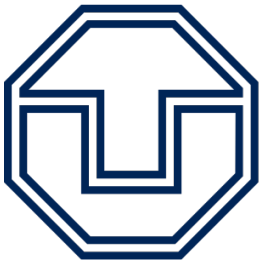

TECHNISCHE UNIVERSITÄT DRESDEN

RoboLab

Assignments

03 - Stack Machine

Florian Lubitz

Fakultät Informatik

5106912

florian.lubitz@mailbox.tu-dresden.de

Task 1

Question 1

One drawback of stack machines is the need of more memory references. For a simple `ADD` operation of two integers, how many times the data cache is referenced? Write down the steps for the operation.

```
PUSH 1
PUSH 2
ADD:
    OP1 = POP
    OP2 = POP
    PUSH OP1 + OP2
```

This results in 5 stack operations (2 pushes, 2 pops, 1 push) and 4 memory references (for reading an writing the operands).

Question 2

For stack machines, we have a very compact object code (instruction set and rules) which fits in 6 bit or less. In comparison, register machines need more bits for the same instruction on the arithmetic logic unit (ALU). Explain briefly why this is the case and give an average length needed for instructions for register machines.

Stack machines need less bits for the instruction set because they only need to specify the operation and not the operands. The operands are implicitly specified by the stack. For register machines, the operands need to be specified explicitly. This results in a larger instruction set and thus more bits needed for the instructions.

The instructions for the stack machine are 6 bits long. The first two bit of an instruction specify the type of operation and the remaining 4 bits specify the parameters or operation. Register machines need to specify the operation and the operands. With the same set of operations we would need the same 4 bits for the operation but also 3 bits to specify the operands register and the target register. This results in at least 7 bits for the instruction set. Register machines also need more operands and average to 16 bits per instruction.

Question 3

Explain briefly how register and stack machines handle interrupts and why stack machines may have an advantage here. A register machine needs to save the state of the registers when an interrupt occurs. This is done by pushing the registers, program counter and other information to memory. After the interrupt is handled, the registers need to be restored from memory. A stack machine does not need to perform this state saving because most parameters are already on the stack. Only the stack pointer needs to be saved and restored.

Task 2

The following expressions are encoded in postfix notation. They can be converted to postfix notation by using the shunting-yard algorithm.

$$4 * (7 + 8 * 9) - 1 \Rightarrow 4789 * + * 1 -$$
$$96 - (4 + 44 * (3 - 1) + 7) * 25 \Rightarrow 9644431 - * + 7 + 25 * -$$
$$5^3/(2 + 3))/5 = ((5 * 5 * 5)/(2 + 3))/5 \Rightarrow 4789 * + * 1 -$$

Task 3

Running the first expression on the stack machine results in the following steps:

1. Instruction list is: `[4, 2, 2, 3, MUL, ADD, MUL, 2, DIV, STP]`
2. Instruction is: `4` Stack is: `[]` Overflow flag is: `False`
   Pushing `4`
   Stack after instruction: `[4]`
3. Instruction is: `2` Stack is: `[4]` Overflow flag is: `False`
   Pushing `2`
   Stack after instruction: `[4, 2]`
4. Instruction is: `2` Stack is: `[4, 2]` Overflow flag is: `False`
   Pushing `2`
   Stack after instruction: `[4, 2, 2]`
5. Instruction is: `3` Stack is: `[4, 2, 2]` Overflow flag is: `False`
   Pushing `3`
   Stack after instruction: `[4, 2, 2, 3]`

6. Instruction is: `MUL` Stack is: `[4, 2, 2, 3]` Overflow flag is: `False`
   Run instruction `MUL`
   Stack after instruction: `[4, 2, 6]`
7. Instruction is: `ADD` Stack is: `[4, 2, 6]` Overflow flag is: `False`
   Run instruction `ADD`
   Stack after instruction: `[4, 8]`
8. Instruction is: `MUL` Stack is: `[4, 8]` Overflow flag is: `False`
   Run instruction `MUL`
   Stack after instruction: `[32]`
9. Instruction is: `2` Stack is: `[32]` Overflow flag is: `False`
   Pushing `2`
   Stack after instruction: `[32, 2]`
10. Instruction is: `DIV` Stack is: `[32, 2]` Overflow flag is: `False`
    Run instruction `DIV`
    Stack after instruction: `[16]`
11. Instruction is: `STP` Stack is: `[16]` Overflow flag is: `False`
    Run instruction `STP`
12. Final stack is: `[16]`

The instruction list results the following steps:

1. Instruction list is: `[10, DUP, DUP, MUL, XOR, 4, SHR, 4, MOD, 6, EXP, ' ', 'S', 'E', 'R', STP]`
2. Instruction is: `10` Stack is: `[]` Overflow flag is: `False`
   Pushing `10`
   Stack after instruction: `[10]`
3. Instruction is: `DUP` Stack is: `[10]` Overflow flag is: `False`
   Run instruction `DUP`
   Stack after instruction: `[10, 10]`
4. Instruction is: `DUP` Stack is: `[10, 10]` Overflow flag is: `False`
   Run instruction `DUP`
   Stack after instruction: `[10, 10, 10]`
5. Instruction is: `MUL` Stack is: `[10, 10, 10]` Overflow flag is: `False`
   Run instruction `MUL`
   Stack after instruction: `[10, 100]`

6. Instruction is: `XOR` Stack is: `[10, 100]` Overflow flag is: `False`
   Run instruction `XOR`
   Stack after instruction: `[110]`

7. Instruction is: `4` Stack is: `[110]` Overflow flag is: `False`
   Pushing `4`
   Stack after instruction: `[110, 4]`

8. Instruction is: `SHR` Stack is: `[110, 4]` Overflow flag is: `False`
   Run instruction `SHR`
   Stack after instruction: `[6]`

9. Instruction is: `4` Stack is: `[6]` Overflow flag is: `False`
   Pushing `4`
   Stack after instruction: `[6, 4]`

10. Instruction is: `MOD` Stack is: `[6, 4]` Overflow flag is: `False`
    Run instruction `MOD`
    Stack after instruction: `[2]`

11. Instruction is: `6` Stack is: `[2]` Overflow flag is: `False`
    Pushing `6`
    Stack after instruction: `[2, 6]`

12. Instruction is: `EXP` Stack is: `[2, 6]` Overflow flag is: `False`
    Run instruction `EXP`
    Stack after instruction: `[64]`

13. Instruction is: `Stack` is: [64] Overflow flag is: `True`
    Pushing
    Stack after instruction: `[64, ' ']`

14. Instruction is: `S` Stack is: `[64, ' ']` Overflow flag is: `False`
    Pushing `S`
    Stack after instruction: `[64, ' ', 'S']`

15. Instruction is: `E` Stack is: `[64, ' ', 'S']` Overflow flag is: `False`
    Pushing `E`
    Stack after instruction: `[64, ' ', 'S', 'E']`

16. Instruction is: `R` Stack is: `[64, ' ', 'S', 'E']` Overflow flag is: `False`
    Pushing `R`
    Stack after instruction: `[64, ' ', 'S', 'E', 'R']`

17. Instruction is: `STP` Stack is: `[64, ' ', 'S', 'E', 'R']` Overflow flag is: `False`
    Run instruction `STP`
    Final stack is: `[64, ' ', 'S', 'E', 'R']`

Appendix

Python code for the shunting-yard algorithm

```python
from enum import Enum


class Associativity(Enum):
    LEFT = 1
    RIGHT = 2


operators: dict[str, dict[str, Associativity]] = {
    "*": {"precedence": 3, "associativity": Associativity.LEFT},
    "/": {"precedence": 3, "associativity": Associativity.LEFT},
    "+": {"precedence": 2, "associativity": Associativity.LEFT},
    "-": {"precedence": 2, "associativity": Associativity.LEFT},
}


def shunting_yard(input: str) -> str:
    input = input.replace(" ", "")
    operations = operators.keys()
    stack = []
    output = []
    for token in input:
        if token.isnumeric():
            output.append(token)
        elif token in operations:
            op1 = operators[token]
            while (
                len(stack) > 0
                and stack[-1] in operations
                and (
                    operators[stack[-1]]["precedence"] > op1["precedence"]
                    or (
```

```
                    operators[stack[-1]]["precedence"] == op1["precedence"]
                    and operators[stack[-1]]["associativity"] == Associativity
                        .LEFT
                )
            )
        ):
            output.append(stack.pop())
        stack.append(token)
    elif token == "(":
        stack.append(token)
    elif token == ")":
        while stack[-1] != "(":
            assert len(stack) > 0
            output.append(stack.pop())
        assert stack[-1] == "("
        stack.pop()
    else:
        raise Exception(f"Unknown token: {token}")
while len(stack) > 0:
    output.append(stack.pop())
return " ".join(output)


examples = [
    "4*(7+8*9)-1",
    "(96 - (4 + 44 * (3 - 1) + 7) * 25)",
    "((5*5*5) / ( 2 + 3)) / 5",
]

for example in examples:
    print(shunting_yard(example))

# Prints:
# 4789*+*1-
# 9644431-*+7+25*-
# 55*5*23+/5/
```

Python code for the stack machine

```
from enum import Enum
from typing import List
```

```
3
4
5  class Instruction(Enum):
6      STP = 0b010000
7      DUP = 0b010001
8      DEL = 0b010010
9      SWP = 0b010011
10     ADD = 0b010100
11     SUB = 0b010101
12     MUL = 0b010110
13     DIV = 0b010111
14     EXP = 0b011000
15     MOD = 0b011001
16     SHL = 0b011010
17     SHR = 0b011011
18     HEX = 0b011100
19     FAC = 0b011101
20     NOT = 0b011110
21     XOR = 0b011111
22     NOP = None
23     SPEAK = 0b100001
24
25     def __str__(self):
26         return self.name
27
28
29 class StackMachine:
30     def __init__(self):
31         self.stack = []
32         self.overflow_flag = False
33
34     def parse_byte(self, byte: int) -> int or Instruction or str:
35         if 0 <= byte <= 15:
36             # Is a number
37             return byte
38         elif 16 <= byte <= 31:
39             # Is an instruction
40             return Instruction(byte)
41         elif 32 <= byte <= 35:
42             # Is a special case
43             if byte == 33:
```

```
44                 return Instruction.SPEAK
45             elif byte == 34:
46                 return " "
47             else:
48                 return Instruction.NOP
49         elif 36 <= byte <= 61:
50             # Is a letter
51             return chr(ord("A") + byte - 36)
52         else:
53             return Instruction.NOP
54
55     def parse_instr_list(self, instr_list: List[str]) -> List[int]:
56         return [self.parse_byte(int(x, 2)) for x in instr_list]
57
58     def rpn_to_instr_list(self, rpn: str) -> List[int]:
59         math_operations = {
60             "+": 0b010100,
61             "-": 0b010101,
62             "*": 0b010110,
63             "/": 0b010111,
64         }
65         instr_list = []
66         for token in rpn:
67             if token in math_operations.keys():
68                 instr_list.append(math_operations[token])
69             else:
70                 instr_list.append(int(token, 16))
71
72         instr_list.append(0b010000)
73         return [bin(x)[2:].zfill(6) for x in instr_list]
74
75     def simulate_instructions(self, instr_list: List[str] or str):
76         # Clear stack and overflow flag
77         self.stack.clear()
78         self.overflow_flag = False
79         if isinstance(instr_list, str):
80             instr_list = self.rpn_to_instr_list(instr_list)
81         instr_list = self.parse_instr_list(instr_list)
82         print("Instruction list is: ", instr_list)
83         for word in instr_list:
84             print(
```

```
85                "Instruction is:",
86                word,
87                "Stack is:",
88                self.stack,
89                "Overflow flag is:",
90                self.overflow_flag,
91            )
92            if isinstance(word, Instruction):
93                print("\tRun instruction", word)
94                if self.run_instruction(word) == 1:
95                    print("Final stack is: ", self.stack)
96                    return
97            else:
98                print("\tPushing", word)
99                self.stack.append(word)
100               self.overflow_flag = False
101           print("\tStack after instruction: ", self.stack)
102
103   def get_operands_from_stack(self, n=2):
104       if len(self.stack) < n:
105           raise ValueError("Stack underflow")
106       else:
107           return tuple(self.stack.pop() for _ in range(n))
108
109   def run_instruction(self, instr: Instruction):
110       MAX_INT = 255
111       if instr == Instruction.STP:
112           return 1
113       elif instr == Instruction.DUP:
114           ops = self.get_operands_from_stack(1)
115           self.stack.append(ops[0])
116           self.stack.append(ops[0])
117       elif instr == Instruction.DEL:
118           print("Do DEL")
119       elif instr == Instruction.SWP:
120           print("Do SWP")
121       elif instr == Instruction.ADD:
122           ops = self.get_operands_from_stack()
123           result = ops[1] + ops[0]
124           if result > MAX_INT:
125               result = result % (MAX_INT + 1)
```

```
126                 self.overflow_flag = True
127             self.stack.append(result)
128         elif instr == Instruction.SUB:
129             ops = self.get_operands_from_stack()
130             result = ops[1] - ops[0]
131             if result < 0:
132                 result = (MAX_INT + 1) + result
133                 self.overflow_flag = True
134             self.stack.append(result)
135         elif instr == Instruction.MUL:
136             ops = self.get_operands_from_stack()
137             result = ops[1] * ops[0]
138 
139             if result > MAX_INT:
140                 result = result % (MAX_INT + 1)
141                 self.overflow_flag = True
142             self.stack.append(result)
143         elif instr == Instruction.DIV:
144             ops = self.get_operands_from_stack()
145             self.overflow_flag = False
146             self.stack.append(ops[1] // ops[0])
147         elif instr == Instruction.EXP:
148             ops = self.get_operands_from_stack()
149             result = ops[1] ** ops[0]
150 
151             if result > MAX_INT:
152                 result = result % (MAX_INT + 1)
153                 self.overflow_flag = True
154             self.stack.append(result)
155         elif instr == Instruction.MOD:
156             self.overflow_flag = False
157             ops = self.get_operands_from_stack()
158             result = ops[1] % ops[0]
159             self.stack.append(result)
160         elif instr == Instruction.SHL:
161             print("Do SHL")
162         elif instr == Instruction.SHR:
163             self.overflow_flag = False
164             ops = self.get_operands_from_stack()
165             result = ops[1] >> ops[0]
166             self.stack.append(result)
```

```
167         elif instr == Instruction.HEX:
168             print("Do HEX")
169         elif instr == Instruction.FAC:
170             print("Do FAC")
171         elif instr == Instruction.NOT:
172             print("Do NOT")
173         elif instr == Instruction.XOR:
174             self.overflow_flag = False
175             ops = self.get_operands_from_stack()
176             result = ops[1] ^ ops[0]
177             self.stack.append(result)
178         elif instr == Instruction.NOP:
179             print("Do NOP")
180         elif instr == Instruction.SPEAK:
181             print("Do SPEAK")
182         return 0
183
184
185 rpn_expr = "4223*+*2/"
186 instr_list = [
187     "001010",
188     "010001",
189     "010001",
190     "010110",
191     "011111",
192     "000100",
193     "011011",
194     "000100",
195     "011001",
196     "000110",
197     "011000",
198     "100010",
199     "110110",
200     "101000",
201     "110101",
202     "010000",
203 ]
204
205 sm = StackMachine()
206 print("1. RPN expression")
207 sm.simulate_instructions(rpn_expr)
```

```
208 print("\n\n2. instruction list")
209 sm.simulate_instructions(instr_list)
210
211 # Prints
212 # 1. RPN expression
213 # Instruction list is: [4, 2, 2, 3, <Instruction.MUL: 22>, <Instruction.ADD:
        20>, <Instruction.MUL: 22>, 2, <Instruction.DIV: 23>, <Instruction.STP: 16>]
214 # Instruction is: 4 Stack is: [] Overflow flag is: False
215 #   Pushing 4
216 #   Stack after instruction: [4]
217 # Instruction is: 2 Stack is: [4] Overflow flag is: False
218 #   Pushing 2
219 #   Stack after instruction: [4, 2]
220 # Instruction is: 2 Stack is: [4, 2] Overflow flag is: False
221 #   Pushing 2
222 #   Stack after instruction: [4, 2, 2]
223 # Instruction is: 3 Stack is: [4, 2, 2] Overflow flag is: False
224 #   Pushing 3
225 #   Stack after instruction: [4, 2, 2, 3]
226 # Instruction is: MUL Stack is: [4, 2, 2, 3] Overflow flag is: False
227 #   Run instruction MUL
228 #   Stack after instruction: [4, 2, 6]
229 # Instruction is: ADD Stack is: [4, 2, 6] Overflow flag is: False
230 #   Run instruction ADD
231 #   Stack after instruction: [4, 8]
232 # Instruction is: MUL Stack is: [4, 8] Overflow flag is: False
233 #   Run instruction MUL
234 #   Stack after instruction: [32]
235 # Instruction is: 2 Stack is: [32] Overflow flag is: False
236 #   Pushing 2
237 #   Stack after instruction: [32, 2]
238 # Instruction is: DIV Stack is: [32, 2] Overflow flag is: False
239 #   Run instruction DIV
240 #   Stack after instruction: [16]
241 # Instruction is: STP Stack is: [16] Overflow flag is: False
242 #   Run instruction STP
243 # Final stack is: [16]
244
245
246 # 2. instruction list
```

```
247 # Instruction list is: [10, <Instruction.DUP: 17>, <Instruction.DUP: 17>, <
        Instruction.MUL: 22>, <Instruction.XOR: 31>, 4, <Instruction.SHR: 27>, 4, <
        Instruction.MOD: 25>, 6, <Instruction.EXP: 24>, ' ', 'S', 'E', 'R', <
        Instruction.STP: 16>]
248 # Instruction is: 10 Stack is: [] Overflow flag is: False
249 #   Pushing 10
250 #   Stack after instruction: [10]
251 # Instruction is: DUP Stack is: [10] Overflow flag is: False
252 #   Run instruction DUP
253 #   Stack after instruction: [10, 10]
254 # Instruction is: DUP Stack is: [10, 10] Overflow flag is: False
255 #   Run instruction DUP
256 #   Stack after instruction: [10, 10, 10]
257 # Instruction is: MUL Stack is: [10, 10, 10] Overflow flag is: False
258 #   Run instruction MUL
259 #   Stack after instruction: [10, 100]
260 # Instruction is: XOR Stack is: [10, 100] Overflow flag is: False
261 #   Run instruction XOR
262 #   Stack after instruction: [110]
263 # Instruction is: 4 Stack is: [110] Overflow flag is: False
264 #   Pushing 4
265 #   Stack after instruction: [110, 4]
266 # Instruction is: SHR Stack is: [110, 4] Overflow flag is: False
267 #   Run instruction SHR
268 #   Stack after instruction: [6]
269 # Instruction is: 4 Stack is: [6] Overflow flag is: False
270 #   Pushing 4
271 #   Stack after instruction: [6, 4]
272 # Instruction is: MOD Stack is: [6, 4] Overflow flag is: False
273 #   Run instruction MOD
274 #   Stack after instruction: [2]
275 # Instruction is: 6 Stack is: [2] Overflow flag is: False
276 #   Pushing 6
277 #   Stack after instruction: [2, 6]
278 # Instruction is: EXP Stack is: [2, 6] Overflow flag is: False
279 #   Run instruction EXP
280 #   Stack after instruction: [64]
281 # Instruction is: Stack is: [64] Overflow flag is: False
282 #   Pushing
283 #   Stack after instruction: [64, ' ']
284 # Instruction is: S Stack is: [64, ' '] Overflow flag is: False
```

```
285 #   Pushing S
286 #   Stack after instruction: [64, ' ', 'S']
287 # Instruction is: E Stack is: [64, ' ', 'S'] Overflow flag is: False
288 #   Pushing E
289 #   Stack after instruction: [64, ' ', 'S', 'E']
290 # Instruction is: R Stack is: [64, ' ', 'S', 'E'] Overflow flag is: False
291 #   Pushing R
292 #   Stack after instruction: [64, ' ', 'S', 'E', 'R']
293 # Instruction is: STP Stack is: [64, ' ', 'S', 'E', 'R'] Overflow flag is: False
294 #   Run instruction STP
295 # Final stack is: [64, ' ', 'S', 'E', 'R']
296
297
298
299 # Prints:
300 # 1. RPN expression
301 # Instruction list is: [4, 2, 2, 3, <Instruction.MUL: 22>, <Instruction.ADD:
        20>, <Instruction.MUL: 22>, 2, <Instruction.DIV: 23>, <Instruction.STP: 16>]
302 # Instruction is: 4 Stack is: [] Overflow flag is: False
303 #   Pushing 4
304 #   Stack after instruction: [4]
305 # Instruction is: 2 Stack is: [4] Overflow flag is: False
306 #   Pushing 2
307 #   Stack after instruction: [4, 2]
308 # Instruction is: 2 Stack is: [4, 2] Overflow flag is: False
309 #   Pushing 2
310 #   Stack after instruction: [4, 2, 2]
311 # Instruction is: 3 Stack is: [4, 2, 2] Overflow flag is: False
312 #   Pushing 3
313 #   Stack after instruction: [4, 2, 2, 3]
314 # Instruction is: MUL Stack is: [4, 2, 2, 3] Overflow flag is: False
315 #   Run instruction MUL
316 #   Stack after instruction: [4, 2, 6]
317 # Instruction is: ADD Stack is: [4, 2, 6] Overflow flag is: False
318 #   Run instruction ADD
319 #   Stack after instruction: [4, 8]
320 # Instruction is: MUL Stack is: [4, 8] Overflow flag is: False
321 #   Run instruction MUL
322 #   Stack after instruction: [32]
323 # Instruction is: 2 Stack is: [32] Overflow flag is: False
324 #   Pushing 2
```

```
325 #   Stack after instruction: [32, 2]
326 # Instruction is: DIV Stack is: [32, 2] Overflow flag is: False
327 #   Run instruction DIV
328 #   Stack after instruction: [16]
329 # Instruction is: STP Stack is: [16] Overflow flag is: False
330 #   Run instruction STP
331 # Final stack is: [16]
332
333
334 # 2. instruction list
335 # Instruction list is: [10, <Instruction.DUP: 17>, <Instruction.DUP: 17>, <
        Instruction.MUL: 22>, <Instruction.XOR: 31>, 4, <Instruction.SHR: 27>, 4, <
        Instruction.MOD: 25>, 6, <Instruction.EXP: 24>, ' ', 'S', 'E', 'R', <
        Instruction.STP: 16>]
336 # Instruction is: 10 Stack is: [] Overflow flag is: False
337 #   Pushing 10
338 #   Stack after instruction: [10]
339 # Instruction is: DUP Stack is: [10] Overflow flag is: False
340 #   Run instruction DUP
341 #   Stack after instruction: [10, 10]
342 # Instruction is: DUP Stack is: [10, 10] Overflow flag is: False
343 #   Run instruction DUP
344 #   Stack after instruction: [10, 10, 10]
345 # Instruction is: MUL Stack is: [10, 10, 10] Overflow flag is: False
346 #   Run instruction MUL
347 #   Stack after instruction: [10, 100]
348 # Instruction is: XOR Stack is: [10, 100] Overflow flag is: False
349 #   Run instruction XOR
350 #   Stack after instruction: [110]
351 # Instruction is: 4 Stack is: [110] Overflow flag is: False
352 #   Pushing 4
353 #   Stack after instruction: [110, 4]
354 # Instruction is: SHR Stack is: [110, 4] Overflow flag is: False
355 #   Run instruction SHR
356 #   Stack after instruction: [6]
357 # Instruction is: 4 Stack is: [6] Overflow flag is: False
358 #   Pushing 4
359 #   Stack after instruction: [6, 4]
360 # Instruction is: MOD Stack is: [6, 4] Overflow flag is: False
361 #   Run instruction MOD
362 #   Stack after instruction: [2]
```

```
363 # Instruction is: 6 Stack is: [2] Overflow flag is: False
364 #   Pushing 6
365 #   Stack after instruction: [2, 6]
366 # Instruction is: EXP Stack is: [2, 6] Overflow flag is: False
367 #   Run instruction EXP
368 #   Stack after instruction: [0]
369 # Instruction is: Stack is: [0] Overflow flag is: True
370 #   Pushing
371 #   Stack after instruction: [0, ' ']
372 # Instruction is: S Stack is: [0, ' '] Overflow flag is: False
373 #   Pushing S
374 #   Stack after instruction: [0, ' ', 'S']
375 # Instruction is: E Stack is: [0, ' ', 'S'] Overflow flag is: False
376 #   Pushing E
377 #   Stack after instruction: [0, ' ', 'S', 'E']
378 # Instruction is: R Stack is: [0, ' ', 'S', 'E'] Overflow flag is: False
379 #   Pushing R
380 #   Stack after instruction: [0, ' ', 'S', 'E', 'R']
381 # Instruction is: STP Stack is: [0, ' ', 'S', 'E', 'R'] Overflow flag is: False
382 #   Run instruction STP
383 # Final stack is: [0, ' ', 'S', 'E', 'R']
```