

2025_2 - COMPILADORES - METATURMA

PAINEL > **MINHAS TURMAS** > **2025_2 - COMPILADORES - METATURMA** > **LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL**
 > **AV12 - RENOMEAÇÃO DE VARIÁVEIS**

[Descrição](#)

[Visualizar envios](#)

AV12 - Renomeação de variáveis

Data de entrega: quarta, 5 Nov 2025, 23:59

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

Tipo de trabalho: Trabalho individual

No último laboratório de programação, foi desenvolvido um gerador de código para um subconjunto de instruções [RISC-V](#). Aquele exercício possuía uma restrição importante: assumia-se 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 era válido, uma vez que a variável `x` é definida duas vezes. Esta restrição, contudo, é artificial, pois o programa mostrado é perfeitamente válido quando interpretado. Seu valor, neste caso, é 3. Neste exercício iremos remover tal restrição, permitindo que o mesmo nome de variável seja reutilizado entre variáveis diferentes. Assim, as igualdades a seguir devem ser válidas:

```
let v <- 1 in let v <- 2 in v end + v end = 3
let v <- 1 in let v <- 2 in let v <- 3 in v end + v end + v end = 6
```

Para tanto, você deve criar variáveis com diferentes nomes quando gerar código para os diferentes programas. Por exemplo, o primeiro programa acima (`let v <- 1 in let v <- 2 in v end + v end`) deveria levar a alguma sequência de instruções como abaixo:

```
v_0 = addi x0 1          -- Define v_0 com o valor 1
v_1 = addi x0 2          -- Define v_1 com o valor 2
v3 = add v_1 v_0         -- Define v3 como a soma de v_0 e v_1
```

Essas três instruções correspondem ao programa `let v_0 <- 1 in let v_1 <- 2 in v_1 end + v_0 end`, que é equivalente ao programa original. Uma forma de permitir o reuso de nomes de variáveis é renomear as variáveis definidas em uma expressão, antes da geração de código. Com tal objetivo, você deverá implementar uma classe `RenameVisitor`, que modifica uma expressão ([in place](#)) para garantir que todos os blocos `let` criem variáveis com nomes diferentes. Note que as modificações devem ocorrer "*in place*", isto é, sobre a expressão original. Os métodos `visit`, neste caso, não precisam retornar nada. Como exemplo, a implementação abaixo deveria ser válida:

```
def visit.eql(self, exp, name_map):
    exp.left.accept(self, name_map)
    exp.right.accept(self, name_map)
```

A modificação *in place* é necessária não somente porque ela é mais eficiente, mas também porque a própria interface de uso do `driver` requer essa abordagem. O código que renomeia variáveis é invocado como abaixo:

```
def rename_variables(exp):
    ren = RenameVisitor()
    exp.accept(ren, {})
    return exp
```

Uma curiosidade sobre este exercício é que ele implementa uma forma de [atribuição estática única](#) sobre [árvores de sintaxe abstrata](#) (AST). O formato SSA (*Static Single Assignment*) é normalmente usado sobre a representação de baixo nível de programas (o [grafo de fluxo de controle](#), por exemplo). Mas, neste exercício estamos obtendo a propriedade da definição única sobre a árvore de sintaxe abstrata.

Submetendo e Testando

Este VPL deve ser construído sobre o VPL 11. A única extensão sobre aquele exercício é a implementação de `RenameVisitor`. 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 def rename_variables(exp):
9     """
10     Esta função invoca o renomeador de variáveis. Ela deve ser usada antes do
11     inicio da fase de geração de código.
12     """
13     ren = RenameVisitor()
14     exp.accept(ren, {})
15     return exp
16
17 if __name__ == "__main__":
18     """
19     Este arquivo não deve ser alterado, mas deve ser enviado para resolver o
20     VPL. O arquivo contém o código que testa a implementação do parser.
21     """
22     text = sys.stdin.read()
23     lexer = Lexer(text)
24     parser = Parser(lexer.tokens())
25     exp = rename_variables(parser.parse())
26     prog = AsmModule.Program({}, [])
27     gen = GenVisitor()
28     var_answer = exp.accept(gen, prog)
29     prog.eval()
30     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         """
55             The constructor of the lexer. It receives the string that shall be
56             scanned.
57             TODO: You will need to implement this method.
58             """
59
60         pass
61
62     def tokens(self):
63
64         """
65             This method is a token generator: it converts the string encapsulated
66             into this object into a sequence of Tokens. Examples:
67
68             >>> l = Lexer("1 + 3")
69             >>> [tk.kind for tk in l.tokens()]
70             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]
71
72             >>> l = Lexer('1 * 2 -- 3\n')
73             >>> [tk.kind for tk in l.tokens()]
74             [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
75
76             >>> l = Lexer("1 + var")
77             >>> [tk.kind for tk in l.tokens()]
78             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]
79
80             >>> l = Lexer("let v <- 2 in v end")
81             >>> [tk.kind.name for tk in l.tokens()]
82             ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
83
84             token = self.getToken()
85             while token.kind != TokenType.EOF:
86                 if (
87                     token.kind != TokenType.WSP
88                     and token.kind != TokenType.COM
89                     and token.kind != TokenType.NLN
90                 ):
91                     yield token
92                 token = self.getToken()
93
94     def getToken(self):
95
96         """
97             Return the next token.
98             TODO: Implement this method (you can reuse Lab 5: Visitors)!
99             """
100            token = None
101            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             >>> exp = parser.parse()
31             >>> exp.eval(None)
32             123
33
34             >>> parser = Parser([Token('True', TokenType.TRU)])
35             >>> exp = parser.parse()
36             >>> exp.eval(None)
37             True
38
39             >>> parser = Parser([Token('False', TokenType.FLS)])
40             >>> exp = parser.parse()
41             >>> exp.eval(None)
42             False
43
44             >>> tk0 = Token('~', TokenType.NEG)
45             >>> tk1 = Token('123', TokenType.NUM)
46             >>> parser = Parser([tk0, tk1])
47             >>> exp = parser.parse()
48             >>> exp.eval(None)
49             -123
50
51             >>> tk0 = Token('3', TokenType.NUM)
52             >>> tk1 = Token('*', TokenType.MUL)
53             >>> tk2 = Token('4', TokenType.NUM)
54             >>> parser = Parser([tk0, tk1, tk2])
55             >>> exp = parser.parse()
56             >>> exp.eval(None)
57             12
58
59             >>> tk0 = Token('3', TokenType.NUM)
60             >>> tk1 = Token('*', TokenType.MUL)
61             >>> tk2 = Token('~', TokenType.NEG)
62             >>> tk3 = Token('4', TokenType.NUM)
63             >>> parser = Parser([tk0, tk1, tk2, tk3])
64             >>> exp = parser.parse()
65             >>> exp.eval(None)
66             -12
67
68             >>> tk0 = Token('30', TokenType.NUM)
69             >>> tk1 = Token('/', TokenType.DIV)
70             >>> tk2 = Token('4', TokenType.NUM)
71             >>> parser = Parser([tk0, tk1, tk2])
72             >>> exp = parser.parse()
73             >>> exp.eval(None)
74             7
75
76             >>> tk0 = Token('3', TokenType.NUM)
77             >>> tk1 = Token('+', TokenType.ADD)
78             >>> tk2 = Token('4', TokenType.NUM)
79             >>> parser = Parser([tk0, tk1, tk2])
80             >>> exp = parser.parse()
81             >>> exp.eval(None)
82             7
83
84             >>> tk0 = Token('30', TokenType.NUM)
85             >>> tk1 = Token('-', TokenType.SUB)
86             >>> tk2 = Token('4', TokenType.NUM)
87             >>> parser = Parser([tk0, tk1, tk2])
88             >>> exp = parser.parse()
89             >>> exp.eval(None)
90             26
91
92             >>> tk0 = Token('2', TokenType.NUM)
93             >>> tk1 = Token('*', TokenType.MUL)
94             >>> tk2 = Token('(', TokenType.LPR)
95             >>> tk3 = Token('3', TokenType.NUM)
96             >>> tk4 = Token('+', TokenType.ADD)
97             >>> tk5 = Token('4', TokenType.NUM)
98             >>> tk6 = Token(')', TokenType.RPR)
99             >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
100            >>> exp = parser.parse()
101            >>> exp.eval(None)
102            14
103

```

```

104     >>> tk0 = Token('4', TokenType.NUM)
105     >>> tk1 = Token('==', TokenType.EQL)
106     >>> tk2 = Token('4', TokenType.NUM)
107     >>> parser = Parser([tk0, tk1, tk2])
108     >>> exp = parser.parse()
109     >>> exp.eval(None)
110     True
111
112     >>> tk0 = Token('4', TokenType.NUM)
113     >>> tk1 = Token('<=', TokenType.LEQ)
114     >>> tk2 = Token('4', TokenType.NUM)
115     >>> parser = Parser([tk0, tk1, tk2])
116     >>> exp = parser.parse()
117     >>> exp.eval(None)
118     True
119
120     >>> tk0 = Token('4', TokenType.NUM)
121     >>> tk1 = Token('<', TokenType.LTH)
122     >>> tk2 = Token('4', TokenType.NUM)
123     >>> parser = Parser([tk0, tk1, tk2])
124     >>> exp = parser.parse()
125     >>> exp.eval(None)
126     False
127
128     >>> tk0 = Token('not', TokenType.NOT)
129     >>> tk1 = Token('4', TokenType.NUM)
130     >>> tk2 = Token('<', TokenType.LTH)
131     >>> tk3 = Token('4', TokenType.NUM)
132     >>> parser = Parser([tk0, tk1, tk2, tk3])
133     >>> exp = parser.parse()
134     >>> exp.eval(None)
135     True
136
137     >>> tk0 = Token('let', TokenType.LET)
138     >>> tk1 = Token('v', TokenType.VAR)
139     >>> tk2 = Token('<-', TokenType.ASN)
140     >>> tk3 = Token('42', TokenType.NUM)
141     >>> tk4 = Token('in', TokenType.INX)
142     >>> tk5 = Token('v', TokenType.VAR)
143     >>> tk6 = Token('end', TokenType.END)
144     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
145     >>> exp = parser.parse()
146     >>> exp.eval({})
147     42
148
149     >>> tk0 = Token('let', TokenType.LET)
150     >>> tk1 = Token('v', TokenType.VAR)
151     >>> tk2 = Token('<-', TokenType.ASN)
152     >>> tk3 = Token('21', TokenType.NUM)
153     >>> tk4 = Token('in', TokenType.INX)
154     >>> tk5 = Token('v', TokenType.VAR)
155     >>> tk6 = Token('+', TokenType.ADD)
156     >>> tk7 = Token('v', TokenType.VAR)
157     >>> tk8 = Token('end', TokenType.END)
158     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
159     >>> exp = parser.parse()
160     >>> exp.eval({})
161     42
162     """
163     # TODO: implement this method.
164     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 acceptuation 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 acceptuation 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 Eq1(BinaryExpression):
85     """
86     This class represents the equality between two expressions. The acceptuation
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_eq1(self, arg)
96
97
98 class Add(BinaryExpression):
99     """
100    This class represents addition of two expressions. The acceptuation 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 acceptuation 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 acceptuation 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         acceptuation 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 acceptuation 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 acceptuation 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     ....
209     return visitor.visit_lth(self, arg)
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
246         return visitor.visit_neg(self, arg)
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
260         return visitor.visit_not(self, arg)
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
281         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 one 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    subclass 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_eql(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 RenameVisitor(ABC):
71 """
72     This visitor traverses the AST of a program, renaming variables to ensure
73     that they all have different names.
74
75 Usage:
76     >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
77     >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
78     >>> e0.identifier == e1.identifier
79     True
80
81     >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
82     >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
83     >>> r = RenameVisitor()
84     >>> e1.accept(r, {})
85     >>> e0.identifier == e1.identifier
86     False
87
88     >>> x0 = Var('x')
89     >>> x1 = Var('x')
90     >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
91     >>> e1 = Let('x', e0, Mul(x1, Num(10)))
92     >>> x0.identifier == x1.identifier
93     True
94
95     >>> x0 = Var('x')
96     >>> x1 = Var('x')
97     >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
98     >>> e1 = Let('x', e0, Mul(x1, Num(10)))
99     >>> r = RenameVisitor()
100    >>> e1.accept(r, {})
101    >>> x0.identifier == x1.identifier
102    False
103 """

```

```

104
105     def visit_var(self, exp, arg):
106         # TODO: Implement this method.
107         raise NotImplementedError
108
109     def visit_bln(self, exp, arg):
110         # TODO: Implement this method.
111         raise NotImplementedError
112
113     def visit_num(self, exp, arg):
114         # TODO: Implement this method.
115         raise NotImplementedError
116
117     def visit_eql(self, exp, arg):
118         # TODO: Implement this method.
119         raise NotImplementedError
120
121     def visit_add(self, exp, arg):
122         # TODO: Implement this method.
123         raise NotImplementedError
124
125     def visit_sub(self, exp, arg):
126         # TODO: Implement this method.
127         raise NotImplementedError
128
129     def visit_mul(self, exp, arg):
130         # TODO: Implement this method.
131         raise NotImplementedError
132
133     def visit_div(self, exp, arg):
134         # TODO: Implement this method.
135         raise NotImplementedError
136
137     def visit_leq(self, exp, arg):
138         # TODO: Implement this method.
139         raise NotImplementedError
140
141     def visit_lth(self, exp, arg):
142         # TODO: Implement this method.
143         raise NotImplementedError
144
145     def visit_neg(self, exp, arg):
146         # TODO: Implement this method.
147         raise NotImplementedError
148
149     def visit_not(self, exp, arg):
150         # TODO: Implement this method.
151         raise NotImplementedError
152
153     def visit_let(self, exp, arg):
154         # TODO: Implement this method.
155         raise NotImplementedError
156
157
158 class GenVisitor(Visitor):
159     """
160     The GenVisitor class compiles arithmetic expressions into a low-level
161     language.
162     """
163
164     def __init__(self):
165         self.next_var_counter = 0
166
167     def next_var_name(self):
168         self.next_var_counter += 1
169         return f"v{self.next_var_counter}"
170
171     def visit_var(self, exp, prog):
172         """
173         Usage:
174             >>> e = Var('x')
175             >>> p = AsmModule.Program({"x":1}, [])
176             >>> g = GenVisitor()
177             >>> v = e.accept(g, p)
178             >>> p.eval()
179             >>> p.get_val(v)
180             1
181         """
182         return exp.identifier
183
184     def visit_bln(self, exp, env):
185         """
186         Usage:
187             >>> e = Bln(True)
188             >>> p = AsmModule.Program({}, [])
189             >>> g = GenVisitor()
190             >>> v = e.accept(g, p)
191             >>> p.eval()
192             >>> p.get_val(v)
193             1
194
195             >>> e = Bln(False)
196             >>> p = AsmModule.Program({}, [])
197             >>> g = GenVisitor()
198             >>> v = e.accept(g, p)
199             >>> p.eval()
200             >>> p.get_val(v)
201             0
202         """
203         # TODO: Implement this method.
204         raise NotImplementedError
205
206     def visit_num(self, exp, prog):
207         """

```

```

208
209     Usage:
210         >>> e = Num(13)
211         >>> p = AsmModule.Program({}, [])
212         >>> g = GenVisitor()
213         >>> v = e.accept(g, p)
214         >>> p.eval()
215         >>> p.get_val(v)
216         13
217
218     # TODO: Implement this method.
219     raise NotImplementedError
220
221 def visit_eql(self, exp, prog):
222     """
223         >>> e = Eq1(Num(13), Num(13))
224         >>> p = AsmModule.Program({}, [])
225         >>> g = GenVisitor()
226         >>> v = e.accept(g, p)
227         >>> p.eval()
228         >>> p.get_val(v)
229         1
230
231         >>> e = Eq1(Num(13), Num(10))
232         >>> p = AsmModule.Program({}, [])
233         >>> g = GenVisitor()
234         >>> v = e.accept(g, p)
235         >>> p.eval()
236         >>> p.get_val(v)
237         0
238
239         >>> e = Eq1(Num(-1), Num(1))
240         >>> p = AsmModule.Program({}, [])
241         >>> g = GenVisitor()
242         >>> v = e.accept(g, p)
243         >>> p.eval()
244         >>> p.get_val(v)
245         0
246
247     # TODO: Implement this method.
248     raise NotImplementedError
249
250 def visit_add(self, exp, prog):
251     """
252         >>> e = Add(Num(13), Num(-13))
253         >>> p = AsmModule.Program({}, [])
254         >>> g = GenVisitor()
255         >>> v = e.accept(g, p)
256         >>> p.eval()
257         >>> p.get_val(v)
258         0
259
260         >>> e = Add(Num(13), Num(10))
261         >>> p = AsmModule.Program({}, [])
262         >>> g = GenVisitor()
263         >>> v = e.accept(g, p)
264         >>> p.eval()
265         >>> p.get_val(v)
266         23
267
268     # TODO: Implement this method (see the example in the lab's page).
269     raise NotImplementedError
270
271 def visit_sub(self, exp, prog):
272     """
273         >>> e = Sub(Num(13), Num(-13))
274         >>> p = AsmModule.Program({}, [])
275         >>> g = GenVisitor()
276         >>> v = e.accept(g, p)
277         >>> p.eval()
278         >>> p.get_val(v)
279         26
280
281         >>> e = Sub(Num(13), Num(10))
282         >>> p = AsmModule.Program({}, [])
283         >>> g = GenVisitor()
284         >>> v = e.accept(g, p)
285         >>> p.eval()
286         >>> p.get_val(v)
287         3
288
289     # TODO: Implement this method.
290     raise NotImplementedError
291
292 def visit_mul(self, exp, prog):
293     """
294         >>> e = Mul(Num(13), Num(2))
295         >>> p = AsmModule.Program({}, [])
296         >>> g = GenVisitor()
297         >>> v = e.accept(g, p)
298         >>> p.eval()
299         >>> p.get_val(v)
300         26
301
302         >>> e = Mul(Num(13), Num(10))
303         >>> p = AsmModule.Program({}, [])
304         >>> g = GenVisitor()
305         >>> v = e.accept(g, p)
306         >>> p.eval()
307         >>> p.get_val(v)
308         130
309
310     # TODO: Implement this method.
311     raise NotImplementedError

```

```

310
311     def visit_div(self, exp, prog):
312         """
313
314         >>> e = Div(Num(13), Num(2))
315         >>> p = AsmModule.Program({}, [])
316         >>> g = GenVisitor()
317         >>> v = e.accept(g, p)
318         >>> p.eval()
319         >>> p.get_val(v)
320         6
321
322         >>> e = Div(Num(13), Num(10))
323         >>> p = AsmModule.Program({}, [])
324         >>> g = GenVisitor()
325         >>> v = e.accept(g, p)
326         >>> p.eval()
327         >>> p.get_val(v)
328         1
329         """
330
331     # TODO: Implement this method.
332     raise NotImplementedError
333
334     def visit_leq(self, exp, prog):
335         """
336
337         >>> e = Leq(Num(3), Num(2))
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 = Leq(Num(3), Num(3))
346         >>> p = AsmModule.Program({}, [])
347         >>> g = GenVisitor()
348         >>> v = e.accept(g, p)
349         >>> p.eval()
350         >>> p.get_val(v)
351         1
352
353         >>> e = Leq(Num(2), Num(3))
354         >>> p = AsmModule.Program({}, [])
355         >>> g = GenVisitor()
356         >>> v = e.accept(g, p)
357         >>> p.eval()
358         >>> p.get_val(v)
359         1
360
361         >>> e = Leq(Num(-3), Num(-2))
362         >>> p = AsmModule.Program({}, [])
363         >>> g = GenVisitor()
364         >>> v = e.accept(g, p)
365         >>> p.eval()
366         >>> p.get_val(v)
367         1
368
369         >>> e = Leq(Num(-3), Num(-3))
370         >>> p = AsmModule.Program({}, [])
371         >>> g = GenVisitor()
372         >>> v = e.accept(g, p)
373         >>> p.eval()
374         >>> p.get_val(v)
375         1
376
377         >>> e = Leq(Num(-2), Num(-3))
378         >>> p = AsmModule.Program({}, [])
379         >>> g = GenVisitor()
380         >>> v = e.accept(g, p)
381         >>> p.eval()
382         >>> p.get_val(v)
383         0
384
385     # TODO: Implement this method.
386     raise NotImplementedError
387
388     def visit_lth(self, exp, prog):
389         """
390
391         >>> e = Lth(Num(3), Num(2))
392         >>> p = AsmModule.Program({}, [])
393         >>> g = GenVisitor()
394         >>> v = e.accept(g, p)
395         >>> p.eval()
396         >>> p.get_val(v)
397         0
398
399         >>> e = Lth(Num(3), Num(3))
400         >>> p = AsmModule.Program({}, [])
401         >>> g = GenVisitor()
402         >>> v = e.accept(g, p)
403         >>> p.eval()
404         >>> p.get_val(v)
405         0
406
407         >>> e = Lth(Num(2), Num(3))
408         >>> p = AsmModule.Program({}, [])
409         >>> g = GenVisitor()
410         >>> v = e.accept(g, p)
411         >>> p.eval()
412         >>> p.get_val(v)
413         1
414         """
415
416     # TODO: Implement this method.
417     raise NotImplementedError

```

```

414
415     def visit_neg(self, exp, prog):
416         """
417             >>> e = Neg(Num(3))
418             >>> p = AsmModule.Program({}, [])
419             >>> g = GenVisitor()
420             >>> v = e.accept(g, p)
421             >>> p.eval()
422             >>> p.get_val(v)
423             -3
424
425             >>> e = Neg(Num(0))
426             >>> p = AsmModule.Program({}, [])
427             >>> g = GenVisitor()
428             >>> v = e.accept(g, p)
429             >>> p.eval()
430             >>> p.get_val(v)
431             0
432
433             >>> e = Neg(Num(-3))
434             >>> p = AsmModule.Program({}, [])
435             >>> g = GenVisitor()
436             >>> v = e.accept(g, p)
437             >>> p.eval()
438             >>> p.get_val(v)
439             3
440             """
441
442     # TODO: Implement this method.
443     raise NotImplementedError
444
445     def visit_not(self, exp, prog):
446         """
447             >>> e = Not(Bln(True))
448             >>> p = AsmModule.Program({}, [])
449             >>> g = GenVisitor()
450             >>> v = e.accept(g, p)
451             >>> p.eval()
452             >>> p.get_val(v)
453             0
454
455             >>> e = Not(Bln(False))
456             >>> p = AsmModule.Program({}, [])
457             >>> g = GenVisitor()
458             >>> v = e.accept(g, p)
459             >>> p.eval()
460             >>> p.get_val(v)
461             1
462
463             >>> e = Not(Num(0))
464             >>> p = AsmModule.Program({}, [])
465             >>> g = GenVisitor()
466             >>> v = e.accept(g, p)
467             >>> p.eval()
468             >>> p.get_val(v)
469             1
470
471             >>> e = Not(Num(-2))
472             >>> p = AsmModule.Program({}, [])
473             >>> g = GenVisitor()
474             >>> v = e.accept(g, p)
475             >>> p.eval()
476             >>> p.get_val(v)
477             0
478
479             >>> e = Not(Num(2))
480             >>> p = AsmModule.Program({}, [])
481             >>> g = GenVisitor()
482             >>> v = e.accept(g, p)
483             >>> p.eval()
484             >>> p.get_val(v)
485             0
486             """
487
488     # TODO: Implement this method.
489     raise NotImplementedError
490
491     def visit_let(self, exp, prog):
492         """
493             Usage:
494                 >>> e = Let('v', Not(Bln(False)), Var('v'))
495                 >>> p = AsmModule.Program({}, [])
496                 >>> g = GenVisitor()
497                 >>> v = e.accept(g, p)
498                 >>> p.eval()
499                 >>> p.get_val(v)
500
501                 >>> e = Let('v', Num(2), Add(Var('v'), Num(3)))
502                 >>> p = AsmModule.Program({}, [])
503                 >>> g = GenVisitor()
504                 >>> v = e.accept(g, p)
505                 >>> p.eval()
506                 >>> p.get_val(v)
507
508                 >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
509                 >>> e1 = Let('y', e0, Mul(Var('y'), Num(10)))
510                 >>> p = AsmModule.Program({}, [])
511                 >>> g = GenVisitor()
512                 >>> v = e1.accept(g, p)
513                 >>> p.eval()
514                 >>> p.get_val(v)
515
516                 50
517             """

```

```
517  
518
```

```
# TODO: Implement this method.  
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            def max(a, b):
101                """
102                    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  $D = V*T + (A*T^2)/2$ .
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     """
208     
```

```

278     sub rd, rs1, rs2: rd = rs1 - rs2
279
280     Example:
281         >>> i = Sub("a", "b0", "b1")
282         >>> str(i)
283         'a = sub b0 b1'
284
285         >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
286         >>> p.eval()
287         >>> p.get_val("a")
288         -1
289
290     """
291
292     def eval(self, prog):
293         rs1 = prog.get_val(self.rs1)
294         rs2 = prog.get_val(self.rs2)
295         prog.set_val(self.rd, rs1 - rs2)
296
297     def get_opcode(self):
298         return "sub"
299
300
301 class Xor(BinOp):
302     """
303     xor rd, rs1, rs2: rd = rs1 ^ rs2
304
305     Example:
306         >>> i = Xor("a", "b0", "b1")
307         >>> str(i)
308         'a = xor b0 b1'
309
310     """
311
312     def eval(self, prog):
313         rs1 = prog.get_val(self.rs1)
314         rs2 = prog.get_val(self.rs2)
315         prog.set_val(self.rd, rs1 ^ rs2)
316
317     def get_opcode(self):
318         return "xor"
319
320
321 class Mul(BinOp):
322     """
323     mul rd, rs1, rs2: rd = rs1 * rs2
324
325     Example:
326         >>> i = Mul("a", "b0", "b1")
327         >>> str(i)
328         'a = mul b0 b1'
329
330         >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
331         >>> p.eval()
332         >>> p.get_val("a")
333         6
334
335     """
336
337     def eval(self, prog):
338         rs1 = prog.get_val(self.rs1)
339         rs2 = prog.get_val(self.rs2)
340         prog.set_val(self.rd, rs1 * rs2)
341
342     def get_opcode(self):
343         return "mul"
344
345
346 class Sub(BinOp):
347     """
348     sub rd, rs1, rs2: rd = rs1 - rs2
349
350     Example:
351         >>> i = Sub("a", "b0", "b1")
352         >>> str(i)
353         'a = sub b0 b1'
354
355         >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
356         >>> p.eval()
357         >>> p.get_val("a")
358         -1
359
360     """
361
362     def eval(self, prog):
363         rs1 = prog.get_val(self.rs1)
364         rs2 = prog.get_val(self.rs2)
365         prog.set_val(self.rd, rs1 - rs2)
366
367     def get_opcode(self):
368         return "sub"
369
370
371 class Xor(BinOp):
372     """
373     xor rd, rs1, rs2: rd = rs1 ^ rs2
374
375     Example:
376         >>> i = Xor("a", "b0", "b1")
377         >>> str(i)
378         'a = xor b0 b1'
379
380     """
381
382     def eval(self, prog):
383         rs1 = prog.get_val(self.rs1)
384         rs2 = prog.get_val(self.rs2)
385         prog.set_val(self.rd, rs1 ^ rs2)
386
387     def get_opcode(self):
388         return "xor"
389
390
391 class Add(BinOp):
392     """
393     add rd, rs1, rs2: rd = rs1 + rs2
394
395     Example:
396         >>> i = Add("a", "b0", "b1")
397         >>> str(i)
398         'a = add b0 b1'
399
400         >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
401         >>> p.eval()
402         >>> p.get_val("a")
403         5
404
405     """
406
407     def eval(self, prog):
408         rs1 = prog.get_val(self.rs1)
409         rs2 = prog.get_val(self.rs2)
410         prog.set_val(self.rd, rs1 + rs2)
411
412     def get_opcode(self):
413         return "add"
414
415
416 class Addi(BinOpImm):
417     """
418     addi rd, rs1, imm: rd = rs1 + imm
419
420     Example:
421         >>> i = Addi("a", "b0", 1)
422         >>> str(i)
423         'a = addi b0 1'
424
425         >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
426         >>> p.eval()
427         >>> p.get_val("a")
428         5
429
430     """
431
432     def eval(self, prog):
433         rs1 = prog.get_val(self.rs1)
434         prog.set_val(self.rd, rs1 + self.imm)
435
436     def get_opcode(self):
437         return "addi"
438
439
440 class Mul(BinOp):
441     """
442     mul rd, rs1, rs2: rd = rs1 * rs2
443
444     Example:
445         >>> i = Mul("a", "b0", "b1")
446         >>> str(i)
447         'a = mul b0 b1'
448
449         >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
450         >>> p.eval()
451         >>> p.get_val("a")
452         6
453
454     """
455
456     def eval(self, prog):
457         rs1 = prog.get_val(self.rs1)
458         rs2 = prog.get_val(self.rs2)
459         prog.set_val(self.rd, rs1 * rs2)
460
461     def get_opcode(self):
462         return "mul"
463
464
465 class Sub(BinOp):
466     """
467     sub rd, rs1, rs2: rd = rs1 - rs2
468
469     Example:
470         >>> i = Sub("a", "b0", "b1")
471         >>> str(i)
472         'a = sub b0 b1'
473
474         >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
475         >>> p.eval()
476         >>> p.get_val("a")
477         -1
478
479     """
480
481     def eval(self, prog):
482         rs1 = prog.get_val(self.rs1)
483         rs2 = prog.get_val(self.rs2)
484         prog.set_val(self.rd, rs1 - rs2)
485
486     def get_opcode(self):
487         return "sub"
488
489
490 class Xor(BinOp):
491     """
492     xor rd, rs1, rs2: rd = rs1 ^ rs2
493
494     Example:
495         >>> i = Xor("a", "b0", "b1")
496         >>> str(i)
497         'a = xor b0 b1'
498
499     """
500
501     def eval(self, prog):
502         rs1 = prog.get_val(self.rs1)
503         rs2 = prog.get_val(self.rs2)
504         prog.set_val(self.rd, rs1 ^ rs2)
505
506     def get_opcode(self):
507         return "xor"
508
509
510 class Add(BinOp):
511     """
512     add rd, rs1, rs2: rd = rs1 + rs2
513
514     Example:
515         >>> i = Add("a", "b0", "b1")
516         >>> str(i)
517         'a = add b0 b1'
518
519         >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
520         >>> p.eval()
521         >>> p.get_val("a")
522         5
523
524     """
525
526     def eval(self, prog):
527         rs1 = prog.get_val(self.rs1)
528         rs2 = prog.get_val(self.rs2)
529         prog.set_val(self.rd, rs1 + rs2)
530
531     def get_opcode(self):
532         return "add"
533
534
535 class Addi(BinOpImm):
536     """
537     addi rd, rs1, imm: rd = rs1 + imm
538
539     Example:
540         >>> i = Addi("a", "b0", 1)
541         >>> str(i)
542         'a = addi b0 1'
543
544         >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
545         >>> p.eval()
546         >>> p.get_val("a")
547         5
548
549     """
550
551     def eval(self, prog):
552         rs1 = prog.get_val(self.rs1)
553         rs2 = prog.get_val(self.rs2)
554         prog.set_val(self.rd, rs1 + rs2)
555
556     def get_opcode(self):
557         return "addi"
558
559
560 class Mul(BinOp):
561     """
562     mul rd, rs1, rs2: rd = rs1 * rs2
563
564     Example:
565         >>> i = Mul("a", "b0", "b1")
566         >>> str(i)
567         'a = mul b0 b1'
568
569         >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
570         >>> p.eval()
571         >>> p.get_val("a")
572         6
573
574     """
575
576     def eval(self, prog):
577         rs1 = prog.get_val(self.rs1)
578         rs2 = prog.get_val(self.rs2)
579         prog.set_val(self.rd, rs1 * rs2)
580
581     def get_opcode(self):
582         return "mul"
583
584
585 class Sub(BinOp):
586     """
587     sub rd, rs1, rs2: rd = rs1 - rs2
588
589     Example:
590         >>> i = Sub("a", "b0", "b1")
591         >>> str(i)
592         'a = sub b0 b1'
593
594         >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
595         >>> p.eval()
596         >>> p.get_val("a")
597         -1
598
599     """
600
601     def eval(self, prog):
602         rs1 = prog.get_val(self.rs1)
603         rs2 = prog.get_val(self.rs2)
604         prog.set_val(self.rd, rs1 - rs2)
605
606     def get_opcode(self):
607         return "sub"
608
609
610 class Xor(BinOp):
611     """
612     xor rd, rs1, rs2: rd = rs1 ^ rs2
613
614     Example:
615         >>> i = Xor("a", "b0", "b1")
616         >>> str(i)
617         'a = xor b0 b1'
618
619     """
620
621     def eval(self, prog):
622         rs1 = prog.get_val(self.rs1)
623         rs2 = prog.get_val(self.rs2)
624         prog.set_val(self.rd, rs1 ^ rs2)
625
626     def get_opcode(self):
627         return "xor"
628
629
630 class Add(BinOp):
631     """
632     add rd, rs1, rs2: rd = rs1 + rs2
633
634     Example:
635         >>> i = Add("a", "b0", "b1")
636         >>> str(i)
637         'a = add b0 b1'
638
639         >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
640         >>> p.eval()
641         >>> p.get_val("a")
642         5
643
644     """
645
646     def eval(self, prog):
647         rs1 = prog.get_val(self.rs1)
648         rs2 = prog.get_val(self.rs2)
649         prog.set_val(self.rd, rs1 + rs2)
650
651     def get_opcode(self):
652         return "add"
653
654
655 class Addi(BinOpImm):
656     """
657     addi rd, rs1, imm: rd = rs1 + imm
658
659     Example:
660         >>> i = Addi("a", "b0", 1)
661         >>> str(i)
662         'a = addi b0 1'
663
664         >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
665         >>> p.eval()
666         >>> p.get_val("a")
667         5
668
669     """
670
671     def eval(self, prog):
672         rs1 = prog.get_val(self.rs1)
673         rs2 = prog.get_val(self.rs2)
674         prog.set_val(self.rd, rs1 + rs2)
675
676     def get_opcode(self):
677         return "addi"
678
679
680 class Mul(BinOp):
681     """
682     mul rd, rs1, rs2: rd = rs1 * rs2
683
684     Example:
685         >>> i = Mul("a", "b0", "b1")
686         >>> str(i)
687         'a = mul b0 b1'
688
689         >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
690         >>> p.eval()
691         >>> p.get_val("a")
692         6
693
694     """
695
696     def eval(self, prog):
697         rs1 = prog.get_val(self.rs1)
698         rs2 = prog.get_val(self.rs2)
699         prog.set_val(self.rd, rs1 * rs2)
700
701     def get_opcode(self):
702         return "mul"
703
704
705 class Sub(BinOp):
706     """
707     sub rd, rs1, rs2: rd = rs1 - rs2
708
709     Example:
710         >>> i = Sub("a", "b0", "b1")
711         >>> str(i)
712         'a = sub b0 b1'
713
714         >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
715         >>> p.eval()
716         >>> p.get_val("a")
717         -1
718
719     """
720
721     def eval(self, prog):
722         rs1 = prog.get_val(self.rs1)
723         rs2 = prog.get_val(self.rs2)
724         prog.set_val(self.rd, rs1 - rs2)
725
726     def get_opcode(self):
727         return "sub"
728
729
730 class Xor(BinOp):
731     """
732     xor rd, rs1, rs2: rd = rs1 ^ rs2
733
734     Example:
735         >>> i = Xor("a", "b0", "b1")
736         >>> str(i)
737         'a = xor b0 b1'
738
739     """
740
741     def eval(self, prog):
742         rs1 = prog.get_val(self.rs1)
743         rs2 = prog.get_val(self.rs2)
744         prog.set_val(self.rd, rs1 ^ rs2)
745
746     def get_opcode(self):
747         return "xor"
748
749
750 class Add(BinOp):
751     """
752     add rd, rs1, rs2: rd = rs1 + rs2
753
754     Example:
755         >>> i = Add("a", "b0", "b1")
756         >>> str(i)
757         'a = add b0 b1'
758
759         >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
760         >>> p.eval()
761         >>> p.get_val("a")
762         5
763
764     """
765
766     def eval(self, prog):
767         rs1 = prog.get_val(self.rs1)
768         rs2 = prog.get_val(self.rs2)
769         prog.set_val(self.rd, rs1 + rs2)
770
771     def get_opcode(self):
772         return "add"
773
774
775 class Addi(BinOpImm):
776     """
777     addi rd, rs1, imm: rd = rs1 + imm
778
779     Example:
780         >>> i = Addi("a", "b0", 1)
781         >>> str(i)
782         'a = addi b0 1'
783
784         >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
785         >>> p.eval()
786         >>> p.get_val("a")
787         5
788
789     """
790
791     def eval(self, prog):
792         rs1 = prog.get_val(self.rs1)
793         rs2 = prog.get_val(self.rs2)
794         prog.set_val(self.rd, rs1 + rs2)
795
796     def get_opcode(self):
797         return "addi"
798
799
800 class Mul(BinOp):
801     """
802     mul rd, rs1, rs2: rd = rs1 * rs2
803
804     Example:
805         >>> i = Mul("a", "b0", "b1")
806         >>> str(i)
807         'a = mul b0 b1'
808
809         >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
810         >>> p.eval()
811         >>> p.get_val("a")
812         6
813
814     """
815
816     def eval(self, prog):
817         rs1 = prog.get_val(self.rs1)
818         rs2 = prog.get_val(self.rs2)
819         prog.set_val(self.rd, rs1 * rs2)
820
821     def get_opcode(self):
822         return "mul"
823
824
825 class Sub(BinOp):
826     """
827     sub rd, rs1, rs2: rd = rs1 - rs2
828
829     Example:
830         >>> i = Sub("a", "b0", "b1")
831         >>> str(i)
832         'a = sub b0 b1'
833
834         >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
835         >>> p.eval()
836         >>> p.get_val("a")
837         -1
838
839     """
840
841     def eval(self, prog):
842         rs1 = prog.get_val(self.rs1)
843         rs2 = prog.get_val(self.rs2)
844         prog.set_val(self.rd, rs1 - rs2)
845
846     def get_opcode(self):
847         return "sub"
848
849
850 class Xor(BinOp):
851     """
852     xor rd, rs1, rs2: rd = rs1 ^ rs2
853
854     Example:
855         >>> i = Xor("a", "b0", "b1")
856         >>> str(i)
857         'a = xor b0 b1'
858
859     """
860
861     def eval(self, prog):
862         rs1 = prog.get_val(self.rs1)
863         rs2 = prog.get_val(self.rs2)
864         prog.set_val(self.rd, rs1 ^ rs2)
865
866     def get_opcode(self):
867         return "xor"
868
869
870 class Add(BinOp):
871     """
872     add rd, rs1, rs2: rd = rs1 + rs2
873
874     Example:
875         >>> i = Add("a", "b0", "b1")
876         >>> str(i)
877         'a = add b0 b1'
878
879         >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
880         >>> p.eval()
881         >>> p.get_val("a")
882         5
883
884     """
885
886     def eval(self, prog):
887         rs1 = prog.get_val(self.rs1)
888         rs2 = prog.get_val(self.rs2)
889         prog.set_val(self.rd, rs1 + rs2)
890
891     def get_opcode(self):
892         return "add"
893
894
895 class Addi(BinOpImm):
896     """
897     addi rd, rs1, imm: rd = rs1 + imm
898
899     Example:
900         >>> i = Addi("a", "b0", 1)
901         >>> str(i)
902         'a = addi b0 1'
903
904         >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
905         >>> p.eval()
906         >>> p.get_val("a")
907         5
908
909     """
910
911     def eval(self, prog):
912         rs1 = prog.get_val(self.rs1)
913         rs2 = prog.get_val(self.rs2)
914         prog.set_val(self.rd, rs1 + rs2)
915
916     def get_opcode(self):
917         return "addi"
918
919
920 class Mul(BinOp):
921     """
922     mul rd, rs1, rs2: rd = rs1 * rs2
923
924     Example:
925         >>> i = Mul("a", "b0", "b1")
926         >>> str(i)
927         'a = mul b0 b1'
928
929         >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
930         >>> p.eval()
931         >>> p.get_val("a")
932         6
933
934     """
935
936     def eval(self, prog):
937         rs1 = prog.get_val(self.rs1)
938         rs2 = prog.get_val(self.rs2)
939         prog.set_val(self.rd, rs1 * rs2)
940
941     def get_opcode(self):
942         return "mul"
943
944
945 class Sub(BinOp):
946     """
947     sub rd, rs1, rs2: rd = rs1 - rs2
948
949     Example:
950         >>> i = Sub("a", "b0", "b1")
951         >>> str(i)
952         'a = sub b0 b1'
953
954         >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
955         >>> p.eval()
956         >>> p.get_val("a")
957         -1
958
959     """
960
961     def eval(self, prog):
962         rs1 = prog.get_val(self.rs1)
963         rs2 = prog.get_val(self.rs2)
964         prog.set_val(self.rd, rs1 - rs2)
965
966     def get_opcode(self):
967         return "sub"
968
969
970 class Xor(BinOp):
971     """
972     xor rd, rs1, rs2: rd = rs1 ^ rs2
973
974     Example:
975         >>> i = Xor("a", "b0", "b1")
976         >>> str(i)
977         'a = xor b0 b1'
978
979     """
980
981     def eval(self, prog):
982         rs1 = prog.get_val(self.rs1)
983         rs2 = prog.get_val(self.rs2)
984         prog.set_val(self.rd, rs1 ^ rs2)
985
986     def get_opcode(self):
987         return "xor"
988
989
990 class Add(BinOp):
991     """
992     add rd, rs1, rs2: rd = rs1 + rs2
993
994     Example:
995         >>> i = Add("a", "b0", "b1")
996         >>> str(i)
997         'a = add b0 b1'
998
999         >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
1000        >>> p.eval()
1001        >>> p.get_val("a")
1002        5
1003
1004    """
1005
1006    def eval(self, prog):
1007        rs1 = prog.get_val(self.rs1)
1008        rs2 = prog.get_val(self.rs2)
1009        prog.set_val(self.rd, rs1 + rs2)
1010
1011    def get_opcode(self):
1012        return "add"
1013
1014
1015 class Addi(BinOpImm):
1016     """
1017     addi rd, rs1, imm: rd = rs1 + imm
1018
1019     Example:
1020         >>> i = Addi("a", "b0", 1)
1021         >>> str(i)
1022         'a = addi b0 1'
1023
1024         >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
1025         >>> p.eval()
1026         >>> p.get_val("a")
1027         5
1028
1029     """
1030
1031     def eval(self, prog):
1032         rs1 = prog.get_val(self.rs1)
1033         rs2 = prog.get_val(self.rs2)
1034         prog.set_val(self.rd, rs1 + rs2)
1035
1036     def get_opcode(self):
1037         return "addi"
1038
1039
1040 class Mul(BinOp):
1041     """
1042     mul rd, rs1, rs2: rd = rs1 * rs2
1043
1044     Example:
1045         >>> i = Mul("a", "b0", "b1")
1046         >>> str(i)
1047         'a = mul b0 b1'
1048
1049         >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
1050         >>> p.eval()
1051         >>> p.get_val("a")
1052         6
1053
1054     """
1055
1056     def eval(self, prog):
1057         rs1 = prog.get_val(self.rs1)
1058         rs2 = prog.get_val(self.rs2)
1059         prog.set_val(self.rd, rs1 * rs2)
1060
1061     def get_opcode(self):
1062         return "mul"
1063
1064
1065 class Sub(BinOp):
1066     """
1067     sub rd, rs1, rs2: rd = rs1 - rs2
1068
1069     Example:
1070         >>> i = Sub("a", "b0", "b1")
1071         >>> str(i)
1072         'a = sub b0 b1'
1073
1074         >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
1075         >>> p.eval()
1076         >>> p.get_val("a")
1077         -1
1078
1079     """
1080
1081     def eval(self, prog):
1082         rs1 = prog.get_val(self.rs1)
1083         rs2 = prog.get_val(self.rs2)
1084         prog.set_val(self.rd, rs1 - rs2)
1085
1086     def get_opcode(self):
1087         return "sub"
1088
1089
1090 class Xor(BinOp):
1091     """
1092     xor rd, rs1, rs2: rd = rs1 ^ rs2
1093
1094     Example:
1095         >>> i = Xor("a", "b0", "b1")
1096         >>> str(i)
1097         'a = xor b0 b1'
1098
1099     """
1100
1101     def eval(self, prog):
1102         rs1 = prog.get_val(self.rs1)
1103         rs2 = prog.get_val(self.rs2)
1104         prog.set_val(self.rd, rs1 ^ rs2)
1105
1106     def get_opcode(self):
1107         return "xor"
1108
1109
1110 class Add(BinOp):
1111     """
1112     add rd, rs1, rs2: rd = rs1 + rs2
1113
1114     Example:
1115         >>> i = Add("a", "b0", "b1")
1116         >>> str(i)
1117         'a = add b0 b1'
1118
1119         >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
1120         >>> p.eval()
1121         >>> p.get_val("a")
1122         5
1123
1124     """
1125
1126     def eval(self, prog):
1127         rs1 = prog.get_val(self.rs1)
1128         rs2 = prog.get_val(self.rs2)
1129         prog.set_val(self.rd, rs1 + rs2)
1130
1131     def get_opcode(self):
1132         return "add"
1133
1134
1135 class Addi(BinOpImm):
1136     """
1137     addi rd, rs1, imm: rd = rs1 + imm
1138
1139     Example:
1140         >>> i = Addi("a", "b0", 1)
1141         >>> str(i)
1142         'a = addi b0 1'
1143
1144         >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
1145         >>> p.eval()
1146         >>> p.get_val("a")
1147         5
1148
1149     """
1150
1151     def eval(self, prog):
1152         rs1 = prog.get_val(self.rs1)
1153         rs2 = prog.get_val(self.rs2)
1154         prog.set_val(self.rd, rs1 + rs2)
1155
1156     def get_opcode(self):
1157         return "addi"
1158
1159
1160 class Mul(BinOp):
1161     """
1162     mul rd, rs1, rs2: rd = rs1 * rs2
1163
1164     Example:
1165         >>> i = Mul("a", "b0", "b1")
1166         >>> str(i)
1167         'a = mul b0 b1'
1168
1169         >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
1170         >>> p.eval()
1171         >>> p.get_val("a")
1172         6
1173
1174     """
1175
1176     def eval(self, prog):
1177         rs1 = prog.get_val(self.rs1)
1178         rs2 = prog.get_val(self.rs2)
1179         prog.set_val(self.rd, rs1 * rs2)
1180
1181     def get_opcode(self):
1182         return "mul"
1183
1184
1185 class Sub(BinOp):
1186     """
1187     sub rd, rs1, rs2: rd = rs1 - rs2
1188
1189     Example:
1190         >>> i = Sub("a", "b0", "b1")
1191         >>> str(i)
1192         'a = sub b0 b1'
1193
1194         >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
1195         >>> p.eval()
1196         >>> p.get_val("a")
1197         -1
1198
1199     """
1200
1201     def eval(self, prog):
1202         rs1 = prog.get_val(self.rs1)
1203         rs2 = prog.get_val(self.rs2)
1204         prog.set_val(self.rd, rs1 - rs2)
1205
1206     def get_opcode(self):
1207         return "sub"
1208
1209
1210 class Xor(BinOp):
1211     """
1212     xor rd, rs1, rs2: rd = rs1 ^ rs2
1213
1214     Example:
1215         >>> i = Xor("a", "b0", "b1")
1216         >>> str(i)
1217         'a = xor b0 b1'
1218
1219     """
1220
1221     def eval(self, prog):
1222         rs1 = prog.get_val(self.rs1)
1223         rs2 = prog.get_val(self.rs2)
1224         prog.set_val(self.rd, rs1 ^ rs2)
1225
1226     def get_opcode(self):
1227         return "xor"
1228
1229
1230 class Add(BinOp):
1231     """
1232     add rd, rs1, rs2: rd = rs1 + rs2
1233
1234     Example:
1235         >>> i = Add("a", "b0", "b1")
1236         >>> str(i)
1237         'a = add b0 b1'
1238
1239         >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
1240         >>> p.eval()
1241         >>> p.get_val("a")
1242         5
1243
1244     """
1245
1246     def eval(self, prog):
1247         rs1 = prog.get_val(self.rs1)
1248         rs2 = prog.get_val(self.rs2)
1249         prog.set_val(self.rd, rs1 + rs2)
1250
1251     def get_opcode(self):
1252         return "add"
1253
1254
1255 class Addi(BinOpImm):
1256     """
1257     addi rd, rs1, imm: rd = rs1 + imm
1258
1259     Example:
1260         >>> i = Addi("a", "b0", 1)
1261         >>> str(i)
1262         'a = addi b0 1'
1263
1264         >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
1265         >>> p.eval()
1266         >>> p.get_val("a")
1267         5
1268
1269     """
1270
1271     def eval(self, prog):
1272         rs1 = prog.get_val(self.rs1)
1273         rs2 = prog.get_val(self.rs2)
1274         prog.set_val(self.rd, rs1 + rs2)
1275
1276     def get_opcode(self):
1277         return "addi"
1278
1279
1280 class Mul(BinOp):
1281     """
1282     mul rd, rs1, rs2: rd = rs1 * rs2
1283
1284     Example:
1285         >>> i = Mul("a", "b0", "b1")
1286         >>> str(i)
1287         'a = mul b0 b1'
1288
1289         >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
1290         >>> p.eval()
1291         >>> p.get_val("a")
1292         6
1293
1294     """
1295
1296     def eval(self, prog):
1297         rs1 = prog.get_val(self.rs1)
1298         rs2 = prog.get_val(self.rs2)
1299         prog.set_val(self.rd, rs1 * rs2)
1300
1301     def get_opcode(self):
1302         return "mul"
1303
1304
1305 class Sub(BinOp):
1306     """
1307     sub rd, rs1, rs2: rd = rs1 - rs2
1308
1309     Example:
1310         >>> i = Sub("a", "b0", "b1")
1311         >>> str(i)
1312         'a = sub b0 b1'
1313
1314         >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
1315         >>> p.eval()
1316         >>> p.get_val("a")
1317         -1
1318
1319     """
1320
1321     def eval(self, prog):
1322         rs1 = prog.get_val(self.rs1)
1323         rs2 = prog.get_val(self.rs2)
1324         prog.set_val(self.rd, rs1 - rs2)
1325
1326     def get_opcode(self):
1327         return "sub"
1328
1329
1330 class Xor(BinOp):
1331     """
1332     xor rd, rs1, rs2: rd = rs1 ^ rs2
1333
1334     Example:
1335         >>> i = Xor("a", "b0", "b1")
1336         >>> str(i)
1337         'a = xor b0 b1'
1338
1339     """
1340
1341     def eval(self, prog):
1342         rs1 = prog.get_val(self.rs1)
1343         rs2 = prog.get_val(self.rs2)
1344         prog.set_val(self.rd, rs1 ^ rs2)
1345
1346     def get_opcode(self):
1347         return "xor"
1348
1349
1350 class Add(BinOp):
1351     """
1352     add rd, rs1, rs2: rd = rs1 + rs2
1353
1354     Example:
1355         >>> i = Add("a", "b0", "b1")
1356         >>> str(i)
1357         'a = add b0 b1'
1358
1359         >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
1360         >>> p.eval()
1361         >>> p.get_val("a")
1362         5
1363
1364     """
1365
1366     def eval(self, prog):
1367         rs1 = prog.get_val(self.rs1)
1368         rs2 = prog.get_val(self.rs2)
1369         prog.set_val(self.rd, rs1 + rs2)
1370
1371     def get_opcode(self):
1372         return "add"
1373
1374
1375 class Addi(BinOpImm):
1376     """
1377     addi rd, rs1, imm: rd = rs1 + imm
1378
1379     Example:
1380         >>> i = Addi("a", "b0", 1)
1381        
```

```

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