Arithmetic

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Addition and Subtraction

- Two instructions used for additions and subtractions: add and sub
- Both instructions can be used on a pair of signed numbers or on a pair of unsigned numbers
  - One of the big advantages of 2's complement storage
  - **No mixing of signed and unsigned numbers**
- IMPORTANT: The CPU does not know whether numbers stored in registers are signed or unsigned!
  - You, the programmer, must keep your own interpretation of the number consistent throughout your program
  - The CPU will happily add whatever registers together using binary addition
- These two instructions each may set some bits of the FLAG register:
  - The *carry* bit
  - The *overflow* bit
  - The *zero* bit (=1 if the result is equal to zero)
  - The *sign* bit (=1 if the result is negative)

The Magic of 2's Complement

- I have two 1-byte values, A3 and 17, and I add them together:

  A3 + 17 = BA

- If my interpretation of the numbers is unsigned:
  - A3h = 163d
  - 17h = 23d
  - BAh = 186d
  - and indeed, 163d + 23d = 186d
- If my interpretation of the numbers is signed:
  - A3h = -93d
  - 17h = 23d
  - BAh = -70d
  - and indeed, -93d + 23d = -70d
- So, as long as I stick to my interpretation, the binary addition will do the right thing.... amazing!
  - Same thing for the subtraction

Overflow???

- Generally speaking, overflow occurs when the result of an arithmetic operation generates a result that's "out of range"
- This happens because a register has a limited number of bits, which means that our interpretation of a number comes with a valid range
- For instance
  - adding 1-byte unsigned quantity 240d to 1-byte unsigned quantity 100d will lead to an overflow because 340d > 255d
  - subtracting 1-byte unsigned quantity 240d from 1-byte unsigned quantity 100d will lead to an overflow because -140d < 0d
  - adding 1-byte signed quantity 100d to 1-byte signed quantity 120d will lead to an overflow because 220d > 127d
  - etc.
- Question: how do we detect overflow in a program?
  - Important otherwise we could be working with bogus numbers
- It depends on whether numbers are signed or unsigned...

Overflow for Unsigned Operations

- There is an overflow with an unsigned operation (i.e., on unsigned quantities) if the carry bit is set
- If the carry bit is set, that means we'd need a larger quantity to hold the result
  - This also works for subtractions (instead of a carry, we have a "borrow", but it's still set in the carry bit)
- 1-byte Example (all in hex):
  - FF + 02 Carry is set (result would be 101h)
    - 255 + 2 > 255
  - 01 - 02 Carry is set (result cannot be negative)
    - 1 - 2 < 0
  - 8A - 0F Carry is not set (result is 7Bh)
    - 138 - 15 = 123

Overflow for Signed Operations

- There is an overflow with a signed operation (i.e., on signed quantities) if the overflow bit is set
  - This bit is set when the sign of the result does not agree with the signs of the operands
- 1-byte Example (all in hex, same as before):
  - FF + 02 Overflow is not set (result is 01h)
    - -1 + 2 = +1
  - 01 - 02 Overflow is not set (result is FFh)
    - 1 - 2 = -1
  - 8A - 0F Overflow is set (result would be < 80h)
    - 8A is negative, and is equal to -76h = -118d
    - -118 - 15 < -128, and thus cannot be represented as a 1-byte signed quantity

Determining Overflow

- Another way to determine whether a particular signed operation would overflow is to look at the sign of the result and see if it makes sense
- Example: 1-byte operation 8A + A2
  - 8A is negative
  - A2 is negative
  - In hex 8A + A2 = 2C (and a carry)
  - 2C is positive
  - The sum of two negative numbers should be negative, so we’ve experienced an overflow

Overflow is your Responsibility

- The processor merely computes bits and puts them into the destination location as if everything were fine, and it's your responsibility to check the overflow!
- Let's look at two examples
  - An unsigned arithmetic example
  - A signed arithmetic example
- Note that we will see later how to "check" the Carry bit and the Overflow bit in the FLAGS register

Unsigned Overflow

On web site as ics312_overflow_unsigned.asm

```
mov     al, 0F0h     ; al = F0h
mov     bl, 0A3h     ; bl = A3h
add     al, bl       ; al = al + bl
movzx   eax, al      ; increase size for printing
call    print_int    ; print al as an integer
```

- As a programmer we decided to do some computation with unsigned values
- We put value F0h in al (unsigned F0h is decimal 240)
- We put value A3h in bl (unsigned A3h is decimal 163)
- We add them together
- The "true" result should be decimal 240+163 = 403, which cannot be encoded on 8 bits (should be < 255)
- But the processor just goes ahead: F0 + A3 = 193h, and then drops the leftmost bits to truncate to a 1-byte value to get 93h!
- To call print_int, we need the integer in eax, so we movzx al into eax
- print_int print the decimal value corresponding to 00000093h, that is: 147!
- This is obviously wrong, and we can tell (or will be able to shortly) because the carry bit is in fact set to 1
- Note that this is all correct if we assume signed values and replace movzx by movsx, but then our initial interpretation of the two values is different

Signed Overflow

On web site as
ics312_overflow_signed.asm

```
mov     al, 09Ah      ; al = 9Ah
mov     bl, 073h      ; bl = 73h
sub     al, bl        ; al = al - bl
movsx   eax, al       ; increase size for printing
call    print_int     ; print al as an integer
```

- As a programmer we decided to do some computation with signed values
- We put value 9Ah in al (signed 9Ah is decimal -102)
- We put value 73h in bl (signed 73h is decimal +115)
- We subtract bl from al
- The “true” result should be decimal -102 - 115 = -217, which cannot be encoded on 8 bits (should be >= -128)
- But the processor just goes ahead: 9A - 73 = 27h
- To call print_int, we need the integer in eax, so we movsx al into eax
- print_int prints the decimal value corresponding to 00000027h, that is: 39!
- This is obviously wrong, and we can tell (or will be able to shortly) because the overflow bit is in fact set to 1
- Note that this is all correct if we assume unsigned values and replace movsx by movzx, but then our initial interpretation of the two values is different

Multiplication

- There are two instructions to perform multiplications
- Multiplying unsigned numbers: mul
- Multiplying signed numbers: imul
- Why do we need two different instructions?
- Consider the multiplication of FF by FF
  - If we assume unsigned quantities, this is 255*255 = 65035 = FE0Bh
  - If we assume signed quantities, this is -1 * -1 = 1 = 0001h

The mul Instruction

- The size of the result of the multiplication is sometimes twice larger than the size of the operands
  - Multiplications just leads to much bigger numbers than additions
  - At most the result will be twice the size of the operands (255 * 255 = 65,025, which is encodable on 2 bytes)
- The oldest form of multiplication is the "mul" instruction, which produce a result twice the size of its unsigned operand

  mul <register or memory reference>

  - If the operand is a byte, then it is multiplied by AL and the result is stored in (16-bit) AX
  - If the operand is 16-bit, it is multiplied by AX and stored in (32-bit) DX:AX
    - There used to be no 32-bit registers
  - If the operand is 32-bit, it is multiplied by EAX and the result is stored in (64-bit) EDX:EAX
    - We don't have 64-bit registers on a 32-bit architecture

The imul instruction

- Imul, which is used for signed numbers, has three formats:

```
imul    src
imul    dst, src1
imul    dst, src1, src2
```

- The different combinations are shown in Table 2.2 in the text book
- This table uses the typical way in which one specifies operands:
  - reg16: a 16-bit register
  - reg32: a 32-bit register
  - immed8: an 8-bit immediate operand (i.e., a number)
  - mem16: a word of memory
  - etc.
- Let’s look at the table

The imul instruction

Will not overflow (although the overflow bit may be set)

dst	src1	src2	action
	reg/mem8		AX = AL * src1
	reg/mem16		DX:AX = AX * src1
	reg/mem32		EDX:EAX = EAX * src1
reg16	reg/mem16		dst *= src1
reg32	reg/mem32		dst *= src1
reg16	immed8		dst *= immed8
reg32	immed8		dst *= immed8
reg16	immed16		dst *= immed16
reg32	immed32		dst *= immed32
reg16	reg/mem16	immed8	dst = src1*src2
reg32	reg/mem32	immed8	dst = src1*src2
reg16	reg/mem16	immed16	dst = src1*src2
reg32	reg/mem32	immed32	dst = src1*src2

Division

- Two instruction:
  - div for unsigned quantities
  - idiv for signed quantities
- They perform integer division
  - e.g.,: 19 / 4 produces quotient = 4 remainder = 3
- Only one format for both:

  div/idiv src
- If src is an 8-bit quantity:
  - AX is divided by src
  - quotient stored into AL
  - remainder stored into AH
- If src is a 16-bit quantity:
  - DX:AX is divided by src
  - quotient stored into AX
  - remainder stored into DX

Division

- If src is a 32-bit quantity:
  - EDX:EAX is divided by src
  - quotient stored into EAX
  - remainder stored into EDX
- Warning: it’s very common for programmers to forget initializing DX or EDX before the division

Negation

- There is a convenient instruction to negate an operand: neg
- It simply computes the 2's complement of a quantity
- Works on 8-bit, 16-bit, or 32-bit quantities
  - either in registers or in memory

- We'll ignore the content of Section 2.1.5 in the textbook

Example Program in Textbook

- Section 2.1.4 shows a sample program that uses all the arithmetic operations we just saw
- There is nothing particularly difficult about it, especially because overflows are not handled (so the numbers entered had better be "small")
- One interesting point: One cannot divide by an immediate value and must use a register
- Make sure you go through this example and understand how it works
  - You may want to run it as well

Conclusion

- One has to be careful when doing arithmetic operations because the processor happily produces results but it's your responsibility to check for overflow/carry bits