Arithmetic

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Addition and Subtraction

- We've seen two instructions 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 (AGAIN): 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
- Let's see an example that shows that addition works on either signed or unsigned numbers

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 :(

- The magic works only with numbers of within acceptable ranges
- Because we encode numbers with finite numbers of bits, sometimes we want to store a number that would require more bits
  - The results is out of range
- In this case we have “overflow”
- The CPU proceeds with the computations, but “drops” bits that can’t fit
  - Again, it has no idea what the numbers mean
- The numerical result of the operations is then wrong and your program buggy

Overflow and Range (1-byte)

- 1-byte unsigned numbers have range 0, 255
- 1-byte signed numbers have range -128, + 127
- Example additions
  - 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.
- Let's see how, as humans, we can detect/understand overflow…
  - Of course one full-proof way is to convert everything to decimal and check whether the result is in range
  - But often we can simply reason about the numbers..

Unsigned Overflow

- Say all our numbers are meant to be unsigned
  - Again, the CPU has no idea about signed/ unsigned
- We have overflow when:
  - An addition would lead to a result that can't be encoded in the required number of bits
    - i.e., add something big to something big
    - i.e., we have a left-over carry that can't fit in the result
  - A subtraction would lead to a negative result
    - i.e., subtract something big from something small
- Let's see 1-byte examples…

Unsigned Overflow Examples

- 1-byte Example (all in hex):
  - FF + 02 OVERFLOW (result would be 101h)
    - 255 + 2 > 255
  - 01 - 05 OVERFLOW (result cannot be negative)
    - 1 - 5 < 0
  - 8A - 0F NO OVERFLOW (result is 7Bh)
    - 138 - 15 = 123
    - We're subtracting something small (0F) from something big (8A), so we can't be negative

- In a nutshell
  - An addition/sub overflows if there is a leftover carry
  - BIGGER - SMALLER never overflows
  - SMALLER - BIGGER always overflows

In-Class Exercise: Unsigned

- Which of these **unsigned** operations cause overflow?
  - 0F12 + F212 (2-byte quantities)
  - 00E3 + F74F (2-byte quantities)
  - F1 - FA (1-byte quantities)
  - FB12 - A3AA (2-byte quantities)
  - A314 - B010 (2-byte quantities)

In-Class Exercise: Solutions

- Which of these unsigned operations cause overflow?

```
    0F12
+   F212
=  10124                OVERFLOW

    00E3
+   F74F
=   F832                NO OVERFLOW
```

  - F1 - FA: smaller - bigger OVERFLOW
  - FB12 - A3AA: bigger - smaller NO OVERFLOW
  - A314 - B010: smaller - bigger OVERFLOW

Signed Overflow

- It's a bit more difficult to think about the range for signed numbers because both positive and negative values are possible
- 1-byte Example (all in hex, same as before):
  - FF + 02 NO OVERFLOW (result is 01h)
    - -1 + 2 = +1
  - 01 - 02 NO OVERFLOW (result is FFh)
    - 1 - 2 = -1
  - 8A - 0F OVERFLOW (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
    - We subtracted from a number that's very close to the edge of the valid range
- Useful trick: look whether the sign of the result is in agreement with the sign of the operands (next slide)

Signed Overflow

- A possible 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
- Same example as before: 8A - 0F
  - 8A < 0 and 0F >0, so the result should be negative
  - Lets compute the result
  - I don't like to hex subtractions, so I instead compute -0F = +F1
    - "flip and add one" to get the opposite number
  - In hex 8A + F1 = 7B   (we dropped the carry to fit in 8 bits)
  - 7B is positive!   OVERFLOW
- In a nutshell:
  - POSITIVE + POSITIVE should be POSITIVE
  - NEGATIVE + NEGATIVE should be NEGATIVE
  - POSITIVE + NEGATIVE never causes overflow!

In-Class Exercise: Signed

- Which of these **signed** operations cause overflow?
  - 00E3 + FF4F (2-byte quantities)
  - F1 - 7A (1-byte quantities)
  - FF847CAA + 78AA0401 (4-byte quantities)
  - DF + EF (1-byte quantities)

In-Class Exercise: Solutions

- Which of these signed operations cause overflow?
  - 00E3 + FF4F
    - POSITIVE + NEGATIVE: NO OVERFLOW
  - F1 - 7A
    - I do the hex addition: F1 - 7A = F1 + 86 = 77
    - Should be negative: OVERFLOW
  - FF847CAA + 78AA0401
    - NEGATIVE + POSITIVE: NO OVERFLOW
  - DF + EF
    - SMALL NEGATIVE + SMALL NEGATIVE: NO OVERFLOW
    - DF + EF = CD (dropped a carry), which is negative

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 prints 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: 9Ah - 73h = 9Ah + 8Dh = 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

Overflow is your Responsibility

- The processor merely computes bits and puts them into the destination location, possibly dropping bits, as if everything were fine, and it’s your responsibility to check the overflow!
- In your program you should have checks for overflow, which is annoying
  - That’s true in high-level languages as well!
  - Which is why we often use too many bits (e.g., 4-byte values for numbers we know to be small)
  - This way we waste memory, but we’re pretty sure to avoid overflow
    - Until we don’t all everything falls apart

The FLAG register

- You probably have forgotten by now, but at the beginning of the semester I mentioned the FLAG register
- It's basically a bunch of bits that are set/unset when instructions are executed
- They have a range of uses
- Two of them have to do with overflow:
  - The carry bit
  - The overflow bit

Detecting Overflow in Code

- There is an overflow with an unsigned operation (i.e., on unsigned quantities) if the **carry bit in the FLAG register 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)
- There is an overflow with a signed operation (i.e., on signed quantities) if the **overflow bit in the FLAG register is set**
  - This bit is set when the sign of the result does not agree with the signs of the operands
- We'll see later how to check those bits

To remember

domain	overflow
unsigned	carry bit
signed	overflow bit

- After a valid unsigned operation, the overflow bit could be set
- After a valid signed operation, the carry bit could be set

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
- So clearly we need two different instructions because we need to get two different results

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 produces 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
- WARNING: Multiplication will thus overwrite DX or EDX for 16- or 32-bit values

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
  - Because you are not expected to memorize all options!
- 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 instructions:
  - 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
- ANNOYING FEATURE: the argument to div/idiv must be a register
  - Not a constant
- And yes, this is all cumbersome
- Let's see an example

Division Example

- Say I want to divide 2042 by 13 and I want to have quotient and remainder as 4-byte values
- Here is the code

```
mov       eax, 2042
mov       edx, 0       // important
mov       ecx, 13
idiv      ecx           // must use register
// eax contains 157
// edx contains  1
// since 2042 = 157 * 13 + 1
```

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")

- Make sure you go through that example and understand how it works
  - You may want to run it as well

High-Level Languages

- Say you have to write a function in C/C++:

```
void f(unsigned int a, unsigned int b) {
  unsigned int x =  a + b;
  for (unsigned int i=0; i  < x; i++) {
    // do something
  }
}
```

- If a user passes numbers whose sum is too big, x will be bogus
- But we cannot check the carry bit
  - We can in assembly, as we’ll see
- So we have to check “by hand” :(
- Let’s see the code…

High-Level Languages (2)

```c
#include <limits.h>

void f(unsigned int a, unsigned int b) {
  if (a > UINT_MAX - b) {
    exit(1);  // Overflow
  }

  unsigned int x =  a + b;
  for (unsigned int i=0; i  < x; i++) {
    // do something
  }

}
```

- You may have had to do this, e.g., when practicing for the coding interview on some sites? (like Leetcode)

High-Level Languages (3)

```
#include <limits.h>

void f(int a, int b) {

  if ((b > 0 ) && (a > INT_MAX - b)) ||
     ((b < 0) && (a < INT_MIN - b)) {
     exit(1);  // Overflow
  }

  int x =  a + b;
}
```

- For signed integers, you have to check “both” ends, since you can overflow on either side

High-Level Languages (4)

- As you can see, it’s pretty inconvenient, but if you want to write robust code, you have to do it
- What you really want to avoid is a silent overflow
  - Easiest way: always use bigger data types than needed, if possible
- But often, overflow is actually a feature of a program
  - i.e., the program relies on the weird “wrap-around” behavior that happens when you have overflow
- Different languages provide different way of dealing with overflow
- In Java, you can use special “overflow catching” methods of the Math package (e.g., Math.addExact())
- In C/C++ you can give flags to the compiler…

High-Level Languages (4)

- We can ask the C/C++ compiler to add (assembly) code to the check for overflow for all integer operations
  - As we'll learn to do in the next module in assembly
- It's easy for the compiler based on the signed-ness of numbers
  - Insert code to check the carry bit or to check the overflow bit
- If overflow is detected, abort the program
  - But if your program uses overflow as a "feature", then that will be a problem!
- With gcc: -ftrapv will do this for signed overflow
- Alternatively, with gcc: you can do __builtin_sadd_overflow(a,b,&c) for an addition that checks overflow and returns true/false
- But that won't work with other compilers….

Do we care?

- Clearly, dealing with overflow is a pain (not in assembly though, as we'll see!)
  - This may be the **one** thing this semester which is better in assembly than in high-level languages
- Let's look at: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html
  - And click at subsequent links
- Many of the examples here says "and then there will be a buffer overflow"
- Stay tuned for a discussion of that vulnerability in the "Subprogram" (post-midterm) module

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

- We will have practice quiz on this module on Thursday