Formally verified 32- and 64-bit integer division using double-precision floating-point arithmetic

David Monniaux Alice Pain

2022-09-14

Kalray KVX core

- VLIW
- fast 64-bit multiply-adder
- fast IEEE-754 single and double precision fused multiply-add
- **no divider unit** except for single-precision reciprocal ($1/x$)

Extant division on KVX

Division function = a loop around a special instruction producing one bit of quotient per iteration

Issues:

- not constant-time
- loop cannot be easily interleaved with other computations
- slow?
- cost of function call (could be inlined)

32-bit integer division $\lfloor a/b \rfloor$ idea

1. compute single-precision $1/b$
2. refine into a better approximation y by iteration using fused multiply-add
3. $ay \simeq a/b$; round to nearest integer, get q
4. $\lfloor a/b \rfloor$ is q or $q+1$ depending on the sign of $a - qb$

64-bit integer division $\lfloor a/b \rfloor$ idea

53-bit precision insufficient for inverses for 64-bit numbers

"Compute a rough quotient, divide the rough remainder, return sum of two quotients"

1. compute single-precision x for $1/b$
2. compute q_0 integer rounding of ax ("rough approximate of the quotient")
3. refine x into a better approximation y by iteration using fused multiply-add
4. divide the "rough remainder": $(a - q_0 b)y \simeq (a - q_0 b)/b$; round to nearest integer, get q_1
5. $q'_1 = \lfloor (a - q_0 b)/b \rfloor$ is q_1 or $q_1 + 1$ depending on the sign of $a - qb$
6. answer $q_0 + q'_1$
7. special cases $b = 1$ and $b \geq 2^{63}$ dealt with separately.

Efficiency remarks

- Straight-line code, no jumps, can be easily interleaved with other computations.
- Most of the computation depends only on b and can be merged or hoisted out by the compiler if common b.

CompCert

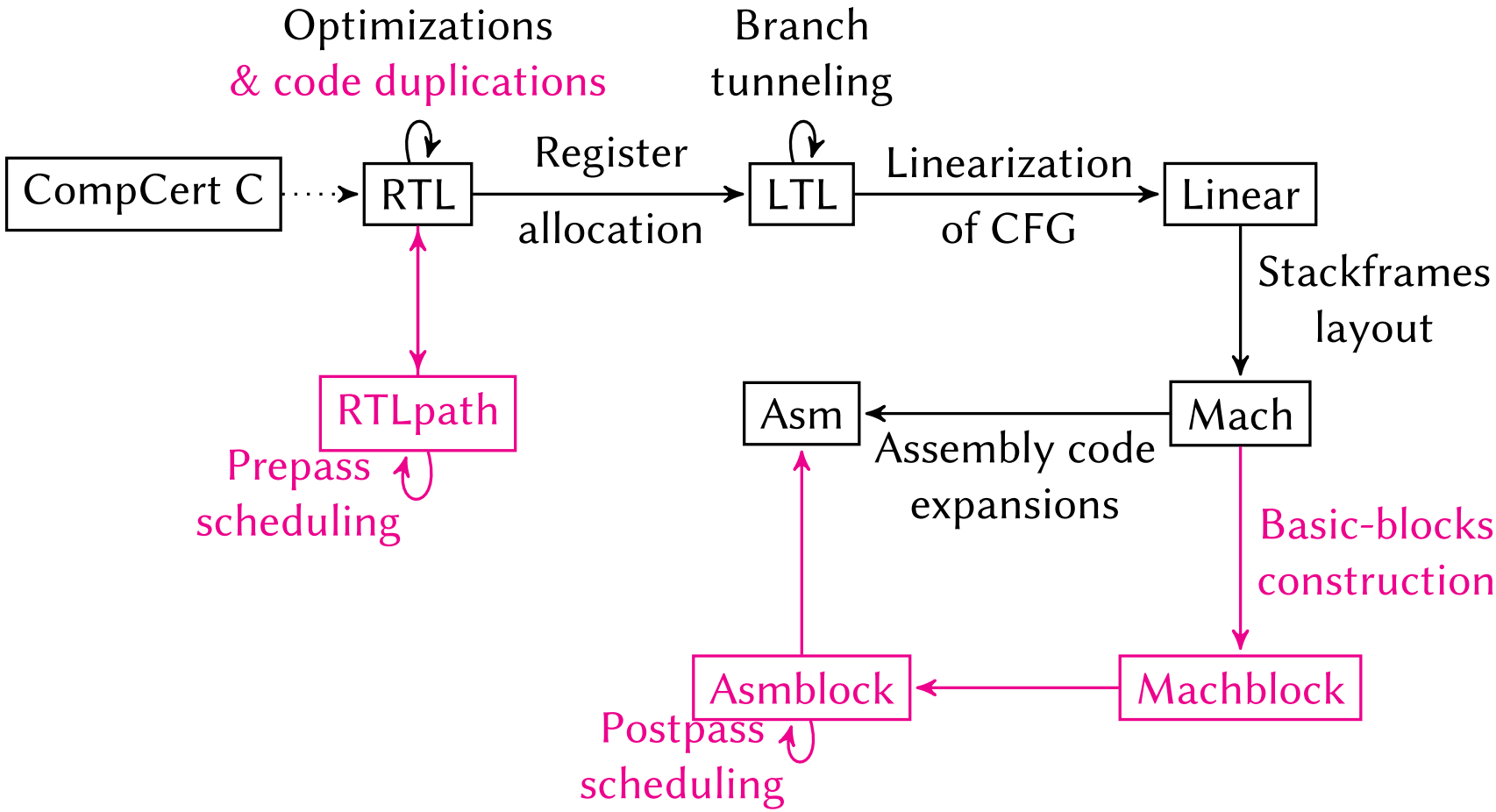
`https://compcert.org`
`https://github.com/AbsInt/CompCert`

Compiler for C (also Lustre, possibly Rust) to ARM, AArch64, x86, x86-64, RISC-V, PowerPC 32/64...and also KVX

Proof executed assembly code "does the same" as source code. (Same sequence of observable events: calls to external functions, accesses to volatile variables)

For safety-critical embedded code

Preservation



Proof goal

Proof in Coq proof assistant that we really compute the integer division.

Using the definition of IEEE-754 operations.

Lemmas from the Flocq library and some proofs produced by the Gappa tool.

Proof insights

- A lot of care due to integer overflows.
- a and b do not necessarily fit exactly into double precision numbers, which may add roundoff error.
- Separate proofs for small b and big b
- A lot of auxiliary lemmas for showing things do not overflow

Performance

64-bit division

Method	Loop	Floating-point
One quotient per iteration	620180	522316
Two quotients per iteration	589696	292314

32-bit division

Method	Loop	Floating-point
One quotient per iteration	469969	442101
Two quotients per iteration	434501	232124

64-bit, common divisor

Method	Loop	Floating-point
One quotient per iteration	608158	342951
Two quotients per iteration	582948	237857

Source code

`FPDivision32.v` and `FPDivision64.v` in
`https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx`