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 only a single instance of carry and overflow makes it difficult to make full use in a compiled programming language, and also makes multi-word multiplication slower. We propose to remedy this by adding carry and overflow to every general-purpose register.

1 Introduction

Multi-word integer addition carries the (unsigned) overflow of a single-word addition to the addition of the two next-higher words of the addends. Similarly for multi-word subtraction and comparison. Multi-word addition and its carries also play a role in the implementation of multi-word multiplication.

Arbitrary-precision (signed) integer arithmetic (also known as bignum arithmetic) needs to know whether signed addition, subtraction or multiplication overflows.

A common architectural way to provide this functionality is through a carry and an overflow bit in a condition-code register (e.g., present in the IA-32, AMD64, and ARM A32, T32, and A64 instruction sets).

However, such a unique condition code register 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.

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 give an overview of how existing architectures deal with the problem (Section 3). Then we present the idea as an instruction-set extension (Section 4), and show its benefits (Section 5).

2 Motivation

2.1 Unsigned multi-precision arithmetics

The main use for carry is (unsigned) multi-precision arithemetics, used, e.g., for cryptography. An architecture without carry bit, e.g., RISC-V, takes 5 instructions with a latency of three cycles[1] for an internal addition of two words (Fig. 1), in comparison to one `adc` instruction on AMD64 with one cycle of latency on recent high-end implementations.

For multi-precision multiplication, Fig. 2 shows the shortcomings of having one dedicated carry bit (in the AMD64 architecture without the ADX extension) [OGGF12, Table 2]:

- In each step the `mulx` instruction produces two words. With ADX, these two words are added to the respective words of the intermediate result (from other steps in the algorithm) right away, and they produce two carry chains, one for the low word and one for the high word of each step.[2] The instruction `adcx` uses the C bit as carry-in and carry-out bit and does not overwrite the O bit; the instruction `adox` uses the O (normally overflow) bit as carry-in carry-out bit and does not overwrite C.

- Without ADX, there is only the C bit for storing carry, and the computation is organized

[1] assuming a one-cycle latency for simple ALU operations and comparisons

[2] You may wonder if architectures that produce the low and high words with separate instructions (e.g., ARM A64's `mul` and `umulh` instructions) have this problem. They do, because you don't want to perform separate loops for dealing with the low and the high results, so you still either have two dependence chains or need workarounds involving extra instructions. Moreover, it is a good idea to keep the two instructions adjacent to each other to allow the microarchitecture to fuse them (so the multiplier is used only once). E.g., RISC-V suggests a specific sequence [WA17, Section 6.1].

```
#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)

```
with ADX                    without ADX
mulx TMP1, R'1, [pA+8*2]    mulx TMP1,R'1, [pA+8*2]
adox R'1, R2                add R'1, R2
adcx R3, TMP1               adc TMP1, 0
                            add R'1, TMP2
                            adc TMP1, 0
```

Figure 2: Two ways ways to code one step in multi-precision multiplication on AMD64 (from [OGGF12])

differently: It adds the intermediate result R2, the low-word result R'1 of the most recent multiplication and the high-word result TMP2 of the previous multiplication, and puts the result in R'1. It adds the carry-outs of these additions to the high-word result of the current multiplication. This approach needs two additional instructions per step. Each step also costs two cycles of latency, compared to one with ADX.

While this shows that having a carry input and carry output of an addition is useful, the implementation as a single dedicated bit, or even as two special-purpose bits (as in ADX) makes them hard for compilers to manage.

E.g., clang has the extension `__builtin_addcl()` for performing an add with carry-in carry-out; however, while AMD64 and ARM A64 each have 1 instruction (`adc` and `adcs`, respectively) that performs the complete functionality of `__builtin_addcl()`, clang-17.0.1 (without unrolling) translates this builtin to 6 instructions on AMD64 and 4 instructions on ARM A64.[3] That's because these instructions use the carry bit for carry-in and carry-out, but many instructions write to this bit, too, including the instructions that clang uses for loop control. Therefore clang takes the carry-in from a general-purpose register and produces the carry-out in a general-purpose register (and it's not particularly clever in doing that).

[3] https://godbolt.org/z/99PdqvMTb

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, a slow multi-word addition needs to be performed. This means that a signed overflow test is needed after every addition of two small growable 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 the code shown in the upper part of Fig. 3.

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.

This code can be improved by using the GCC extension `__builtin_add_overflow()` (lower part of Fig. 3). In this case, the overflow flag is used right away and does not need to be preserved, resulting

[5] 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. The C and assembler code for the lower part can also be found at https://godbolt.org/z/TroqhsrrM

```
/* close to original */
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) ;
}
```

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

```
/* using __builtin_add_overflow() */
Scheme_Object *ADD_tagged1(
       Scheme_Object *tagged_a,
       Scheme_Object *tagged_b)
{
  intptr_t a1=(intptr_t) tagged_a;
  intptr_t b1=(intptr_t) tagged_b;
  intptr_t r ;

  if (!__builtin_add_overflow(a1,(b1-1),&r))
    return (Scheme_Object *)r;

  intptr_t a = ((intptr_t)tagged_a)>>1;
  intptr_t b = ((intptr_t)tagged_b)>>1;
  return ADD_slow (a , b );
}
```

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

  #RISC-V
  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
```

Figure 3: Two versions of adding two small growable integers in the non-JIT Racket-8.6 BC run-time system[5]and the resulting AMD64 and RISC-V code

in good code produced for AMD64.

However, for RISC-V the resulting code is quite a bit longer, because RISC-V has to synthesize the overflow result by performing three comparisons. This is still shorter than the code produced for the `ADD_tagged` source code without `__builtin_add_overflow()`.

3 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, among them AMD64 and ARM A64, have flags for Negative/Sign, Zero, Carry, and oVerflow (NZCV) in a condition code register. Implementations of these architectures with out-of-order execution (dominant in every-

thing from smartphones to servers) need to have many physical condition-code registers to avoid slowdowns from write-after-write hazards. E.g., Intel's Golden Cove P-core has 280 physical flags registers, as many as physical integer registers.[6]

Yet, the architectural interface of providing only one architectural flags register means that Intel had to add the ADX extension [OGGF12] to allow using two carry chains in multi-precision arithmetic, as we saw in Section 2.1. We also saw the difficulties that compilers have with this kind of interface.

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, covering the case discussed in Section 2.2. The lack of addition with carry-in carry-out leads to the five-instruction three-cycle workaround (see Section 2.1).

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.

Overall, the importance of carry and overflow seems to have increased over time, as is witnessed by the addition of the `ALCR` instruction in the IBM ESA/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.

Our explanation for the non-use of add with carry input is that other operations between the invocations 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 materialize some carry bits in a general-purpose 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.

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.

[6] https://chipsandcheese.com/2021/12/21/gracemont-revenge-of-the-atom-cores/

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.[7]

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 (if necessary) `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.

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 on an implementation with out-of-order execution.

... 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.

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

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.

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). Figure 4 shows a comparison of one iteration of the inner loop (with an unrolling factor of 2).

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. Figure 5 shows 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.

For the growable integers in the Racket implementation, the common path of the code for `ADD_tagged1` (Fig. 3) shrinks from 7 to 5 instructions (factor 1.4). The saved instructions are on the dependence path of a predictable branch, so their latency is usually not an issue (Fig. 6).

6 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. Our solution is to add carry and overflow to general-purpose registers. This idea can be used in any architecture. To present something concrete, we describe an extension to RISC-V, which adds these bits and 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. 3

[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

[IBM05] IBM. *PowerPC User Instruction Set Architecture – Book I*, version 2.02 edition, 2005. 3

[MIP14] MIPS. *MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS64 Architecture*, revision 6.01 edition, 2014. 3

[Mot90] Motorola, Inc. *MC88100 RISC Microprocessor User's Manual*, second edition, 1990. 3

[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. 2.1, 2, 3

[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. 2

```
RV64G                 extended RV64G
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
```

Figure 4: Inner loop body of multi-precision addition

```
#RV64G               #extended RV64G
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
```

Figure 5: Inner loop body of multi-precision multiplication

```
#RV64G                  #extended RV64G
  addi a4,a1,-1           addi a4,a1,-1
  mv   a5,a0              mv   a5,a0
  add  a0,a0,a4           add  a0,a0,a4
  slti a3,a5,0
  slt  a4,a0,a4
  bne  a3,a4,.L8          bo   a0,zero,.L8
  ret                     ret
.L8:                    .L8:
  srai a1,a1,1            srai a1,a1,1
  srai a0,a5,1            srai a0,a5,1
  tail ADD_slow           tail ADD_slow
```

Figure 6: Growable integer addition