[[!tag standards]]

Analysis </>

DRAFT SVP64

- Revision 0.0: 21apr2022 https://www.youtube.com/watch?v=8hrIG7-E77o
- Revision 0.01: 22apr2022 removal of msubed because sv.maddedu and sv.subfe works
- Revision 0.02: 22apr2022 128/64 scalar divide, investigate Goldschmidt
- Revision 0.03: 24apr2022 add 128/64 divmod2du, similar loop to maddedu
- Revision 0.04: 26apr2022 Knuth original uses overflow on scalar div
- Revision 0.05: 27apr2022 add vector shift section (no new instructions)

Introduction

This page covers an analysis of big integer operations, to work out optimal Scalar Instructions to propose be submitted to the OpenPOWER ISA WG, that when combined with Draft SVP64 give high performance compact Big Integer Vector Arithmetic. Leverage of existing Scalar Power ISA instructions is also explained.

Use of smaller sub-operations is a given: worst-case in a Scalar context, addition is O(N) whilst multiply and divide are O(N^2), and their Vectorization would reduce those (for small N) to O(1) and O(N). Knuth's big-integer scalar algorithms provide useful real-world grounding into the types of operations needed, making it easy to demonstrate how they would be Vectorized.

The basic principle behind Knuth's algorithms is to break the problem down into a single scalar op against a Vector operand. *This fits naturally with a Scalable Vector ISA such as SVP64.* It only remains to exploit Carry (1-bit and 64-bit) in a Scalable Vector context and the picture is complete.

Links

- https://en.wikipedia.org/wiki/The_Art_of_Computer_Programming
- https://web.archive.org/web/20141021201141/https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf
- https://lists.libre-soc.org/pipermail/libre-soc-dev/2022-April/004700.html
- https://news.ycombinator.com/item?id=21151646
- https://twitter.com/lkcl/status/1517169267912986624
- https://www.youtube.com/watch?v=8hrIG7-E77o
- https://www.reddit.com/r/OpenPOWER/comments/u8r4vf/draft_svp64_biginteger_vector_arithmetic_for_the/
- https://bugs.libre-soc.org/show_bug.cgi?id=817

Vector Add and Subtract </>

Surprisingly, no new additional instructions are required to perform a straightforward big-integer add or subtract. Vectorized `adde` or `addex` is perfectly sufficient to produce arbitrary-length big-integer add due to the rules set in SVP64 that all Vector Operations are directly equivalent to the strict Program Order Execution of their element-level operations. Assuming that the two bigints (or a part thereof) have been loaded into sequentially-contiguous registers, with the least-significant bits being in the lowest-numbered register in each case:

```
R0,CA = A0+B0+CA  adde r0,a0,b0
    |
    +----------+
               |
R1,CA = A1+B1+CA  adde r1,a1,b1
    |
    +----------+
               |
R2,CA = A2+B2+CA  adde r2,a2,b2
```

This pattern - sequential execution of individual instructions with incrementing register numbers - is precisely the very definition of how SVP64 works! Thus, due to sequential execution of `adde` both consuming and producing a CA Flag, with no additions to SVP64 or to the v3.0 Power ISA, `sv.adde` is in effect an alias for Big-Integer Vectorized add. As such, implementors are entirely at liberty to recognise Horizontal-First Vector adds and send the vector of registers to a much larger and wider back-end ALU, and short-cut the intermediate storage of XER.CA on an element level in back-end hardware that need only:

- read the first incoming XER.CA
- implement a large Vector-aware carry propagation algorithm
- store the very last XER.CA in the batch

The size and implementation of the underlying back-end SIMD ALU is entirely at the discretion of the implementer, as is whether to deploy the above strategy. The only hard requirement for implementors of SVP64 is to comply with strict and precise Program Order even at the Element level.

If there is pressure on the register file (or multi-million-digit big integers) then a partial-sum may be carried out with LD and ST in a standard Cray-style Vector Loop:

```
  aptr = A address
  bptr = B address
  rptr = Result address
  li r0, 0        # used to help clear CA
  addic r0, r0, 0 # CA to zero as well
```

```
  setmvli 8       # set MAXVL to 8
loop:
  setvl t0, n         # n is the number of digits
  mulli t1, t0, 8     # 8 bytes per digit/element
  sv.ldu a0, aptr, t1 # update advances pointer
  sv.ldu b0, bptr, t1 # likewise
  sv.adde r0, a0, b0  # takes in CA, updates CA
  sv.stu rptr, r0, t1 # pointer advances too
  sub. n, n, t0       # should not alter CA
  bnz loop            # do more digits
```

This is not that different from a Scalar Big-Int add, it is just that like all Cray-style Vectorization, a variable number of elements are covered by one instruction. Of interest to people unfamiliar with Cray-style Vectors: if VL is not permitted to exceed 1 (because MAXVL is set to 1) then the above actually becomes a Scalar Big-Int add algorithm.

Vector Multiply </>

Long-multiply, assuming an O(N^2) algorithm, is performed by summing NxN separate smaller multiplications together. Karatsuba's algorithm reduces the number of small multiplies at the expense of increasing the number of additions. Some algorithms follow the Vedic Multiply pattern by grouping together all multiplies of the same magnitude/power (same column) whilst others perform row-based multiplication: a single digit of B multiplies the entirety of A, summed a row at a time. A Row-based algorithm is the basis of the analysis below (Knuth's Algorithm M).

Multiply is tricky: 64 bit operands actually produce a 128-bit result, which clearly cannot fit into an orthogonal register file. Most Scalar RISC ISAs have separate `mul-low-half` and `mul-hi-half` instructions, whilst some (OpenRISC) have "Accumulators" from which the results of the multiply must be explicitly extracted. High performance RISC advocates recommend "macro-op fusion" which is in effect where the second instruction gains access to the cached copy of the HI half of the multiply result, which had already been computed by the first. This approach quickly complicates the internal microarchitecture, especially at the decode phase.

Instead, Intel, in 2012, specifically added a `mulx` instruction, allowing both HI and LO halves of the multiply to reach registers with a single instruction. If however done as a multiply-and-accumulate this becomes quite an expensive operation: (3 64-Bit in, 2 64-bit registers out).

Long-multiplication may be performed a row at a time, starting with B0:

```
C4 C3 C2 C1 C0
         A0xB0
      A1xB0
   A2xB0
A3xB0
R4 R3 R2 R1 R0
```

- R0 contains C0 plus the LO half of A0 times B0
- R1 contains C1 plus the LO half of A1 times B0 plus the HI half of A0 times B0.
- R2 contains C2 plus the LO half of A2 times B0 plus the HI half of A1 times B0.

This would on the face of it be a 4-in operation: the upper half of a previous multiply, two new operands to multiply, and an additional accumulator (C). However if C is left out (and added afterwards with a Vector-Add) things become more manageable.

Demonstrating in c, a Row-based multiply using a temporary vector. Adapted from a simple implementation of Knuth M: https://git.libre-soc.org/?p=libreriscv.git;a=blob;f=openpower/sv/bitmanip/mulmnu.c;hb=HEAD

```
      // this becomes the basis for sv.maddedu in RS=RC Mode,
      // where k is RC. k takes the upper half of product
      // and adds it in on the next iteration
      k = 0;
      for (i = 0; i < m; i++) {
         unsigned product = u[i]*v[j] + k;
         k = product>>16;
         plo[i] = product; // & 0xffff
      }
      // this is simply sv.adde where k is XER.CA
      k = 0;
      for (i = 0; i < m; i++) {
         t = plo[i] + w[i + j] + k;
         w[i + j] = t;          // (I.e., t & 0xFFFF).
         k = t >> 16; // carry: should only be 1 bit
      }
```

We therefore propose an operation that is 3-in, 2-out, that, noting that the connection between successive mul-adds has the UPPER half of the previous operation as its input, writes the UPPER half of the current product into a second output register for exactly the purpose of letting it be added onto the next BigInt digit.

```
product = RA*RB+RC
RT = lowerhalf(product)
RC = upperhalf(product)
```

Horizontal-First Mode therefore may be applied to just this one instruction. Successive sequential iterations effectively use RC as a kind of 64-bit carry, and as noted by Intel in their notes on mulx, `RA*RB+RC+RD` cannot overflow, so does not require setting an additional CA flag. We first cover the chain of `RA*RB+RC` as follows:

```
RT0, RC0 = RA0 * RB0 + 0
      |
      +----------------+
                       |
RT1, RC1 = RA1 * RB1 + RC0
      |
      +----------------+
                       |
RT2, RC2 = RA2 * RB2 + RC1
```

Following up to add each partially-computed row to what will become the final result is achieved with a Vectorized big-int `sv.adde`. Thus, the key inner loop of Knuth's Algorithm M may be achieved in four instructions, two of which are scalar initialisation:

```
li r16, 0                     # zero accumulator
addic r16, r16, 0             # CA to zero as well
sv.maddedu *r0, *r8, r17, r16 # mul vector
sv.adde *r24, *r24, *r0   # big-add row to result
```

Normally, in a Scalar ISA, the use of a register as both a source and destination like this would create costly Dependency Hazards, so such an instruction would never be proposed. However: it turns out that, just as with repeated chained application of `adde`, macro-op fusion may be internally applied to a sequence of these strange multiply operations. (*Such a trick works equally as well in a Scalar-only Out-of-Order microarchitecture, although the conditions are harder to detect*).

Application of SVP64

SVP64 has the means to mark registers as scalar or vector. However the available space in the prefix is extremely limited (9 bits). With effectively 5 operands (3 in, 2 out) some compromises are needed. A little thought gives a useful workaround: two modes, controlled by a single bit in `RM.EXTRA`, determine whether the 5th register is set to RC or whether to RT+MAXVL. This then leaves only 4 registers to qualify as scalar/vector, which can use four EXTRA2 designators and fits into the available 9-bit space.

RS=RT+MAXVL Mode:

```
product = RA*RB+RC
RT = lowerhalf(product)
RS=RT+MAXVL = upperhalf(product)
```

and RS=RC Mode:

```
product = RA*RB+RC
RT = lowerhalf(product)
RS=RC = upperhalf(product)
```

Now there is much more potential, including setting RC to a Scalar, which would be useful as a 64 bit Carry. RC as a Vector would produce a Vector of the HI halves of a Vector of multiplies. RS=RT+MAXVL Mode would allow that same Vector of HI halves to not be an overwrite of RC. Also it is possible to specify that any of RA, RB or RC are scalar or vector. Overall it is extremely powerful.

Vector Shift </>

The decision here was made to follow the same principle as for multiply-and-accumulate. Whilst Intel has srd which is very similar (but only 2-in 2-out), the decision was made to create a 3-in 2-out instruction that effectively uses one of the inputs and one of the outputs as a "sort of 64-bit carry".

Without the `dsrd` instruction, it is necessary to have three instructions in an inner loop. Keeping the shift amount within the range of the element (64 bit) a Vector bit-shift may be synthesised from a pair of shift operations and an OR, all of which are standard Scalar Power ISA instructions. that when Vectorized are exactly what is needed, but that critically requires Vertical-First Mode.

```
void bigrsh(unsigned s, uint64_t r[], uint64_t un[], int n) {
    for (int i = 0; i < n - 1; i++)
        r[i] = (un[i] >> s) | (un[i + 1] << (64 - s));
    r[n - 1] = un[n - 1] >> s;
}
```

With SVP64 being on top of the standard scalar regfile the offset by one of the elements may be achieved simply by referencing the same vector data offset by one. Given that all three instructions (`srd`, `sld`, `or`) are an SVP64 type `RM-1P-2S1D` and are `EXTRA3`, it is possible to reference the full 128 64-bit registers (r0-r127):

```
subfic t1, t0, 64     # compute 64-s (s in t0)
sv.srd *r8, *r24, t0  # shift each element of r24 vector up by s
sv.sld *r16, *r25, t1 # offset start of vector by one (r25)
sv.or  *r8, *r8, *r16 # OR two parts together
```

Predication with zeroing may be utilised on sld to ensure that the last element is zero, avoiding over-run.

The reason why three instructions are needed instead of one in the case of big-add is because multiple bits chain through to the next element, where for add it is a single bit (carry-in, carry-out), and this is precisely what `adde` already does. For multiply, divide and shift it is worthwhile to use one scalar register effectively as a full 64-bit carry/chain.

The limitations of this approach therefore become pretty clear: not only must Vertical-First Mode be used but also the predication with zeroing trick. Worse than that, an entire temporary vector is required which wastes register space. A better way would be to create a single scalar instruction that can do the long-shift in-place.

The basic principle of the 3-in 2-out `dsrd` is:

```
# r[i] = (un[i] >> s) | (un[i + 1] << (64 - s));
temp <- ROT128(RA || RC, RB[58:63])
RT <- temp[64:127]
RS <- temp[0:63]
```

A 128-bit shift is performed, taking the lower half into RS and the upper half into RT. However there is a trick that may be applied, which only requires `ROT64`:

```
n <- (RB)[58:63]
v <- ROTL64((RA), 64-n)
mask <- MASK(n, 63)
RT <- (v[0:63] & mask) | ((RC) & ¬mask)
RS <- v[0:63] & ¬mask
```

The trick here is that the *entirety* of `RA` is rotated, then parts of it are masked into the destinations. RC, if also properly masked, can be ORed into RT, as long as the bits of RC are in the right place. The really interesting bit is that when Vectorised, the upper bits (now in RS) *are* in the right bit-positions to be ORed into the second `dsrd` operation. This allows us to create a chain `sv.dsrd`, and a single instruction replaces all four above:

```
sv.dsrd *r8, *r24, t1, t0
```

For larger shift amounts beyond an element bitwidth standard register move operations may be used, or, if the shift amount is static, to reference an alternate starting point in the registers containing the Vector elements because SVP64 sits on top of a standard Scalar register file. `sv.sld r16.v, r26.v, t1` for example is equivalent to shifting by an extra 64 bits, compared to `sv.sld r16.v, r25.v, t1`.

Vector Divide </>

The simplest implementation of big-int divide is the standard schoolbook "Long Division", set with RADIX 64 instead of Base 10. Donald Knuth's Algorithm D performs estimates which, if wrong, are compensated for afterwards. Essentially there are three phases:

- Calculation of the quotient estimate. This uses a single Scalar divide, which is covered separately in a later section
- Big Integer multiply and subtract.
- Carry-Correction with a big integer add, if the estimate from phase 1 was wrong by one digit.

From Knuth's Algorithm D, implemented in divmnu64.c, Phase 2 is expressed in c, as:

```
// Multiply and subtract.
k = 0;
for (i = 0; i < n; i++) {
   p = qhat*vn[i]; // 64-bit product
   t = un[i+j] - k - (p & 0xFFFFFFFFLL);
   un[i+j] = t;
   k = (p >> 32) - (t >> 32);
}
```

Where analysis of this algorithm, if a temporary vector is acceptable, shows that it can be split into two in exactly the same way as Algorithm M, this time using subtract instead of add.

```
uint32_t carry = 0;
// this is just sv.maddedu again
for (int i = 0; i <= n; i++) {
    uint64_t value = (uint64_t)vn[i] * (uint64_t)qhat + carry;
    carry = (uint32_t)(value >> 32); // upper half for next loop
    product[i] = (uint32_t)value;    // lower into vector
}
bool ca = true;
// this is simply sv.subfe where ca is XER.CA
for (int i = 0; i <= n; i++) {
    uint64_t value = (uint64_t)~product[i] + (uint64_t)un_j[i] + ca;
    ca = value >> 32 != 0;
    un_j[i] = value;
}
bool need_fixup = !ca; // for phase 3 correction
```

In essence then the primary focus of Vectorized Big-Int divide is in fact big-integer multiply

Detection of the fixup (phase 3) is determined by the Carry (borrow) bit at the end. Logically: if borrow was required then the qhat estimate was too large and the correction is required, which is, again, nothing more than a Vectorized big-integer add (one instruction). However this is not the full story

128/64-bit divisor

As mentioned above, the first part of the Knuth Algorithm D involves computing an estimate for the divisor. This involves using the three most significant digits, performing a scalar divide, and consequently requires a scalar division with *twice* the number of bits of the size of individual digits (for example, a 64-bit array). In this example taken from divmnu64.c the digits are 32 bit and, special-casing the overflow, a 64/32 divide is sufficient (64-bit dividend, 32-bit divisor):

```
        // Compute estimate qhat of q[j] from top 2 digits
        uint64_t dig2 = ((uint64_t)un[j + n] << 32) | un[j + n - 1];
        if (un[j+n] >= vn[n-1]) {
            // rhat can be bigger than 32-bit when the division overflows
            qhat = UINT32_MAX;
            rhat = dig2 - (uint64_t)UINT32_MAX * vn[n - 1];
        } else {
            qhat = dig2 / vn[n - 1]; // 64/32 divide
            rhat = dig2 % vn[n - 1]; // 64/32 modulo
        }
        // use 3rd-from-top digit to obtain better accuracy
        b = 1UL<<32;
        while (rhat < b || qhat * vn[n - 2] > b * rhat + un[j + n - 2]) {
            qhat = qhat - 1;
            rhat = rhat + vn[n - 1];
        }
```

However when moving to 64-bit digits (desirable because the algorithm is `O(N^2)`) this in turn means that the estimate has to be computed from a *128* bit dividend and a 64-bit divisor. Such an operation simply does not exist in most Scalar 64-bit ISAs. Although Power ISA comes close with `divdeu`, by placing one operand in the upper half of a 128-bit dividend, the lower half is zero. Again Power ISA has a Packed SIMD instruction `vdivuq` which is a 128/128 (quad) divide, not a 128/64, and its use would require considerable effort to move registers to and from GPRs. Some investigation into soft-implementations of 128/128 or 128/64 divide show it to be typically implemented bit-wise, with all that implies.

The irony is, therefore, that attempting to improve big-integer divide by moving to 64-bit digits in order to take advantage of the efficiency of 64-bit scalar multiply when Vectorized would instead lock up CPU time performing a 128/64 scalar division. With the Vector Multiply operations being critically dependent on that `qhat` estimate, and because that scalar is as an input into each of the vector digit multiples, as a Dependency Hazard it would cause *all* Parallel SIMD Multiply back-ends to sit 100% idle, waiting for that one scalar value.

Whilst one solution is to reduce the digit width to 32-bit in order to go back to 64/32 divide, this increases the completion time by a factor of 4 due to the algorithm being `O(N^2)`.

Reducing completion time of 128/64-bit Scalar division

Scalar division is a known computer science problem because, as even the Big-Int Divide shows, it requires looping around a multiply (or, if reduced to 1-bit per loop, a simple compare, shift, and subtract). If the simplest approach were deployed then the completion time for the 128/64 scalar divide would be a whopping 128 cycles. To be workable an alternative algorithm is required, and one of the fastest appears to be Goldschmidt Division. Whilst typically deployed for Floating Point, there is no reason why it should not be adapted to Fixed Point. In this way a Scalar Integer divide can be performed in the same time-order as Newton-Raphson, using two hardware multipliers and a subtract.

Back to Vector carry-looping

There is however another reason for having a 128/64 division instruction, and it's effectively the reverse of `maddedu`. Look closely at Algorithm D when the divisor is only a scalar (`v[0]`):

```
        k = 0; // the case of a
        for (j = m - 1; j >= 0; j--)
        {                                   // single-digit
            uint64_t dig2 = ((k << 32) | u[j]);
            q[j] = dig2 / v[0]; // divisor here.
            k = dig2 % v[0]; // modulo back into next loop
        }
```

Here, just as with `maddedu` which can put the hi-half of the 128 bit product back in as a form of 64-bit carry, a scalar divisor of a vector dividend puts the modulo back in as the hi-half of a 128/64-bit divide.

```
RT0      = ((  0<<64) | RA0) / RB0
     RC0 = ((  0<<64) | RA0) % RB0
      |
      +-------+
              |
RT1      = ((RC0<<64) | RA1) / RB1
     RC1 = ((RC0<<64) | RA1) % RB1
      |
      +-------+
              |
RT2      = ((RC1<<64) | RA2) / RB2
```

```
RC2 = ((RC1<<64) | RA2) % RB2
```

By a nice coincidence this is exactly the same 128/64-bit operation needed (once, rather than chained) for the `qhat` estimate if it may produce both the quotient and the remainder. The pseudocode cleanly covering both scenarios (leaving out overflow for clarity) can be written as:

```
divmod2du RT,RA,RB,RC

 dividend = (RC) || (RA)
 divisor = EXTZ128(RB)
 RT = UDIV(dividend, divisor)
 RS = UREM(dividend, divisor)
```

Again, in an SVP64 context, using EXTRA mode bit 8 allows for selecting whether `RS=RC` or `RS=RT+MAXVL`. Similar flexibility in the scalar-vector settings allows the instruction to perform full parallel vector div/mod, or act in loop-back mode for big-int division by a scalar, or for a single scalar 128/64 div/mod.

Again, just as with `sv.maddedu` and `sv.adde`, adventurous implementors may perform massively-wide DIV/MOD by transparently merging (fusing) the Vector element operations together, only inputting a single RC and outputting the last RC. Where efficient algorithms such as Goldschmidt are deployed internally this could dramatically reduce the cycle completion time for massive Vector DIV/MOD. Thus, just as with the other operations the apparent limitation of creating chains is overcome: SVP64 is, by design, an "expression of intent" where the implementor is free to achieve that intent in any way they see fit as long as strict precise-aware Program Order is preserved (even on the VL for-loops).

Just as with `divdeu` on which this instruction is based an overflow detection is required. When the divisor is too small compared to the dividend then the result may not fit into 64 bit. Knuth's original algorithm detects overflow and manually places 0xffffffff (all ones) into `qhat`. With there being so many operands already in `divmod2du` a `cmpl` instruction can be used instead to detect the overflow. This saves having to add an Rc=1 or OE=1 mode when the available space in VA-Form EXT04 is extremely limited.

Looking closely at the loop however we can see that overflow will not occur. The initial value k is zero: as long as a divide-by-zero is not requested this always fulfils the condition `RC < RA`, and on subsequent iterations the new k, being the modulo, is always less than the divisor as well. Thus the condition (the loop invariant) `RC < RA` is preserved, as long as RC starts at zero.

Limitations

One of the worst things for any ISA is that an algorithm's completion time is directly affected by different implementations having instructions take longer or shorter times. Knuth's Big-Integer division is unfortunately one such algorithm.

Assuming that the computation of qhat takes 128 cycles to complete on a small power-efficient embedded design, this time would dominate compared to the 64 bit multiplications. However if the element width was reduced to 8, such that the computation of qhat only took 16 cycles, the calculation of qhat would not dominate, but the number of multiplications would rise: somewhere in between there would be an elwidth and a Vector Length that would suit that particular embedded processor.

By contrast a high performance microarchitecture may deploy Goldschmidt or other efficient Scalar Division, which could complete 128/64 qhat computation in say only 5 to 8 cycles, which would be tolerable. Thus, for general-purpose software, it would be necessary to ship multiple implementations of the same algorithm and dynamically select the best one.

The very fact that programmers even have to consider multiple implementations and compare their performance is an unavoidable nuisance. SVP64 is supposed to be designed such that only one implementation of any given algorithm is needed. In some ways it is reassuring that some algorithms just don't fit. Slightly more reassuring is that Goldschmidt Divide, which uses two multiplications that can be performed in parallel, would be a much better fit with SVP64 (and Vector Processing in general), the only downside being that it is regarded as worthwhile for much larger integers.

Conclusion </>

The majority of ISAs do not have 3-in 2-out instructions for very good reasons: they are horrendously expensive if implemented as operations that can write to two registers every clock cycle. Implementations that take two clock cycles to write to a single write port are frowned-upon, and the Hazard Management complexity also goes up.

However when looked at from a Vectorised perspective and using "chaining" (64-bit carry-out becomes the 64-bit output for the next operation) then "Operand Forwarding" makes one read and one write effectively disappear, reducing most of the operations to a manageable efficent 2-in 1-out. In normal Scalar hardware being used to attempt Vectored bigint operations the opportunity for "Forwarding" is quite hard to spot, but because of the Vectorisation it is known *in advance* that a Vector of values is being used, i.e. only *one* instruction issued not several. Therefore it is easy to micro-code the Operand Forwarding.

In essence by taking things that extra step further the complexity of Scalar ISAs disappears through the introduction of some uniform Vector-Looping that, although complex in itself in hardware, at least only has to be done once and becomes uniform-RISC applicable to *all* instructions. These comprehensive powerful Scalar arithmetic instructions will significantly reduce the complexity of big-integer operations.