



**TECHNISCHE  
UNIVERSITÄT  
DRESDEN**

RoboLab

## **Assignments**

03 - Stack Machine

**Florian Lubitz**

Fakultät Informatik

5106912

[florian.lubitz@mailbox.tu-dresden.de](mailto: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 and 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 bits 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.

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

### 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: [0]
13. Instruction is: **Stack** is: [0] Overflow flag is: **True**  
Pushing  
Stack after instruction: [0, ' ']
14. Instruction is: **S** Stack is: [0, ' '] Overflow flag is: **False**  
Pushing **S**  
Stack after instruction: [0, ' ', 'S']
15. Instruction is: **E** Stack is: [0, ' ', 'S'] Overflow flag is: **False**  
Pushing **E**  
Stack after instruction: [0, ' ', 'S', 'E']
16. Instruction is: **R** Stack is: [0, ' ', 'S', 'E'] Overflow flag is: **False**  
Pushing **R**  
Stack after instruction: [0, ' ', 'S', 'E', 'R']

17. Instruction is: STP Stack is: [0, ' ', 'S', 'E', 'R'] Overflow flag is: False  
Run instruction STP  
Final stack is: [0, ' ', 'S', 'E', 'R']

## Appendix

### Python code for the shunting-yard algorithm

```
1  from enum import Enum
2
3
4  class Associativity(Enum):
5      LEFT = 1
6      RIGHT = 2
7
8
9  operators: dict[str, dict[str, Associativity]] = {
10     "*": {"precedence": 3, "associativity": Associativity.LEFT},
11     "/": {"precedence": 3, "associativity": Associativity.LEFT},
12     "+": {"precedence": 2, "associativity": Associativity.LEFT},
13     "-": {"precedence": 2, "associativity": Associativity.LEFT},
14 }
15
16
17 def shunting_yard(input: str) -> str:
18     input = input.replace(" ", "")
19     operations = operators.keys()
20     stack = []
21     output = []
22     for token in input:
23         if token.isnumeric():
24             output.append(token)
25         elif token in operations:
26             op1 = operators[token]
27             while (
28                 len(stack) > 0
29                 and stack[-1] in operations
30                 and (
31                     operators[stack[-1]]["precedence"] > op1["precedence"]
32                     or (
```

```

33         operators[stack[-1]]["precedence"] == op1["precedence"]
34         and operators[stack[-1]]["associativity"] == Associativity
           .LEFT
35     )
36 )
37 ):
38     output.append(stack.pop())
39     stack.append(token)
40 elif token == "(":
41     stack.append(token)
42 elif token == ")":
43     while stack[-1] != "(":
44         assert len(stack) > 0
45         output.append(stack.pop())
46     assert stack[-1] == "("
47     stack.pop()
48 else:
49     raise Exception(f"Unknown token: {token}")
50 while len(stack) > 0:
51     output.append(stack.pop())
52 return " ".join(output)
53
54
55 examples = [
56     "4*(7+8*9)-1",
57     "(96 - (4 + 44 * (3 - 1) + 7) * 25)",
58     "((5*5*5) / ( 2 + 3)) / 5",
59 ]
60
61 for example in examples:
62     print(shunting_yard(example))
63
64 # Prints:
65 # 4789*+*1-
66 # 96444431-*+7+25*-
67 # 55*5*23+/5/

```

### Python code for the stack machine

```

1 from enum import Enum
2 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        if instr == Instruction.STP:
111            return 1
112        elif instr == Instruction.DUP:
113            ops = self.get_operands_from_stack(1)
114            self.stack.append(ops[0])
115            self.stack.append(ops[0])
116        elif instr == Instruction.DEL:
117            print("Do DEL")
118        elif instr == Instruction.SWP:
119            print("Do SWP")
120        elif instr == Instruction.ADD:
121            ops = self.get_operands_from_stack()
122            result = ops[1] + ops[0]
123            if result > 15:
124                result = result % 16
125            self.overflow_flag = True

```

```
126         self.stack.append(result)
127     elif instr == Instruction.SUB:
128         ops = self.get_operands_from_stack()
129         result = ops[1] - ops[0]
130         if result < 0:
131             result = 16 + result
132             self.overflow_flag = True
133         self.stack.append(result)
134     elif instr == Instruction.MUL:
135         ops = self.get_operands_from_stack()
136         result = ops[1] * ops[0]
137         if result < 0:
138             result = 16 + result
139             self.overflow_flag = True
140         self.stack.append(result)
141     elif instr == Instruction.DIV:
142         ops = self.get_operands_from_stack()
143         self.overflow_flag = False
144         self.stack.append(ops[1] // ops[0])
145     elif instr == Instruction.EXP:
146         ops = self.get_operands_from_stack()
147         result = ops[1] ** ops[0]
148         if result > 15:
149             result = result % 16
150             self.overflow_flag = True
151         self.stack.append(result)
152     elif instr == Instruction.MOD:
153         self.overflow_flag = False
154         ops = self.get_operands_from_stack()
155         result = ops[1] % ops[0]
156         self.stack.append(result)
157     elif instr == Instruction.SHL:
158         print("Do SHL")
159     elif instr == Instruction.SHR:
160         self.overflow_flag = False
161         ops = self.get_operands_from_stack()
162         result = ops[1] >> ops[0]
163         self.stack.append(result)
164     elif instr == Instruction.HEX:
165         print("Do HEX")
166     elif instr == Instruction.FAC:
```

```
167         print("Do FAC")
168     elif instr == Instruction.NOT:
169         print("Do NOT")
170     elif instr == Instruction.XOR:
171         self.overflow_flag = False
172         ops = self.get_operands_from_stack()
173         result = ops[1] ^ ops[0]
174         self.stack.append(result)
175     elif instr == Instruction.NOP:
176         print("Do NOP")
177     elif instr == Instruction.SPEAK:
178         print("Do SPEAK")
179     return 0
180
181
182 rpn_expr = "4223*++2/"
183 instr_list = [
184     "001010",
185     "010001",
186     "010001",
187     "010110",
188     "011111",
189     "000100",
190     "011011",
191     "000100",
192     "011001",
193     "000110",
194     "011000",
195     "100010",
196     "110110",
197     "101000",
198     "110101",
199     "010000",
200 ]
201
202 sm = StackMachine()
203 print("1. RPN expression")
204 sm.simulate_instructions(rpn_expr)
205 print("\n\n2. instruction list")
206 sm.simulate_instructions(instr_list)
207
```

```
208
209 # Prints:
210 # 1. RPN expression
211 # Instruction list is: [4, 2, 2, 3, <Instruction.MUL: 22>, <Instruction.ADD:
      20>, <Instruction.MUL: 22>, 2, <Instruction.DIV: 23>, <Instruction.STP: 16>]
212 # Instruction is: 4 Stack is: [] Overflow flag is: False
213 #   Pushing 4
214 #   Stack after instruction: [4]
215 # Instruction is: 2 Stack is: [4] Overflow flag is: False
216 #   Pushing 2
217 #   Stack after instruction: [4, 2]
218 # Instruction is: 2 Stack is: [4, 2] Overflow flag is: False
219 #   Pushing 2
220 #   Stack after instruction: [4, 2, 2]
221 # Instruction is: 3 Stack is: [4, 2, 2] Overflow flag is: False
222 #   Pushing 3
223 #   Stack after instruction: [4, 2, 2, 3]
224 # Instruction is: MUL Stack is: [4, 2, 2, 3] Overflow flag is: False
225 #   Run instruction MUL
226 #   Stack after instruction: [4, 2, 6]
227 # Instruction is: ADD Stack is: [4, 2, 6] Overflow flag is: False
228 #   Run instruction ADD
229 #   Stack after instruction: [4, 8]
230 # Instruction is: MUL Stack is: [4, 8] Overflow flag is: False
231 #   Run instruction MUL
232 #   Stack after instruction: [32]
233 # Instruction is: 2 Stack is: [32] Overflow flag is: False
234 #   Pushing 2
235 #   Stack after instruction: [32, 2]
236 # Instruction is: DIV Stack is: [32, 2] Overflow flag is: False
237 #   Run instruction DIV
238 #   Stack after instruction: [16]
239 # Instruction is: STP Stack is: [16] Overflow flag is: False
240 #   Run instruction STP
241 # Final stack is: [16]
242
243
244 # 2. instruction list
245 # 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>]
246 # Instruction is: 10 Stack is: [] Overflow flag is: False
247 #   Pushing 10
248 #   Stack after instruction: [10]
249 # Instruction is: DUP Stack is: [10] Overflow flag is: False
250 #   Run instruction DUP
251 #   Stack after instruction: [10, 10]
252 # Instruction is: DUP Stack is: [10, 10] Overflow flag is: False
253 #   Run instruction DUP
254 #   Stack after instruction: [10, 10, 10]
255 # Instruction is: MUL Stack is: [10, 10, 10] Overflow flag is: False
256 #   Run instruction MUL
257 #   Stack after instruction: [10, 100]
258 # Instruction is: XOR Stack is: [10, 100] Overflow flag is: False
259 #   Run instruction XOR
260 #   Stack after instruction: [110]
261 # Instruction is: 4 Stack is: [110] Overflow flag is: False
262 #   Pushing 4
263 #   Stack after instruction: [110, 4]
264 # Instruction is: SHR Stack is: [110, 4] Overflow flag is: False
265 #   Run instruction SHR
266 #   Stack after instruction: [6]
267 # Instruction is: 4 Stack is: [6] Overflow flag is: False
268 #   Pushing 4
269 #   Stack after instruction: [6, 4]
270 # Instruction is: MOD Stack is: [6, 4] Overflow flag is: False
271 #   Run instruction MOD
272 #   Stack after instruction: [2]
273 # Instruction is: 6 Stack is: [2] Overflow flag is: False
274 #   Pushing 6
275 #   Stack after instruction: [2, 6]
276 # Instruction is: EXP Stack is: [2, 6] Overflow flag is: False
277 #   Run instruction EXP
278 #   Stack after instruction: [0]
279 # Instruction is: Stack is: [0] Overflow flag is: True
280 #   Pushing
281 #   Stack after instruction: [0, ' ']
282 # Instruction is: S Stack is: [0, ' '] Overflow flag is: False
283 #   Pushing S
284 #   Stack after instruction: [0, ' ', 'S']
```

```
285 # Instruction is: E Stack is: [0, ' ', 'S'] Overflow flag is: False
286 #   Pushing E
287 #   Stack after instruction: [0, ' ', 'S', 'E']
288 # Instruction is: R Stack is: [0, ' ', 'S', 'E'] Overflow flag is: False
289 #   Pushing R
290 #   Stack after instruction: [0, ' ', 'S', 'E', 'R']
291 # Instruction is: STP Stack is: [0, ' ', 'S', 'E', 'R'] Overflow flag is: False
292 #   Run instruction STP
293 # Final stack is: [0, ' ', 'S', 'E', 'R']
```