Big Integer Arithmetic Instruction design

An analysis of big-integer arithmetic instructions (why not to put all eggs in Custom Silicon basket)

Silicon Salon 2023

Sponsored by NLnet's Assure Programme

May 3, 2023

Who are we?

- Libre-SOC: a fully Libre Project with the goal of creating a Hybrid 3D CPU-VPU-GPU including designing a powerful Vector Extension (for the Power ISA). https://libre-soc.org
- RED Semiconductor Ltd: a commercial realisation of Libre-SOC designs. https://redsemiconductor.com
- Libre-SOC researches and designs instructions that are then proposed to the OpenPOWER Foundation ISA Technical Workgroup; RED Semiconductor (as an OPF ISA WG Voting Member) then keeps an eye on the RFC.
- RED Semiconductor Ltd seeks VC funding and commercial business propositions, Libre-SOC covers Research.

What are the challenges faced by Biginteger?

- Algorithms especially post-quantum are now fast-moving. This does not go down well! It typically takes 5-10 years for an algorithm to become "trustable".
- Custom Cryptographic Hardware will typically take 3 years from design concept to first production silicon: Certification even longer. If a fault is found in the algorithm, the entire investment is wasted.
- Performance on 32-bit and 64-bit Embedded Hardware sucks. Algorithms are roughly $O(N^2)$ which wreaks havoc. The temptation therefore is to add SIMD instructions or dedicated "custom" instructions which makes the problem worse.
- So how can these polar opposites be solved?

Go back to the algorithms.

- https://libre-soc.org/openpower/sv/biginteger/analysis/
- Starting with Knuth's Algorithm D and M, if a True-Scalable Vector ISA can cope with those, chances are good it'll cope with more (Karatsuba, and so on).
- SVP64 has "looping" as a primary construct:
  loop i 0..VL-1: GPR(RT+i) = ADD(GPR(RA+i), GPR(RB+i)
- If however Carry-in and Carry-out are included in that, we have arbitrary-length Big-Integer Vector Add!
- For all other operations as long as Scalar-Vector is ok, it turns out to be possible to do 64-bit carry-in and 64-bit carry-out, without significant hardware disruption.
- Irony: all relevant Scalar instructions (shift, mul, div) usually drop 1/2 the result on the floor!

Turning add-with-carry into Vector-Add

- Add-with-Carry is the building-block of larger operations
- Let's simply chain them together.
- sv.adde (Add-Carry with Vector loop) creates chains

```
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
```

Vector-Scalar Shift

- Shift by 64-bit is just "pick a register"
- Add a 2nd input register with what needs to be shifted IN (64-bit carry in)
- Add 2nd output saving what normally gets thrown away (64-bit carry-out)
- Again: a chain of these performs Vector-by-Scalar shift

```
brs(uint64_t 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;
}
```

Vector-Scalar Multiply

- Normally in FMAC the top 64-bits is thrown away.
- What if we stored those 64-bits in a 2nd register? (64-bit carry-out)
- And what if the next FMAC added that "digit" on? (64-bit carry-in)
- Again: a chain of these performs Vector-by-Scalar Multiply

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

Vector-Scalar Divide

- Same story. special-case for overflow.

```
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
```

Summary so far

- Extending the usual 1-bit Carry-in Carry-out to 64-bit and adding a loop-construct inherently turns Scalar operations into arbitrary-length Vectorised ones
- Irony: 30 years ago Power ISA actually had a "Carry SPR", where the normally-discarded upper half of multiply would be placed in that SPR (it was deprecated).
- Hardware is NOT made more complex because in all shift multiply and divide operations these bits are discarded in other ISAs, which is why you end up with complex carry workarounds. This gives ISAs a "bad rep" for doing Big-int
- The "complication" is that you need 3-in 2-out instructions, but actually in Micro-code you can do operand-forwarding. 1st op: 3-in 1-out. chain: 2-in 1-out. Last: 2-in 2-out.

OpenTITAN

- https://opentitan.org/book/hw/ip/otbn/index.html
- 256b wide data path with 32 256b wide registers
- Zero-Overhead Loop Control would have been better https://ieeexplore.ieee.org/abstract/document/1692906/
- Formal verification completion time is a factor of the operation bit-width. 256-bit unlikely to be reasonable time.
- 256-bit is great for EC25519 but for RSA (etc.) you run into exactly the same problem as a Scalar ISA, made worse.
- Opportunities to optimise algorithms not possible (efficient power-optimised Karatsuba, etc.)

OpenTITAN shift

- Immediate-only. what about shift-by-reg?
- merges 2 operands, still not chainable.
- needs a copy of the vector input (double number of regs)
- needs massive 256-bit shifter! 8 layers of muxes!

```
a = WDRs[wrs1]
b = WDRs[wrs2]

result = (((a << 256) | b) >> imm) & ((1 << 256) - 1)
WDRs[wrd] = result
```

Draft Double-Shift

- Remarkably similar to x86 dsld
- Does not need 128-bit ROT: simple mod to existing hardware
- Hardware may macro-op fuse Vector-shift for better efficiency
- Chainable and in-place (no copy of vector needed).

```
n <- (RB)[58:63]     # Power ISA MSB0 numbering. sigh
v <- ROTL64((RA), n)
mask <- MASK(0, 63-n)
RT <- (v[0:63] & mask) | ((RC) & ~mask)
RS <- v[0:63] & ~mask
```

Conclusion

- We went back to the algorithms (Knuth D and M) and examined what they are trying to achieve.
- Turns out they need a 64-bit carry-in and carry-out
- Keeping to 64-bit maximum hardware means Formal Proofs complete in reasonable time (less than heat-death of universe)
- Reasonably straightforward: creates and uses partial results normally thrown away (needing more instructions)
- Freaks out pure-RISC proponents (3-in 2-out) but look at the number of instructions (and temporary registers) needed otherwise, and the overall algorithm efficiency, and the case for these instructions is clear.
- They also speed up **general-purpose** code

The end
Thank you
Questions?

- https://redsemiconductor.com
- Discussion: http://lists.libre-soc.org
- Libera.Chat IRC #libre-soc
- http://libre-soc.org/
- http://nlnet.nl/assure
- https://libre-soc.org/nlnet/#faq