# A Proposal For Hardware Assisted Arithmetic Overflow Detection

Darek Mihocka
Jens Troeger

Parallel Hybrid Computing, Intel Corporation

April 25 2010

# Overview

- Definition of arithmetic overflow
- Common mechanisms for dealing with overflow and their drawbacks
- Understanding how overflow can be detected
- Handling arbitrary integer sizes
- Overflow vectors
- Array summation
- Packed integer bitfield overflow
- A simple ISA extension proposal to accelerate overflow vector generation

# What is Overflow?

- Arithmetic overflow results when the integer result of an addition, subtraction, or other arithmetic operation cannot be represented exactly for a given integer size and type.
- Two types of integer overflow – signed and unsigned.
- ***Signed overflow*** occurs when the resulting sign bit is incorrect.
- ***Unsigned overflow*** occurs when the result is truncated and ends up being smaller than either of the two input values.
- For example, for 8-bit signed integers:
  - 0x7F + 0x02 = 0x81 is a signed integer overflow because positive + positive resulted in a negative result
  - 0x7F + 0x82 = 0x01 is an unsigned integer overflow because the true sum 0x101 cannot be represented
  - 0x81 + 0x82 = 0x03 is both a signed integer overflow and unsigned integer overflow condition
- Detecting arithmetic overflow during addition and subtraction is vital to ensure correct and secure behaviour of software. Malware exploits have made use of overflow!

# Detecting Overflow using CPU Flags

- Almost all microprocessors provide arithmetic status flags which are updated by the ALU for common operations such as addition and subtraction, e.g. EFLAGS, CCR

- These flags generally consist of (but are not limited to):
  - **Zero Flag** – indicates that the resulting value is zero (i.e. all bits are zero)
  - **Sign** (or **Negative**) **Flag** – indicates that the resulting value is negative (i.e. high bit is set)
  - **Overflow Flag** – indicates a signed integer overflow condition (i.e. sign bit is incorrect)
  - **Carry Flag** – indicates an unsigned integer overflow condition (i.e. unsigned result is truncated)

- This permits high-level languages such as C# to explicitly expose overflow detection by means of the "**checked**" keyword and throw an exception when the resulting data would have been invalid.

```
add     eax,1
jo      throw_overflow_exception
```

- A limitation of this approach is that it is serializing on x86: only one ALU operation per clock cycle can generate arithmetic status flags, which must be forwarded to a conditional branch instruction before any other flag-altering ALU operation can occur.

# Detecting Overflow using SIMD

- Modern processors with SIMD instruction set extensions such as SSE2 provide the ability to perform packed integer operations and optionally saturate the results of each packed bitfield.
- Detecting that the saturated and unsaturated results are different is one way to detect overflow in parallel:

```
movq     xmm1, vector1     ; input vector 1
movq     xmm2, vector2     ; input vector 2
movq     xmm0, xmm1
paddb    xmm0, xmm2        ; xmm0 = vector 1 + vector 2, unsaturated result
paddsb   xmm1, xmm2        ; xmm1 = vector 1 + vector 2, saturated result
pcmpeqb xmm0, xmm1         ; are the results the same? All zeroes indicates no overflow
pmovmskb eax, xmm0         ; copy the CMPEQB sign bits results to a GPR
test     eax,eax
jnz      throw_overflow_exception
```

- The limitations of this approach are that this is limited to only a specific set of integer sizes (for example, Larrabee does not support 8-bit packed integers at all), and several cycles of overhead required to transfer the vector result to a general purpose register in order to generate flags and conditionally branch.

# Detecting Overflow in a HLL

- Without directly having access to the Carry Flag and Overflow Flag, a high-level language can detect overflow based on the definitions previously given:
  - For signed overflow, detect whether the sign of the result is different from the sign of both of the inputs:

```
int32_t s = a + b;
bool overflow = ((s ^ a) & (s ^ b)) < 0;
```

  - For unsigned overflow, detect whether the result is smaller in magnitude than either of the inputs:

```
uint32_t s = a + b;
bool overflow = (s < a) && (s < b);
```

- A disadvantage of this straightforward approach is that mainstream Windows compilers will actually generate *two* conditional branches per check!

```
lea     ebx, DWORD PTR [edi+eax]
cmp     ebx, edi
jae     SHORT $LN4@unsigned_c
cmp     ebx, eax
jae     SHORT $LN4@unsigned_c
mov     ecx, 1
jmp     SHORT $LN5@unsigned_c
xor     ecx, ecx
```

# The Challenges

- Can the overflow detection be generalized to work on arbitrary sized integers?
  - e.g. if the sum of two numbers a valid 10-bit integer
- Even in an arbitrary sized packed integer bitfield?
  - e.g. increment a bitfield within an integer and detect if that overflowed (and thus corrupted other bits)
- Can more parallelism be found by not serializing the code on a conditional branch every 1 (or small N) number of arithmetic operations?
  - e.g. increment multiple bitfields within an integer and detect overflow in any of them
  - e.g. calculate the sum of an array and know the result is valid using perhaps only one conditional check!
- ***The key to solving these challenges is in understanding how the Carry Flag and Overflow Flag are generated in the first place and then finding alternate ways of deriving them.***
- Fortunately, the mathematics behind arithmetic flags is well understood
  - Read “Hacker’s Delight” for the bit twiddling tricks
  - Read “Hardware Without Virtualization“ (Mihocka, Shwartsman, 2008) for real world implementation details

# Understanding the Carry Flag

- Start with the fundamentals – 1-bit integer addition and subtraction
- Given two input bits A and B, and sum bit S and difference bit D, we see that:
  - S = A + B = A XOR B and D = A – B = A XOR B
  - (S XOR (A XOR B)) == 0 and also (D XOR (A XOR B)) == 0
  - Without carry bits, ADD and SUB are just XOR operations, exactly how half adders implement them
- When there is a carry-in bit involved:
  - S = A + B + Cin = (A + B) + Cin = (A XOR B) XOR Cin
  - Cin = (S XOR (A XOR B)), also Cin = (D XOR (A XOR B))
- If A B S D Cin are multi-bit integers (i.e. bit vectors), Cin for each bit can be derived
- Carry-In bit for bit position *n* is really the Carry-Out bit from bit position *n-1*
- So for addition or subtraction:
  - Cout(A+B) = S XOR (A XOR B) SHR 1
  - Cout(A-B) = D XOR (A XOR B) SHR 1

- In other words, for an n-bit operation, bitwise logical operations can determine the Carry Flag of any of the lower n-1 bits of an addition or subtraction
  - Perfectly fine if using 64-bit integers to derive carry-out bits for say, 8- 10- 16- or 32-bit operations

# Integer Overflow Vectors

- Recalling the expression which determines signed integer overflow…

```
bool overflow = ((s ^ a) & (s ^ b)) < 0;
```

- One can now realize that what the expression is merely returning the high bit of a complete signed overflow vector similarly generated from bitwise logical operations.

- That is, one can derive integer overflow vectors for addition and subtraction:

  OVout(A+B) = (S XOR A) AND (S XOR B)

  OVout(A-B) = (S XOR A) AND (A XOR B)

- Alternatively, microprocessor manuals and textbooks point out that the signed overflow bit is equivalent to the XOR of the top two carry-out bits, i.e.:

  OVout(A+B) = Cout(A+B) XOR (Cout(A+B) SHL 1) ➔ Cout(A+B) = OVout(A+B) XOR Cin(A+B)

- ***This allows us to derive the complete carry-out vector formulas for all bits!***

  Cout(S=A+B) = (((A XOR S)) AND (B XOR S)) XOR (A XOR (B XOR S)))

  ➔ ((A AND B) OR ((A OR B) ANDNOT S))

  Cout(D=A-B) = ((A XOR B) AND (B XOR D)) XOR (A XOR (B XOR D)))

# Lazy Overflow Checking

- Being able to generate the carry-out and overflow-out vectors explicitly allows the overflow state to be accumulated such that the overflow check is deferred.
- This is similar to the “lazy flags” approach used in Bochs and other simulators, where arithmetic flags are derived only as needed based on saving away such intermediate information in a canonical state. Thus we call this “lazy overflow checking”.
- It is not necessary to accumulate both carry-out and overflow-out vectors, since signed overflow is determined by the XOR of two adjacent carry-out bits.
- A canonical lazy flags state to derive common arithmetic flags can be implemented as shown here, saving sign-extended result and shifting down top two carry bits:

```
uint8_t s = a + b;
uint8_t cout = ((a & b) | ((a | b) & ~s));
int64_t lazy_result = (int64_t)(int8_t)s;
uint8_t lazy_carry_vec = (cout >> (8-2)) | (cout & 8);
```

- All six common IA32 and Intel64 arithmetic status flags can be derived:

```
ZF = (lazy_result != 0);
SF = (lazy_result < 0);
CF = (lazy_cvec & 2) != 0;
OF = ((lazy_cvec + 1) & 2) != 0;
AF = (lazy_cvec & 8) != 0;
PF = parity_table[lazy_result & 255];
```

| 63 | 62 | 61 | 60 | … | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sign -extended result determines ZF | | | | | | | | | | | | | |
| SF | x | x | x | x | x | Parity bits | | | | | | | |

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | AF | 0 | CF | OF’ |

# Overflow Checks in Array Addition

- Let's construct some benchmark tests to add two integer arrays, without any overflow checking, with signed integer overflow checking, and with unsigned integer overflow checking, and see how the relative speeds compare of checking and branching on overflow each array element vs. accumulating the overflow state.
- Compiled using mainstream 64-bit Windows compiler, both with inlined and non-inlined versions of the overflow check.

```
signed int unsigned_check(unsigned int a, unsigned int b) {
    unsigned int r = a + b;
    return (signed int)((a & b) | ((a | b) & ~r));
}
signed int signed_check(signed int a, signed int b) {
    signed int r = a + b;
    return ((a ^ r) & (b ^ r));
}

for (i = 0; i < count; i++) {
    if ([un]signed_check(array1[i], array2[i]) < 0)    // check and branch version
        return OVERFLOW_ERROR;
    overflow |= [un]signed_check(array1[i], array2[i]); // accumulate overflow version
    array1[i] += array2[i];
}
return (overflow < 0) ? OVERFLOW_ERROR : NO_ARITH_ERROR;
```

# Accumulating Overflow Is Faster

- The 5 variants of the array summation loop were run on 64-bit Windows 7 on both Core 2 and Atom host processors. The relative times (normalized to milliseconds per million array elements) for inlined and non-inlined builds are shown here:

| | Core 2 func call | Core 2 inlined | Atom func call | Atom inlined |
|---|---|---|---|---|
| No overflow checking | 2.7 | 2.7 | 4.6 | 4.6 |
| Unsigned compare/branch | 3.8 | 2.8 | 11.7 | 9.1 |
| Unsigned accumulate | 3.4 | 2.8 | 10.1 | 7.2 |
| Signed compare/branch | 3.5 | 2.7 | 10.8 | 7.6 |
| Signed accumulate | 3.1 | 2.7 | 9.2 | 5.6 |

- The out-of-order core of the Core 2 processor mostly hides the differences. However, on the in-order Atom core, representative of the type of cores found in netbooks, phones, and game consoles, the deferred overflow checking is over 10% faster.
- ***Advantageous to replace flow control with a little extra computation.***

# Packed Bitfield Operations

- Operating over arrays can be parallelized using SIMD, but with the overflow detection overhead previously shown which requires moving data to a GPR as well.
- Packed data and bitfields can be handled directly in a GPR without using SIMD, since the carry-out or overflow-out vectors provide overflow status of *every* bit position.
- This permits multiple bitfields in the same integer to be operated on in parallel.
- Consider a structure of three oddly sized bitfields and the task of incrementing them:

```
struct {
        int data0:9;
        int data1:11;
        int data2:12;
};
```

- Mainstream compilers will increment each field separately. Using a carry-out vector, all three fields can be incremented simultaneously in roughly half the cycles by using the carry-out information to prevent carry propagation between bitfields:

```
int add_bits = 0x00100201;  // low bits of each of the three bitfields
int add_mask = 0x00080100;  // high carry-out bits of each of the three bitfields
int carrys = carry_out(array[i].raw_bits, add_bits) & add_mask;
array[i].raw_bits += add_bits - carrys;
```

# Packed Bitfield Saturation

- Using the carry-out vector to block leakage between adjacent bitfields, common SIMD operations, including signed/unsigned saturating addition and subtraction operations can be implemented purely using integer operations.
- Saturation is merely the use of overflow condition to clip a bitfield's value. Consider a typical 64-bit integer implementation of a packed unsigned addition of 8 8-bit fields:

```
uint64_t mask1 = 0x7F7F7F7F7F7F7F7F;
uint64_t mask2 = ~mask1;
uint64_t result_lo = (a & mask1) + (b & mask1);
uint64_t result_hi = ((a ^ b) & mask2);
uint64_t result = result_lo ^ result_hi;
```

- The result potentially contains bitfields which have wrapped around beyond 0xFF. The result can be adjusted to correctly saturate by applying the carry-out bits:

```
uint64_t carrys = ((a & b) | ((a | b) & ~(a + b))) & mask2;
// writes 0xFF to byte lanes that carried out
result |= (carrys << 1) - (carrys >> 7);
```

- This is a trivial example of implementing SIMD style saturating packed operations from a high-level language, with the added benefit of being able to operate support non-standard sized bitfield widths.

# New Instruction Proposal

- We propose adding two new instructions for directly generating carry-out vectors for addition and subtraction: ADDCOUT and SUBCOUT.

- These instructions eliminate the sequence of XOR and AND operations needed to construct the carry-out vector, thus reducing code size and cycle count.

- Would be implemented as 2-input 1-destination operations on either the general purpose integer register file (similar to LEA), or the multimedia SIMD register file, enabling rather trivial implementation of lazy flags and lazy overflow detection, e.g.:

```
mov     rcx,input1
mov     rdx,input2
subcout rax,rcx,rdx
sub     rcx,rdx
mov     carrys,rax
mov     result,rcx
```

- Could be exposed as intrinsics to high-level languages such as C++, benefiting performance of simulators and allowing for more efficient overflow detection.

# Summary

- A quick refresher course on the definition of arithmetic integer overflow, how it relates to the Carry and Overflow arithmetic status flags on the processor, and how those flags are derived.
- Shown how to construct the carry-out and overflow-out bit vectors for every bit in an integer addition and subtraction operation.
- Demonstrated useful applications of these carry-out vectors for CPU simulation, deferring overflow checks in array summation, and parallelizing bitfield operations.
- Introduced the concept of separating overflow checking from the actual arithmetic operation to increase performance.
- Proposed a simple instruction set extension to provide the carry-out vector directly to reduce the computation required.

- THE END

# Backup Slides