Materials

Ćw2 Programowanie BPMK -- różne rodzaje adresowania.

Omawiane zagadnienia j.w.

Improvements, part I: various addressing modes

Extend set of instructions by INC and DEC

Instruction list missed instruction to increment or decrement given value. Without this, instead of one instruction, three have to be used, sequence like:

```
CPA X ; X - address of the value to increment
ADD Y ; add value from address Y (very often simply equal to 1)
STO X ; store X incremented by Y
```

It seems to be a detail but this type of operation is very, very common and that's why it's good to extend instruction list with two instructions:

```
01xxx INC address
02xxx DEC address
```

where `address` is coded on three digits `xxx` in machine instruction.

For example:

```
0101 01034 ; INC 34
```

means: increas value at address 34 by 1.

In this case I intentionaly avoid the number 9 as the first digit in the code (having in mind that 9 was reserved for extensions) to get more "handy" pattern for instructon numbering -- see next part of this chapter.

Introduce immediate addressing

Addressing mode used so far is a type of **direct addressing** e.g addressing which uses operand as a value of memory address where actual argument is stored:

```
+-code for ADD
|
| +-operand (0123)
| |
| |       Address     Value
30123         ...  |        |
  |         (0122) |        |
  +-------> (0123) | 00035  |
            (0124) |        |
              ...  |        |
```

In the example above instruction `ADD` adds value `35` at the address `0123`. In other words, operand points to a memory cell and to execute this type of instruction **two memory accesses are needed**: one to get instruction and second to get value.

There are situation when it is useful to treat operand not as memory address but as value. For example, when you want to add 5 to value in accumulator, instead of:

```
ADD 35 ; Assume that value 5 is stored at address 35
```

more intuitive is to write:

```
ADD 5 ; 5 is not an address but value
```

The question is:

- How to distinguish between these two variants of `ADD` instruction?
- When operand treat as address and when as value?

To overcome this difficulties you will use the following convention is. Notation:

```
mnemonic operand
```

means: executing instruction **mnemonic** as a value (as an argument) use number taken from the address **operand**, while notation:

```
mnemonic (operand)
```

means: executing instruction **mnemonic** as a value (as an argument) use number **operand**.

This leads to the second type of addressing -- addressing when value is "**in**" instruction and is accessible **immediately** just after instruction read is complete -- so called **immediate addressing**.

```
+-code for ADD
|
| +-operand (0123) - value of the argument
| |
| |
30123
```

Introducing this type of addressing entails new codes for instruction because computers, like humans, have to distinguisg variants of addressing:

```
              Direct addressing   Immediate addressing
Human         ADD 35              ADD (5)

Computer      30035               92305

9xxxx - to indicate extension of basic instruction set
x2xxx - addressing mode (2 for immediate, 1 byte length)
xx3xx - code for addition in basic instructions set
xxx05 - immediate value - notice that this value is stored "in" ins
```

Notice that value `5` is stored "in" instruction and there is no need of an extra memory access -- it means that this type of instruction is faster.

Two bytes instructions

Unfortunately immediate addressing solves one problem and at the same time generates a new one -- what about instruction like:

```
ADD (128)
```

It is not possible to "squeeze" value 128 and put "into" instruction like it was in case of value

5.

The solution for this is to put another code for addition which assumes that value of the argument is put just after instruction, like in the following example:

```
address   value
x         93300 - add
x + 1     00128 - value for add of code 9230

9xxxx - to indicate extension of basic instruction set
x3xxx - addressing mode (3 for immediate, 2 byte length)
xx3xx - code for addition in basic instructions set
```

This is in some sens a mixture of direct and immediate addresing: you have two memory access (one for instruction and the second to get value) but argument is always located next to instruction (after instruction) -- you could say that you (almost) immediately know where the argument is. If you take into account that modern CPU reads always few memory cell at once, it may turn out that when first number (instruction) `93300` is read, `00128` will also be read and will be available for CPU just after it finish processing instruction.

When you use assembler, you simply write in your code:

```
ADD (7)
ADD (342)
ADD (9)
```

and you don't have to care about all low level details. The compiler translates your code into machine code. Even though both instructions mean adding in the same addressing mode, each will have a different machine code generated:

```
address value
x       92307      ADD (7)
x + 1   93300      ADD (342)
x + 2   00342
x + 3   92309      ADD (9)
```

Excercise 1

Calculate the dot product (sometimes scalar product or inner product) of two vectors of length `N` (for simplicity you can assume for example `N=10` or `N=30` but in general it could be any natural number).

Solution 1.1

The solution is possible however it's not very sophisticated with multiple code repeated:

```
0010 ?  ; a1
0011 ?  ; a2
...
0019 ?  ; a10
0020 ?  ; b1
0021 ?  ; b2
...
0029 ?  ; b10
0030 ?  ; result
...
0050 CPA 10 ; a1 * b1
0051 MUL 20
0052 STO 30
0053 CPA 11 ; a2 * b2
0054 MUL 21
0055 ADD 30
0056 STO 30
0057 CPA 12 ; a3 * b3
0058 MUL 22
0059 ADD 30
0060 STO 30
0061 CPA 13 ; a4 * b4
0062 MUL 23
0063 ADD 30
0064 STO 30
0065 CPA 14 ; a5 * b5
0066 MUL 24
0067 ADD 30
0068 STO 30
0069 CPA 15 ; a6 * b6
0070 MUL 25
0071 ADD 30
0072 STO 30
0073 CPA 16 ; a7 * b7
0074 MUL 26
0075 ADD 30
0076 STO 30
0077 CPA 17 ; a8 * b8
0078 MUL 27
0079 ADD 30
0080 STO 30
0081 CPA 18 ; a9 * b9
0082 MUL 28
```

```
0083 ADD 30
0084 STO 30
0085 CPA 19 ; a10 * b10
0086 MUL 29
0087 ADD 30
0088 STO 30
0089 HLT
```

Introduce indirect addressing

The problem of calculating dot product seems to be unsolvable in satisfactory way without concept of memory **indirect addressing**.

Notation:

```
mnemonic operand
```

means: executing instruction `mnemonic` as an address of the argument use `operand`, while notation:

```
mnemonic [operand]
```

means: executing instruction `mnemonic` as an address of the argument use value taken from the address `operand`.

This leads to the third type of addressing -- addressing when value is "**pointed**" by value at given address -- so called **indirect addressing**:

```
+-code for ADD [x] ->--+
|                      +->-- finally: ADD [6] and it adds 123
|   +-operand (6) -->--+              to acumulator
|   |
|   |       Address     Value
94306          ...  |        |
    |        (0005) |        |
    +------> (0006) | 00009  | ---+
             (0007) |        |    |
               ...  |        |    |
             (0009) | 00123  | <--+
               ...  |        |

9xxxx - to indicate extension of basic instruction set
x4xxx - addressing mode (4 for indirect)
xx3xx - code for addition in basic instructions set
```

You can think about `[ ]` "operator" as an substitution: having instruction `mnemonic [operand]` take first value from the address `operand`, name it `val`, substitute `[operand]` by `val` and finally execute instruction `mnemonic val`.

```
ADD [5] -> value at address 5 = 36 -> ADD 36
```

To execute instruction **in this addressing mode three memory accesses are needed**: first to get instruction, second to get address and third to get value at that address.

So this is the "slowest" instruction because it requires the most references to memory of all addressing known so far. On the other hand, it gives you possibility to access much wider memory space because to point an address you can use a full number stored in memory cell (`00009` -- five-digit number in example above), while in direct addressing one digit is reserved for instruction number (so you can specify only four-digit numbers). Other words in direct addressing you can specify addresses from `0000` to `9999`, while in indirect addressing from `00000` to `99999`.

Solution 1.2 (second approach)

```
Address Value      Instruction
0001    00010       ; Address of the first component of vector 1
0002    00020       ; Address of the first component of vector 2
0003    00000       ; Result
0004    00010       ; n - length of vector
...
0010    xxxxx       ; First component of vector 1
...
0019    xxxxx       ; Last component of vector 1
0020    xxxxx       ; First component of vector 2
...
0029    xxxxx       ; Last component of vector 2

0030    10004       CPA 4
0031    80040       BRZ 40
0032    94101       CPA [1]
0033    94502       MUL [2]
0034    30003       ADD 3
0035    20003       STO 3
0036    01001       INC 1
0037    01002       INC 2
0038    02004       DEC 4
0039    60030       BRA 30
0040    00000       HLT
```