PRCMP PL04

Registo de estado e ordenação de bytes

May 2023

1 Status register

A status register is a set of status flags that provide information about the status of the processor. Typically, flags are updated (set or cleared) upon an instruction executed by the Arithmetic Logic Unit (ALU). Status flags allow an instruction to act based on the result of the previous instruction.

The most common flags are:

- Z – the *zero* flag
- S – the *sign* flag
- C – the *carry* flag
- O – the *overflow* flag

The electronic implementation of each ALU instruction includes specific circuitry to update these flags.

Note that the values of flags C and O depend on the operation performed, usually resulting from different rules for different arithmetic operations.

While the value of the carry flag is straightforward – a sum generated a carry bit, or a subtraction generated a borrow bit – the overflow flag has more complex rules, depending on the arithmetic operation. For simplicity, the overflow rules for addition and subtraction are summarized below.

Overflow Rule for addition: If 2 two's complement signed integers are added, and they both have the same sign (both positive or both negative), then overflow occurs if and only if the result has the opposite sign. Overflow never occurs when adding operands with different signs.

Overflow Rule for Subtraction: If 2 two's complement signed integers are subtracted, and their signs are different, then overflow occurs if and only if the result has the same sign as the subtrahend. *Note: the subtrahend is what is being subtracted; the minuhend is what it is being subtracted from:*

$$minuhend - subtrahend = result$$

Questions

1. What is the function of each of the following flags in the status register?
   (a) *Zero* flag.
   (b) *Sign* flag.
   (c) *Carry* flag.
   (d) *Overflow* flag.
2. How does the execution of each of the following instructions update the status register of an 8-bit processor?
   (a) 0x02 + 0x60

(b) `0xF0` + `0x10`

(c) `0x90` + `0x20`

(d) `0xFF XOR 0x00`

(e) `0x80 OR 0x0F`

(f) `0x0F AND 0x80`

(g) `0x70` + `0x20`

(h) `0x9C` − `0x70`

3. Assume that on an 8-bit processor, the word `0b01000000` is shifted one bit to the left.
   (a) Discuss if and how the ALU should update the *overflow* flag.
   (b) Assume the ALU sets the *overflow* flag after executing the instruction. Is the flag meaningful if the word is an unsigned integer?
4. Think about the possible results of the overflow flag when adding two signed numbers: $r = a + b$.
   (a) Construct a truth table where the variables A, B and R are the most significant bit of words a, b, and r, respectively.
   (b) *(Optional)* Can you derive from the truth table a logical function to determine overflow on a signed integer addition?
5. Do the same reasoning exercise, thinking about the possible results of the overflow flag when subtracting two signed numbers: $r = a - b$.
   (a) Construct a truth table where the variables A, B and R are the most significant bit of words a, b, and r, respectively.
   (b) *(Optional)* Can you derive from the truth table a logical function to determine overflow on a signed integer subtraction?

2 Endianess

Current computer memories have an 8-bit width, meaning each memory cell stores a word of 8 bits. Each memory cell has a unique address used to access (to read or write) its contents.

However, many of the values in practical programs do not fit 8 bits, and most program variables store 32- or 64-bit integers. Consequently, a value longer than 8-bits is stored in multiple sequential addresses.

There are two approaches on storing multiple byte words in memory:

- big-endian, where the least significant byte (LSB) is stored in the highest (big) address
  decreasing numeric significance with increasing memory addresses
- little-endian, where the LSB is stored in the lowest (little) address
  increasing numeric significance with increasing memory addresses.

Questions

6. A 64-bit variable is allocated in memory starting at address `0x1000`.
   (a) Knowing that any variable is always allocated to consecutive addresses to match its size, what is the range of addresses used by the mentioned variable?
   (b) Which address stores the most significant byte if the computer uses big-endian?
   (c) And which address stores the most significant byte if the computer uses little-endian?
7. Assume a computer that uses big-endian and stores a given variable x starting from address `0x5000`. Distribute the bits of x on the necessary addresses if this variable is:

(a) (8-bit) $x = 0\text{x}\,95$.

(b) (16-bit) $x = 0\text{x}\,1\text{A}\,2\text{B}$

(c) (32-bit) $x = 0\text{x}\,\text{A0}\,\text{B0}\,\text{C0}\,\text{D0}$

(d) (64-bit) $x = 0\text{x}\,\text{FF}\,00\,00\,00\,00\,00\,00\,11$

8. Now, assume another computer that uses little-endian, instead. Again, the given variable x is allocated to memory starting from address 0x5000. Distribute the bits of x on the necessary addresses if this variable is:

(a) (8-bit) $x = 0\text{x}\,0\text{F}$.

(b) (16-bit) $x = 0\text{x}\,\text{A1}\,\text{B2}$

(c) (32-bit) $x = 0\text{x}\,10\,20\,30\,40$

(d) (64-bit) $x = 0\text{x}\,\text{FF}\,00\,00\,00\,00\,00\,00\,11$

3 Solutions

1. (a) Indicates that the result of the last ALU operation is zero.

   (b) Indicates that the result of the last ALU operation is negative if the result is signed.

   (c) Indicates that the result of the last unsigned integer addition/subtraction generated a carry/borrow from the most significant position. It is an overflow condition exclusively for operations with unsigned integers.

   (d) Indicates that the result of the last signed ALU operation generated a result that does not fit the number of bits used for the result.

2. (a) `0x02` + `0x60` → `0x62`
   Z: 0 S: 0 C: 0 O: 0

   (b) `0xF0` + `0x10` → `0x00`
   Z: 1 S: 0 C: 1 O: 0

   (c) `0x90` + `0x20` → `0xB0`
   Z: 0 S: 1 C: 0 O: 0

   (d) `0xFF XOR 0x00` → `0xFF`
   Z: 0 S: 1 C: 0 O: 0

   (e) `0x80 OR 0x0F` → `0x8F`
   Z: 0 S: 1 C: 0 O: 0

   (f) `0x0F AND 0x80` → `0x00`
   Z: 1 S: 0 C: 0 O: 0

   (g) `0x70` + `0x20` → `0x90`
   Z: 0 S: 1 C: 0 O: 1

   (h) `0x9C` − `0x70` → `0x2C`
   Z: 0 S: 0 C: 0 O: 1

3. (a) If the shift operation is strictly bitwise (logical, not arithmetic, not considering the bit word as a number) then the overflow condition has no meaning. However, if the left shift is used to simulate a multiplication of a signed integer by 2, then setting the overflow flag would be helpful to indicate that the sign changed and, consequently, the result is wrong.

   (b) No, because the resulting unsigned number would be correct, as the most significant bit would not represent the sign of the number. Thus, a programmer should not consider the *overflow* flag in this case.

4. (a)

   (b)

5. (a)

   (b)

6. (a) A 64-bit variable requires 8 bytes of storage: $64 \div 8 = 8$ bytes. Consequently, this variable is stored in addresses starting from `0x1000` to `0x1007`.

   (b) The most significant byte is stored at address `0x1000` on a computer that uses big-endian.

   (c) The most significant byte is stored at address `0x1007` on a computer that uses little-endian.

7. (a) `0x5000 : 95`

   (b) `0x5000 : 1A`
   `0x5001 : 2B`

(c) 0x5000 : A0
0x5001 : B0
0x5002 : C0
0x5003 : D0

(d) 0x5000 : FF
0x5001 : 00
0x5002 : 00
0x5003 : 00
0x5004 : 00
0x5005 : 00
0x5006 : 00
0x5007 : 11

8. (a) 0x5000 : 0F

(b) 0x5000 : B2
0x5001 : A1

(c) 0x5000 : 40
0x5001 : 30
0x5002 : 20
0x5003 : 10

(d) 0x5000 : 11
0x5001 : 00
0x5002 : 00
0x5003 : 00
0x5004 : 00
0x5005 : 00
0x5006 : 00
0x5007 : FF