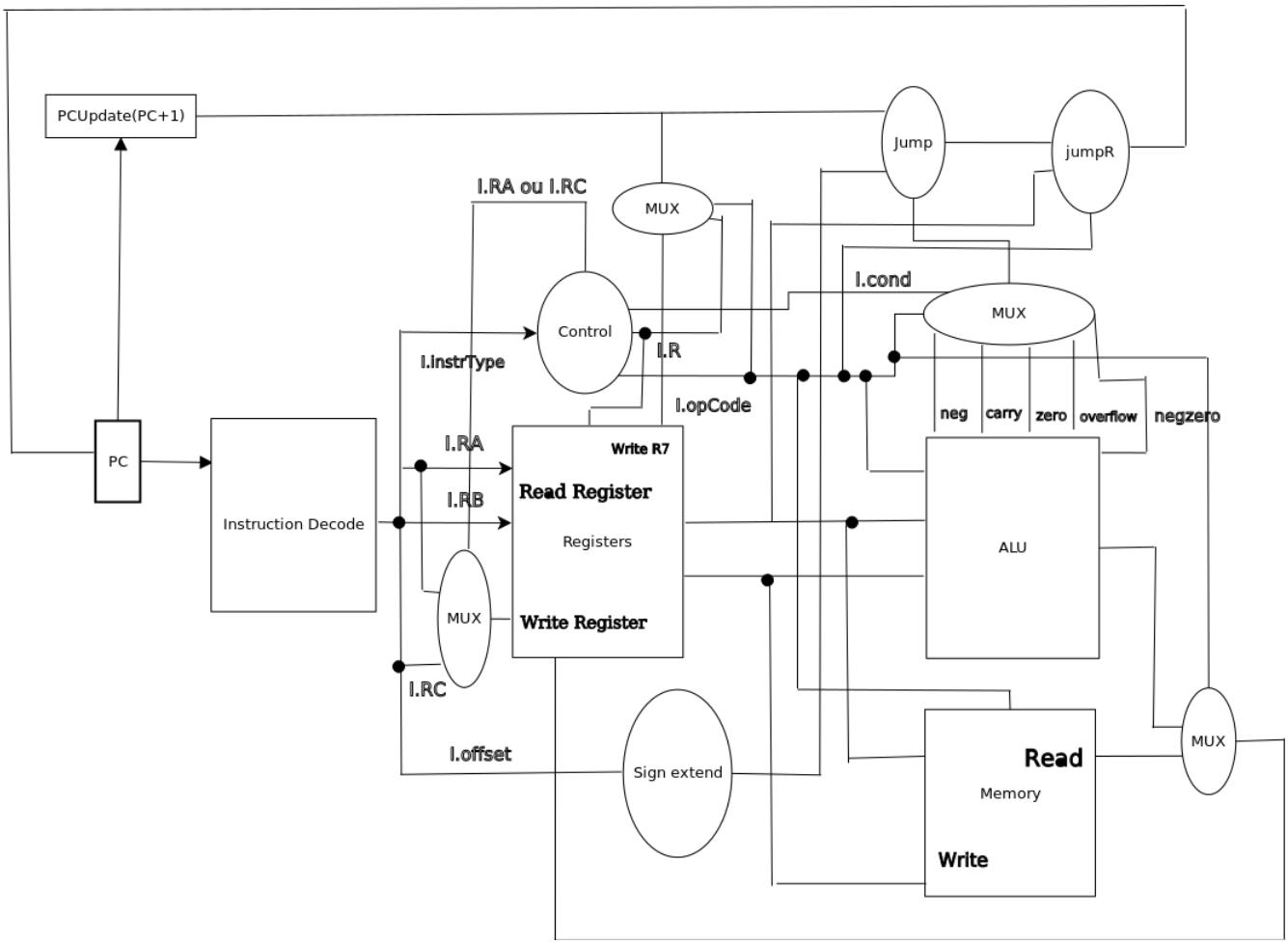
Exemplo:

[user@*localhost]$ python uRisc.py jupitermaca.asm -d 03e8 20 -s*

Ou, sem o parâmetro opcional:

[user@*localhost]$ python uRisc.py ximira.asm -d 01ff 42*

Desenho do datapath:



Descrição dos módulos e estruturas utilizadas:

Registers.py:

É o módulo responsável por interagir com os registros, que são representados por uma lista de *strings*, cada *string* representando a informação de um registro em hexadecimal. As funções providas para a dita interação são: *WriteToRegister*, função que escreve informação num registro; *ReadFromRegister,* lê a informação contida num registro e a retorna como um inteiro.

ALU.py:

É a classe responsável por manipular as funções lógicas e aritméticas, possuindo os atributos *carry, zero, neg, negzero* e *overflow*. Seus métodos são: *Exe*, o método principal, que utiliza os outros métodos quando preciso; *Adder*, método responsável pela realização da adição binária; *Sub*, método responsável pela subtração binária; *Notb,*

Andb, Orb e *Xorb* são os métodos responsáveis pelas funções binárias lógicas *not, and, or* e *xor*, respectivamente.

InstructionDecode.py:

É o módulo responsável pelo tipo *Instruction*, que contém os atributos: *instrType,* atributo que indica o tipo da instrução; *opCode,* atributo que indica o código da operação da instrução; *RC*, atributo que indica o registro destinatário; *RA* e *RB*, os registros utilizados para operações; *offset*, o atributo que armazena o campo de *offset*; *R*, o atributo que armazena o campo *R* de algumas instruções; *cond,* o atributo que armazena o código da condição das instruções *jump*.

Também é o módulo responsável pelas funções que decodificam a instrução, recebida como lista de *strings* do PC. Estas são: *Decode,* função responsável pelo controle de decodificação; *decodeTypeI, decodeTypeII, decodeTypeIII, decodeTypeIVtoVI* e *decoteTypeVII* são as funções de decodificação específicas a cada tipo de instrução.

PC.py:

É a classe responsável pela leitura das instruções no arquivo, além de armazenar o PC durante a execução e fazer sua atualização. Seus métodos são: *UpdatePC,* é o método responsável por atualizar o PC, somando-o com 1 ou definindo-o como o *offset,* passado por parâmetro; *ReadInstruction*, é o método responsável por ler a instrução do arquivo, retirar os comentários e as possíveis *labels*, e separar os campos numa lista de *strings*; *loadIR*, método responsável por carregar uma instrução na lista de instruções *IR*, atributo da classe; *InstructionFetch*, método responsável por buscar uma instrução na lista *IR* baseando-se no valor do atributo PC.

uRisc.py

É o módulo principal, responsável por realizar a execução do simulador, baseando-se nos parâmentros passados por linha de comando. Suas funções são: *main*, função principal, que organiza os parâmetros passados por linha de comando e determina a execução de todo o resto; *fetchLabels,* função responsável por detectar todas as *labels* presentes no código fonte e armazená-las como chave num dicionário, sendo seu valor o PC da próxima instrução que segue a *label*; *screen*, função responsável pela impressão na saída padrão do estado no processador num determinado instante; *uRisc*, função responsável pela execução do simulador em si.

Testes:

Foram utilizados os arquivos providos pelo professor para a realização dos testes, que contêm uma grande combinação de instruções diferentes e comentários auxiliando a correção. Também foram utilizados dois testes feitos por mim para verificar o uso de subrotinas um pouco mais elaboradas. Estes são: *fibonacci.asm*, que escreve na memória os dez primeiros números da sequência de fibonacci; *fatorial.asm*, que escreve na memória o fatorial do número 5.

Listagem do código fonte:

Registers.py:

```
Regs = [format(0, "#06x") for i in range(8)]

def WriteToRegister(n, data, R=None):
    data = format(int(data, 2), "#06x")
    if R != None:
        #0xff00 if lcl(R=0) else 0x00ff (lch-R=1)
        if R == 1:
            Regs[n] = data
        else:
            data = switchBytes(data)
            Regs[n] = data
        return
    Regs[n] = data

def ReadFromRegister(n):
    if n == None:
        return
    regData = Regs[n]
    regData = int(regData, 16)
    return regData
```

ALU.py:

```
from Registers import WriteToRegister

class ALU():
    def __init__(self):
        self.overflow = 0b0
        self.carry = 0b0
        self.neg = 0b0
        self.zero = 0b0
        self.negzero = 0b0
    def Exe(self, opCode, RC, ra, rb):
        if ra != None:
            ra = format(ra, "#018b")
        if rb != None:
            rb = format(rb, "#018b")
        if opCode == 0b0: #zero
            WriteToRegister(RC, format(0, "#018b"))
```

```
            self.zero = 1
            self.neg = 0
            self.carry = 0
            self.overflow = 0
            self.negzero = 1
        if opCode == 0b1001:
            WriteToRegister(RC, ra)
            self.carry = 0
            self.overflow = 0
            if ra[2] == '1':
                self.neg = 1
                self.zero = 0
                self.negzero = 1
            elif int(ra, 2) == 0:
                self.zero = 1
                self.neg = 0
                self.negzero = 1
        elif opCode == 0b11000 or opCode == 0b11010 or opCode == 0b11100 or
opCode == 0b11101 or opCode == 0b11011 or opCode == 0b11001:
            rc = 0
            if opCode == 0b11000: #arithmetic op
                rc = self.Adder(ra, rb)#ra+rb
            elif opCode == 0b11010:
                rc = self.Adder(ra, rb)
                rc = self.Adder(rc, format(1, "#018b"))#ra+rb+1
            elif opCode == 0b11100:
                rc = self.Adder(ra, format(1, "#018b"))#ra+1
            elif opCode == 0b11101:
                rc = self.Sub(ra, format(1, "#018b"))#ra-1
            elif opCode == 0b11011:
                rc = self.Sub(ra, rb)
                rc = self.Sub(ra, format(1, "#018b"))#ra-rb-1
            elif opCode == 0b11001:
                rc = self.Sub(ra, rb)#ra-rb
            if int(rc,2) == 0:
                self.zero = 1
                self.neg = 0
                self.negzero = 1
            else:
                self.zero = 0
                if rc[2] == '1':
                    self.neg = 1
                    self.negzero = 1
                else:
                    self.neg = 0
                    self.negzero = 0
            WriteToRegister(RC, rc)
```

```
        elif opCode == 0b110 or opCode == 0b111 or opCode == 0b1000 or
opCode == 0b1011 or opCode == 0b1011 or opCode == 0b01 or opCode ==
0b00011 or opCode == 0b00101 or opCode == 0b00100 or opCode == 0b1010
or opCode == 0b10:
            self.carry = 0   #logical op
            self.overflow = 0
            if opCode == 0b110:
                WriteToRegister(RC, self.Xorb(ra,rb))
            elif opCode == 0b1010:
                WriteToRegister(RC, self.Andb(self.Notb(ra), rb))
            elif opCode == 0b111:
                WriteToRegister(RC, self.Notb(self.Xorb(ra,rb)))
            elif opCode == 0b1000:
                WriteToRegister(RC, self.Notb(ra))
            elif opCode == 0b1011:
                WriteToRegister(RC, self.Orb(ra, self.Notb(rb)))
            elif opCode == 0b01:
                WriteToRegister(RC, format(1, "#018b"))
            elif opCode == 0b00011:
                WriteToRegister(RC, self.Orb(self.Notb(ra), self.Notb(rb)))
            elif opCode == 0b00101:
                WriteToRegister(RC, self.Andb(self.Notb(ra), self.Notb(rb)))
            elif opCode == 0b00100:
                WriteToRegister(RC, self.Orb(ra,rb))
            elif opCode == 0b10:
                WriteToRegister(RC, self.Andb(ra,rb))
        elif opCode == 0b10001 or opCode == 0b10000: #pode estar errado
            ra = format(int(ra, 2)<<1, "#018b")
            if(len(ra) > 18):
                ra = "0b"+ra[3:]
            bRa = ra
            r15 = int(bRa[2])
            r14 = int(ra[2])
            self.carry = r15
            if opCode == 0b10000:
                self.overflow = 0
            else:
                self.overflow = r15 ^ r14
            if ra[2] == '1':
                self.neg = 1
                self.negzero = 1
            WriteToRegister(RC, ra) #shift aritmético à esquerda msm q lógico.
        elif opCode == 0b10011:
            sign = ra[2]
            ra = format(int(ra, 2)>>1, "#018b")
            newRa = "0b"+sign+ra[3:]
            if sign == '1':
                self.neg = 1
```

```
            self.negzero = 1
        WriteToRegister(RC, newRa) #tbm
        self.carry = 0
        self.overflow = 0
    elif opCode == 0b10010:
        ra = format(int(ra, 2)>>1, "#018b")
        WriteToRegister(RC, ra)
        if int(ra, 2) == 0:
            self.zero = 1
            self.negzero = 1
        else:
            self.neg = 0
            self.zero = 0
            self.negzero = 0
        self.carry = 0
        self.overflow = 0

def Adder(self, a, b):#16bits input com o 0b
    self.carry = 0
    c = ['0']*16
    carryIn = 0
    carryOut = 0
    for i in range(17, 1, -1):
        carryIn = self.carry
        if a[i] == '1' and b[i] == '1':
            if self.carry == 0:
                c[i-2] = '0'
            else:
                c[i-2] = '1'
            self.carry = 1
        elif a[i] == '1' or b[i] == '1':
            if self.carry == 1:
                c[i-2] = '0'
            else:
                c[i-2] = '1'
                self.carry = 0
        else:
            if self.carry == 1:
                c[i-2] = '1'
            else:
                c[i-2] = '0'
            self.carry = 0
        carryOut = self.carry
    self.overflow = carryIn ^ carryOut
    result = "0b"
    for i in c:
        result += i
    return result
```

```
def Sub(self, a, b):#16bits input com o 0b
    self.carry = 0
    c = ['0']*16
    borrowIn = 0
    borrowOut = 0
    for i in range(17, 1, -1):
        borrowIn = self.carry
        if a[i] == b[i]:
            if self.carry == 1:
                c[i-2] = '1'
            else:
                c[i-2] = '0'
                self.carry = 0
        elif a[i] == '1':
            if self.carry == 1:
                c[i-2] = '0'
                self.carry = 0
            else:
                c[i-2] = '1'
        else:
            if self.carry == 1:
                c[i-2] = '0'
            else:
                c[i-2] = '1'
                self.carry = 1
        borrowOut = self.carry
    self.overflow = borrowIn ^ borrowOut
    result = "0b"
    for i in c:
        result += i
    return result

def Notb(self, c):
    res = "0b"
    for i in range(2, 18):
        if c[i] == '1':
            res += '0'
        else:
            res += '1'
    return res

def Andb(self, a, b):
    res = "0b"
    for i in range(2, 18):
        if a[i] != b[i]:
            res += '0'
        else:
```

```
            res += a[i]
        return res

    def Orb(self, a, b):
        res = "0b"
        for i in range(2, 18):
            if a[i] != b[i]:
                res += '1'
            else:
                res += a[i]
        return res

    def Xorb(self, a, b):
        res = "0b"
        for i in range(2, 18):
            if a[i] != b[i]:
                res += '1'
            else:
                res += '0'
        return res
```

PC.py

```
class ProgramCounter():
    def __init__(self):
        self.PC = 0
        self.IR = []

    def UpdatePC(self, offset=None):
        if offset == None:
            self.PC += 1
        else:
            self.PC = offset

    def ReadInstruction(self, line):
        preIR = ""
        line = line.strip('\n')
        if line == "":
            return line
        commentPos = line.find(';')
        if commentPos != -1:
            preIR = line[:commentPos]
        else:
            preIR = line
        labelPos = preIR.find(':')
        if labelPos != -1:
```

```
            preIR = preIR[labelPos+1:]
        return preIR.split()

    def loadIR(self, I):
        self.IR.append(I)

    def InstructionFetch(self):
        return self.IR[self.PC]
```

uRisc.py:

```
from InstructionDecode import Decode, typeI, typeII, typeIII, typeIV, typeV,
typeVI, typeVII
from PC import ProgramCounter
from ALU import ALU
from random import randint
from Registers import *
import argparse

MEMORY = ["0x0000" for i in range(65535)]

def signed16b(n):#nem sei se será preciso usar
    if n > 32767:
        return n-65536
    else:
        return n

def screen(A, PC):
    print("-----Estado do processador-----")
    print("R0:{}\tR1:{}\tcarry:{}\nR2:{}\tR3:{}\toverflow:{}\nR4:{}\tR5:
{}\tzero:{}\nR6:{}\tR7:{}\t\tneg:{}\n\t\tPC:{}\tnegzero:{}".format(Regs[0],
Regs[1], A.carry, Regs[2], Regs[3], A.overflow, Regs[4], Regs[5], A.zero, Regs[6],
Regs[7], A.neg, PC, A.negzero))

def fetchLabels(Labels, srcFile):
    i = 0
    f = open(srcFile, 'r')
    pendingLabel = ""
    for line in f:
        if line == '\n':
            continue
        commentPos = line.find(';')
        if commentPos != -1:
            line = line[:commentPos]
            if line == "":
                continue
        labelPos = line.find(':')
```

```
        if labelPos != -1:
            pendingLabel = line[:labelPos]
            line = line[labelPos+2:]
            if line == "":
                continue
        if pendingLabel != "":
            Labels.update({pendingLabel.lower():i})
            pendingLabel = ""
        i += 1

def uRisc(srcFile, screenFlag):
    f = open(srcFile, 'r')
    Labels = dict()
    PC = ProgramCounter()
    A = ALU()
    fetchLabels(Labels, srcFile)
    for line in f:
        unpreparedI = []
        unpreparedI = PC.ReadInstruction(line)
        if unpreparedI == [] or unpreparedI == "":
            continue
        I = Decode(unpreparedI, Labels)
        if I == None:
            continue
        else:
            PC.loadIR(I)
    PCMax = len(PC.IR)
    while True:
        if PC.PC == PCMax:
            break
        I = PC.InstructionFetch()
        PrevPC = PC.PC
        PC.UpdatePC()
        if I.instrType == 0b01:
            if I.opCode == 0b10100 or I.opCode == 0b10110:
                a = ReadFromRegister(I.RA)
                b = ReadFromRegister(I.RB)
                if I.opCode == 0b10100:
                    WriteToRegister(I.RA, MEMORY[b]) #load
                else:
                    MEMORY[a] = format(b, "#06x") #store
            else:
                a = ReadFromRegister(I.RA)
                b = ReadFromRegister(I.RB)
                A.Exe(I.opCode, I.RC, a, b)
        elif I.instrType == 0b10:
            WriteToRegister(I.RC, I.offset)
        elif I.instrType == 0b11:
```

```
            if I.R == 1:
                WriteToRegister(I.RC, I.offset, I.R)
            else:
                WriteToRegister(I.RC, I.offset, I.R)
        elif I.instrType == 0b00:#jumps
            jumpCond = None
            if I.opCode == 0b00:
                jumpCond = 0
            elif I.opCode == 0b01:
                jumpCond = 1
            elif I.opCode == 0b10:
                jumpCond = 1#vai dar jump
            elif I.opCode == 0b11:
                if I.R == 0:
                    WriteToRegister(7, format(PC.PC, "#016b")) #jump and link
                    b = ReadFromRegister(I.RC)
                    PC.UpdatePC(b)
                    continue
                else:
                    b = ReadFromRegister(I.RC)#jump register
                    PC.UpdatePC(b)
                    continue
            #determinando cond agora
            jump = None
            if I.cond != None:
                if I.cond == 0b100:
                    jump = jumpCond == A.neg
                elif I.cond == 0b101:
                    jump = jumpCond == A.zero
                elif I.cond == 0b110:
                    jump = jumpCond == A.carry
                elif I.cond == 0b111:
                    jump = jumpCond == A.negzero
                elif I.cond == 0b0:
                    jump = True
                elif I.cond == 0b011:
                    jump = jumpCond == A.overflow
            else:
                jump = True
            if jump:
                PC.UpdatePC(I.offset)
        if screenFlag:
            screen(A, format(PrevPC, "#06x"))
            wait = input()

def main():
```

```
    parser = argparse.ArgumentParser(prog="uRisc", description="Um simulador
uRisc.", epilog="E é isso aí. Até mais, e obrigado pelos peixes!", usage="%
(prog)s filename [-h] [-d inicio nPalavras] [-s] [-p]")
    parser.add_argument('filename', help="nome do arquivo que possui o código
fonte")
    parser.add_argument('-d', '--dump', help="dump de memória a partir de uma
posição inicial, seguido de n palavras", nargs=2)
    parser.add_argument('-s', '--screen', help="escreve na saída padrão o estado
do processador após cada instrução", action='store_true')
    args = parser.parse_args()
    srcFile = args.filename
    screen = args.screen
    uRisc(srcFile, screen)
    if args.dump != None:
        aux = "0x"+args.dump[0]
        memDump = int(aux, 16)
        nWords = int(args.dump[1])
        print(memDump)
        print(nWords)
        print(MEMORY[memDump:memDump+nWords])

if __name__ == "__main__":
    main()
```

InstructionDecode.py:

```
typeI = ["zeros", "and", "andnota", "passa", "xor", "or", "nor", "xnor", "passnota",
        "ornotb", "nand", "ones", "add", "addinc", "inca", "subdec", "sub", "deca",
        "lsl", "lsr", "asr", "asl"]
typeII = ["loadlit", "lc"]
typeIII = ["lcl", "lch"]
typeIV = ["jf", "jt"]
typeV = ["j"]
typeVI = ["jal", "jr"]
typeVII = ["store", "st", "load", "ld"]

#, instrType, opCode=None, RC=None, RA=None, RB=None, offset=None,
R=None, cond=None


class Instruction():
    def __init__(self):
        self.instrType = None
```

```
        self.opCode = None
        self.RC = None
        self.RA = None
        self.RB = None
        self.offset = None
        self.R = None
        self.cond = None
    def littleToBig(self, attr):
        c1 = attr & 255
        c2 = (attr >> 8) & 255
        attr = (c1 << 8) + c2
    def setRi(self, n, i):
        if n > 8:
            raise RuntimeError #registro inexistente
        else:
            if i == 1:
                self.RC = n
            elif i == 2:
                self.RA = n
            elif i == 3:
                self.RB = n
    def setOffset(self, n): #deve extender constante n
        extendedSignal = "0b"
        b = format(n, "#018b")
        if self.instrType != 0b10:
            self.offset = b
            return
        sign = b[7]
        for i in range(5):
            extendedSignal += sign
        extendedSignal += b[7:]
        self.offset = extendedSignal

    def printInstr(self):#teste
        print("Type:{}\tOpCode:{}\nRC:{}\toffset:{}\nRA:{}\tRB:{}\nR:{}\tcond:
{}".format(bin(self.instrType), self.opCode, self.RC, self.offset, self.RA, self.RB,
self.R, self.cond))


def Decode(line, Labels):
    I = Instruction()
    s = ""
    for elem in line[1:]:
        s += elem
    op = line[0]
    F = s.split(',')
    if op in typeI:
        I.instrType = 0b01
```

```
        decodeTypeI(op, I, F)
    elif op in typeII:
        I.instrType = 0b10
        decodeTypeII(I, F)
    elif op in typeIII:
        I.instrType = 0b11
        if op == "lcl":
            I.R = 0b0
        else:
            I.R = 0b1
        decodeTypeIII(I, F)
    elif op[:2] in typeIV or op in typeV or op in typeVI:
        cond = ""
        I.instrType = 0b00
        if op[:2] in typeIV:
            if op[:2] == "jf":
                I.opCode = 0b00
            else:
                I.opCode = 0b01
            cond += op[3:]
        elif op in typeV:
            I.opCode = 0b10
        elif op in typeVI:
            I.opCode = 0b11
            if op == "jal":
                I.R = 0b0
            else:
                I.R = 0b1
        decodeTypeIVtoVI(I, F, cond, Labels)
    elif op in typeVII:
        I.instrType = 0b01
        if op == "load" or op == "ld":
            I.opCode = 0b10100
        else:
            I.opCode = 0b10110
        decodeTypeVII(I, F)
    else:
        raise SyntaxError("Instrução errada")
    return I


def decodeTypeI(op, I, F):
    #adicionar tratamento para o zero e outras instruções especiais.
    opCode = -1
    if op == "zeros" or op == "ones":
        if op == "zeros":
            I.opCode = 0b0
        else:
```

```
            I.opCode = 0b1
        try:
            I.setRi(int(F[0][1]), 1)
        except IndexError:
            raise SyntaxError("Instrução incompleta")
        if F[0][0].lower() != 'r':
            raise SyntaxError("Instrução errada")
        return
    elif op == "and":
        I.opCode = 0b10
    elif op == "andnota":
        I.opCode = 0b1010
    elif op == "passa" or op == "passnota" or  op == "inca" or op == "deca" or op
== "lsl" or op == "lsr" or op == "asr" or op == "asl":
        if op == "passa":
            I.opCode = 0b1001
        elif op == "passnota":
            I.opCode = 0b1000
        elif op == "inca":
            I.opCode = 0b11100
        elif op == "deca":
            I.opCode = 0b11101
        elif op == "lsl":
            I.opCode = 0b10000
        elif op == "lsr":
            I.opCode = 0b10010
        elif op == "asr":
            I.opCode = 0b10011
        elif op == "asl":
            I.opCode = 0b10001
        try:
            I.setRi(int(F[0][1]), 1)
            I.setRi(int(F[1][1]), 2)
        except IndexError:
            raise SyntaxError("Instrução incompleta")
        for i in F:
            if i[0].lower() != 'r':
                raise SyntaxError("Instrução errada")
        return
    elif op == "xor":
        I.opCode = 0b110
    elif op == "or":
        I.opCode = 0b100
    elif op == "nor":
        I.opCode = 0b101
    elif op == "xnor":
        I.opCode = 0b111
    elif op == "ornotb":
```

```
        I.opCode = 0b1011
    elif op == "nand":
        I.opCode = 0b11
    elif op == "add":
        I.opCode = 0b11000
    elif op == "addinc":
        I.opCode = 0b11010
    elif op == "subdec":
        I.opCode = 0b11011
    elif op == "sub":
        I.opCode = 0b11001
    try:
        I.setRi(int(F[0][1]), 1)
        I.setRi(int(F[1][1]), 2)
        I.setRi(int(F[2][1]), 3)
    except IndexError:
        raise SyntaxError("Instrução incompleta")
    for i in F:
        if i[0].lower() != 'r':
            raise SyntaxError("Instrução errada")


def decodeTypeII(I, F):
    I.RC = int(F[0][1])
    I.setOffset(int(F[1]))

def decodeTypeIII(I, F):
    if F[0][0].lower() != 'r':
        raise SyntaxError("Instrução errada")
    I.setRi(int(F[0][1]), 1)
    const = F[1].lower().split("const")
    if len(const) == 2:
        I.setOffset(int(const[1]))
    else:
        I.setOffset(int(const[0]))

def decodeTypeIVtoVI(I, F, cond, Labels):
    if I.opCode == 0b00 or I.opCode == 0b01:
        condCode = 0
        if cond == "neg":
            condCode = 0b100
        elif cond == "zero":
            condCode = 0b101
        elif cond == "carry":
            condCode = 0b110
        elif cond == "negzero":
            condCode = 0b111
        elif cond == "true":
```

```
            condCode = 0b0
        elif cond == "overflow":
            condCode = 0b011
        else:
            raise SyntaxError("Instrução errada")
        try:
            I.setOffset(Labels[F[0].lower()])
        except KeyError:
            I.setOffset(int(F[0]))
        I.cond = condCode
    elif I.opCode == 0b10:
        try:
            I.setOffset(Labels[F[0].lower()])
        except KeyError:
            I.setOffset(Labels[F[0].lower()])
    elif I.opCode == 0b11:
        if F[0][0].lower() != 'r':
            raise SyntaxError("Instrução errada")
        I.setRi(int(F[0][1]), 1)

def decodeTypeVII(I, F):
    for i in F:
        if i[0].lower() != 'r':
            raise SyntaxError("Instrução errada")
    I.setRi(int(F[0][1]), 2)
    I.setRi(int(F[1][1]), 3)
```

Testes:

fibonacci.asm:

```
;Programa que armazena na memória os 10 primeiros números da sequência de fibonacci

loadlit r0, 1000
loadlit r6, 10
zeros r1
ones r2
store r0, r1
inca r0, r0
store r0, r2
inca r0, r0

LOOP: add r3, r2, r1
```

```
    store r0, r3
    inca r0, r0
    passa r1, r2
    passa r2, r3
    deca r6, r6
    jf.zero LOOP
```

fatorial.asm:

```
;Programa que calcula o fatorial de 5 e armazena na memória

loadlit r0, 1000
loadlit r4, 5
loadlit r5, 7
loadlit r6, 5
jal r5
store r0,r6
L: j L

FATORIAL:
    deca r4, r4
    passa r1, r4
    jf.zero MULTIPLICA
    jr r7

MULTIPLICA: add r2,r2,r6
    deca r1,r1
    jf.zero MULTIPLICA
    passa r6, r2
    zeros r2
    j FATORIAL
```

multiplica.asm:

```
; Programa MULTIPLICA.ASM
; Este programa multiplica dois valores de ate 11 bits
; POSITIVOS, armazenados em r0 e r1. O resultado eh
; armazenado em Mem[1000]
;
;
        loadlit r0,50               ; Operando A
        loadlit r1,5          ; Operando B
        zeros r2

LOOP:       add r2,r2,r1
        deca r0,r0
```

```
        jf.zero LOOP
        loadlit r1,1000
        store r1,r2         ; armazena o produto AxB em Mem[1000]
```

logica.asm:

```
; Programa LOGICA.ASM
; Este programa testa todas as operacoes logicas da ALU
; Cada teste incrementa o registrador r4 se for bem sucedido.
; no final, se todos os testes forem bem sucedidos, a pontuacao
; atinge valor 10, que eh armazenada em Mem[1000]
; Cada teste verifica se a tabela verdade para a operacao
; esta sendo gerada corretamente.

zeros r4

teste0:
        zeros r0            ; Operacao C = A & B;
        zeros r1
        and r1,r0,r1
        jf.zero teste1
        zeros r0
        ones r1
        and r1,r0,r1
        jf.zero teste1
        ones r0
        zeros r1
        and r1,r0,r1
        jf.zero teste1
        ones r0
        ones r1
        and r1,r0,r1
       passnota r1,r1
        jf.zero teste1
        inca r4,r4

teste1:
        zeros r0            ; Operacao C = !A & B
        zeros r1
        andnota r1,r0,r1
        jf.zero teste2
        zeros r0
        ones r1
        andnota r1,r0,r1
       passnota r1,r1
        jf.zero teste2
        ones r0
        zeros r1
```

```
        andnota r1,r0,r1
        jf.zero teste2
        ones r0
        ones r1
        andnota r1,r0,r1
        jf.zero teste2
        inca r4,r4

teste2:
        zeros r0            ; Operacao C = A
        passa r1,r0
        jf.zero teste3
        ones r0
        passa r1,r0
       passnota r1,r0
        jf.zero teste3
        inca r4,r4

teste3:
        zeros r0            ; Operacao C = A ^ B
        zeros r1
        xor r1,r0,r1
        jf.zero teste4
        zeros r0
        ones r1
        xor r1,r0,r1
        passnota r1,r1
        jf.zero teste4
        ones r0
        zeros r1
        xor r1,r0,r1
        passnota r1,r1
        jf.zero teste4
        ones r0
        ones r1
        xor r1,r0,r1
        jf.zero teste4
        inca r4,r4

teste4:
        zeros r0            ; Operacao C = A | B
        zeros r1
        or r1,r0,r1
        jf.zero teste5
        zeros r0
        ones r1
        or r1,r0,r1
        passnota r1,r1
```

```
        jf.zero teste5
        ones r0
        zeros r1
        or r1,r0,r1
        passnota r1,r1
        jf.zero teste5
        ones r0
        ones r1
        or r1,r0,r1
       passnota r1,r1
        jf.zero teste5
        inca r4,r4

teste5:
        zeros r0            ; Operacao C = !A & !B
        zeros r1
        nor r1,r0,r1
        passnota r1,r1
        jf.zero teste6
        zeros r0
        ones r1
        nor r1,r0,r1
        jf.zero teste6
        ones r0
        zeros r1
        nor r1,r0,r1
        jf.zero teste6
        ones r0
        ones r1
        nor r1,r0,r1
        jf.zero teste6
        inca r4,r4

teste6:
        zeros r0            ; Operacao C = !A ^ B
        zeros r1
        xnor r1,r0,r1
        passnota r1,r1
        jf.zero teste7
        zeros r0
        ones r1
        xnor r1,r0,r1
        jf.zero teste7
        ones r0
        zeros r1
        xnor r1,r0,r1
        jf.zero teste7
        ones r0
```

```
        ones r1
        xnor r1,r0,r1
        passnota r1,r1
        jf.zero teste7
        inca r4,r4

teste7:
        zeros r0            ; Operacao C = !A
        passnota r0,r0
       passnota r0,r0
        jf.zero teste8
        ones r0
        passnota r0,r0
        jf.zero teste8
        inca r4,r4

teste8:
        zeros r0            ; Operacao C = A | !B
        zeros r1
        ornotb r1,r0,r1
        passnota r1,r1
        jf.zero teste9
        zeros r0
        ones r1
        ornotb r1,r0,r1
        jf.zero teste9
        ones r0
        zeros r1
        ornotb r1,r0,r1
        passnota r1,r1
        jf.zero teste9
        ones r0
        ones r1
        ornotb r1,r0,r1
       passnota r1,r1
        jf.zero teste9
        inca r4,r4

teste9:
        zeros r0            ; Operacao C = !A | !B
        zeros r1
        nand r1,r0,r1
       passnota r1,r1
        jf.zero result
        zeros r0
        ones r1
        nand r1,r0,r1
        passnota r1,r1
```

```
        jf.zero result
        ones r0
        zeros r1
        nand r1,r0,r1
        passnota r1,r1
        jf.zero result
        ones r0
        ones r1
        nand r1,r0,r1
        jf.zero result
        inca r4,r4

result: loadlit r0, 1000
        store r0,r4
```