Національний університет «Львівська політехніка»

Ісак Володимир

8 грудня 2023 р.

Курсова Робота

З дисципліни «Архітектура Комп'ютерів»

Варіант №9

Зміст

1 Постановка задачі

Адресація	Безадресна
Розрядність шини (Біт)	24
Розмір пам'яті (Байт)	4096
Розмір регістрового файлу	8
Прапорці	ZF, PF
Арифметичні операції	ADD, SUB, INC, DEC, IMUL, IDIV
Логічні/побітові операції	AND, OR, XOR, SHL, SHR, CMP
Операції передачі керування	JE, JNE, JMA

Табл. 1: Дані на проектування

2 Опис стекової та асоцітивної організації пам'яті

У безадресній пам'яті для пошуку інформації в запам'ятовуючому масиві використовують не адреси комірок, а інші принципи. До цього типу пам'яті відноситься асоціативна та стекова пам'яті. [1]

2.1 Асоціативна пам'ять

У пам'яті цього типу пошук потрібної інформації проводиться не за адресою, а за її змістом (за асоціативною ознакою). При цьому пошук за асоціативною ознакою (або послідовно за окремими розрядами цієї ознаки) відбувається паралельно у часі для всіх комірок запам'ятовуючого масиву. В багатьох випадках асоціативний пошук дозволяє істотно спростувати та прискорити обробку даних. Це досягається за рахунок того, що в пам'яті цього типу операція зчитування інформації суміщена з виконанням ряду логічних операцій. Запам'ятовуючий масив має N (n+1)–розрядних комірок. Для вказування зайнятості комірки використовують службовий n-й розряд (0 – комірка вільна, 1 – в комірці записане слово). По вхідній інформаційній шині ШВх в регістр асоціативної ознаки (РгАО) в розряди 0 ÷ n-1 надходить n-розрядний асоціативний запит, а в регістр маски (РгМ) – код маски пошуку, при цьому n-й розряд РгМ установлюється в 0. Асоціативний пошук проводиться лише для сукупності розрядів РгАО, яким відповідає 1 в РгМ (не замасковані розряди РгАО). Для слів, у яких цифри в розрядах збіглися з незамаскованими розрядами РгАО, комбінаційна схема (КС) установлює 1 у відповідні розряди регістра збігання (РгЗб) і 0 в решту розрядів. Комбінаційна схема формування результатів асоціативного звертання (ФС) формує із слова, яке утворилося в РгЗб, сигнали a0, a1, a2, що відповідають випадкам відсутності слів у ЗМ, які задовольняють асоціативні ознаки та наявність одного (a1) і більше (a2) таких слів. Формування вмісту РгЗб та сигналів a0, a1, a2, за вмістом РгАО, РгМ та ЗМ, називається операцією контролю асоціації. Ця операція є складовою частиною операцій

зчитування та запису, хоч вона має самостійне значення. Під час зчитування спочатку проводиться контроль асоціації за асоціативною ознакою в РгАО. Потім при а0=1 зчитування скасовується через відсутність потрібної інформації, при а1=1 зчитується слово, на яке вказує одиниця в РгЗб, при а2=1 в РгІ зчитується слово із комірки, яка має найменший номер серед комірок, позначених 1 в РгЗб. Із РгІ зчитане слово видається на вихідну інформаційну шину (ШВих). Під час запису інформації спочатку відшукується вільна комірка. Для цього виконується операція контролю асоціації при РгАО = 111...10 та РгМ = 000...01, при цьому вільні комірки позначаються 1 в РгЗб. Для запису вибирається вільна комірка з найменшим номером або та, яка довше не використовувалася. В неї записують слово, яке надійшло із ШВх в РгІ. За допомогою операції контролю асоціації можна, не зчитуючи слів з пам'яті, визначити за вмістом РгЗб, скільки в пам'яті слів, які задовольняють асоціативні ознаки, наприклад реалізувати запити, типу скільки студентів мають відмінні оцінки з даної дисципліни. Під час використання відповідних комбінаційних схем в асоціативній пам'яті можуть використовуватись досить складні логічні операції, такі, як пошук більшого (меншого) числа та інше. Відзначимо, що для асоціативної пам'яті необхідні запам'ятовуючі елементи, які дозволяють виконувати зчитування без знищення записаної в них інформації. Через відносно високу вартість асоціативна пам'ять рідко використовується як самостійний вид пам'яті.

2.2 Стекова пам'ять

Стекову пам'ять можна розглядати як сукупність комірок, що утворюють одновимірний масив, у якому сусідні комірки зв'язані одна з одною розрядними ланцюгами передачі слів. Запис нового слова проводиться в верхню комірку (вершину стека – комірку 0), при цьому всі записані раніше слова (включаючи і слова, записані в комірці 0) зсуваються вниз, в сусідні комірки з більшими на 1 номерами. Зчитування можливе тільки з верхньої (нульової) комірки пам'яті, при цьому, якщо проводиться зчитування з видаленням, всі інші слова в пам'яті зсуваються вверх, в сусідні комірки з більшими номерами. В цій пам'яті порядок зчитування слів відповідає правилу: останнім надійшов – першим обслужився. У ряді пристроїв розглянутого типу передбачається також операція простого зчитування слова з нульової комірки (без його видалення і зсуву слова в пам'яті). Іноді стекова пам'ять має лічильник стека (ЛчСт), який показує кількість занесених у пам'ять слів. Сигнал ЛчСт = 0 відповідає порожньому стеку. Пошук і переміщення інформації в СтП здійснюється з допомогою реверсивного лічильника (Лч). Він у будь-який момент часу вказує кількість зайнятих комірок СтП. Під час запису інформації показання Лч збільшуються на одиницю, а вміст СтП переміщується на одну комірку вниз, а при зчитуванні – показання Лч зменшується на одиницю і вміст СтП переміщується на одну комірку вверх. Оперативний запам'ятовуючий пристрій (ОЗП) має зв'язок тільки з верхньою коміркою пам'яті (Рг1), а процесор – з двома верхніми (Рг1 і Рг2). Операція, яка задається програмою, виконується над вмістом двох верхніх

комірок – Рг1 і Рг2, а результат операції завжди залишається в Рг2. Якщо Рг1 вільний, то перед виконанням наступної операції вміст Рг2 передається в Рг1, із Рг3 в Рг2 і т.д. (вміст СтП переміщується на одну комірку вверх), а показання Лч зменшуються на одиницю. Стекова пам'ять (її ще називають магазинна пам'ять) широко застосовується як в обчислювальних системах високої продуктивності, так і в міні- і мікро-ЕОМ, і пропонує використання безадресних команд обробки масивів даних, які утворюють стек.

3 Функціональна схема комп'ютера

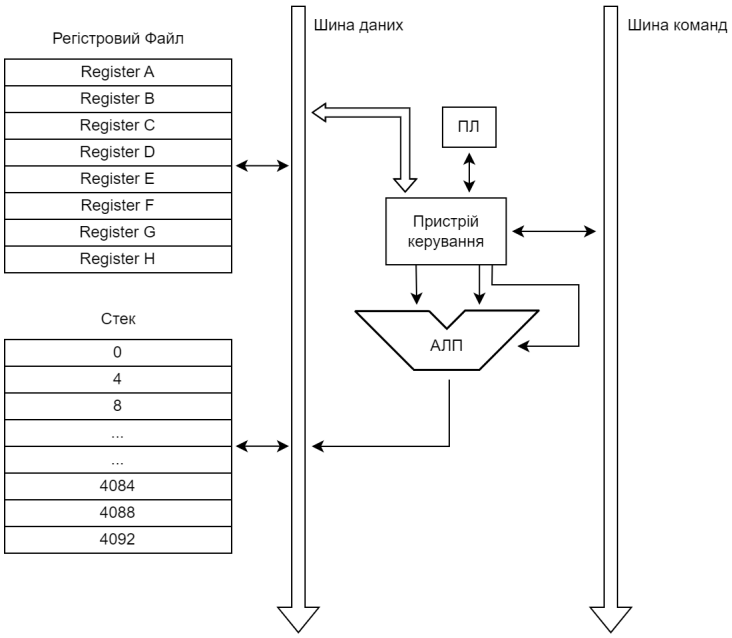

Рис 1. Функціональна схема комп'ютера

4 Вихідний код

```
class CPU:
    def __init__(self):
        self._stack_ = []
        self._lines_ = []
        self._bus_length_ = 24

        self._pc_ = 0
        self._stack_pointer_ = -1
        self._delay_ = 0

        self._zf_ = 0
        self._pf_ = 0

        self._registers = [0 for _ in range(8)]

        self._register_mapping_ = {
            "reg_a": 0,
            "reg_b": 1,
            "reg_c": 2,
            "reg_d": 3,
            "reg_e": 4,
            "reg_f": 5,
            "reg_g": 6,
            "reg_h": 7,
        }

        self._command_mapping_ = {
            "push": self.push,
            "pop": self.pop,
            "add": self.add,
            "sub": self.sub,
            "inc": self.inc,
            "dec": self.dec,
            "imul": self.imul,
            "idiv": self.idiv,
            "and": self.and_operation,
            "or": self.or_operation,
            "xor": self.xor,
            "shl": self.shl,
            "shr": self.shr,
            "cmp": self.cmp,
            "je": self.je,
            "jma": self.jma,
            "jne": self.jne,
            "skip": self.skip
        }

    @property
    def mapping(self) -> dict:
        return self._command_mapping_

    def execute(self, operation: str, *arguments) -> None:
        self._command_mapping_.get(operation)(*arguments)
```

```
def run(self, lines: list[str, ...]) -> None:
    self._lines_ = lines

    while self._pc_ < len(lines):
        self.execute(*lines[self._pc_])

        self._pc_ += 1

def get_register(self, register: str) -> int:
    return self._registers[self._register_mapping_.get(register
)]

def info(self) -> None:
    print("\nStack\n")

    for index, value in enumerate(self._stack_):
        print(f"{index}: {value}")

    print("\nRegisters\n")

    for index, value in enumerate(self._registers):
        print(f"Register {index}: {value}")

def push(self, value: str = None) -> None:
    if value:
        if value.replace("-", "").isnumeric():
            self._stack_.append(int(value))
        else:
            raise Exception("Incorrect value")
    else:
        self._stack_.append(self._registers[0])

        self._registers.pop(0)
        self._registers.append(0)

    self._stack_pointer_ += 1

def pop(self) -> None:
    self._registers = [self._stack_.pop(self._stack_pointer_)]
+ self._registers[:-1]

    self._stack_pointer_ -= 1

def _pop_(self) -> None:
    self._stack_.pop(-1)
    self._stack_pointer_ -= 1

def add(self, first: str = None, second: str = None,
destination: str = None) -> None:
    if all((first, second, destination)):
        self._registers[self._register_mapping_.get(destination
)] = self.get_register(first) + self.get_register(second)

        return
    elif any((first, second, destination)):
        raise Exception("One of arguments unfilled")
```

```
    self._stack_[self._stack_pointer_ - 1] += self._stack_[self
._stack_pointer_]

    self._pop_()

def sub(self, first: str = None, second: str = None,
destination: str = None) -> None:
    if all((first, second, destination)):
        self._registers[self._register_mapping_.get(destination
)] = self.get_register(first) - self.get_register(second)

        return
    elif any((first, second, destination)):
        raise Exception("One of arguments unfilled")

    self._stack_[self._stack_pointer_ - 1] -= self._stack_[self
._stack_pointer_]

    self._pop_()

def inc(self, first: str = None) -> None:
    if first:
        self._registers[self._register_mapping_.get(first)] =
self.get_register(first) + 1

        return
    self._stack_[self._stack_pointer_] += 1

def dec(self, first: str = None) -> None:
    if first:
        self._registers[self._register_mapping_.get(first)] =
self.get_register(first) - 1

        return
    self._stack_[self._stack_pointer_] -= 1

def imul(self, first: str = None, second: str = None,
destination: str = None) -> None:
    if all((first, second, destination)):
        self._registers[self._register_mapping_.get(destination
)] = self.get_register(first) * self.get_register(second)

        return
    elif any((first, second, destination)):
        raise Exception("One of arguments unfilled")

    self._stack_[self._stack_pointer_ - 1] *= self._stack_[self
._stack_pointer_]

    self._pop_()

def idiv(self, first: str = None, second: str = None,
destination: str = None) -> None:
    if all((first, second, destination)):
        self._registers[self._register_mapping_.get(destination
)] = self.get_register(first) // self.get_register(second)
```

```
        return
    elif any((first, second, destination)):
        raise Exception("One of arguments unfilled")

    self._stack_[self._stack_pointer_ - 1] //= self._stack_[
self._stack_pointer_]

    self._pop_()

def and_operation(self, first: str = None, second: str = None,
destination: str = None) -> None:
    if all((first, second, destination)):
        self._registers[self._register_mapping_.get(destination
)] = self.get_register(first) & self.get_register(second)

        return
    elif any((first, second, destination)):
        raise Exception("One of arguments unfilled")

    self._stack_[self._stack_pointer_ - 1] &= self._stack_[self
._stack_pointer_]

    self._pop_()

def or_operation(self, first: str = None, second: str = None,
destination: str = None) -> None:
    if all((first, second, destination)):
        self._registers[self._register_mapping_.get(destination
)] = self.get_register(first) | self.get_register(second)

        return
    elif any((first, second, destination)):
        raise Exception("One of arguments unfilled")

    self._stack_[self._stack_pointer_ - 1] |= self._stack_[self
._stack_pointer_]

    self._pop_()

def shl(self, first: str = None, second: str = None,
destination: str = None) -> None:
    if all((first, second, destination)):
        self._registers[self._register_mapping_.get(destination
)] = self.get_register(first) << self.get_register(second)

        return
    elif any((first, second, destination)):
        raise Exception("One of arguments unfilled")

    self._stack_[self._stack_pointer_ - 1] = self._stack_[self.
_stack_pointer_ - 1] << self._stack_[
        self._stack_pointer_]

    self._pop_()

def shr(self, first: str = None, second: str = None,
destination: str = None) -> None:
```

```
    if all((first, second, destination)):
        self._registers[self._register_mapping_.get(destination
)] = self.get_register(first) >> self.get_register(second)

        return
    elif any((first, second, destination)):
        raise Exception("One of arguments unfilled")

    self._stack_[self._stack_pointer_ - 1] = self._stack_[self.
_stack_pointer_ - 1] >> self._stack_[
        self._stack_pointer_]

    self._pop_()

def xor(self, first: str = None, second: str = None,
destination: str = None) -> None:
    if all((first, second, destination)):
        self._registers[self._register_mapping_.get(destination
)] = self.get_register(first) ^ self.get_register(second)

        return
    elif any((first, second, destination)):
        raise Exception("One of arguments unfilled")

    self._stack_[self._stack_pointer_ - 1] ^= self._stack_[self
._stack_pointer_]

    self._pop_()

def cmp(self, first: str = None, second: str = None) -> None:
    if all((first, second)):
        self._zf_ = int((self.get_register(first) - self.
get_register(second)) == 0)
        self._pf_ = int((self.get_register(first) - self.
get_register(second)) > 0)

        return
    elif any((first, second)):
        raise Exception("One of arguments unfilled")

    self._zf_ = int((self._stack_[self._stack_pointer_ - 1] -
self._stack_[self._stack_pointer_]) == 0)
    self._pf_ = int((self._stack_[self._stack_pointer_ - 1] -
self._stack_[self._stack_pointer_]) > 0)

def je(self, value: str) -> None:
    if (self._pc_ + self._stack_[self._stack_pointer_] not in
range(len(self._lines_))) and not value:
        raise Exception("Jump value out of bounds")

    if value:
        if not self._zf_:
            self._pc_ = int(value) - 2

            return
    else:
        if not self._zf_:
```

```
                self._pc_ += self._stack_[self._stack_pointer_]

            self._pop_()

    def jma(self, value: str) -> None:
        if (self._pc_ + self._stack_[self._stack_pointer_] not in
    range(len(self._lines_))) and not value:
            raise Exception("Jump value out of bounds")

        if value:
            if not self._zf_:
                self._pc_ = int(value) - 2

                return
        else:
            if not self._zf_:
                self._pc_ += self._stack_[self._stack_pointer_]

            self._pop_()

    def jne(self, value: str) -> None:
        if self._stack_:
            if (self._pc_ + self._stack_[self._stack_pointer_] not
    in range(len(self._lines_))) and not value:
                raise Exception("Jump value out of bounds")

        if value:
            if not self._zf_:
                self._pc_ = int(value) - 2

                return
        else:
            if not self._zf_:
                self._pc_ += self._stack_[self._stack_pointer_]

            self._pop_()

    def skip(self) -> None:
        pass


class Simulator:
    def __init__(self):
        self._CPU_ = CPU()

        self._lines_ = []

    def preprocess(self, path: str) -> None:
        with open(path, "r") as file:
            for index, line in enumerate(file.readlines()):
                if line[0] == ";" or line[0] == "\n":
                    self._lines_.append(["skip"])

                    continue

                splitted = " ".join(line.split()).split(" ")
```

```
                command = splitted[0]

                if command not in self._CPU_.mapping:
                    raise Exception("Unknown command")

                self._lines_.append(splitted)

    def run(self, path: str, info: bool = True) -> None:
        self.preprocess(path)

        self._CPU_.run(self._lines_)

        if info:
            self.info()

    def info(self) -> None:
        self._CPU_.info()


def main():
    simulator = Simulator()

    simulator.run("program.txt")


if __name__ == '__main__':
    main()
```

5 Приклад тестової програми

```
; reg_d = ((8 + 6) - 2) * 100 / 13

push 8
push 6
add
push 2
sub
push 100
imul
push 13
idiv
pop

; 123 + 321 on stack

push 123
push 321

; reg_c = 5!

add
push 1
pop
push 1
pop
push 5
pop
imul reg_a reg_c reg_c
sub reg_a reg_b reg_a
cmp reg_a reg_f
jne 28
```

6 Результат роботи симуляції

```
Stack

0: 444

Registers

Register 0: 0
Register 1: 1
Register 2: 120
Register 3: 92
Register 4: 0
Register 5: 0
Register 6: 0
Register 7: 0
```

7 Висновок

В ході виконання курсової роботи було досліджено стекову та асоціативну організацію пам'яті, вивчено принципо конструювання та побудови комп'ютера на їх основі. Складена функціональна схема роботи системи та безпосередньо комп'ютерну програму-симулятор. Написано тестову програму, яка демострує певний функціонал системи та наведено результати роботи. Вихідний код тестової програми та безпосередньо симулятора наведено також.

8 Посилання

[1] - Вовк П. Б. Лекція 1. Луцький національний технічний університет.
[2] - Мельник А. О. Архітектура комп'ютера. Наукове видання. – Луцьк: Волинська обласна друкарня, 2008. – 470 с.