Materials

Ćw3 Programowanie BPMK - etykiety i rejestr flagowy.

Omawiane zagadnienia j.w.

Improvements, part II: labels

Problem with code reallocation

Recall last code from previous part -- code to compute the dot product of two vectors:

```
Address    Instruction
0001       10  ; Address of the first component of vector 1
0002       20  ; Address of the first component of vector 2
0003       0   ; Result
0004       10  ; n - length of vector
...
0010       1   ; First component of vector 1
...
0019       10  ; Last component of vector 1
0020       10  ; First component of vector 2
...
0029       1   ; Last component of vector 2

0030       CPA 4
0031       BRZ 40
0032       CPA [1]
0033       MUL [2]
0034       ADD 3
0035       STO 3
0036       INC 1
0037       INC 2
0038       DEC 4
0039       BRA 30
0040       HLT
```

Assume now that you want to reallocate the code to a different area of your memory, for

example you want to start the code not at address `1` but `101` ("shift" all the code by 100). You may want to simply enumerate all addresses like this:

```
Address    Instruction
0101       ; Address of the first component of vector 1
0102       ; Address of the first component of vector 2
0103       ; Result
0104       ; n - length of vector
...
0110       ; First component of vector 1
...
0119       ; Last component of vector 1
0120       ; First component of vector 2
...
0129       ; Last component of vector 2

0130       CPA 104
0131       BRZ 140
0132       CPA [101]
0133       MUL [102]
0134       ADD 103
0135       STO 103
0136       INC 101
0137       INC 102
0138       DEC 104
0139       BRA 130
0140       HLT
```

Initial solution is correct, but when the code is reallocated into other place in the memory, instruction names, mnemonics, stays the same, but their binary code changes significantly and in consequences addresses are not correct because it's not enough to shift all of them by 100.

Explanation is as follow: the same instruction may have different length in bytes depending on the values of their operands. Code:

```
CPA [1]
```

generates machine code different than:

```
CPA [101]
```

In the first case you have:

```
Address Value      Instruction
x       94101       CPA [1]
```

while in the second instruction `CPA` requires two bytes:

```
Address Value      Instruction
x       95100       CPA [101]
x+1     00101
```

In consequence the binary code changes (but the assembler stays the same, except addresses you use):

```
Address Value      Instruction
0101    00010       ; Address of the first component of vector 1
0102    00020       ; Address of the first component of vector 2
0103    00000       ; Result
0104    00010       ; n - length of vector
...
0110    xxxxx       ; First component of vector 1
...
0119    xxxxx       ; Last component of vector 1
0120    xxxxx       ; First component of vector 2
...
0129    xxxxx       ; Last component of vector 2

0130    10004       CPA 104
0131    80040       BRZ 142    ; Must be 142 not 140!!!
0132    94100       CPA [101]
0133    00101                  ; "Extra" byte
0134    94700       MUL [102]
0135    00102                  ; "Extra" byte
0136    30003       ADD 103
0137    20003       STO 103
0138    01001       INC 101
0139    01002       INC 102
0140    02004       DEC 104
0141    60030       BRA 130
0142    00000       HLT
```

The source of the problem with variable length instructions is that you have to use real addresses in menmonics, like `CPA [101]`. Solution seems to be quite natural: stop using explicit addresses. Instead use, **labels** to indicate locations in the memory and let the compiler to translate your set of assembler mnemonic instruction into adequate machine code and (re)calculate all required addresses.

With label you may code initial solution as:

```
Address    Instruction
.data 1              ; Start data block at address 0
v1:        xxxxx     ; First component of vector 1
           ...
           xxxxx     ; Last component of vector 1
v2:        xxxxx     ; First component of vector 2
           ...
           xxxxx     ; Last component of vector 2
a_v1:      v1        ; Address of the first component of vector 1
a_v2:      v2        ; Address of the first component of vector 2
result:    0         ; Result
vec_len:   10        ; n - length of vector

.code 30             ; Start code block at address 30

begin:     CPA vec_len
           BRZ end
           CPA [a_v1]
           MUL [a_v2]
           ADD result
           STO result
           INC a_v1
           INC a_v2
           DEC vec_len
           BRA begin
end:       HLT
```

"Shifted" version of this code is:

```
Address    Instruction
.data 101           ; Start data block at address 0
v1:        xxxxx    ; First component of vector 1
           ...
           xxxxx    ; Last component of vector 1
v2:        xxxxx    ; First component of vector 2
           ...
           xxxxx    ; Last component of vector 2
a_v1:      v1       ; Address of the first component of vector 1
a_v2:      v2       ; Address of the first component of vector 2
result:    0        ; Result
vec_len:   10       ; n - length of vector

.code 130           ; Start code block at address 130

begin:     CPA vec_len
           BRZ end
           CPA [a_v1]
           MUL [a_v2]
           ADD result
           STO result
           INC a_v1
           INC a_v2
           DEC vec_len
           BRA begin
end:       HLT
```

As you can see, the only thing which has changed is:

```
.data 1   ->  .data 101
.code 30  ->  .code 130
```

The job of recalculating addresses and generating correct machine code has been transferred to compiler.

Now it is instructive to write again some of the program from previous chapters but replacing raw addresses with labels.

Solution of xxx from chapter yyy

```

```

Improvements, part III: flag register

Consider now a following sequence of instructions:

```
DEC counter
CPA counter
BRN end
```

The idea behind this is very simple: decrease variable (an iterator) and if it is negative (or zero if you use `BRZ`) then jump somewhere. The odd thing is that after you decrease your counter with `DEC` instruction you have to load it with `CPA` into accumulator because jump instructions can work only on values stored in accumulator.

You can solve this if you take a following agreement: every numerical instruction (`INC`, `DEC`, `ADD`, `SUB`, `MUL`) after execution sets some dedicated memory cells (registers) -- called **flags** -- located in CPU (like accumulator is located in CPU):

- `ZF`: **Zero Flag** this flag is set to 1 if last instruction's result is equal to zero, othervise is set to 0;
- `NF`: **Negative Flag** this flag is set to 1 if last instruction's result is neqative, othervise is set to 0.

Now you can introduce new set of jump instructions:

- `BRNF`: jump if `NF` flag is set (is equal to 1);
- `BRZF`: jump if `ZF` flag is set.

Having them sequence:

`DEC counter CPA counter BRN end`

can be replace by more intuitive sequence:

```
DEC counter
BRNF end
```