MPSoC'25

Megève, France, June 20, 2025

Experimental Software and Hardware Evaluation of Ad-Hoc Constant Division Routines

Frédéric Pétrot
Grenoble INP/Université Grenoble Alpes,
CNRS, TIMA Lab,
F-38000 Grenoble, France
frederic.petrot@univ-grenoble-alpes.fr

Why divide by integer constants?

Initially Motivated by AI Computations

Average Pooling
Kernel shapes known beforehand for hw implementations
Kernel usually square $k \times k$, with k odd and ”small”
Look for efficient $\frac{x}{k^2}$ implementation

Many algorithms need that!

Printing decimal numbers, filters in signal processing, ...
Lots of prior art, since the mists of time, summarized in:

- Hacker’s Delight, Warren, Chapter 10
- Application-Specific Arithmetic, De Dinechin & Kumm, Chapter 13
- **Focus of this talk: Ad-Hoc Cookbook proposed by Li in 1985**

General approach: multiply by the reciprocal

Compute $\frac{1}{p}$, with p odd and $p \geq 3$

$$\frac{1}{p} = 0.(0b_1b_2b_3\ldots b_{n-1}b_1b_2b_3\ldots b_{n-1})$$

$$\frac{x}{p} = \frac{x}{b_1 2^1} + \frac{x}{b_2 2^2} + \frac{x}{b_3 2^3} + \frac{x}{b_4 2^4} + \cdots$$

Note that binary pattern is repeating ad infinitum:
Quickly for some numbers, not so much for others

$$\begin{aligned}
\frac{1}{3} &= 0.01010101010101010101010101010101010... = 0.(01)* \\
\frac{1}{23} &= 0.00001011001000010110010000101100 1... = 0.(00001011001001)* \\
\frac{1}{47} &= 0.00000101011100100110001000001010 1... = 0.(0000010101110010011000 1)* \\
\frac{1}{63} &= 0.00000100000010000001000000100000 1000... = 0.(000001)*
\end{aligned}$$

Use infinite product rather than infinite series

Finding the pattern and its period

Li's first proposal based on Fermat Euler's theorem

Assuming $p > 1$ odd, find smallest n such that p divides $2^n - 1$

$\frac{2^n-1}{p} = b_1 b_2 b_3 \dots b_{n-1}$

$\frac{1}{p} = 0.(0b_1 b_2 b_3 \dots b_{n-1}) \prod_{i=0}^{\infty} (1 + \frac{1}{2^{n \times 2^i}})$

Example: $\frac{x}{23}$

Smallest integer n such that 23 divides $2^n - 1$ is 11

$\frac{2047}{23} = 89$ or 1011001_2

Must be written on 11 bits to have the proper magnitude 00001011001_2

$$\frac{x}{23} = \left(\frac{x}{2^5} + \frac{x}{2^7} + \frac{x}{2^8} + \frac{x}{2^{11}}\right)\left(1 + \frac{1}{2^{11}}\right)\left(1 + \frac{1}{2^{22}}\right)\cdots$$

5 adds and 6 shifts instead of 12 adds and 11 shifts for infinite series

Computing the division

Continuing with $\frac{x}{23}$: unfortunately $\frac{x}{2^n} \neq \lfloor \frac{x}{2^n} \rfloor$ (that is $x \gg n$)

```
uint32_t bad_divu23(uint32_t x)
{
    x = (x >> 5) + (x >> 7) + (x >> 8) + (x >> 11);
    x = (x >> 11) + x;
    x = (x >> 22) + x;
    return x;
}
```

Incorrect, rounding is very soon an issue
General approach computes remainder and corrects:

```
uint32_t good_divu23(uint32_t n)
{   /* needs also mult and test to be correct! */
    uint32_t x = n, r;
    x = (x >> 1) + (x >> 3) + (x >> 4) + (x >> 7);
    x = (x >> 11) + x;
    x = (x >> 22) + x;
    x = x >> 4;
    r = n - x * 23;
    return x + (r > 22);
}
```

Li's Routines Sample

Handmade routines for unsigned division

Li: *Playing with the binary decomposition, its complement, add and sub, simpler expressions can be found empirically*

```c
uint32_t good_divu23(uint32_t n)
{
    /* 7 add and 7 shifts, but also
       mult and test to be correct! */
    uint32_t x = n, r;
    x = (x >> 1) + (x >> 3)
        + (x >> 4) + (x >> 7);
    x = (x >> 11) + x;
    x = (x >> 22) + x;
    x = x >> 4;
    r = n - x * 23;
    return x + (r > 22);
}
```

```c
uint32_t <ug?>li_divu23(uint32_t n)
{
    /* just 6 shifts and 6 adds, magic ! */
    uint32_t x = n + 1;
    x = (((x >> 3) + x) >> 1) + x + (x << 2);
    x = (x >> 11) + x;
    x = (x >> 22) + x;
    x = (x >> 7);
    return x;
}
```

Quite mind blowing!
But there is not free lunch:
Erroneous from `0x2e000000` on

Li's Routines Sample

Handmade routines for unsigned division

Li: *Playing with the binary decomposition, its complement, add and sub, simpler expressions can be found empirically*

Divisor	*Division Procedure*		
3	X ← XL2+X+5 X ← XR16+X	X ← XR4+X X ← XR4	X ← XR8+X
5	X ← XL1+X+3 X ← XR16+X	X ← XR4+X X ← XR4	X ← XR8+X
7	X ← X+1 X ← XR24+X	X ← XL2+XR1 X ← XR5	X ← XR6+X
9	X ← X+1 X ← XR12+X	X ← XL1+X+XR1 X ← XR24+X	X ← XR6+X X ← XR5
11	X ← X+1 X ← XR10+X	Y ← XL2+X X ← XR20+X	X ← Y+YR4+XR1 X ← XR6
13	X ← X+1 X ← XR12+X	X ← XL2+XR1 X ← XR24+X	X ← X+(XR1+X)R4 X ← XR6
15	X ← X+1 X ← XR16+X	X ← XL2+XR2 X ← XR6	X ← XR8+X
17	X ← XL2+X+5 X ← XR16+X	X ← XR1+X X ← XR7	X ← XR8+X
19	X ← XL1+X+2 X ← XR18+X	X ← XR3+X X ← XR6	X ← X-XR9
21	X ← XL1+X+3 X ← XR24+X	X ← XR6+X X ← XR6	X ← XR12+X
[illegible]	[illegible]	[illegible]	[illegible]

```cpp
template<unsigned int S = 32>
sc_uint<S> divu23(sc_uint<S> n)
{
    sc_uint<S + 3> x = n + 1;
    x = (((x >> 3) + x) >> 1) + x + (x << 2);
    x = (x >> 11) + x;
    x = (x >> 22) + x;
    x = (x >> 7);
    return x;
}
```

Works for all 32-bit values,
at the cost of "just" 3 more bits

$\Rightarrow$ Expensive in software,

$\Rightarrow$ Cheap in hardware

Li's Routines

Unsigned division for odd numbers between 3 and 55 included

Not all are correct

5 erroneous routines, among which:

- 3 transcription errors: 7, 27, 39,
- 2 seemingly just wrong: 49, 53

```c
static inline uint32_t divu49(uint32_t n)
{   /* 1 more shift than Li's */
    uint32_t x = n;
    x = (x << 2) - (x >> 5) + 2;
    x = x + (x >> 2) + (((x >> 4) + x) >> 4);
    x = (x >> 21) + x;
    x = x >> 8;
    return x;
}
```

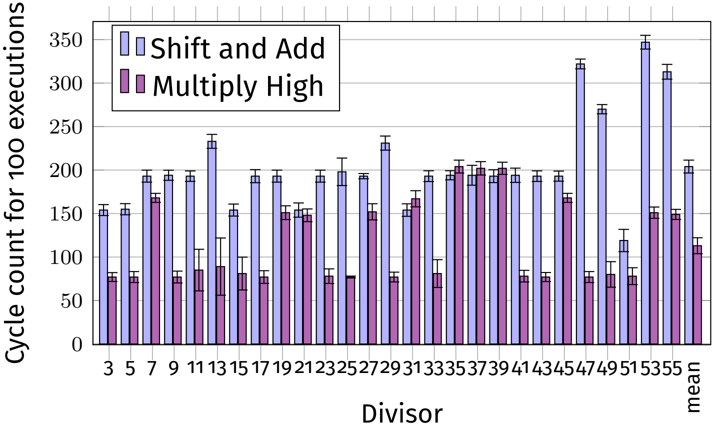
```c
static inline uint32_t divu53(uint32_t n)
{   /* 2 more shifts and adds than Li's */
    uint32_t x = n + 1, y;
    y = (x << 1) + (x >> 2);
    y = x + y + (y >> 5);
    y = y + ((x - (x >> 3)) >> 11)
        + ((x - (x >> 5)) >> 16)
        + (x >> 23) + (x >> 24);
    x = (x << 2) + (y >> 2);
    x = (x >> 8);
    return x;
}
```

Software evaluation: Cycle count on x86_64

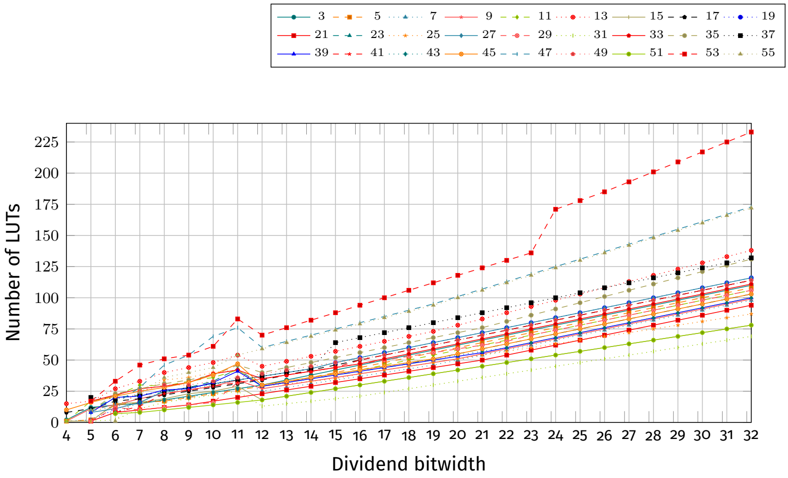
Granlund & Montgomery

Variations used by `gcc` and `clang`: mult by a magic constant and correction

```
uint32_t gm_divu23(uint32_t n)
{
    uint64_t x = n;
    uint64_t u = 0xd79435f11 * x;
    uint64_t h = u >> 32;
    u = u - h * 0xd;
    return (uint32_t) (u >> 32);
}
```

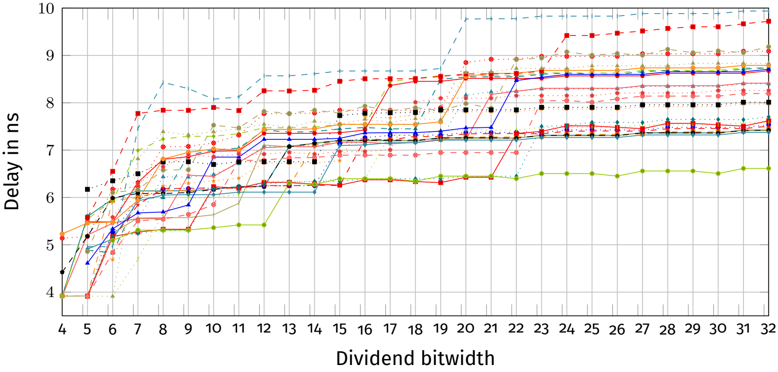


Hardware Evaluation : Li’s Resources on FPGA



Li: Number of LUTs on Virtex 7.

Hardware Evaluation : Li’s Timings on FPGA



Li: Circuit delay in ns on Virtex 7.

Hardware Evaluation : Comparison with SoA on FPGA

Delay and area after FPGA synthesis (BR: Block Ram)

		This work		SC (Arith 2023)		LinArch (TC 2017)		BTCD (TC 2017)	
		Delay	Area	Delay	Area	Delay	Area	Delay	Area
d	n	ns	LUTs	ns	LUTs	ns	LUTs	ns	LUTs + BR
3	16	7.2	46	4.1	40	3.6	17	3.7	37
	32	8.6	114	11.4	98	6.0	32	4.8	95
	64	9.0	277	27.5	379	13.5	63	6.2	225
5	16	7.2	44	4.0	52	4.4	21	3.8	44
	32	8.6	111	10.6	123	9.3	45	4.7	109
	64	9.0	274	27.3	386	20.1	93	6.7	270
11	16	7.5	42	3.9	53	8.0	39	3.8	79
	32	8.7	106	10.7	159	17.9	87	6.1	212
	64	10.4	265	26.9	436	39.0	183	8.8	526
23	16	7.5	41	3.9	52	7.4	69	5.6	197
	32	8.7	103	10.4	187	18.5	165	6.8	436 + 1BR
	64	10.4	256	26.1	493	36.6	357	6.5	959 + 2.5BR

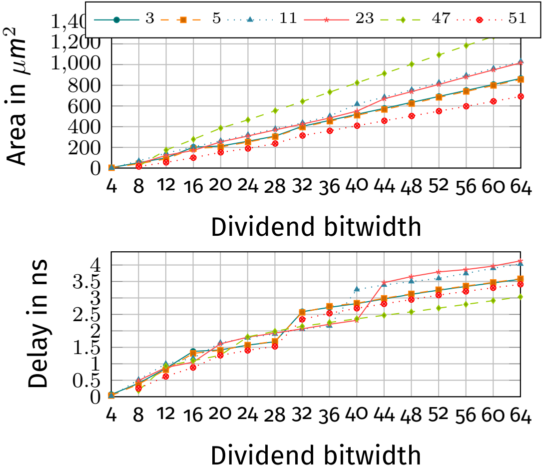
Hardware Evaluation : Comparison with SoA on FPGA

Delay and area after FPGA synthesis (BR: Block Ram)

		This work		SC (Arith 2023)		LinArch (TC 2017)		BTCD (TC 2017)	
		Delay	Area	Delay	Area	Delay	Area	Delay	Area
d	n	ns	LUTs	ns	LUTs	ns	LUTs	ns	LUTs + BR
3	16	7.2	46	4.1	40	3.6	17	3.7	37
	32	8.6	114	11.4	98	6.0	32	4.8	95
	64	9.0	277	27.5	379	13.5	63	6.2	225
5	16	7.2	44	4.0	52	4.4	21	3.8	44
	32	8.6	111	10.6	123	9.3	45	4.7	109
	64	9.0	274	27.3	386	20.1	93	6.7	270
11	16	7.5	42	3.9	53	8.0	39	3.8	79
	32	8.7	106	10.7	159	17.9	87	6.1	212
	64	10.4	265	26.9	436	39.0	183	8.8	526
23	16	7.5	41	3.9	52	7.4	69	5.6	197
	32	8.7	103	10.4	187	18.5	165	6.8	436 + 1BR
	64	10.4	256	26.1	493	36.6	357	6.5	959 + 2.5BR

Hardware Evaluation: ASIC

STMicroelectronics 28 nm FDSOI technology



Comparison with SOA (TC 2017)

- Slower than SOA for small to medium bit sizes, but SOA slope steeper
- Easily pipelineable for high-throughput
- Area about half the size of SOA solutions

Take away

Li's approach hard to scale

Erroneous above ”some” point

Li's approach hard to generalize

No nice formulation found for division with SA
Not even some kind of algorithm
$\Rightarrow$ Trial and error approach not satisfactory

Nevertheless...

Provides a different trade-off compared to existing hardware approaches
Very dependent on the value of the divisor and the target technology
$\Rightarrow$ Can be useful on a case-by-case basis

Code, of the PoC kind, ...

`https://github.com/fpetrot/divbysmallcst`

Thanks for listening

Thanks to French DGE and EU that is financing the EdgeAI project

ACKNOWLEDGEMENTS

EdgeAI "Edge AI Technologies for Optimised Performance Embedded Processing" project has received funding from Chips Joint Undertaking (Chips JU), under grant agreement No 101097300. The Chips JU receives support from the European Union's Horizon Europe research and innovation program and Austria, Belgium, France, Greece, Italy, Latvia, Netherlands, Norway.