


2025_2 - COMPILADORES - METATURMA


PAINEL > **MINHAS TURMAS** > **2025_2 - COMPILADORES - METATURMA** > **LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL**



> **AV11 - CÓDIGO PARA EXPRESSÕES ARITMÉTICAS**

 Descrição

 Visualizar envios

AV11 - Código para expressões aritméticas

 **Data de entrega:** quarta, 29 Out 2025, 23:59

 **Arquivos requeridos:** driver.py, Lexer.py, Parser.py, Expression.py, Visitor.py, Asm.py ( [Baixar](#))

Tipo de trabalho:  Trabalho individual

O objetivo deste trabalho é compilar expressões lógicas e aritméticas para um subconjunto de instruções [RISC-V](#). Por hora, iremos considerar uma máquina com somente nove instruções, a saber:

- add rd, rs1, rs2: $rd = rs1 + rs2$
- addi rd, rs1, imm: $rd = rs1 + imm$
- mul rd, rs1, rs2: $rd = rs1 * rs2$
- sub rd, rs1, rs2: $rd = rs1 - rs2$
- xor rd, rs1, rs2: $rd = rs1 \wedge rs2$
- xori rd, rs1, imm: $rd = rs1 \wedge imm$
- div rd, rs1, rs2: $rd = rs1 // rs2$ (signed integer division)
- slt rd, rs1, rs2: $rd = (rs1 < rs2) ? 1 : 0$ (signed comparison)
- slti rd, rs1, imm: $rd = (rs1 < imm) ? 1 : 0$

Este trabalho possui um emulador para estas instruções, implementado no arquivo `Asm.py`. Você não precisa alterar este arquivo para resolver este trabalho. Por outro lado, pode ser instrutivo ler o código do emulador. No arquivo há duas funções, `max` e `distance_with_acceleration` que mostram como criar novas instruções. Essas funções podem ser vistas nas Figuras 1 e 2 abaixo. Note que em RISC-V existe um registrador especial, chamado "x0", que sempre possui o valor zero. Você pode usar este registrador para "montar" instruções complexas a partir de instruções simples. Por exemplo, para definir a variável "a" com o valor 2, você pode usar a instrução de soma com valor imediato: `addi("a", "x0", 2)`.

```
def max(a, b):
    """
    This function computes the maximum between a and b.
    >>> max(2, 3)
    3
    """
    p = Program({}, [])
    p.set_val("rs1", a)
    p.set_val("rs2", b)
    p.add_inst(Slt("t0", "rs2", "rs1"))
    p.add_inst(Slt("t1", "rs1", "rs2"))
    p.add_inst(Mul("t0", "t0", "rs1"))
    p.add_inst(Mul("t1", "t1", "rs2"))
    p.add_inst(Add("rd", "t0", "t1"))
    p.eval()
    return p.get_val("rd")
```

Figura 1: Uma função que calcula o máximo valor entre dois números

```
def distance_with_acceleration(V, A, T):
    """
    This function computes the position of an accelerated object.
    >>> distance_with_acceleration(3, 4, 5)
    65
    """
    p = Program({"rs1": V, "rs2": A, "rs3": T, "x0": 0}, [])
    p.add_inst(Addi("two", "x0", 2))
    p.add_inst(Mul("t0", "rs1", "rs3"))
    p.add_inst(Mul("t1", "rs3", "rs3"))
    p.add_inst(Mul("t2", "rs2", "t1"))
    p.add_inst(Div("t2", "t2", "two"))
    p.add_inst(Add("rd", "t0", "t2"))
    p.eval()
    return p.get_val("rd")
```

Figura 2: Uma função que calcula a distância percorrida por um objeto sujeito a aceleração (A) constante, com velocidade inicial (V), dado um tempo (T).

Neste trabalho, você deverá gerar código para as seguintes expressões, que fazem parte do Laboratório 5 (*Visitors*):

1. Variáveis, como `Var("x")`.
2. Booleanos, como `Bin(True)`.
3. Números como `Num(2)`.
4. Igualdade, como `Eq(Exp0, Exp1)`.
5. Adição, como `Add(Exp0, Exp1)`.
6. Multiplicação, como `Mul(Exp0, Exp1)`.
7. Subtração, como `Sub(Exp0, Exp1)`.
8. Divisão, como `Div(Exp0, Exp1)`.
9. Comparações de menor ou igual, como `Leq(Exp0, Exp1)`.

10. Comparações de menor, como Lth(Exp0, Exp1).
11. Inversão de sinal, como Neg(Exp).
12. Negação de booleano, como Not(Exp).
13. Criação de variáveis, como Let("x", Exp0, Exp1). **Importante:** você pode assumir que todas as variáveis definidas em um programa são diferentes.

Para gerar código, você deverá implementar um novo Visitor, chamado GenVisitor. Cada método visit recebe como entrada uma expressão e um "Programa", e retorna o nome da variável que irá armazenar o valor da expressão. Um Programa é uma instância da classe Program, que está definida no arquivo Asm.py. A tarefa de GenVisitor é inserir instruções no programa, o que pode ser feito via o método Program.add_inst. As Figuras 3 e 4 mostram dois exemplos de métodos visit. Note que algumas expressões, como Add, possuem instruções equivalentes em nossa máquina RISC-V reduzida. Porém, outras expressões, como Not e Eq1 não possuem instruções equivalentes. Você terá de resolver um pequeno quebra-cabeças, neste caso: "Como implementar essas expressões com as instruções que estão disponíveis?" Note que em nenhuma hipótese você deve modificar Asm.py para adicionar novas instruções. Você precisa usar somente as instruções que já estão definidas na máquina reduzida.

Cada método visit retorna o nome da variável que irá armazenar o valor da expressão. Pense neste nome como um "endereço de memória". Em um próximo lab iremos implementar endereços realmente. Por hora, podemos assumir a memória é endereçável pelo nome de variáveis.

```
def visit_var(self, exp, prog):
    """
    Usage:
    >>> e = Var('x')
    >>> p = AsmModule.Program({"x":1}, [])
    >>> g = GenVisitor()
    >>> v = e.accept(g, p)
    >>> p.eval()
    >>> p.get_val(v)
    1
    """
    return exp.identifier
```

Figura 3: Implementação de geração de código para variáveis.

A classe GenVisitor possui um método para criar novos nomes de variáveis: GenVisitor.next_var_name(). Você pode usá-lo sempre que precisar de armazenar um novo valor.

```
def visit_add(self, exp, prog):
    """
    >>> e = Add(Num(13), Num(-13))
    >>> p = AsmModule.Program({}, [])
    >>> g = GenVisitor()
    >>> v = e.accept(g, p)
    >>> p.eval()
    0
    """
    l_name = exp.left.accept(self, prog)
    r_name = exp.right.accept(self, prog)
    v_name = self.next_var_name()
    prog.add_inst(AsmModule.Add(v_name, l_name, r_name))
    return v_name
```

Instâncias de Program possuem um método eval, que avalia as instruções presentes naquele programa, dado o environment que ele contém.

Figura 4: Implementação de geração de código para adição.

Observação sobre o nome de variáveis em blocos let

Conforme explicado anteriormente, neste exercício você pode assumir que blocos let sempre definem variáveis com nomes diferentes. Em outras palavras, um programa como let x <- 1 in let x <- 2 in x end + x end não é válido. Um programa equivalente, válido, seria: let x0 <- 1 in let x1 <- 2 in x1 end + x0 end. Assim, você não precisa tratar situações em que variáveis são definidas com o mesmo nome. No próximo laboratório iremos implementar um visitor simples para fazer essa renomeação automaticamente.

Submetendo e Testando

Para completar este VPL, você deverá entregar seis arquivos: Expression.py, Lexer.py, Parser.py, Visitor.py, Asm.py e driver.py. Você não deverá alterar Asm.py, driver.py ou Expression.py. Para testar sua implementação localmente, você pode usar o comando abaixo:

```
python3 driver.py
2 + 3 # CTRL+D
5
```

A implementação dos diferentes arquivos possui vários comentários doctest, que testam sua implementação. Caso queira testar seu código, simplesmente faça:

```
python3 -m doctest xx.py
```

No exemplo acima, substitua xx.py por algum dos arquivos que você queira testar (experimente com Visitor.py, por exemplo). Caso você não gere mensagens de erro, então seu trabalho está (quase) completo!

Arquivos requeridos

driver.py

```
1 import sys
2 from Expression import *
3 from Visitor import *
4 from Lexer import Lexer
5 from Parser import Parser
6 import Asm as AsmModule
7
8 if __name__ == "__main__":
9     """
10     Este arquivo nao deve ser alterado, mas deve ser enviado para resolver o
11     VPL. O arquivo contem o codigo que testa a implementacao do parser.
12     """
13     text = sys.stdin.read()
14     lexer = Lexer(text)
15     parser = Parser(lexer.tokens())
16     exp = parser.parse()
17     prog = AsmModule.Program({}, [])
18     gen = GenVisitor()
19     var_answer = exp.accept(gen, prog)
20     prog.eval()
21     print(f"{prog.get_val(var_answer)}")
```

Lexer.py

```

1  import sys
2  import enum
3
4
5  class Token:
6      """
7      This class contains the definition of Tokens. A token has two fields: its
8      text and its kind. The "kind" of a token is a constant that identifies it
9      uniquely. See the TokenType to know the possible identifiers (if you want).
10     You don't need to change this class.
11     """
12     def __init__(self, tokenText, tokenKind):
13         # The token's actual text. Used for identifiers, strings, and numbers.
14         self.text = tokenText
15         # The TokenType that this token is classified as.
16         self.kind = tokenKind
17
18
19  class TokenType(enum.Enum):
20      """
21      These are the possible tokens. You don't need to change this class at all.
22      """
23
24      EOF = -1 # End of file
25      NLN = 0 # New line
26      WSP = 1 # White Space
27      COM = 2 # Comment
28      NUM = 3 # Number (integers)
29      STR = 4 # Strings
30      TRU = 5 # The constant true
31      FLS = 6 # The constant false
32      VAR = 7 # An identifier
33      LET = 8 # The 'let' of the let expression
34      INX = 9 # The 'in' of the let expression
35      END = 10 # The 'end' of the let expression
36      EQL = 201
37      ADD = 202
38      SUB = 203
39      MUL = 204
40      DIV = 205
41      LEQ = 206
42      LTH = 207
43      NEG = 208
44      NOT = 209
45      LPR = 210
46      RPR = 211
47      ASN = 212 # The assignment '<-' operator
48
49
50  class Lexer:
51
52     def __init__(self, source):
53         """
54         The constructor of the lexer. It receives the string that shall be
55         scanned.
56         TODO: You will need to implement this method.
57         """
58         pass
59
60     def tokens(self):
61         """
62         This method is a token generator: it converts the string encapsulated
63         into this object into a sequence of Tokens. Examples:
64
65         >>> l = Lexer("1 + 3")
66         >>> [tk.kind for tk in l.tokens()]
67         [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]
68
69         >>> l = Lexer('1 * 2 -- 3\\n')
70         >>> [tk.kind for tk in l.tokens()]
71         [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
72
73         >>> l = Lexer("1 + var")
74         >>> [tk.kind for tk in l.tokens()]
75         [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]
76
77         >>> l = Lexer("let v <- 2 in v end")
78         >>> [tk.kind.name for tk in l.tokens()]
79         ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
80         """
81         token = self.getToken()
82         while token.kind != TokenType.EOF:
83             if (
84                 token.kind != TokenType.WSP
85                 and token.kind != TokenType.COM
86                 and token.kind != TokenType.NLN
87             ):
88                 yield token
89             token = self.getToken()
90
91     def getToken(self):
92         """
93         Return the next token.
94         TODO: Implement this method (you can reuse Lab 5: Visitors)!
95         """
96         token = None
97         return token

```

Parser.py

```

1  import sys
2
3  from Expression import *
4  from Lexer import Token, TokenType
5
6  """
7  This file implements the parser of arithmetic expressions. The same rules of
8  precedence and associativity from Lab 5: Visitors, apply.
9
10 References:
11 see https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm#classic
12 """
13
14 class Parser:
15     def __init__(self, tokens):
16         """
17         Initializes the parser. The parser keeps track of the list of tokens
18         and the current token. For instance:
19         """
20         self.tokens = list(tokens)
21         self.cur_token_idx = 0 # This is just a suggestion!
22         # You can (and probably should!) modify this method.
23
24     def parse(self):
25         """
26         Returns the expression associated with the stream of tokens.
27
28         Examples:
29         >>> parser = Parser([Token('123', TokenType.NUM)])
30         >>> g = GenVisitor()
31         >>> p = AsmModule.Program({}, [])
32         >>> exp = parser.parse()
33         >>> v = exp.accept(g, p)
34         >>> p.eval()
35         >>> p.get_val(v)
36         123
37
38         >>> parser = Parser([Token('True', TokenType.TRU)])
39         >>> g = GenVisitor()
40         >>> p = AsmModule.Program({}, [])
41         >>> exp = parser.parse()
42         >>> v = exp.accept(g, p)
43         >>> p.eval()
44         >>> p.get_val(v)
45         1
46
47         >>> parser = Parser([Token('False', TokenType.FLS)])
48         >>> g = GenVisitor()
49         >>> p = AsmModule.Program({}, [])
50         >>> exp = parser.parse()
51         >>> v = exp.accept(g, p)
52         >>> p.eval()
53         >>> p.get_val(v)
54         0
55
56         >>> tk0 = Token('~', TokenType.NEG)
57         >>> tk1 = Token('123', TokenType.NUM)
58         >>> parser = Parser([tk0, tk1])
59         >>> g = GenVisitor()
60         >>> p = AsmModule.Program({}, [])
61         >>> exp = parser.parse()
62         >>> v = exp.accept(g, p)
63         >>> p.eval()
64         >>> p.get_val(v)
65         -123
66
67         >>> tk0 = Token('3', TokenType.NUM)
68         >>> tk1 = Token('*', TokenType.MUL)
69         >>> tk2 = Token('4', TokenType.NUM)
70         >>> parser = Parser([tk0, tk1, tk2])
71         >>> g = GenVisitor()
72         >>> p = AsmModule.Program({}, [])
73         >>> exp = parser.parse()
74         >>> v = exp.accept(g, p)
75         >>> p.eval()
76         >>> p.get_val(v)
77         12
78
79         >>> tk0 = Token('3', TokenType.NUM)
80         >>> tk1 = Token('*', TokenType.MUL)
81         >>> tk2 = Token('~', TokenType.NEG)
82         >>> tk3 = Token('4', TokenType.NUM)
83         >>> parser = Parser([tk0, tk1, tk2, tk3])
84         >>> g = GenVisitor()
85         >>> p = AsmModule.Program({}, [])
86         >>> exp = parser.parse()
87         >>> v = exp.accept(g, p)
88         >>> p.eval()
89         >>> p.get_val(v)
90         -12
91
92         >>> tk0 = Token('30', TokenType.NUM)
93         >>> tk1 = Token('/', TokenType.DIV)
94         >>> tk2 = Token('4', TokenType.NUM)
95         >>> parser = Parser([tk0, tk1, tk2])
96         >>> g = GenVisitor()
97         >>> p = AsmModule.Program({}, [])
98         >>> exp = parser.parse()
99         >>> v = exp.accept(g, p)
100        >>> p.eval()
101        >>> p.get_val(v)
102        7
103

```

```
104 >>> tk0 = Token('3', TokenType.NUM)
105 >>> tk1 = Token('+', TokenType.ADD)
106 >>> tk2 = Token('4', TokenType.NUM)
107 >>> parser = Parser([tk0, tk1, tk2])
108 >>> g = GenVisitor()
109 >>> p = AsmModule.Program({}, [])
110 >>> exp = parser.parse()
111 >>> v = exp.accept(g, p)
112 >>> p.eval()
113 >>> p.get_val(v)
114 7
115
116 >>> tk0 = Token('30', TokenType.NUM)
117 >>> tk1 = Token('-', TokenType.SUB)
118 >>> tk2 = Token('4', TokenType.NUM)
119 >>> parser = Parser([tk0, tk1, tk2])
120 >>> g = GenVisitor()
121 >>> p = AsmModule.Program({}, [])
122 >>> exp = parser.parse()
123 >>> v = exp.accept(g, p)
124 >>> p.eval()
125 >>> p.get_val(v)
126 26
127
128 >>> tk0 = Token('2', TokenType.NUM)
129 >>> tk1 = Token('*', TokenType.MUL)
130 >>> tk2 = Token('(', TokenType.LPR)
131 >>> tk3 = Token('3', TokenType.NUM)
132 >>> tk4 = Token('+', TokenType.ADD)
133 >>> tk5 = Token('4', TokenType.NUM)
134 >>> tk6 = Token(')', TokenType.RPR)
135 >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
136 >>> g = GenVisitor()
137 >>> p = AsmModule.Program({}, [])
138 >>> exp = parser.parse()
139 >>> v = exp.accept(g, p)
140 >>> p.eval()
141 >>> p.get_val(v)
142 14
143
144 >>> tk0 = Token('4', TokenType.NUM)
145 >>> tk1 = Token('==', TokenType.EQL)
146 >>> tk2 = Token('4', TokenType.NUM)
147 >>> parser = Parser([tk0, tk1, tk2])
148 >>> g = GenVisitor()
149 >>> p = AsmModule.Program({}, [])
150 >>> exp = parser.parse()
151 >>> v = exp.accept(g, p)
152 >>> p.eval()
153 >>> p.get_val(v)
154 1
155
156 >>> tk0 = Token('4', TokenType.NUM)
157 >>> tk1 = Token('<=', TokenType.LEQ)
158 >>> tk2 = Token('4', TokenType.NUM)
159 >>> parser = Parser([tk0, tk1, tk2])
160 >>> g = GenVisitor()
161 >>> p = AsmModule.Program({}, [])
162 >>> exp = parser.parse()
163 >>> v = exp.accept(g, p)
164 >>> p.eval()
165 >>> p.get_val(v)
166 1
167
168 >>> tk0 = Token('4', TokenType.NUM)
169 >>> tk1 = Token('<', TokenType.LTH)
170 >>> tk2 = Token('4', TokenType.NUM)
171 >>> parser = Parser([tk0, tk1, tk2])
172 >>> g = GenVisitor()
173 >>> p = AsmModule.Program({}, [])
174 >>> exp = parser.parse()
175 >>> v = exp.accept(g, p)
176 >>> p.eval()
177 >>> p.get_val(v)
178 0
179
180 >>> tk0 = Token('not', TokenType.NOT)
181 >>> tk1 = Token('4', TokenType.NUM)
182 >>> tk2 = Token('<', TokenType.LTH)
183 >>> tk3 = Token('4', TokenType.NUM)
184 >>> parser = Parser([tk0, tk1, tk2, tk3])
185 >>> g = GenVisitor()
186 >>> p = AsmModule.Program({}, [])
187 >>> exp = parser.parse()
188 >>> v = exp.accept(g, p)
189 >>> p.eval()
190 >>> p.get_val(v)
191 1
192
193 >>> tk0 = Token('let', TokenType.LET)
194 >>> tk1 = Token('v', TokenType.VAR)
195 >>> tk2 = Token('<-', TokenType.ASN)
196 >>> tk3 = Token('42', TokenType.NUM)
197 >>> tk4 = Token('in', TokenType.INX)
198 >>> tk5 = Token('v', TokenType.VAR)
199 >>> tk6 = Token('end', TokenType.END)
200 >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
201 >>> g = GenVisitor()
202 >>> p = AsmModule.Program({}, [])
203 >>> exp = parser.parse()
204 >>> v = exp.accept(g, p)
205 >>> p.eval()
206 >>> p.get_val(v)
207 --
```

```
207 42
208
209 >>> tk0 = Token('let', TokenType.LET)
210 >>> tk1 = Token('v', TokenType.VAR)
211 >>> tk2 = Token('<->', TokenType.ASN)
212 >>> tk3 = Token('21', TokenType.NUM)
213 >>> tk4 = Token('in', TokenType.INX)
214 >>> tk5 = Token('v', TokenType.VAR)
215 >>> tk6 = Token('+', TokenType.ADD)
216 >>> tk7 = Token('v', TokenType.VAR)
217 >>> tk8 = Token('end', TokenType.END)
218 >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
219 >>> g = GenVisitor()
220 >>> p = AsmModule.Program({}, [])
221 >>> exp = parser.parse()
222 >>> v = exp.accept(g, p)
223 >>> p.eval()
224 >>> p.get_val(v)
225 42
226 """
227
228 # TODO: implement this method.
229 return None
```

Expression.py

```
1 """
2 This file implements the data structures that represent Expressions. You don't
3 need to modify it for this assignment.
4 """
5
6 from abc import ABC, abstractmethod
7 from Visitor import *
8
9
10 class Expression(ABC):
11     @abstractmethod
12     def accept(self, visitor, arg):
13         raise NotImplementedError
14
15
16 class Var(Expression):
17     """
18     This class represents expressions that are identifiers. The value of an
19     identifier is the value associated with it in the environment table.
20     """
21
22     def __init__(self, identifier):
23         self.identifier = identifier
24
25     def accept(self, visitor, arg):
26         """
27         Variables don't need to be implemented for this exercise.
28         """
29         return visitor.visit_var(self, arg)
30
31
32 class Bln(Expression):
33     """
34     This class represents expressions that are boolean values. There are only
35     two boolean values: true and false. The acceptance of such an expression
36     is the boolean itself.
37     """
38
39     def __init__(self, bln):
40         self.bln = bln
41
42     def accept(self, visitor, arg):
43         """
44         booleans don't need to be implemented for this exercise.
45         """
46         return visitor.visit_bln(self, arg)
47
48
49 class Num(Expression):
50     """
51     This class represents expressions that are numbers. The acceptance of such
52     an expression is the number itself.
53     """
54
55     def __init__(self, num):
56         self.num = num
57
58     def accept(self, visitor, arg):
59         """
60         Example:
61         >>> e = Num(3)
62         >>> ev = EvalVisitor()
63         >>> e.accept(ev, None)
64         3
65         """
66         return visitor.visit_num(self, arg)
67
68
69 class BinaryExpression(Expression):
70     """
71     This class represents binary expressions. A binary expression has two
72     sub-expressions: the left operand and the right operand.
73     """
74
75     def __init__(self, left, right):
76         self.left = left
77         self.right = right
78
79     @abstractmethod
80     def accept(self, visitor, arg):
81         raise NotImplementedError
82
83
84 class Eql(BinaryExpression):
85     """
86     This class represents the equality between two expressions. The acceptance
87     of such an expression is True if the subexpressions are the same, or false
88     otherwise.
89     """
90
91     def accept(self, visitor, arg):
92         """
93         Equality doesn't need to be implemented for this exercise.
94         """
95         return visitor.visit_eql(self, arg)
96
97
98 class Add(BinaryExpression):
99     """
100     This class represents addition of two expressions. The acceptance of such
101     an expression is the addition of the two subexpression's values.
102     """
103
```



```
104     def accept(self, visitor, arg):
105         """
106         Example:
107         >>> n1 = Num(3)
108         >>> n2 = Num(4)
109         >>> e = Add(n1, n2)
110         >>> ev = EvalVisitor()
111         >>> e.accept(ev, None)
112         7
113         """
114         return visitor.visit_add(self, arg)
115
116
117     class Sub(BinaryExpression):
118         """
119         This class represents subtraction of two expressions. The acceptance of
120         such an expression is the subtraction of the two subexpression's values.
121         """
122
123         def accept(self, visitor, arg):
124             """
125             Example:
126             >>> n1 = Num(3)
127             >>> n2 = Num(4)
128             >>> e = Sub(n1, n2)
129             >>> ev = EvalVisitor()
130             >>> e.accept(ev, None)
131             -1
132             """
133             return visitor.visit_sub(self, arg)
134
135
136     class Mul(BinaryExpression):
137         """
138         This class represents multiplication of two expressions. The acceptance of
139         such an expression is the product of the two subexpression's values.
140         """
141
142         def accept(self, visitor, arg):
143             """
144             Example:
145             >>> n1 = Num(3)
146             >>> n2 = Num(4)
147             >>> e = Mul(n1, n2)
148             >>> ev = EvalVisitor()
149             >>> e.accept(ev, None)
150             12
151             """
152             return visitor.visit_mul(self, arg)
153
154
155     class Div(BinaryExpression):
156         """
157         This class represents the integer division of two expressions. The
158         acceptance of such an expression is the integer quotient of the two
159         subexpression's values.
160         """
161
162         def accept(self, visitor, arg):
163             """
164             Example:
165             >>> n1 = Num(28)
166             >>> n2 = Num(4)
167             >>> e = Div(n1, n2)
168             >>> ev = EvalVisitor()
169             >>> e.accept(ev, None)
170             7
171             >>> n1 = Num(22)
172             >>> n2 = Num(4)
173             >>> e = Div(n1, n2)
174             >>> ev = EvalVisitor()
175             >>> e.accept(ev, None)
176             5
177             """
178             return visitor.visit_div(self, arg)
179
180
181     class Leq(BinaryExpression):
182         """
183         This class represents comparison of two expressions using the
184         less-than-or-equal comparator. The acceptance of such an expression is a
185         boolean value that is true if the left operand is less than or equal the
186         right operand. It is false otherwise.
187         """
188
189         def accept(self, visitor, arg):
190             """
191             Comparisons don't need to be implemented for this exercise.
192             """
193             return visitor.visit_leq(self, arg)
194
195
196     class Lth(BinaryExpression):
197         """
198         This class represents comparison of two expressions using the
199         less-than comparison operator. The acceptance of such an expression is a
200         boolean value that is true if the left operand is less than the right
201         operand. It is false otherwise.
202         """
203
204         def accept(self, visitor, arg):
205             """
206             Comparisons don't need to be implemented for this exercise.
207             """
208             return visitor.visit_lth(self, arg)
```

```
207         return visitor.visit_lth(self, arg)
208     return visitor.visit_lth(self, arg)
209
210
211 class UnaryExpression(Expression):
212     """
213     This class represents unary expressions. A unary expression has only one
214     sub-expression.
215     """
216
217     def __init__(self, exp):
218         self.exp = exp
219
220     @abstractmethod
221     def accept(self, visitor, arg):
222         raise NotImplementedError
223
224
225 class Neg(UnaryExpression):
226     """
227     This expression represents the additive inverse of a number. The additive
228     inverse of a number n is the number -n, so that the sum of both is zero.
229     """
230
231     def accept(self, visitor, arg):
232         """
233         Example:
234         >>> n = Num(3)
235         >>> e = Neg(n)
236         >>> ev = EvalVisitor()
237         >>> e.accept(ev, None)
238         -3
239         >>> n = Num(0)
240         >>> e = Neg(n)
241         >>> ev = EvalVisitor()
242         >>> e.accept(ev, None)
243         0
244         """
245         return visitor.visit_neg(self, arg)
246
247
248 class Not(UnaryExpression):
249     """
250     This expression represents the negation of a boolean. The negation of a
251     boolean expression is the logical complement of that expression.
252     """
253
254     def accept(self, visitor, arg):
255         """
256         No need to implement negation for this exercise, for we don't even have
257         booleans at this point.
258         """
259         return visitor.visit_not(self, arg)
260
261
262 class Let(Expression):
263     """
264     This class represents a let expression. The semantics of a let expression,
265     such as "let v <- e0 in e1" on an environment env is as follows:
266     1. Evaluate e0 in the environment env, yielding e0_val
267     2. Evaluate e1 in the new environment env' = env + {v:e0_val}
268     """
269
270     def __init__(self, identifier, exp_def, exp_body):
271         self.identifier = identifier
272         self.exp_def = exp_def
273         self.exp_body = exp_body
274
275     def accept(self, visitor, arg):
276         """
277         We don't have bindings at this point. So, nothing to be done here, for
278         this exercise.
279         """
280         return visitor.visit_let(self, arg)
```

Visitor.py

```

1  import sys
2  from abc import ABC, abstractmethod
3  from Expression import *
4  import Asm as AsmModule
5
6
7  class Visitor(ABC):
8      """
9      The visitor pattern consists of two abstract classes: the Expression and the
10     Visitor. The Expression class defines on method: 'accept(visitor, args)'.
11     This method takes in an implementation of a visitor, and the arguments that
12     are passed from expression to expression. The Visitor class defines one
13     specific method for each subclass of Expression. Each instance of such a
14     subclasse will invoke the right visiting method.
15     """
16
17     @abstractmethod
18     def visit_var(self, exp, arg):
19         pass
20
21     @abstractmethod
22     def visit_bln(self, exp, arg):
23         pass
24
25     @abstractmethod
26     def visit_num(self, exp, arg):
27         pass
28
29     @abstractmethod
30     def visit_eq1(self, exp, arg):
31         pass
32
33     @abstractmethod
34     def visit_add(self, exp, arg):
35         pass
36
37     @abstractmethod
38     def visit_sub(self, exp, arg):
39         pass
40
41     @abstractmethod
42     def visit_mul(self, exp, arg):
43         pass
44
45     @abstractmethod
46     def visit_div(self, exp, arg):
47         pass
48
49     @abstractmethod
50     def visit_leq(self, exp, arg):
51         pass
52
53     @abstractmethod
54     def visit_lth(self, exp, arg):
55         pass
56
57     @abstractmethod
58     def visit_neg(self, exp, arg):
59         pass
60
61     @abstractmethod
62     def visit_not(self, exp, arg):
63         pass
64
65     @abstractmethod
66     def visit_let(self, exp, arg):
67         pass
68
69
70     class GenVisitor(Visitor):
71         """
72         The GenVisitor class compiles arithmetic expressions into a low-level
73         language.
74         """
75
76         def __init__(self):
77             self.next_var_counter = 0
78
79         def next_var_name(self):
80             self.next_var_counter += 1
81             return f"v{self.next_var_counter}"
82
83         def visit_var(self, exp, prog):
84             """
85             Usage:
86             >>> e = Var('x')
87             >>> p = AsmModule.Program({"x":1}, [])
88             >>> g = GenVisitor()
89             >>> v = e.accept(g, p)
90             >>> p.eval()
91             >>> p.get_val(v)
92             1
93             """
94             return exp.identifier
95
96         def visit_bln(self, exp, env):
97             """
98             Usage:
99             >>> e = Bln(True)
100            >>> p = AsmModule.Program({}, [])
101            >>> g = GenVisitor()
102            >>> v = e.accept(g, p)
103            >>> p.eval()

```

```

104         >>> p.get_val(v)
105         1
106
107         >>> e = Bln(False)
108         >>> p = AsmModule.Program({}, [])
109         >>> g = GenVisitor()
110         >>> v = e.accept(g, p)
111         >>> p.eval()
112         >>> p.get_val(v)
113         0
114     """
115     # TODO: Implement this method.
116     raise NotImplementedError
117
118     def visit_num(self, exp, prog):
119         """
120         Usage:
121         >>> e = Num(13)
122         >>> p = AsmModule.Program({}, [])
123         >>> g = GenVisitor()
124         >>> v = e.accept(g, p)
125         >>> p.eval()
126         >>> p.get_val(v)
127         13
128     """
129     # TODO: Implement this method.
130     raise NotImplementedError
131
132     def visit_eql(self, exp, prog):
133         """
134         >>> e = Eql(Num(13), Num(13))
135         >>> p = AsmModule.Program({}, [])
136         >>> g = GenVisitor()
137         >>> v = e.accept(g, p)
138         >>> p.eval()
139         >>> p.get_val(v)
140         1
141
142         >>> e = Eql(Num(13), Num(10))
143         >>> p = AsmModule.Program({}, [])
144         >>> g = GenVisitor()
145         >>> v = e.accept(g, p)
146         >>> p.eval()
147         >>> p.get_val(v)
148         0
149
150         >>> e = Eql(Num(-1), Num(1))
151         >>> p = AsmModule.Program({}, [])
152         >>> g = GenVisitor()
153         >>> v = e.accept(g, p)
154         >>> p.eval()
155         >>> p.get_val(v)
156         0
157     """
158     # TODO: Implement this method.
159     raise NotImplementedError
160
161     def visit_add(self, exp, prog):
162         """
163         >>> e = Add(Num(13), Num(-13))
164         >>> p = AsmModule.Program({}, [])
165         >>> g = GenVisitor()
166         >>> v = e.accept(g, p)
167         >>> p.eval()
168         >>> p.get_val(v)
169         0
170
171         >>> e = Add(Num(13), Num(10))
172         >>> p = AsmModule.Program({}, [])
173         >>> g = GenVisitor()
174         >>> v = e.accept(g, p)
175         >>> p.eval()
176         >>> p.get_val(v)
177         23
178     """
179     # TODO: Implement this method (see the example in the lab's page).
180     raise NotImplementedError
181
182     def visit_sub(self, exp, prog):
183         """
184         >>> e = Sub(Num(13), Num(-13))
185         >>> p = AsmModule.Program({}, [])
186         >>> g = GenVisitor()
187         >>> v = e.accept(g, p)
188         >>> p.eval()
189         >>> p.get_val(v)
190         26
191
192         >>> e = Sub(Num(13), Num(10))
193         >>> p = AsmModule.Program({}, [])
194         >>> g = GenVisitor()
195         >>> v = e.accept(g, p)
196         >>> p.eval()
197         >>> p.get_val(v)
198         3
199     """
200     # TODO: Implement this method.
201     raise NotImplementedError
202
203     def visit_mul(self, exp, prog):
204         """
205         >>> e = Mul(Num(13), Num(2))
206         >>> p = AsmModule.Program({}, [])
207         >>> p.eval()
208         >>> p.get_val(v)
209         26
210     """
211     # TODO: Implement this method.
212     raise NotImplementedError

```

```

207     >>> g = GenVisitor()
208     >>> v = e.accept(g, p)
209     >>> p.eval()
210     >>> p.get_val(v)
211     26
212
213     >>> e = Mul(Num(13), Num(10))
214     >>> p = AsmModule.Program({}, [])
215     >>> g = GenVisitor()
216     >>> v = e.accept(g, p)
217     >>> p.eval()
218     >>> p.get_val(v)
219     130
220     """
221     # TODO: Implement this method.
222     raise NotImplementedError
223
224     def visit_div(self, exp, prog):
225         """
226         >>> e = Div(Num(13), Num(2))
227         >>> p = AsmModule.Program({}, [])
228         >>> g = GenVisitor()
229         >>> v = e.accept(g, p)
230         >>> p.eval()
231         >>> p.get_val(v)
232         6
233
234         >>> e = Div(Num(13), Num(10))
235         >>> p = AsmModule.Program({}, [])
236         >>> g = GenVisitor()
237         >>> v = e.accept(g, p)
238         >>> p.eval()
239         >>> p.get_val(v)
240         1
241         """
242         # TODO: Implement this method.
243         raise NotImplementedError
244
245     def visit_leq(self, exp, prog):
246         """
247         >>> e = Leq(Num(3), Num(2))
248         >>> p = AsmModule.Program({}, [])
249         >>> g = GenVisitor()
250         >>> v = e.accept(g, p)
251         >>> p.eval()
252         >>> p.get_val(v)
253         0
254
255         >>> e = Leq(Num(3), Num(3))
256         >>> p = AsmModule.Program({}, [])
257         >>> g = GenVisitor()
258         >>> v = e.accept(g, p)
259         >>> p.eval()
260         >>> p.get_val(v)
261         1
262
263         >>> e = Leq(Num(2), Num(3))
264         >>> p = AsmModule.Program({}, [])
265         >>> g = GenVisitor()
266         >>> v = e.accept(g, p)
267         >>> p.eval()
268         >>> p.get_val(v)
269         1
270
271         >>> e = Leq(Num(-3), Num(-2))
272         >>> p = AsmModule.Program({}, [])
273         >>> g = GenVisitor()
274         >>> v = e.accept(g, p)
275         >>> p.eval()
276         >>> p.get_val(v)
277         1
278
279         >>> e = Leq(Num(-3), Num(-3))
280         >>> p = AsmModule.Program({}, [])
281         >>> g = GenVisitor()
282         >>> v = e.accept(g, p)
283         >>> p.eval()
284         >>> p.get_val(v)
285         1
286
287         >>> e = Leq(Num(-2), Num(-3))
288         >>> p = AsmModule.Program({}, [])
289         >>> g = GenVisitor()
290         >>> v = e.accept(g, p)
291         >>> p.eval()
292         >>> p.get_val(v)
293         0
294         """
295         # TODO: Implement this method.
296         raise NotImplementedError
297
298     def visit_lth(self, exp, prog):
299         """
300         >>> e = Lth(Num(3), Num(2))
301         >>> p = AsmModule.Program({}, [])
302         >>> g = GenVisitor()
303         >>> v = e.accept(g, p)
304         >>> p.eval()
305         >>> p.get_val(v)
306         0
307
308         >>> e = Lth(Num(3), Num(3))
309         >>> p = AsmModule.Program({}, [])
310         >>> g = GenVisitor()

```

```

310         g = GenVisitor()
311         v = e.accept(g, p)
312         p.eval()
313         p.get_val(v)
314         0
315
316         >>> e = Lth(Num(2), Num(3))
317         >>> p = AsmModule.Program({}, [])
318         >>> g = GenVisitor()
319         >>> v = e.accept(g, p)
320         >>> p.eval()
321         >>> p.get_val(v)
322         1
323         """
324         # TODO: Implement this method.
325         raise NotImplementedError
326
327     def visit_neg(self, exp, prog):
328         """
329         >>> e = Neg(Num(3))
330         >>> p = AsmModule.Program({}, [])
331         >>> g = GenVisitor()
332         >>> v = e.accept(g, p)
333         >>> p.eval()
334         >>> p.get_val(v)
335         -3
336
337         >>> e = Neg(Num(0))
338         >>> p = AsmModule.Program({}, [])
339         >>> g = GenVisitor()
340         >>> v = e.accept(g, p)
341         >>> p.eval()
342         >>> p.get_val(v)
343         0
344
345         >>> e = Neg(Num(-3))
346         >>> p = AsmModule.Program({}, [])
347         >>> g = GenVisitor()
348         >>> v = e.accept(g, p)
349         >>> p.eval()
350         >>> p.get_val(v)
351         3
352         """
353         # TODO: Implement this method.
354         raise NotImplementedError
355
356     def visit_not(self, exp, prog):
357         """
358         >>> e = Not(Bln(True))
359         >>> p = AsmModule.Program({}, [])
360         >>> g = GenVisitor()
361         >>> v = e.accept(g, p)
362         >>> p.eval()
363         >>> p.get_val(v)
364         0
365
366         >>> e = Not(Bln(False))
367         >>> p = AsmModule.Program({}, [])
368         >>> g = GenVisitor()
369         >>> v = e.accept(g, p)
370         >>> p.eval()
371         >>> p.get_val(v)
372         1
373
374         >>> e = Not(Num(0))
375         >>> p = AsmModule.Program({}, [])
376         >>> g = GenVisitor()
377         >>> v = e.accept(g, p)
378         >>> p.eval()
379         >>> p.get_val(v)
380         1
381
382         >>> e = Not(Num(-2))
383         >>> p = AsmModule.Program({}, [])
384         >>> g = GenVisitor()
385         >>> v = e.accept(g, p)
386         >>> p.eval()
387         >>> p.get_val(v)
388         0
389
390         >>> e = Not(Num(2))
391         >>> p = AsmModule.Program({}, [])
392         >>> g = GenVisitor()
393         >>> v = e.accept(g, p)
394         >>> p.eval()
395         >>> p.get_val(v)
396         0
397         """
398         # TODO: Implement this method.
399         raise NotImplementedError
400
401     def visit_let(self, exp, prog):
402         """
403         Usage:
404         >>> e = Let('v', Not(Bln(False)), Var('v'))
405         >>> p = AsmModule.Program({}, [])
406         >>> g = GenVisitor()
407         >>> v = e.accept(g, p)
408         >>> p.eval()
409         >>> p.get_val(v)
410         1
411
412         >>> e = Let('v', Num(2), Add(Var('v'), Num(3)))
413         >>> n = AsmModule.Program({}, [])

```

```
414     >>> g = GenVisitor()
415     >>> v = e.accept(g, p)
416     >>> p.eval()
417     >>> p.get_val(v)
418     5
419
420     >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
421     >>> e1 = Let('y', e0, Mul(Var('y'), Num(10)))
422     >>> p = AsmModule.Program({}, [])
423     >>> g = GenVisitor()
424     >>> v = e1.accept(g, p)
425     >>> p.eval()
426     >>> p.get_val(v)
427     50
428     """
429     # TODO: Implement this method.
430     raise NotImplementedError
```

Asm.py

```

1 """
2 This file contains the implementation of a simple interpreter of low-level
3 instructions. The interpreter takes a program, represented as an array of
4 instructions, plus an environment, which is a map that associates variables with
5 values. The following instructions are recognized:
6
7 * add rd, rs1, rs2: rd = rs1 + rs2
8 * addi rd, rs1, imm: rd = rs1 + imm
9 * mul rd, rs1, rs2: rd = rs1 * rs2
10 * sub rd, rs1, rs2: rd = rs1 - rs2
11 * xor rd, rs1, rs2: rd = rs1 ^ rs2
12 * xori rd, rs1, imm: rd = rs1 ^ imm
13 * div rd, rs1, rs2: rd = rs1 // rs2 (signed integer division)
14 * slt rd, rs1, rs2: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)
15 * slti rd, rs1, imm: rd = (rs1 < imm) ? 1 : 0
16
17 This file uses doctests all over. To test it, just run python 3 as follows:
18 "python3 -m doctest Asm.py". The program uses syntax that is exclusive of
19 Python 3. It will not work with standard Python 2.
20 """
21
22 import sys
23 from collections import deque
24 from abc import ABC, abstractmethod
25
26
27 class Program:
28     """
29     The 'Program' is a list of instructions plus an environment that associates
30     names with values, plus a program counter, which marks the next instruction
31     that must be executed. The environment contains a special variable x0,
32     which always contains the value zero.
33     """
34
35     def __init__(self, env, insts):
36         self.__env = env
37         self.__insts = insts
38         self.pc = 0
39         self.__env["x0"] = 0
40
41     def get_inst(self):
42         if self.pc >= 0 and self.pc < len(self.__insts):
43             inst = self.__insts[self.pc]
44             self.pc += 1
45             return inst
46         else:
47             return None
48
49     def add_inst(self, inst):
50         self.__insts.append(inst)
51
52     def set_pc(self, pc):
53         self.pc = pc
54
55     def set_val(self, name, value):
56         self.__env[name] = value
57
58     def get_val(self, name):
59         """
60         The register x0 always contains the value zero:
61
62         >>> p = Program({}, [])
63         >>> p.get_val("x0")
64         0
65         """
66         if name in self.__env:
67             return self.__env[name]
68         else:
69             sys.exit("Def error")
70
71     def print_env(self):
72         for name, val in sorted(self.__env.items()):
73             print(f"{name}: {val}")
74
75     def print_insts(self):
76         for inst in self.__insts:
77             print(inst)
78
79     def eval(self):
80         """
81         This function evaluates a program until there is no more instructions to
82         evaluate.
83
84         Example:
85         >>> insts = [Add("x0", "b0", "b1"), Sub("x1", "x0", "b2")]
86         >>> p = Program({"b0":2, "b1":3, "b2": 4}, insts)
87         >>> p.eval()
88         >>> p.print_env()
89         b0: 2
90         b1: 3
91         b2: 4
92         x0: 5
93         x1: 1
94         """
95         inst = self.get_inst()
96         while inst:
97             inst.eval(self)
98             inst = self.get_inst()
99
100
101     def max(a, b):
102         """
103         This example computes the maximum between a and b.

```



```

104
105 Example:
106 >>> max(2, 3)
107 3
108
109 >>> max(3, 2)
110 3
111
112 >>> max(-3, -2)
113 -2
114
115 >>> max(-2, -3)
116 -2
117 """
118 p = Program({}, [])
119 p.set_val("rs1", a)
120 p.set_val("rs2", b)
121 p.add_inst(Slt("t0", "rs2", "rs1"))
122 p.add_inst(Slt("t1", "rs1", "rs2"))
123 p.add_inst(Mul("t0", "t0", "rs1"))
124 p.add_inst(Mul("t1", "t1", "rs2"))
125 p.add_inst(Add("rd", "t0", "t1"))
126 p.eval()
127 return p.get_val("rd")
128
129
130 def distance_with_acceleration(V, A, T):
131     """
132     This example computes the position of an object, given its velocity (V),
133     its acceleration (A) and the time (T), assuming that it starts at position
134     zero, using the formula  $D = V \cdot T + (A \cdot T^2) / 2$ .
135
136     Example:
137     >>> distance_with_acceleration(3, 4, 5)
138     65
139     """
140     p = Program({}, [])
141     p.set_val("rs1", V)
142     p.set_val("rs2", A)
143     p.set_val("rs3", T)
144     p.add_inst(Addi("two", "x0", 2))
145     p.add_inst(Mul("t0", "rs1", "rs3"))
146     p.add_inst(Mul("t1", "rs3", "rs3"))
147     p.add_inst(Mul("t2", "rs2", "t1"))
148     p.add_inst(Div("t2", "t2", "two"))
149     p.add_inst(Add("rd", "t0", "t2"))
150     p.eval()
151     return p.get_val("rd")
152
153
154 class Inst(ABC):
155     """
156     The representation of instructions. Every instruction refers to a program
157     during its evaluation.
158     """
159
160     def __init__(self):
161         pass
162
163     @abstractmethod
164     def get_opcode(self):
165         raise NotImplementedError
166
167     @abstractmethod
168     def eval(self, prog):
169         raise NotImplementedError
170
171
172 class BinOp(Inst):
173     """
174     The general class of binary instructions. These instructions define a
175     value, and use two values.
176     """
177
178     def __init__(self, rd, rs1, rs2):
179         assert isinstance(rd, str) and isinstance(rs1, str) and isinstance(rs2, str)
180         self.rd = rd
181         self.rs1 = rs1
182         self.rs2 = rs2
183
184     def __str__(self):
185         op = self.get_opcode()
186         return f"{self.rd} = {op} {self.rs1} {self.rs2}"
187
188
189 class BinOpImm(Inst):
190     """
191     The general class of binary instructions where the second operand is an
192     integer constant. These instructions define a value, and use one variable
193     and one immediate constant.
194     """
195
196     def __init__(self, rd, rs1, imm):
197         assert isinstance(rd, str) and isinstance(rs1, str) and isinstance(imm, int)
198         self.rd = rd
199         self.rs1 = rs1
200         self.imm = imm
201
202     def __str__(self):
203         op = self.get_opcode()
204         return f"{self.rd} = {op} {self.rs1} {self.imm}"
205
206

```

```

207 class Add(BinOp):
208     """
209     add rd, rs1, rs2: rd = rs1 + rs2
210
211     Example:
212     >>> i = Add("a", "b0", "b1")
213     >>> str(i)
214     'a = add b0 b1'
215
216     >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
217     >>> p.eval()
218     >>> p.get_val("a")
219     5
220     """
221
222     def eval(self, prog):
223         rs1 = prog.get_val(self.rs1)
224         rs2 = prog.get_val(self.rs2)
225         prog.set_val(self.rd, rs1 + rs2)
226
227     def get_opcode(self):
228         return "add"
229
230
231 class Addi(BinOpImm):
232     """
233     addi rd, rs1, imm: rd = rs1 + imm
234
235     Example:
236     >>> i = Addi("a", "b0", 1)
237     >>> str(i)
238     'a = addi b0 1'
239
240     >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
241     >>> p.eval()
242     >>> p.get_val("a")
243     5
244     """
245
246     def eval(self, prog):
247         rs1 = prog.get_val(self.rs1)
248         prog.set_val(self.rd, rs1 + self.imm)
249
250     def get_opcode(self):
251         return "addi"
252
253
254 class Mul(BinOp):
255     """
256     mul rd, rs1, rs2: rd = rs1 * rs2
257
258     Example:
259     >>> i = Mul("a", "b0", "b1")
260     >>> str(i)
261     'a = mul b0 b1'
262
263     >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
264     >>> p.eval()
265     >>> p.get_val("a")
266     6
267     """
268
269     def eval(self, prog):
270         rs1 = prog.get_val(self.rs1)
271         rs2 = prog.get_val(self.rs2)
272         prog.set_val(self.rd, rs1 * rs2)
273
274     def get_opcode(self):
275         return "mul"
276
277
278 class Sub(BinOp):
279     """
280     sub rd, rs1, rs2: rd = rs1 - rs2
281
282     Example:
283     >>> i = Sub("a", "b0", "b1")
284     >>> str(i)
285     'a = sub b0 b1'
286
287     >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
288     >>> p.eval()
289     >>> p.get_val("a")
290     -1
291     """
292
293     def eval(self, prog):
294         rs1 = prog.get_val(self.rs1)
295         rs2 = prog.get_val(self.rs2)
296         prog.set_val(self.rd, rs1 - rs2)
297
298     def get_opcode(self):
299         return "sub"
300
301
302 class Xor(BinOp):
303     """
304     xor rd, rs1, rs2: rd = rs1 ^ rs2
305
306     Example:
307     >>> i = Xor("a", "b0", "b1")
308     >>> str(i)
309     'a = xor b0 b1'
310

```

```

311         >>> p = Program(env={"b0":2, "b1":3}, insts=[Xor("a", "b0", "b1")])
312         >>> p.eval()
313         >>> p.get_val("a")
314         1
315     """
316
317     def eval(self, prog):
318         rs1 = prog.get_val(self.rs1)
319         rs2 = prog.get_val(self.rs2)
320         prog.set_val(self.rd, rs1 ^ rs2)
321
322     def get_opcode(self):
323         return "xor"
324
325
326 class Xori(BinOpImm):
327     """
328     xori rd, rs1, imm: rd = rs1 ^ imm
329
330     Example:
331     >>> i = Xori("a", "b0", 10)
332     >>> str(i)
333     'a = xori b0 10'
334
335     >>> p = Program(env={"b0":2}, insts=[Xori("a", "b0", 3)])
336     >>> p.eval()
337     >>> p.get_val("a")
338     1
339     """
340
341     def eval(self, prog):
342         rs1 = prog.get_val(self.rs1)
343         prog.set_val(self.rd, rs1 ^ self.imm)
344
345     def get_opcode(self):
346         return "xori"
347
348
349 class Div(BinOp):
350     """
351     div rd, rs1, rs2: rd = rs1 // rs2 (signed integer division)
352     Notice that RISC-V does not have an instruction exactly like this one.
353     The div operator works on floating-point numbers; not on integers.
354
355     Example:
356     >>> i = Div("a", "b0", "b1")
357     >>> str(i)
358     'a = div b0 b1'
359
360     >>> p = Program(env={"b0":8, "b1":3}, insts=[Div("a", "b0", "b1")])
361     >>> p.eval()
362     >>> p.get_val("a")
363     2
364     """
365
366     def eval(self, prog):
367         rs1 = prog.get_val(self.rs1)
368         rs2 = prog.get_val(self.rs2)
369         prog.set_val(self.rd, rs1 // rs2)
370
371     def get_opcode(self):
372         return "div"
373
374
375 class Slt(BinOp):
376     """
377     slt rd, rs1, rs2: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)
378
379     Example:
380     >>> i = Slt("a", "b0", "b1")
381     >>> str(i)
382     'a = slt b0 b1'
383
384     >>> p = Program(env={"b0":2, "b1":3}, insts=[Slt("a", "b0", "b1")])
385     >>> p.eval()
386     >>> p.get_val("a")
387     1
388
389     >>> p = Program(env={"b0":3, "b1":3}, insts=[Slt("a", "b0", "b1")])
390     >>> p.eval()
391     >>> p.get_val("a")
392     0
393
394     >>> p = Program(env={"b0":3, "b1":2}, insts=[Slt("a", "b0", "b1")])
395     >>> p.eval()
396     >>> p.get_val("a")
397     0
398     """
399
400     def eval(self, prog):
401         rs1 = prog.get_val(self.rs1)
402         rs2 = prog.get_val(self.rs2)
403         prog.set_val(self.rd, 1 if rs1 < rs2 else 0)
404
405     def get_opcode(self):
406         return "slt"
407
408
409 class Slti(BinOpImm):
410     """
411     slti rd, rs1, imm: rd = (rs1 < imm) ? 1 : 0
412     (signed comparison with immediate)
413

```

```
414 Example:
415 >>> i = Slti("a", "b0", 0)
416 >>> str(i)
417 'a = slti b0 0'
418
419 >>> p = Program(env={"b0":2}, insts=[Slti("a", "b0", 3)])
420 >>> p.eval()
421 >>> p.get_val("a")
422 1
423
424 >>> p = Program(env={"b0":3}, insts=[Slti("a", "b0", 3)])
425 >>> p.eval()
426 >>> p.get_val("a")
427 0
428
429 >>> p = Program(env={"b0":3}, insts=[Slti("a", "b0", 2)])
430 >>> p.eval()
431 >>> p.get_val("a")
432 0
433 """
434
435 def eval(self, prog):
436     rs1 = prog.get_val(self.rs1)
437     prog.set_val(self.rd, 1 if rs1 < self.imm else 0)
438
439 def get_opcode(self):
440     return "slti"
```

[VPL](#)