A Proposal for Hardware-Assisted Arithmetic Overflow Detection for Array and Bitfield Operations

Darek Mihocka
Intel Corp.
Darek.Mihocka@Intel.com

Jens Troeger
Intel Corp.
Jens.Troeger@Intel.com

Abstract

Detecting arithmetic overflow during summation operations is vital to ensuring correct and secure behavior of many types of code. For example, applying transformations to signed integer pixel co-ordinates without any overflow detection may result in pixels rendering at unexpected negative co-ordinates, summing a large array of signed or unsigned integers without overflow detection can result in bogus totals, or performing arithmetic operations on packed bitfields without overflow detection could result in corruption of data in adjacent bitfields.

A traditional way to detect arithmetic overflow is to insert specific checks of the host processor's Overflow arithmetic condition flag after each arithmetic operation to detect signed integer overflow, or a check of the host processor's Carry arithmetic flag to detect unsigned integer overflow. The C# language for example includes the keyword "checked"[1] which directs the code generator to inject such checks. Similarly, some versions of the gcc native compiler support a switch to generate such checks.[2]

A drawback of this approach is that since it relies on the host arithmetic flags to detect arithmetic overflow, the addition or subtraction operation must immediately be followed by the check as well as by some conditional branch or trap for when the overflow is detected. This severely serializes the ability to parallelize array summation or packed bitfield arithmetic, as it requires a separate test and branch for each arithmetic operation.

Modern processors such as the Intel Core and Intel Atom series offset SIMD extensions which allow for packed data operations on signed and unsigned integers.[3] This permits up to 8 additions or subtractions to be performed in one operation, with overflow detection (in the form of a Saturate status bit). However, even the SIMD approach is limited to the data sizes and operations supported by the particular SIMD instruction set.

An additional problem for either the traditional arithmetic flags or SIMD based overflow detection mechanisms is that they cannot operate on small or unconventional data sizes. The Intel Larrabee[4] SIMD instruction set for example, only supports packed 32-bit and 64-bit integer types, and therefore cannot operate on 8-bit or 16-bit integers directly without additional data conversion operations.

Neither approach can operate directly on bitfield data types such as the common 5-6-5 packing of RGB pixel values, or something even more trivial such as incrementing a 4-bit bitfield, since the smallest arithmetic flags generating operations on most processors require data elements at least 8 bits wide.

This paper examines an alternative and purely integer-based "lazy flags"[5] method of detecting signed and unsigned arithmetic overflow conditions which decouples the generation of the sums (or differences) from the detection of the arithmetic overflow in a manner that is not dependent on the specific integer data size capabilities of the host processor. Such a decoupling allows for the vectorization of both the arithmetic operation as well as the overflow detection without the need for specific SIMD extensions. This has practical uses for runtime written in high-level languages such as bytecode interpreters and simulators where direct access to hardware arithmetic flags or SIMD extensions is difficult.

This paper will then propose simple instruction set extensions for implementing the lazy flags approach to allow for hardware-assisted acceleration of overflow detection across bitfields and arrays without the data size restrictions of existing integer or SIMD implementations.

Keywords

Lazy Flags, Arithmetic Overflow, Hardware Acceleration

1.0 Introduction

Integer arithmetic on modern microprocessors is generally performed on 8-, 16-, 32-, or 64-bit integers. These integers may represent unsigned values or signed two's complement values. An 8-bit integer for example may represent the unsigned values 0 through 255, or the signed values ranging from -128 to +127.

Given two 4-bit integers A and B and the 4-bit sum S, the individual bits in the addition can be represented as follows:

$$S_3S_2S_1S_0 = A_3A_2A_1A_0 + B_3B_2B_1B_0$$

The least significant bit position of the sum, bit S_0, is the sum of bits $A_0 + B_0$, which in an adder circuit can be implemented using an XOR operation. The bit position may also generate a carry-out bit which is propagated to the next bit position, denoted as C_{out0}. Therefore,

$$S_1 = A_1 \text{ XOR } B_1 \text{ XOR } C_{out(0)}$$

In general,

$$S_n = A_n \text{ XOR } B_n \text{ XOR } C_{out(n-1)}$$

Note that the carry-out bit from bit position n-1 is the same as the carry-in bit to bit position n, therefore we can also write each bit position's addition operation as:

$$S_n = A_n \text{ XOR } B_n \text{ XOR } C_{in(n)}$$

An integer overflow condition occurs when the resulting value of an operation cannot correctly be represented in the same number of bits. For example, using 8-bit signed integers, the values 127 + 1 should result in the value 128. This is a correct result for unsigned integers. However, since the number 127 is the largest positive 8-bit integer, represented as 0x7F in hexadecimal, adding 1 result in 0x80, which is the signed value -128, not +128. This result represents a signed integer overflow condition.

It is possible for an arithmetic operation to simultaneously generate an unsigned overflow and a signed overflow. For example, the 8-bit values 0xA0 + 0xA0 result in the 8-bit sum 0x40. As unsigned integers this represents 160 + 160 = 64, and as signed integers this represents -96 + -96 = 64. Obviously neither result is valid.

Modern microprocessors almost universally signal arithmetic integer overflows using arithmetic condition flags status bits following an arithmetic operation. For example, IA32 instruction set architecture (ISA) based processors such as the Intel Pentium and Intel Core series signal overflow in the EFLAGS status register, which contains 6 arithmetic flags – Carry Flag, Overflow Flag, Zero Flag, Parity Flag, Adjust Flag, and Sign Flag. Unsigned integer overflow is signaled by setting the Carry Flag, while signed integer overflow is signaled by the Overflow Flag.

A native compiler or dynamic code generator for IA32 architecture can simply append a "conditional branch if Carry flag set" instruction to jump to an unsigned integer overflow handler following an arithmetic operation. Similarly, the compiler can append a "conditional branch if Overflow flag set" instruction to jump to a signed integer overflow handler.

Since the arithmetic conditional flags are generally overwritten following each arithmetic operation and since most architectures only contain a single set of these flags per hardware thread, this results in the following restrictions in the handling of arithmetic overflow:

- Overflow must be detected and acted upon almost immediately after the arithmetic operation has been performed, because a subsequent operation will destroy the flags state.
- Overflow can be detected for standard integer data sizes supported by the native ISA instructions, but the condition flags will not directly reflect accurately if non-standard sizes wish to be operated on.
- The single instance of the condition flags implies that packed integer operations, such as SIMD addition operations, cannot directly report an overflow condition.

Additionally, the arithmetic condition flags are not directly accessible by high level languages (HLLs).

1.1 Explicitly Detecting Integer Overflow

To detect integer overflow in a high-level language, it is not possible to directly read the hardware Carry Flag or Overflow Flag. Instead, one can check for unsigned overflow by checking whether the sum is of a smaller magnitude than either of the two input operands, i.e. given input exact-width integers[6] A and B and sum S:

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

The compiled code ends up performing three arithmetic operations in this case – the original addition operation as well as two comparison operations (which are effectively subtractions) – and also three jump operations – two conditional branches as well as direct jump. This is the actual optimized IA32 code generated by a mainstream C compiler for performing and addition and overflow check for 32-bit integers:

```
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
```

This code introduces inefficiency in the form of multiple potentially mispredicted jumps which are generated and the fact that all execution paths require at least one branch taken. If this code appeared in the inner loop of a sum of two arrays function, or a sum of array function, it would add measurable overhead.

A similar expression can be used to detect signed overflow, based on understanding two things:

- the addition of a positive number to a negative number cannot generate overflow, implying that both input values must be of the same sign.
- An overflowed result will have the incorrect sign, implying that it will be the opposite sign as the inputs.

Checking that both inputs are of the opposite sign as the result is sufficient to detect signed integer overflow. This can be coded up as follows[7], exploiting the property of the XOR (^) operation which generates a negative number if its inputs are of opposite sign:

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

The two expressions presented are standard methods by which code in a high level language can explicitly detect an integer overflow condition without having direct access to the processor's Carry or Overflow flags. But these are still limited to supporting specific data sizes and potentially generating multiple conditional branches per operation. This would not be the ideal code to use in something like a CPU simulator or bytecode interpreter written in a high level language where performance was desired.

1.2 Handling Arbitrary Integer Sizes

There are times when built-in ISA data types do not match the size of the actual integers being operated on. For example, one could be operating on 5-bit bitfields, such as when manipulating certain representations of RGB color data. In these instances it is often necessary to extend the integer to a larger size to match the size of a supported type. Unsigned bitfields can be zero-extended and signed bitfields can be sign-extended up to say, a standard 32-bit integer width.

This size extension renders overflow detection useless. Consider the case of the 0x7F + 0x01 = 0x80 signed overflow previous presented. If the input values are sign-extended to 32-bit and the added, the resulting addition gives a perfectly valid result:

$$0x0000007F + 0x00000001 = 0x00000080$$

The host processor's Overflow flag will not be set! This problem arises not only when dealing with non-standard integer sizes, but can also occur for any sized integer not natively supported by a given ISA's ALU unit. For example, 64-bit PowerPC processors have no notion of 8-bit addition or subtraction, so how would one go about detecting overflow in that scenario? Simple sign extension of the inputs is not the answer.

One solution is to modify the expressions from the previous section to operate on the specific subsets of N bits equal to the width of the input operands. For the case of unsigned addition, the modified source code could look like this:

```
uint32_t s = (a + b) & ((1 << N)-1);
uint32_t overflow = (s < a) && (s < b);
```

And the case of signed addition, the modified source code could look like this:

```
int32_t s = a + b;
bool overflow =
(((s ^ a) & (s ^ b)) & (1 << (N-1))) != 0;
```

This modification permits an architecture which supports 32-bit and 64-bit integer operations to detect integer overflow for smaller width inputs and results, even down to a single bit!

2.0 Generating Overflow Vectors

With the realization that overflow conditions can be detected for any arbitrary bit position by just applying a bitmask, the next logical step is to derive expressions which generate an overflow vector. That is, an integer which represents the bitwise vector of every bit position's overflow status. The signed overflow expression above actually contains a bitwise logical sub-expression which derives the necessary overflow bit vector:

```
(s ^ a) & (s ^ b)
```

The unsigned integer overflow vector is not as obvious, as the expression:

```
(s < a) && (s < b)
```

is an arithmetic expression, not a bitwise logical expression.

However, we already know that:

$$S_n = A_n \text{ XOR } B_n \text{ XOR } C_{in(n)}$$

which can be rewritten as following:

$$C_{in(n)} = A_n \text{ XOR } B_n \text{ XOR } S_n$$

This permits one to derive the carry-in vector of an integer addition operation:

```
uint64_t s = a + b;
uint64_t carryin = s ^ a ^ b;
```

For all but the highest bit position, the carry-in vector can be converted to a carry-out vector using a single bit right shift:

```
uint64_t s = a + b;
uint64_t carryout = (s ^ a ^ b) >> 1;
```

An expression to generate the full carry-out vector can be derived by considering that when both input bits are set a carry-out must occur, or if at least one input bit is set but the output bit is different due to a carry-in then a carry-out must also have occurred. This can be coded up as:

```
uint64_t s = a + b;
uint64_t carryout =
              ((a & b) | ((a | b) & ~s));
```

2.1 Lazy Arithmetic Flags for CPU Emulation

As previously published[8] in work related to the Bochs[9] open source IA32 emulation project, the method of generating carry-out vectors is extremely useful for efficiently deriving the arithmetic flags without having direct access to the processor's hardware arithmetic condition flags. This technique is sometimes referred to as the "lazy flags" approach, because it minimally requires only recording some variant of the inputs and outputs of an arithmetic operation and deriving the specific flags as needed later on. The lazy flags approach is both portable (as it has no dependency on specific host hardware) and efficient and allows Bochs and similar CPU simulators written in high level languages to achieve speeds in excess of 100 MIPS.

As discussed in the Bochs work, all six of the IA32 arithmetic flags can be derived from just two values: the result of an operation, and the carry-out vector of that operation. Unlike some lazy flags variants, it is not necessary to explicitly record the input values, only the result and the carry-out vector.

The recoded result allows the derivation of the Zero Flag by simply check if that result is zero. Similarly, checking the sign of the result derives the Sign Flags. And examining the lower 8 bits of the result determines the Parity Flag. To avoid the need to record the original size of the arithmetic operation, Bochs and similar simulators only need to sign-extend the result to some canonical width, such as a 64-bit integer.

The carry-out vector is used to derive the IA32 Carry Flag of course, but also the Overflow Flag, and the Adjust Flag. One thing to note is that the hardware definition of the signed Overflow flag is purely derived from the upper two carry bits of an arithmetic operation[10]. Therefore storing the entire carry-out vector is sufficient to also derive the overflow vector. It is not necessary to explicitly calculate and record the expression `(s ^ a) & (s ^ b)`.

By definition[11], the Adjust Flag is simply the carry-out bit of the 3rd least significant bit, and therefore can be derived by logically AND-ing the carry-out vector by 0x08.

The canonical form of the carry-out vector cannot simply be sign-extended as is the case with the result. This would cause an incorrect derivation of the Overflow flag for the extended bit position. Instead, the most significant bit of the carry-out vector needs to be shifted to a fixed bit position. We recommend two approaches, one which shifts the carry-out vector to the highest possible bit position, and one which shifts the carry-out bits to the lowest two bits, as demonstrated by these two code samples which demonstrated how an 8-bit addition would be recorded into a canonical 64-bit wide lazy flags state:

```
uint8_t s = a + b;
uint8_t cout =
              ((a & b) | ((a | b) & ~s));
uint64_t lazy_result = (int64_t)(int8_t)s;
uint8_t lazy_cvec = (cout>>6) | (cout&8);
```

Deriving the common arithmetic flags can now be done as follows :

```
ZF = (lazy_result != 0);
SF = (lazy_result < 0);
CF = (lazy_cvec & 2) != 0;
OF = ((lazy_cvec + 1) & 2) != 0;
```

Note that simply adding 1 to lazy_cvec is sufficient to XOR the lowest two bits together, instead of using the expression `((lazy_cvec ^ (lazy_cvec << 1)) & 2 != 0)`.

2.2 Optimizing Array Summation

An important concept learned from the lazy flags approach is the realization that the act of performing an arithmetic operation and the act of checking for an integer overflow condition can be separated from each other. By recording the carry-out vector the check can be postponed until well after the host processor's hardware arithmetic flags have been lost.

A performance benefit of the lazy flags approach is that the recording of the carry-out vector is a straight-line block of code. Since no conditional branches are required to generate the lazy flags state, it can be beneficial to use the lazy flags approach in hot loops such as array summation to defer the overflow check instead of performing a conditional check and branch on each element.

2.3 Handling Packed Addition and Saturation

The act of walking arrays to compute a total of an array or sum two arrays is can parallelized using SIMD. Arrays of 16-bit integers can be added, using SIMD packed addition operations, 8 elements at a time. IA32 SIMD operations do not explicitly set any kind of arithmetic overflow bit, so in place of such a status bit, SIMD offers saturating versions of signed and unsigned packed addition and subtraction operations. These saturating operations avoid the need to check for overflow by implicitly clamping the output value.

To even detect overflow in a packed saturating operation, software has to explicitly perform a non-saturating operation and compare that result to the saturating operation. If the results are different, then one or more of the packed result values was clamped.

There is an alternative approach which allows for performing packed addition and subtraction operations without the explicit use of SIMD. Using the carry-out vector it is possible to detect the exact bit positions which generate a carry-out, mask out the ones that correspond to the high bits of each packed integer, and then use those to saturate the integers as needed.

Consider the inner loop of a traditional saturating signed 8-bit integer addition:

```
int8_t sum = A[i] + B[i];
if (sum > 127)
    sum = 127;
else if (sum < -128)
    sum = -128;
```

Instead, 64-bit wide integers can be used to perform a packed addition of eight 8-bit values:

```
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;
```

Note that the packing size is determined purely by the constant mask that is applied. A mask of 0x7F7F7F… treats the data as packed 8-bit integer, while a mask of 0x7FFF7FFF… treats the data as packed 16-bit integers. This approach allows for an arbitrary sized data lane, i.e. one could in theory add packed 9-bit numbers.

To apply saturation, the carry-out vector for unsigned integers, or the overflow vector for signed integers, is then applied to adjust the summed values. In this example, the result is adjust to apply signed 8-bit saturation:

```
uint64_t ovrflw, sat;

ovrflw = (~(a ^ b)) & (result ^ b);
ovrflw &= mask2;

// flip sign of bytes that overflowed
result ^= ovrflw;

// stuff 1 bits into overflowed bytes
result |= ((ovrflw >> 7) * 0x7F);

// invert to give 0x7F or 0x80
result ^= (((result&ovrflw)>>7) * 0x7F);

// record any saturated bytes
sat |= ovrflw;
```

Unsigned 8-bit packed saturation would be applied as follows:

```
uint64_t carrys, sat;

carrys = (a & b) | ((a | b) & ~(a + b));
carrys &= mask2;

// write 0xFF to bytes that carried
result |= (carrys << 1) - (carrys >> 7);

// record any saturated bytes
sat |= carrys;
```

This approach allows for the simulation of SIMD operations in a portable manner using a high level language, and support non-standard packed data width.

3.0 Proposed ISA Extension

We have demonstrated several useful applications of the carry-out and overflow vectors to detect integer overflow, simulate arithmetic condition flags, accelerate array operations, and even emulate SIMD operations of arbitrary element width. The fundamental operation involved in all these cases is the efficient generation of a carry-out vector for addition and subtraction operations.

We propose an ISA extension to IA32 to make the carry-out generation be a single instruction, similar to an addition of subtraction but one that stores the carry-out bits instead of the result bits. For 64-bit operations, the full 64-bit carry-out vector would be generated. For smaller 8-, 16-, and 32-bit operations, the valid vector would be replicated in the high bits. The high two bits of such a result would always derive the correct Carry Flag and Overflow Flag, and bit 3 of such a result would give the correct Adjust Flag.

Two instructions would be needed: ADDCOUT, a 3-operand form of the carry-out generating addition, and SUBCOUT, a 3-operand form of the carry-out generating subtraction. The use of the 3-operand form, similar to the widely used IA32 "LEA" instruction, allows the carry-out vector to be generated in one clock cycle without the need to emit additional MOV instructions to copy registers. In most uses, the carry-out generation would be performed before the actual arithmetic operation.

For example, if adding two registers, the standard IA32 integer instruction form requires destroying one of the two input registers. Therefore, it is optimal to generate the carry-out vector first, as shown here:

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

The main benefits of this ISA extension – which would be accessible to high level languages such as C and C++ via compiler intrinsics – would be to allow for faster integer overflow detection, to allow for deferred overflow checking, to reduce CPU arithmetic flags emulation overhead in simulators, and to permit performing certain packed byte and packed bitfield operations using plain integers instead of SIMD operations.

4.0 Conclusions and Further Research

We have demonstrated the lazy flags based techniques for detecting unsigned and signed integer overflow in a CPU agnostic manner. These techniques are suitable for use with high-level languages, which makes them attractive for use in CPU simulators and bytecode interpreters.

We have also shown how the lazy flags approach is able to detect overflow on non-standard integer sizes, permitting more efficient handling of bitfields and packed data structures using ordinary integer operations in lieu of specific SIMD instruction capabilities.

Finally, we propose simple ALU instruction extensions for addition and subtraction to facilitate efficient generation of a carry-out vector.

5.0 References

[1] **Checked keyword**, http://msdn.microsoft.com/en-us/library/74b4xzyw%28VS.71%29.aspx

[2] **Options for Code Generation Conventions**, https://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/gcc/code-gen-options.html

[3] **Intel64 and IA32 Architectures Software Developer's Manual**, http://www.intel.com/Assets/PDF/manual/253665.pdf

[4] **A First Look at the Larrabee New Instructions**, http://software.intel.com/sites/billboard/archive/larrabee-new-instructions.php

[5] **Arithmetic Flags**, Darek Mihocka, http://www.emulators.com/docs/nx11_flags.htm

[6] **stdint.h**, http://en.wikipedia.org/wiki/Stdint.h

[7] **Signalling Integer Overflows in Java**, http://www.drdobbs.com/open-source/210500001

[8] **Virtulization Without Direct Execution**, Darek Mihocka, Stanislav Shwartsman, ISCA 2008, http://www.emulators.com/docs/VirtNoJit_Paper.pdf

[9] **Bochs IA32 Emulation Project**, http://bochs.sourceforge.net/

[10] **PowerPC Programmer's Reference Guide**, XER Register, https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600741775/$file/prg.pdf

[11] **Adjust Flag**, http://en.wikipedia.org/wiki/Adjust_flag