Numerical Overflow

ICS312
Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Overflow

- We've seen **`add`** and **`sub`** for additions and subtractions
- Both instructions can be used either on a pair of signed numbers or on a pair of unsigned numbers
  - The 2's complement "magic"
  - Mixing of signed and unsigned numbers will "work" but give bogus results
- We encode numbers with finite numbers of bits
- Sometimes the numerical result would require more bits!
- In this case, the CPU proceeds with the computation, but **drops extra bits** that can't fit
  - As we saw in the "NASM Basics" module
- The numerical result of the operations is then wrong
- We call this overflow

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 it's much easier to reason about the numbers…

Unsigned Overflow

- Say all our numbers are meant to be **unsigned** for now
- We have overflow when:
  - An addition would lead to left-over carry
    - i.e., the result that can't be encoded in the required number of bits
    - Happens when adding something big to something big
  - A subtraction would lead to a negative result
    - Happens when subtracting 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
  - **Addition:** overflows if there is a leftover carry
  - **Subtraction:**
    - BIGGER - SMALLER **never** overflows
    - SMALLER - BIGGER **always** overflows

In-Class Exercise: Unsigned

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

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

Nuclear Ghandi

- And of course the "Nuclear Ghandi" **urban legend**: https://en.wikipedia.org/wiki/Nuclear_Gandhi

- Although the Nuclear Ghandi is made up, integer overflows are horrible bugs that exit in the real world (see the end of these lecture notes)

Signed Overflow

- It's more difficult to think about ranges 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 a 1-byte signed quantity
    - We subtract a positive number from a number that's already very close to the left edge of the valid range, we get out of range

- So how can we, as humans, easily tell whether something will overflow or not?

Signed Overflow

- A way to determine whether a particular signed operation overflows is to see whether the sign of the result makes sense
  - If it doesn't that means we "wrapped around" the range
- Same example as before: 8A - 0F
  - 8A < 0 and 0F >0, so the result should be negative
  - Let's compute the result
  - I don't like hex subtractions, so I instead compute -0F = +F1
    - "flip and add one" to negate the number (the `neg` instruction)
  - In hex: 8A + F1 = 7B (the carry is dropped 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 values)
  - F1 - 7A (1-byte values)
  - FF847CAA + 78AA0401 (4-byte values)
  - DF + EF (1-byte values)

- Recall that, in a nutshell:
  - POSITIVE + POSITIVE should be POSITIVE
  - NEGATIVE + NEGATIVE should be NEGATIVE
  - POSITIVE + NEGATIVE never causes overflow!

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 = CE (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, and it’s your responsibility to check the overflow!
- In your program you should have checks for overflow, which is more work
  - 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 wastes memory, but we’re pretty sure to avoid overflow in most cases
    - Until we don’t and everything falls apart!!!

The FLAG register

- You probably have forgotten it 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 many different uses
- Two of those bits have to do with overflow:
  - The carry bit
  - The overflow bit
- The CPU sets those bits for you….

Detecting Overflow in Code

- If there is an overflow assuming UNSIGNED values then the **carry bit in the FLAG register is set (to 1) otherwise it is unset (set to 0)**
  - If the carry bit is set to 1, that means there was a leftover carry or borrow, and we'd need more bits to store the result
- If there is an overflow assuming SIGNED values then the **overflow bit in the FLAG register is set (to 1) otherwise it is unset (set to 0)**
  - This bit is set to 1 when the sign of the result does not agree with the signs of the operands
- **Both bits are set/unset each time an arithmetic operation is performed**
  - We'll see later how to check the values of those bits

To remember

domain	overflow detector
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

- Both bits are set/unset because the CPU does not know your interpretation of your numbers
  - It’s your job to check the bit you should care about

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, the value of variable **x** will be bogus
- In high-level code we cannot check the carry bit
  - We can in assembly, as we'll see
- So we have to check overflow "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
  }

}
```

Note that writing **`a + b > UINT_MAX`** doesn't work because the sum can overflow! But bigger - smaller never overflows, so we can safely compute **`UINT_MAX - b`**.

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)

```c
#include <limits.h>

void f(int a, int b) {
   int x;
   if ((b > 0  &&  a < INT_MAX - b) ||
        (b < 0  &&  a > INT_MIN - b)) {
     x =  a + b;
   } else {
     exit(1);   // Overflow
   }


}
```

- For signed integers, you have to check "both" ends, since you can overflow on either side
  - If b > 0 then check that a + b < INT_MAX
  - If b < 0 then check that a + b > INT_MIN

High-Level Languages (4)

- If you want to write robust code, you **have** to catch overflows, to avoid the deadly silent overflow bug
- Note that sometimes 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
- Easiest but far from full proof approach: always use bigger data types than what you think is needed, and pray that you’ll never use really big values (scary….)
- Different languages provide different way of dealing with overflow in a much better approach
- 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 assembly in the next module
- 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 call __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/2023/2023_top25_list.html
  - https://cwe.mitre.org/top25/archive/2024/2024_top25_list.html
- Some of the examples on that site say "and then there will be a buffer overflow"
  - Buffer overflow has **nothing** to do with integer overflow
  - Stay tuned for a discussion of that vulnerability in the "Subprogram" (post-midterm) module

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/2023/2023_top25_list.html
  - https://cwe.mitre.org/top25/archive/2024/2024_top25_list.html
  - Some of the examples on these sites say "and then there will be a buffer overflow"
    - Buffer overflow has **nothing** to do with integer overflow
    - Stay tuned for a discussion of that vulnerability in the "Subprogram" (post-midterm) module
- Good news: LLMs are great at popping up overflow warnings in your IDE via static code analysis you're too lazy to do :)

Important Takeaways

- Overflow occurs when we “don’t have enough bits” when doing computer arithmetic
- Unsigned overflow:
  - We have a leftover carry that would require our data size to be larger by 1 bit
  - In this case, the CPU sets the carry bit to 1
- Signed overflow:
  - The sign of the result doesn’t make sense given the signs of the operands
  - In this case the CPU sets the overflow bit to 1
- In high-level languages, one can
  - Check for overflow “by hand” (making sure the check never computes anything that could overflow)
  - Use “special” functions/APIs

Conclusion

- One has to be careful when doing arithmetic operations because the processor happily produces result bits regardless
- It's your responsibility to check for overflow/ carry bits (in assembly) or to check for overflow manually (in high-level languages)

- Let's look at some practice problems…
- There is a sample homework as well…