Extending General-Purpose Registers with Carry and Overflow Bits

Abstract

Many architectures have condition-code registers that include carry and overflow bits. Carry is used for multi-word precision arithmetic (e.g., for cryptography). Overflow is used for growable (arbitrarily long) integers available in many programming languages. Having a single instance of carry and overflow in a condition-code register makes it hard to make full use in a compiled programming language. We propose to remedy this by adding carry and overflow to every general-purpose register.

1 Introduction

Many architectures have carry and overflow bits in a condition code register.

Carry usually indicates an overflow after an unsigned addition, underflow after an unsigned subtraction[1], and this is used in (fixed-size unsigned) multi-word arithmetics. Carry is also generated and sometimes used in other instructions, in particular, comparison, multiply, and certain shift or rotate instructions.

The overflow bit indicates an overflow or underflow after a signed addition, subtraction, or multiplication. One of its uses is in implementing growable signed integers (also known as bignums, although the common case is that the number fits in one word).

The unique condition code register present in many architectures poses difficulties like other unique special-purpose registers. In particular, it limits the use of the condition codes in programming languages. Carry and overflow have hardly been exposed at the programming language level, and where they have been exposed, compilers have had problems making efficient use of the hardware feature.

Despite this problem, the importance of carry and overflow seems to have increased over time, as is witnessed by the addition of the ALC instruction in the IBM S/390, the addition of the ADX extension to Intel CPUs since Broadwell (2014) and to AMD CPUs since Zen (2017), and the addition of BOVC/BNVC to MIPS64r6 (2014). This increased importance is probably due to the increasing use of cryptography, and the increasing popularity of programming languages that support big integers.

We propose to add carry and overflow bits to the general-purpose registers instead of having a condition-code register. This makes these bits easier to deal with in compilers, as well as providing increased opportunities for instruction-level parallelism (Intel added the ADX extension for this purpose).

The main contribution of the present paper is the idea that carry and overflow are part of the general-purpose registers. First we motivate it by showing how carry and overflow are used (Section 2) and why a condition code register is inadequate (Section 3). Then we present the idea as an instruction-set extension (Section 4), and show its benefits (Section 5). Finally, we compare our extension with the approaches various architectures have taken (Section 7) over the decades.

2 Why carry and overflow are useful

There are architectures like RISC-V that do not have carry and overflow, and one can work around this lack, but at a significant cost.

[1] If the carry bit is set on an underflow (e.g., on x86-64), it is also known as borrow bit. Some architectures (e.g., ARM A64) set the carry bit on non-underflow and clear it on underflow; then it is called the carry bit even in subtraction.

```
#inputs: partm, partn, carry     #C equivalent: all variables are uint64_T
add  partsum0, partm, partn      #partsum0 = partm+partn
sltu carry0, partsum0, partm     #carry0   = partsum0<partm
add  partsum, partsum0, carry    #partsum  = partsum0+carry
sltu carry1, partsum, partsum0   #carry1   = parstsum<partsum0
add  carry, carry0, carry1       #carry    = carry0+carry1
#outputs: partsum, carry
```

Figure 1: Addition with carry-in and carry-out on RISC-V (based on https://gmplib.org/repo/gmp-6.2/file/402b9c4efacb/mpn/riscv/64/aors_n.asm)

2.1 Unsigned multi-precision arithmetics

For multi-precision addition $m + n$, one first adds the first word of m and n with a word-wide result and a carry, then adds the next two words and the carry, again resulting in a word and a carry, and so on. Figure 1 shows how this is coded on RISC-V.

This costs not only five instructions, but also three cycles of latency for the carry propagation. By comparison, on architectures with carry bits these five instructions can typically be replaced with one add-with-carry instruction, with typically one cycle of latency from carry-in to carry-out.

2.2 Growable Integers

Signed variable-length integers in various programming languages can have arbitrary length. At run time, in most cases the operands of, e.g., an addition fit in the smallest container available, but if the addition results in a signed overflow, the slow multi-word addition needs to be performed. This means that a signed overflow test is needed after every addition of two such integers.

As an example, in the non-JIT variant of the BC run-time system for Racket-8.6 a value is represented by a machine word; the machine word is either such a small integer (tagged with 1), or a pointer to a more elaborate (boxed) representation `Scheme_Object` of all other kinds of values, including larger integers. After checking the tags, the addition of two small tagged integers is performed by:[2]

```
Scheme_Object *ADD_tagged(
       Scheme_Object *tagged_a,
       Scheme_Object *tagged_b)
{
  intptr_t a = ((intptr_t)tagged_a)>>1;
  intptr_t b = ((intptr_t)tagged_b)>>1;
  intptr_t r;
  Scheme_Object *o;
  r = (uintptr_t)a + (uintptr_t)b;
  o = (Scheme_Object *)
         ((((uintptr_t)r)<<1)|1);
  r = ((intptr_t )o) >> 1;
  if (b == (uintptr_t)r - (uintptr_t)a)
    return o ;
  else
  return ADD_slow (a , b) ;
}
```

This code uses the additional bit originally used by the tag to check for the overflow, yet the overflow test still looks quite complex: The result of the addition is first tagged (giving `o`) and then untagged (into `r`) again, so that `r` now contains the wrong sign in case of an overflow. Then `r` is checked for correctness by trying to undo the addition. In the common (not overflowing) case, `o` is the tagged result, otherwise `ADD_slow()` produces a boxed bignum.

The resulting x86-64 code is

```
sarq    %rdi
sarq    %rsi
leaq    (%rdi,%rsi), %rax
leaq    1(%rax,%rax), %rax
movq    %rax, %rdx
sarq    %rdx
subq    %rdi, %rdx
cmpq    %rdx, %rsi
jne     .L6
```

[2] The shown code is derived from the original `ADD` function by putting the untagging at the callee rather than the caller side, expanding the macros, and simplifying the resulting code.

```
        ret
.L6:
        jmp     ADD_slow@PLT
```

This code can be optimized by using the overflow flag of the x86-64 architecture (using source code[3] that employs the GCC feature `__builtin_add_overflow`):

```
        leaq    -1(%rsi), %rax
        addq    %rdi, %rax
        jo      .L8
        ret
.L8:
        sarq    %rsi
        sarq    %rdi
        jmp     ADD_slow
```

However, the same source code results in the following RISC-V code:

```
ADD_tagged1:
        addi    a4,a1,-1
        mv      a5,a0
        add     a0,a0,a4
        slti    a3,a5,0
        slt     a4,a0,a4
        bne     a3,a4,.L8
        ret
.L8:    srai    a1,a1,1
        srai    a0,a5,1
        tail    ADD_slow
```

This is still shorter than the code produced for the `ADD_tagged` source code above, but this example shows that working around the lack of signed-integer overflow has its costs.

3 Why a condition code register is inadequate

As can be seen in the `ADD_tagged` example above, an overflow bit in a condition-code register is better than no overflow bit at all, but condition code registers suffer from the same shortcomings as other special-purpose registers.

One disadvantage is that there is usually only one carry flag, which can be insufficient for the natural expression of an algorithm. In particular, in multi-precision multiplication each component multiplication produces two words of multiplication results[4] that have to be added to the intermediate results, which can be done without overhead if the architecture has at least two carry bits [OGGF12, Table 2]. So Intel added the ADCX and ADOX instructions, where ADOX treats the O(verflow) bit as another carry (and ADCX does not overwrite O, unlike ADC).

Another disadvantage of condition-code registers is that many instructions tend to overwrite condition codes, which may still be needed. E.g., the MUL instruction on x86-64 overwrites the C and O flags, so in multi-precision multiply there would be overhead required for preserving these bits across the MUL instruction. To avoid that problem, Intel added the MULX instruction that does not change the condition-code register.

The latter problem is a major obstacle in making use of condition-code bits in compiler-generated code: E.g., the clang compiler supports `__builtin_addcl` for performing an add-with-carry, yet for a straightforward implementation of multi-precision addition results in inefficient code on both x86-64 (16 instructions with at least 6 cycles of latency for two iterations) and on ARM A64 (10 instructions with at least 3 cycles of latency for one iteration).[5] This code converts the carry bit into a value in a general-purpose register right after the addition and then does not use the add instruction with carry input in all (ARM A64) or half (x86-64) of the iterations. By contrast, the corresponding gmp-6.2.1 code written in assembly language takes 17 instructions with at least 4 cycles of latency per 4 iterations (x86-64[6]) and 12 instructions with at least 4 cycles of latency per 4 iterations (ARM A64[7]).

Our explanation for the non-use of add with carry input is that other operations between the invo-

[3] https://godbolt.org/z/TroqhsrrM

[4] While a number of architectures use separate instructions to generate the upper and lower result words, it is a good idea to keep them adjacent to each other to allow the microarchitecture to fuse them and use the multiplier only once for the sequence; e.g. a specific sequence is suggested for this for RISC-V [WA17, Section 6.1].

[5] https://godbolt.org/z/3916jffdq

[6] https://gmplib.org/repo/gmp-6.2/file/gmp-6.2.1/mpn/x86_64/aors_n.asm#l142

[7] https://gmplib.org/repo/gmp-6.2/file/gmp-6.2.1/mpn/arm64/aors_n.asm#l107

cations of `__builtin_addcl` might be (and indeed are) compiled to code that overwrites the condition-code carry, so the compiler needs to save the condition code into a general-purpose register which is under the control of the register allocator. To make better use of the carry flag, the compiler would need to select code that does not change the carry flag for all these other operations; given that the use of an explicit carry flag is rare in most code, it is not surprising that compilers are not set up for doing this.

4 How to add carry and overflow to general-purpose registers

Our solution to these problems is to enhance the general-purpose registers with bits for carry and overflow.

In most of this section we present this as an extension to RISC-V, but the basic idea can also be applied to other architectures, including architectures that have a condition code register with carry and overflow.

The benefits to RISC-V and other architectures without carry and overflow flags are the reduced instruction count and latency for multi-precision arithmetic and for overflow detection, both in assembly language and in compiled languages.

The benefits to architectures that have carry and overflow flags in condition codes are: 1) Code that has more than one carry bit alive at the same time, e.g., multi-precision multiplication can be written without having to reify some carry bits in a register. 2) Compilers have an easier time managing these bits, allowing for compiled code that is much closer in performance to hand-written code by assembly programmers.

We first describe one particular extension for RV64G (64-bit RISC-V with the general-purpose extensions), later describe an alternative to a part of our design (Section 4.5), and finally a possible clean-sheet variant (Section 4.6).

4.1 Extending the general-purpose registers

Each general-purpose register gets two additional bits. The way these bits are set depends on the instructions; below we describe the typical meaning.

carry An instruction typically sets this if the computation overflows the unsigned range when the inputs are interpreted as unsigned.

overflow An instruction typically sets this if the computation overflows or underflows the unsigned range when the inputs are interpreted as signed.

In x0 (the register always containing zero) the carry and overflow bits are 0.

4.2 Adapting existing RV64G instructions

All existing instructions write the carry and overflow bits along with all the other bits of their destination register (if they have one). Unless otherwise noted, the carry and overflow bits are cleared.

None of the existing instructions use the carry or overflow bits as inputs (not even for computing carry or overflow).

Addition instructions compute the carry bit as the 65^{th} bit of the sum of the zero-extended 64-bit operands. There are various ways to compute signed overflow; one way is to compute the 65^{th} bit of the sum of the sign-extended operands and xor it with the 64^{th} bit.

Subtraction instructions work like additions that first two's-complement the subtrahend in 64 bits. As a result, the carry bit is set if the subtraction did not underflow and is clear on underflow. I.e., in the schism between carry and borrow, our extension is on the carry side.

Bitwise operations (and, or, xor) operate on all the bits, including carry and overflow. This allows to perform boolean operations on the carry and overflow bits before checking them.

RISC-V normally implements a register-to-register move (`mv`) as addition with `x0`. This clears both carry and overflow, so after adding the extension the preferred way for the register-to-register move is to use `or rd, x0, rs2` or

`xor rd, x0, rs2`; unfortunately, there are no compressed (16-bit) encodings (yet) for either of these instructions.

Shift-left instructions set the carry bit if any of the shifted-out bits is set (i.e., if the shift, interpreted as unsigned multiplication by a power of two, overflows), and set the overflow bit if any of the shifted-out bits are not equal to the sign bit of the result.

The lower-part multiplication instruction (`mul`) sets the carry bit if the result of the unsigned multiplication does not fit in 64 bits, and sets the overflow bit if any of the upper bits of the result of signed multiplication is different from the sign bit of the result.

Division and remainder instructions set carry on division by zero, and set the overflow bit on division by zero and on division overflow (signed division of the smallest value by -1).

The instructions for dealing with 32-bit values in RV64G (those with the `w` suffix) perform the 32-bit analogues of the operations described above, i.e., they treat the 32^{nd} bit as sign bit, the carry and overflow bits represent the unsigned or signed overflow beyond the 32-bit range.

The other instructions clear the carry and overflow bits.

4.3 New instructions

This section describes the new instructions in ordinary (user-level) code.

`addc rd, rs1, rs2`

The intention of this instruction is that rs1 contains the result of an addition instruction. `Addc` adds the carry bit of rs2 to the 65-bit unsigned and signed data that the earlier addition has left in rs1. `Addc` can also be used with the result of some other instruction in rs1, and will produce a deterministic result in the carry and overflow bits (by just interpreting them as if they came from an addition), but these bits of the result may not be very useful.

For the unsigned case, the 65^{th} bit is the carry bit.

For the signed case, the 65^{th} bit has to be reconstructed from the overflow bit: Since the overflow bit is the xor of the 65^{th} and 64^{th} bit, the 65^{th} bit can be reconstructed by xoring the 64^{th} bit with the overflow bit. After adding the carry bit of rs2, the overflow has to be recomputed again by xoring the 65^{th} and 64^{th} bit of the result.

The sequence

```
add  r3, r1, r2
addc r3, r3, r4
```

performs an add of r1 and r2 with carry-in from r4, giving a result with carry-out and overflow in r5. This fits nicely in RISC-V's usual scheme of having two input operands. For some architectures one instruction with three input operands may be a better fit. For RISC-V, the instruction decoder can fuse a sequence of two such instructions into a single fused instruction [CDPA16].

Note that, even if the instructions are executed separately, the latency of multi-precision arithmetic is one cycle per result word, because the only long dependence chain is from one `addc` instruction to the next; the `add`s are all independent of each other.

For subtraction with carry-in, one can use the following sequence:

```
not  r3, r1
add  r3, r2, r3
addc r3, r3, r4
```

This subtracts r1 from r3, with carry-in from r4. If multi-precision subtraction is important enough, we can also add a `subc` instruction that works with the `sub` instruction. Or, alternatively, the instruction decoder could recognize the sequence above and replace it with, e.g., a three-operand subtract-with-carry micro-instruction.

Like for multi-precision addition, the latency of multi-precision subtraction is one cycle per result word, due to the carry chain.

`bo rs1, rs2, target`

If the overflow bit of rs1 or the overflow bit of rs2 is set, branch to the target. Checking for two overflows at once can reduce the number of branch instructions needed for catching overflows in code that checks the overflow status of all operations. Picking apart which register contains the overflow can be left to the slow overflow-handling path.

Other instructions?

We considered including a branch on carry (analogous to `bo`), but it can be replaced with the sequence

```
or r3, r1, r2   #only if checking two registers
addc r3, x0, r3
bnez r3, target
```

and it's not clear that branch on carry is needed frequently enough to merit a separate instruction. Alternatively, the `bo` instruction could be replaced with an instruction that checks the overflow flag of one source register and the carry flag of the other source register.

X86-64 has instructions that rotate through the carry bit (`rcl rcr`). However, they seem to be no longer useful enough to justify adding an instruction for them to an architecture that does not have them already.[8]

4.4 Dealing with memory

While we have widened the registers, instructions that deal with memory are still limited to 64 bits or less, so we lose the contents of the carry and overflow bits when storing values to memory in the straightforward way.

... in user-level code

When a register is spilled and later reloaded (typically around a function call), the carry and overflow bits are gone. That's normally no problem: On architectures with flags in a special-purpose register, that register is normally not preserved across calls, either.

In the rare case when it is necessary to preserve these bits across spilling and reloading, the bits can be reified into the regular 64 bits of a general-purpose register by using the `addc` and `bo` instructions and the resulting values can be stored into memory in a separate word. On reloading the overflow bit can be used where it is in the general-purpose register, while the carry bit can be transferred into the carry bit of a general-purpose register by adding -1.

[8] https://stackoverflow.com/questions/26913354/practical-uses-for-rotate-carry-left-right

The biggest problem of this approach is that it loses a part of the compiler benefit of the idea proposed in this work. The compiler now has to keep track of which registers contain live carry and/or overflow bits, and has to emit code for saving and restoring them when necessary. It is probably very rarely necessary, so the main problem is not in the resulting code, but in the compiler complexity.

One may want to avoid that complexity by providing a not-too-expensive way to spill and refill carry and overflow along with the rest of the register, but we have not devised a way that we do not deem too expensive, in particular on an out-of-order implementation.

... on context switching

On context switching the carry and overflow bits have to be preserved. If we found good instructions for user-level spilling and reloading, we could use them for context switching as well. For now our approach is:

Every store instruction updates two bits in a 64-bit special-purpose register `storeextra`; the register number of the register containing the stored value determines which two bits in `storeextra` are updated; e.g., writing r1 updates bits 2 and 3. Updates from stores from different registers are indepenent and may be performed out-of-order. After the general-purpose registers are stored, the value in `storeextra` is transferred to a general-purpose register; this operation needs to serialize the pipeline in an implementation with out-of-order execution, i.e., it has to wait for all earlier instructions to finish. Finally, that value is stored.

For reloading, first the word containing all the carry and overflow bits is loaded and transferred into a special-purpose register `loadextra`. Given that we do not want to keep track of this register in the out-of-order execution engine, the pipeline has to be serialized at this point, but this can be the same pipeline drain as before reading `storeextra`. Afterwards, special load instructions `ldx` load the 64 regular bits from memory and fill the carry and overflow bits from the bits in `loadextra` corresponding to the target registers.

This approach is not very elegant, but it's relatively cheap to implement in hardware. How about using it for spilling? That's possible in principle, but the pipeline serialization makes this approach

slow and the values would have to be reloaded into the same registers from which they were spilled. The slowness of the serialization is less of a problem for context switching, because context switching is less frequent, and is already a relatively slow operation.

4.5 Alternative

One variant we have considered is the register sibling of a sticky overflow bit: If the overflow bit of any of the source registers of an arithmetic instruction is set, set it in the destination register. This is probably not useful for growable integers, but it may be useful for reducing the number of `bo` invocations when implementing integers that produce errors on overflow. However, other uses of the overflow bit (e.g., for growable integers) need the non-propagating variant, and we see no way to accomodate both without adding instructions.

4.6 Clean-sheet architecture

Many possible clean-sheet architectures are possible, but we do not want to cover every possibility, so for this section we describe one based on RISC-V, but that is not required to be compatible with RISC-V.

Many architectures have four essential condition-code bits: zero, sign, carry, and overflow. With our extension every general-purpose register contains all that information for the operation that generated it: Sign, carry, and overflow are there as individual bits, and zero holds if all regular bits of a register are zero.

So we can leave away `slt` and `sltu`, and instead use `sub` and `addiu` to produce comparison results in a general-purpose register. Then a conditional-branch instruction can read from that register and branch based on a number of conditions, similar to architectures with a condition-code register, except that we use a general-purpose register instead of a condition-code register. Likewise, we can also have an instruction that sets a destination register to 0 or 1 depending on whether a condition (in the source register) is true.

For cases that are covered by the RISC-V's branch and set instructions, this design would need an additional instruction (`sub`) to achieve the same result. But there are also cases that take as many instructions (e.g., when branching after comparing with a constant) or are shorter to express in this design, e.g., when you check for the flags resulting from an addition, shift, or a boolean operation on the flags.

Another advantage of this design is that the branch instructions now only need to encode one register and a (slightly longer) condition instead of two registers and a condition, leaving more bits for the branch target address.

5 Benefits

To evaluate the potential benefits, we created some traces of the instructions executed by functions from the multi-precision library gmp on RV64GC, and computed the benefits in instruction count and minimal latency that this code would enjoy if rewritten to use `addc`. We selected the low-level functions `mpn_add_n()` and `mpn_mul_n()`, which are very good cases for our extension. In real-world usage of multi-precision arithmetic, other code is executed between calls to these functions, so the benefits will be much less.

A 1024-bit `mpn_add_n()` performs 174 RV64GC instructions with an overall latency of 51 cycles (assuming a latency of 3 for `ld`, 1 for `add` and `sltu`, and 0 for `mv`); the latency is due to the carry chain. By replacing the current RISC-V carry-computing idiom with `addc`, the executed instructions can be reduced to 126 instructions (factor 1.38), and the latency can be reduced to 20 cycles (factor 2.55). Here's the comparison of one iteration of the inner loop (with an unrolling factor of 2):

```
ld   a4,0(a1)        ld   a4,0(a1)
ld   a6,0(a2)        ld   a6,0(a2)
addi a3,a3,-2        addi a3,a3,-2
addi a1,a1,16        addi a1,a1,16
add  t0,a4,a6        add  t4,a4,a6
sltu t2,t0,a4        addc t4,t4,t1
add  t4,t0,t6
sltu t3,t4,t0
sd   t4,0(a0)        sd   t4,0(a0)
add  t6,t2,t3
ld   a5,-8(a1)       ld   a5,-8(a1)
ld   a7,8(a2)        ld   a7,8(a2)
addi a2,a2,16        addi a2,a2,16
addi a0,a0,16        addi a0,a0,16
```

```
add  t1,a5,a7        add  t1,a5,a7
sltu t2,t1,a5        addc t1,t1,t4
add  t4,t1,t6
sltu t3,t4,t1
sd   t4,-8(a0)       sd   t4,-8(a0)
add  t6,t2,t3
bnez a3, loop        bnez a3, loop
```

A 1024-bit × 1024-bit multiplication performs 4109 RV64GC instructions. They could be reduced to 3613 instructions (factor 1.14). However, we think that it is possible to keep more intermediate results in registers; this would help both versions, but would increase the improvement factor from using our extension. Here is a comparison of one iteration of the inner loop. The latency chain works through the loop-carried dependency through a6, and is reduced from 3 cycles to 2 cycles (factor 1.5), or 48 to 32 cycles for the complete inner loop.

```
ld    a7,0(a1)       ld    a7,0(a1)
addi  a1,a1,8        addi  a1,a1,8
ld    a4,0(a0)       ld    a4,0(a0)
addi  a0,a0,8        addi  a0,a0,8
mul   a5,a7,a3       mul   a5,a7,a3
addi  a2,a2,-1       addi  a2,a2,-1
mulhu a7,a7,a3       mulhu a7,a7,a3
add   a5,a5,a4       add   a5,a5,a4
add   a6,a6,a5       add   a6,a6,a5
sltu  a4,a5,a4
add   a4,a4,a7       addc  a4,a7,a5
sltu  a5,a6,a5
sd    a6,-8(a0)      sd    a6,-8(a0)
add   a6,a4,a5       addc  a6,a4,a6
bnez  a2, loop       bnez  a2, loop
```

We also planned to show results for growable integers in racket-8.7, but ran out of time for preparing this paper.

6 Costs

This section discusses the costs that this extension incurs.

6.1 Hardware

We did not implement these features in hardware, so we have to go by rough estimates.

For RISC-V, the proposed extension means that we have to:

- Add two bits to each general-purpose register; this is 3% more bits for the registers, but far less area increase overall. For comparison: when IA-32 was extended to AMD64, the size of the general-purpose registers doubled, but extra hardware cost was reported to be 5% overall (which included changes in the decoder and in the ALUs in addition to the longer registers).

- Hardware in the ALU for generating the carry and overflow flags. These flags only take a few gates to compute, as evidened by the fact that already the 6502 with its 3500 transistors generated carry and overflow.

- The registers `loadextra` and `storeextra`. These will likely be located in the load/store unit. The store and `ldx` instructions would need to carry the physical register number R of the stored-from or loaded-into register with them, and these registers would need addressing hardware for accessing the carry and overflow bits corresponding to R. In an out-of-order execution engine, this is probably a vanishing part of the cost (these two registers are not renamed); for a small in-order implementation the cost may be noticable, though.

- The three additional instructions `addc bo ldx` need to be decoded and executed. Executing them is a minor variations on the execution of `add`, branch instructions, and `ld`, respectively, so the hardware cost for the execution of these instructions is probably small. We expect the decoding cost of a few additional instructions which follow the typical encoding schemes of RISC-V (e.g., two source registers) to be small, too.

An architecture such as AMD64 or ARM A64 that has carry and overflow bits already, but in condition codes, already incurs many of these hardware costs. In particular, a number of AMD64 implementations (such as the most recent performance core from Intel, Golden Cove) renames the flags with the integer registers[9], and likely keeps the flags as extra bits in the general-purpose register file (re-

[9] https://chipsandcheese.com/2022/11/05/amds-zen-4-part-1-frontend-and-execution-engine/

naming them together would not make much sense otherwise).

Compared to such an implementation, an implementation of an architecture where carry and overflow are explicitly part of the general-purpose registers is actually simpler (and probably costs less hardware), because the implementor does not have to deal with cases such as the register being updated while the flags are not, or the flags being updated while the registers are not.

6.2 System Software

The context-switch code has to be adapted to dealing with `storeextra` and `loadextra`. The result will be slower than a context switch that does not have to deal with this extension.

The main performance cost here is serializing the pipeline after transferring the data into `loadextra` and before transferring the data from `storeextra`. The serializing instruction `cpuid` costs 100–250 cycles on Intel's Ice Lake CPU and 140–170 cycles on AMD's Zen4 [Fog22]. Lmbench [MS96] (with 0KB of data exchanged between processes[10]) reports 1.72μs of context switch latency on a 4.2GHz Tiger Lake, i.e., about 7000 cycles; so adding this serializing instruction increases the context switch cost by 1.4%-3.5%. The overall cost on the system depends on the frequency of context switches. For a desktop and a laptop running Linux the tool `sar -w` reported 1600–4000 context switches per second; with 4000 context switches per second, 250 cycles and 4.2GHz, the additional overhead would cost 0.02% of the CPU time. Of course, for a busy server the number of context switches and the contribution of the additional overhead could be quite a bit higher; but a busy server probably also uses more cryptography and benefits from performance improvements from the proposed extension.

Other costs of this instruction set extension (like every instruction set extension) are the cost of extending the tool chain (assemblers, disassemblers, debuggers, compilers); and of course, to benefit from it, you want to use it in libraries like gmp, and language implementations like Racket.

A C compiler can ignore the extra bits and incur no cost and no benefit from them. But if it makes use of them, it has to either ensure that carry and overflow do not survive across calls (typically by restricting code generation, resulting in a suboptimal benefit); or it has to perform liveness analysis for the carry and overflow flags, and has to spill and reload them in the rare case where they are indeed alive at a call site.

[10] `lat_ctx -N 1000 -s 0 2`

6.3 Is the benefit worth the cost?

We have been unsuccessful in finding numbers on the proportion of time spent in multi-precision arithmetics and in bignum processing, so we look at what architects have done who know more about the needs of their customers:

In particular, Intel added the ADX extension in 2014 for improving multi-precision arithmetic beyond what a single carry flag allows, so apparently they have customers for whom multi-precision arithmetic matters a lot. AMD adopted ADX in 2017.

MIPS has added the BOVC instruction, which helps bignum addition (but only addition).

On the other hand, one might argue that fast carry processing has not been important enough (yet?) for MIPS, Alpha, and RISC-V to add a carry flag. One reason for that may be the disadvantages that a single carry flag or condition-code register has in implementation and in usage. The proposed extension relieves some of these disadvantages, however.

One cost that the proposed extension has that the architectures with condition codes do not have is a significantly increased context switch cost. It may be worthwhile to add more hardware to allow faster context switching (i.e., without pipeline serialization).

7 Previous work

There has been a lot of variation in dealing with carry and overflow in instruction sets.

The IBM S/360 [36022] has, e.g., signed (e.g. `A`) and unsigned (e.g., `AL`) addition instructions, which indicate type-specific overflow through instruction-specific settings of the two condition-code bits; signed overflows trap unless masked by bit 20 in the program status word. Instructions that support carry-in, such as `ALCR` were added with ESA/390 in 1990.

Several architectures have flags for Negative/Sign, Zero, Carry, and oVerflow (NZCV) flags in a condition code register. While the implementors of x86-64 and ARM A64 CPUs have demonstrated that these condition codes are no obstacle to high-performance implementations, they still limit what kind of code can be written to use them. This has been demonstrated by Intel's ADX extension, which added three instructions to work around the limitations coming from the use of condition codes in existing instructions condition codes [OGGF12].

Some other architectures have therefore tried to avoid condition-code registers:

Instruction sets in the MIPS [MIP14] family (e.g., Alpha, DLX, RISC-V) have no condition codes and replace them with various combinations of comparison instructions that put their results in registers and compare-and-branch instructions; MIPS, Alpha and DLX (but not RISC-V) also have signed arithmetic instructions that trap (but, e.g., C compilers generate the "unsigned" instructions for signed arithmetic instead). MIPS64r6 has added `BOVC` and `BNVC`; `BOVC` branches (and `BNVC` does not branch) if the signed addition of its two operands overflows. The lack of addition with carry leads to the five-instruction three-cycle workaround (see Section 2).

The 88000 [Mot90] has a compare instruction that writes a collection of flags to a general-purpose register. But it also has a carry bit in the processor status register PSR, with carry-in and carry-out explicitly controlled in the instruction word.

Power [IBM05] has a condition code register that can hold 8 4-bit condition codes, improving on the situation with a single condition code. However, many integer instructions can only write to CR0, but at least the programmer can control (with a bit in the instruction encoding) whether or not the instruction overwrites CR0. These condition codes don't include carry (CA) and overflow (OV), which are kept separate in the XER register; Power also has summary overflow (SO), i.e., a sticky overflow bit. Writing to OV and SO is controlled by another bit in the instruction encoding. In addition to normal add instructions that perform neither carry-in nor carry-out, there are carry-out instructions (e.g., `addc` and carry-in carry-out instructions (e.g., `adde`). As a result, while there is still only one carry and one overflow bit, at least it is easy to avoid accidentally overwriting them.

8 Conclusion

Carry and overflow are useful for multi-precision arithmetic and growable integers. In existing instruction sets they are either single-instance flags (even in architectures that avoid single-instance comparison results like the 88000 and Power), or they are not directly supported and have to be synthesized from sequences of other instructions. We present an extension to RV64G that adds carry and overflow bits to the general-purpose registers. The extension also adds three instructions (`addc bo ldx`) and two special-purpose registers (`storeextra loadextra`). The benefits of this extension are reductions in executed instructions and in latency when performing multi-precision arithmetic, and a reduction in executed instructions when dealing with growable integers.

References

[36022] Wikibook: 360 assembly. https://en.wikibooks.org/w/index.php?title=360_Assembly/360_Instructions&stableid=4078098, 2022. 7

[CDPA16] Christopher Celio, Daniel Dabbelt, David A. Patterson, and Krste Asanović. The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v. Technical Report UCB/EECS-2016-130, Berkeley, 2016. 4.3

[Fog22] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. https://www.agner.org/optimize/instruction_tables.pdf, 2022. 6.2

[IBM05] IBM. *PowerPC User Instruction Set Architecture – Book I*, version 2.02 edition, 2005. 7

[MIP14] MIPS. *MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS64 Architecture*, revision 6.01 edition, 2014. 7

[Mot90] Motorola, Inc. *MC88100 RISC Microprocessor User's Manual*, second edition, 1990. 7

[MS96] Larry McVoy and Carl Staelin. `lmbench`: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, pages 279–294, 1996. 6.2

[OGGF12] Erdinc Ozturk, James Guilford, Vinodh Gopal, and Wajdi Feghali. New instructions supporting large integer arithmetic on Intel architecture processors. White Paper 327831-001, Intel, 2012. 3, 7

[WA17] Andrew Waterman and Krste Asanovič, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, document version 2.2 edition, May 2017. 4