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.

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

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

```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
```

```python
class Instruction(Enum):
    STP = 0b010000
    DUP = 0b010001
    DEL = 0b010010
    SWP = 0b010011
    ADD = 0b010100
    SUB = 0b010101
    MUL = 0b010110
    DIV = 0b010111
    EXP = 0b011000
    MOD = 0b011001
    SHL = 0b011010
    SHR = 0b011011
    HEX = 0b011100
    FAC = 0b011101
    NOT = 0b011110
    XOR = 0b011111
    NOP = None
    SPEAK = 0b100001

    def __str__(self):
        return self.name


class StackMachine:
    def __init__(self):
        self.stack = []
        self.overflow_flag = False

    def parse_byte(self, byte: int) -> int or Instruction or str:
        if 0 <= byte <= 15:
            # Is a number
            return byte
        elif 16 <= byte <= 31:
            # Is an instruction
            return Instruction(byte)
        elif 32 <= byte <= 35:
            # Is a special case
            if byte == 33:
                return Instruction.SPEAK
            elif byte == 34:
                return " "
            else:
                return Instruction.NOP
        elif 36 <= byte <= 61:
            # Is a letter
            return chr(ord("A") + byte - 36)
        else:
            return Instruction.NOP

    def parse_instr_list(self, instr_list: List[str]) -> List[int]:
        return [self.parse_byte(int(x, 2)) for x in instr_list]

    def rpn_to_instr_list(self, rpn: str) -> List[int]:
        math_operations = {
            "+": 0b010100,
            "-": 0b010101,
            "*": 0b010110,
            "/": 0b010111,
        }
        instr_list = []
        for token in rpn:
            if token in math_operations.keys():
                instr_list.append(math_operations[token])
            else:
                instr_list.append(int(token, 16))

        instr_list.append(0b010000)
        return [bin(x)[2:].zfill(6) for x in instr_list]

    def simulate_instructions(self, instr_list: List[str] or str):
        # Clear stack and overflow flag
        self.stack.clear()
        self.overflow_flag = False
        if isinstance(instr_list, str):
            instr_list = self.rpn_to_instr_list(instr_list)
        instr_list = self.parse_instr_list(instr_list)
        print("Instruction list is: ", instr_list)
        for word in instr_list:
            print(
                "Instruction is:",
                word,
                "Stack is:",
                self.stack,
                "Overflow flag is:",
                self.overflow_flag,
            )
            if isinstance(word, Instruction):
                print("\tRun instruction", word)
                if self.run_instruction(word) == 1:
                    print("Final stack is: ", self.stack)
                    return
            else:
                print("\tPushing", word)
                self.stack.append(word)
                self.overflow_flag = False
            print("\tStack after instruction: ", self.stack)

    def get_operands_from_stack(self, n=2):
        if len(self.stack) < n:
            raise ValueError("Stack underflow")
        else:
            return tuple(self.stack.pop() for _ in range(n))

    def run_instruction(self, instr: Instruction):
        if instr == Instruction.STP:
            return 1
        elif instr == Instruction.DUP:
            ops = self.get_operands_from_stack(1)
            self.stack.append(ops[0])
            self.stack.append(ops[0])
        elif instr == Instruction.DEL:
            print("Do DEL")
        elif instr == Instruction.SWP:
            print("Do SWP")
        elif instr == Instruction.ADD:
            ops = self.get_operands_from_stack()
            result = ops[1] + ops[0]
            if result > 15:
                result = result % 16
                self.overflow_flag = True
            self.stack.append(result)
        elif instr == Instruction.SUB:
            ops = self.get_operands_from_stack()
            result = ops[1] - ops[0]
            if result < 0:
                result = 16 + result
                self.overflow_flag = True
            self.stack.append(result)
        elif instr == Instruction.MUL:
            ops = self.get_operands_from_stack()
            result = ops[1] * ops[0]
            if result < 0:
                result = 16 + result
                self.overflow_flag = True
            self.stack.append(result)
        elif instr == Instruction.DIV:
            ops = self.get_operands_from_stack()
            self.overflow_flag = False
            self.stack.append(ops[1] // ops[0])
        elif instr == Instruction.EXP:
            ops = self.get_operands_from_stack()
            result = ops[1] ** ops[0]
            if result > 15:
                result = result % 16
                self.overflow_flag = True
            self.stack.append(result)
        elif instr == Instruction.MOD:
            self.overflow_flag = False
            ops = self.get_operands_from_stack()
            result = ops[1] % ops[0]
            self.stack.append(result)
        elif instr == Instruction.SHL:
            print("Do SHL")
        elif instr == Instruction.SHR:
            self.overflow_flag = False
            ops = self.get_operands_from_stack()
            result = ops[1] >> ops[0]
            self.stack.append(result)
        elif instr == Instruction.HEX:
            print("Do HEX")
        elif instr == Instruction.FAC:
                print("Do FAC")
            elif instr == Instruction.NOT:
                print("Do NOT")
            elif instr == Instruction.XOR:
                self.overflow_flag = False
                ops = self.get_operands_from_stack()
                result = ops[1] ^ ops[0]
                self.stack.append(result)
            elif instr == Instruction.NOP:
                print("Do NOP")
            elif instr == Instruction.SPEAK:
                print("Do SPEAK")
            return 0


rpn_expr = "4223*+*2/"
instr_list = [
    "001010",
    "010001",
    "010001",
    "010110",
    "011111",
    "000100",
    "011011",
    "000100",
    "011001",
    "000110",
    "011000",
    "100010",
    "110110",
    "101000",
    "110101",
    "010000",
]

sm = StackMachine()
print("1. RPN expression")
sm.simulate_instructions(rpn_expr)
print("\n\n2. instruction list")
sm.simulate_instructions(instr_list)

```

```

# Prints:
# 1. RPN expression
# Instruction list is: [4, 2, 2, 3, <Instruction.MUL: 22>, <Instruction.ADD: 20>, <Instruction.MUL: 22>, 2, <Instruction.DIV: 23>, <Instruction.STP: 16>]
# Instruction is: 4 Stack is: [] Overflow flag is: False
#   Pushing 4
#   Stack after instruction: [4]
# Instruction is: 2 Stack is: [4] Overflow flag is: False
#   Pushing 2
#   Stack after instruction: [4, 2]
# Instruction is: 2 Stack is: [4, 2] Overflow flag is: False
#   Pushing 2
#   Stack after instruction: [4, 2, 2]
# Instruction is: 3 Stack is: [4, 2, 2] Overflow flag is: False
#   Pushing 3
#   Stack after instruction: [4, 2, 2, 3]
# Instruction is: MUL Stack is: [4, 2, 2, 3] Overflow flag is: False
#   Run instruction MUL
#   Stack after instruction: [4, 2, 6]
# Instruction is: ADD Stack is: [4, 2, 6] Overflow flag is: False
#   Run instruction ADD
#   Stack after instruction: [4, 8]
# Instruction is: MUL Stack is: [4, 8] Overflow flag is: False
#   Run instruction MUL
#   Stack after instruction: [32]
# Instruction is: 2 Stack is: [32] Overflow flag is: False
#   Pushing 2
#   Stack after instruction: [32, 2]
# Instruction is: DIV Stack is: [32, 2] Overflow flag is: False
#   Run instruction DIV
#   Stack after instruction: [16]
# Instruction is: STP Stack is: [16] Overflow flag is: False
#   Run instruction STP
# Final stack is: [16]


# 2. instruction list
# 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>]
# Instruction is: 10 Stack is: [] Overflow flag is: False
#   Pushing 10
#   Stack after instruction: [10]
# Instruction is: DUP Stack is: [10] Overflow flag is: False
#   Run instruction DUP
#   Stack after instruction: [10, 10]
# Instruction is: DUP Stack is: [10, 10] Overflow flag is: False
#   Run instruction DUP
#   Stack after instruction: [10, 10, 10]
# Instruction is: MUL Stack is: [10, 10, 10] Overflow flag is: False
#   Run instruction MUL
#   Stack after instruction: [10, 100]
# Instruction is: XOR Stack is: [10, 100] Overflow flag is: False
#   Run instruction XOR
#   Stack after instruction: [110]
# Instruction is: 4 Stack is: [110] Overflow flag is: False
#   Pushing 4
#   Stack after instruction: [110, 4]
# Instruction is: SHR Stack is: [110, 4] Overflow flag is: False
#   Run instruction SHR
#   Stack after instruction: [6]
# Instruction is: 4 Stack is: [6] Overflow flag is: False
#   Pushing 4
#   Stack after instruction: [6, 4]
# Instruction is: MOD Stack is: [6, 4] Overflow flag is: False
#   Run instruction MOD
#   Stack after instruction: [2]
# Instruction is: 6 Stack is: [2] Overflow flag is: False
#   Pushing 6
#   Stack after instruction: [2, 6]
# Instruction is: EXP Stack is: [2, 6] Overflow flag is: False
#   Run instruction EXP
#   Stack after instruction: [0]
# Instruction is: Stack is: [0] Overflow flag is: True
#   Pushing
#   Stack after instruction: [0, ’ ’]
# Instruction is: S Stack is: [0, ’ ’] Overflow flag is: False
#   Pushing S
#   Stack after instruction: [0, ’ ’, ’S’]
# Instruction is: E Stack is: [0, ’ ’, ’S’] Overflow flag is: False
#   Pushing E
#   Stack after instruction: [0, ’ ’, ’S’, ’E’]
# Instruction is: R Stack is: [0, ’ ’, ’S’, ’E’] Overflow flag is: False
#   Pushing R
#   Stack after instruction: [0, ’ ’, ’S’, ’E’, ’R’]
# Instruction is: STP Stack is: [0, ’ ’, ’S’, ’E’, ’R’] Overflow flag is: False
#   Run instruction STP
# Final stack is: [0, ’ ’, ’S’, ’E’, ’R’]
```