

Report 15.07.20

15/07/2020

Matteo Perotti

Luca Bertaccini

Pasquale Davide Schiavone

Stefan Mach

Professor Luca Benini

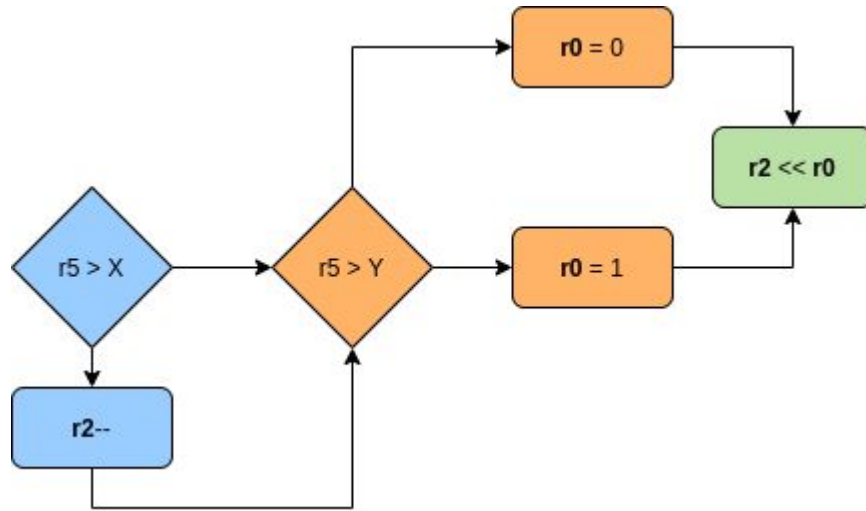
Integrated Systems Laboratory

ETH Zürich

Summary

- **OPUS** analysis
- **FP** multiplication

<silk_resampler_init> - ARM - fragment



```

f5b5 5f7a    cmp.w r5, #16000
bfc8        it  gt
3a01        subgt r2, #1
  
```

```

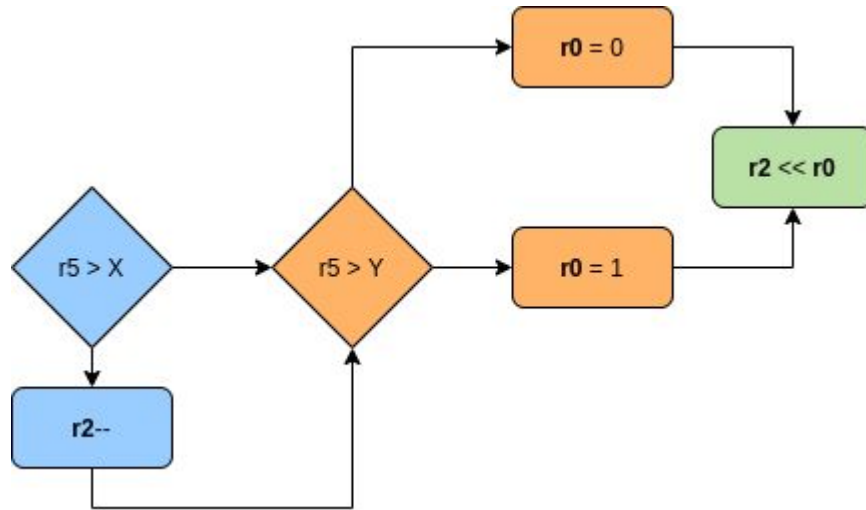
f645 53c0    movw r3, #24000
429d        cmp r5, r3
bfd4        ite le
2000        movle r0, #0
2001        movgt r0, #1
  
```

```

4102        asrs r2, r0
  
```

Code size: 22 B

<silky_resampler_init> - RISC-V - fragment



```

6791      lui a5,0x4
e8178793  addi a5,a5,-383
00f4a733  slt a4,s1,a5
00174713  xori a4,a4,1
40e60733  sub a4,a2,a4
  
```

```

6619      lui a2,0x6
dc160613  addi a2,a2,-575
00c4a6b3  slt a3,s1,a2
0016c693  xori a3,a3,1
  
```

```

40d75733  sra a4,a4,a3
  
```

Code size: 36 B

Comparison

if (**rX** > **imm**) then **rY--**

ARM

```
f5b5 5f7a    cmp.w r5, #16000
bfc8         it gt
3a01         subgt r2, #1
```

- ARM loads/uses immediates with 4 B
- Conditional subtraction 4 B
- Code size: 8 B

RISC-V

```
6791         lui a5,0x4
e8178793     addi a5,a5,-383
00f4a733     slt a4,s1,a5
00174713     xori a4,a4,1
40e60733     sub a4,a2,a4
```

- RISC-V loads immediates from 2k to 128k using 6 B (l.i does NOT help)
- Conditional subtraction 12 B
- Code size : 18 B

Comparison

RISC-V can do better:

$a4 = a2 - !(s1 < a5)$
 $= a2 - (s1 \geq a5)$
 $= a2 - (a5 \leq s1)$
 $= a2 - (a5 - 1 < s1)$

We remove the XORI instruction!

`addi a5,a5,-383`
`slt a4,s1,a5`
`xori a4,a4,1`

→

`addi a5,a5,-384`
`slt a4,a5,s1`

RISC-V

6791	lui a5,0x4
e8178793	addi a5,a5,-383
00f4a733	slt a4,s1,a5
00174713	xori a4,a4,1
40e60733	sub a4,a2,a4

- RISC-V loads immediates from 2k to 128k using 6 B (l.i does NOT help)
- Conditional subtraction 12 B
- Code size : 18 B

Comparison

if ($rX > \text{imm}$) then $rY = 1$ else $rY = 0$

ARM

f645 53c0	movw r3, #24000
429d	cmp r5, r3
bfd4	ite le
2000	movle r0, #0
2001	movgt r0, #1

- ARM loads/uses immediates with 4 B
- Conditional unit assignment 8 B
- Code size: 12 B

RISC-V

6619	lui a2,0x6
dc160613	addi a2,a2,-575
00c4a6b3	slt a3,s1,a2
0016c693	xori a3,a3,1

- RISC-V loads immediates from 2k to 128k using 6 B (l.li does NOT help)
- Conditional unit assignment 8 B
- Code size: 14 B

Comparison

if ($rX > \text{imm}$) then $rY = 1$ else $rY = 0$

slt is better!



ARM

```
f645 53c0    movw r3, #24000
429d         cmp r5, r3
bfd4         ite le
2000         movle r0, #0
2001         movgt r0, #1
```

- ARM loads/uses immediates with 4 B
- Conditional unit assignment 8 B
- Code size: 12 B

RISC-V

```
6619         lui a2,0x6
dc160613     addi a2,a2,-575
00c4a6b3     slt a3,s1,a2
0016c693     xori a3,a3,1
```

- RISC-V loads immediates from 2k to 128k using 6 B (l.li does NOT help)
- Conditional unit assignment 8 B
- Code size: 14 B

Comparison

Same concept as before:

$a3 = !(s1 < a2)$
 $= (s1 \geq a2)$
 $= (a2 \leq s1)$
 $= (a2 - 1 < s1)$

We remove the XORI instruction!

`addi a2,a2,-575`
`slt a3,s1,a2`
`xori a3,a3,1`

→

`addi a2,a2,-576`
`slt a3,a2,s1`

RISC-V

6619	lui a2,0x6
dc160613	addi a2,a2,-575
00c4a6b3	slt a3,s1,a2
0016c693	xori a3,a3,1

- RISC-V loads immediates from 2k to 128k using 6 B (l.li does NOT help)
- Conditional unit assignment 8 B
- Code size: 14 B

Other inefficiencies

- Uncompressed **srai**
- Jump on **comparison** with **big immediates** (ARM does not use registers!)
 - ARM (6 B)
 - f5b5 5ffa cmp.w r5, #8000
 - d00e beq.n 1d564
 - RISC-V (10 B)
 - 6789 lui a5, 0x2
 - f407 8793 addi a5, a5, -192
 - 02f4 8663 beq s1, a5, 27c4c

Other inefficiencies

- Uncompressed srai
- Even when the **immediate cannot be immediately compared...**
 - ARM (8 B)
 - f642 63e0 movw r3, #12000
 - 429d cmp r5, r3
 - d00a beq.n 1d564
 - RISC-V (10 B)
 - 678d lui a5, 0x3
 - ee07 8793 addi a5, a5, -288
 - 02f4 8163 beq s1, a5, 27c4c

Other inefficiencies

- Load **small immediates** (ARM loads “101” using 2 B only)
- muliadd:
 - RISC-V (16 B)
 - 4615 li a2, 5
 - 000486b7 lui a3, 0x48
 - 55468693 addi a3, a3, 1364
 - 02c70733 mul a4, a4, a2
 - 9736 add a4, a4, a3
 - HCC (14 B) -> Good job HCC!
 - 177d addi a4, a4, -1
 - 00048637 lui a2, 0x48
 - f8460613 addi a2, a2, -124
 - 04e6575b muliadd a4, a2, a4, 5
 - ARM (10 B + 4 B of constant pool)
 - 3a01 subs r2, #1
 - 4b3e ldr r3, [pc, #248]
 - eb02 0282 add.w r2, r2, r2, lsl #2
 - 441a add r2, r3

Other inefficiencies

- GCC strange choices...

- 4585 li a1,1
- 00e58793 addi a5,a1,14
- 00b494b3 sll s1,s1,a1
- 65c1 lui a1,0x10

← **a1 is overwritten, not needed!**

- Why not the following?

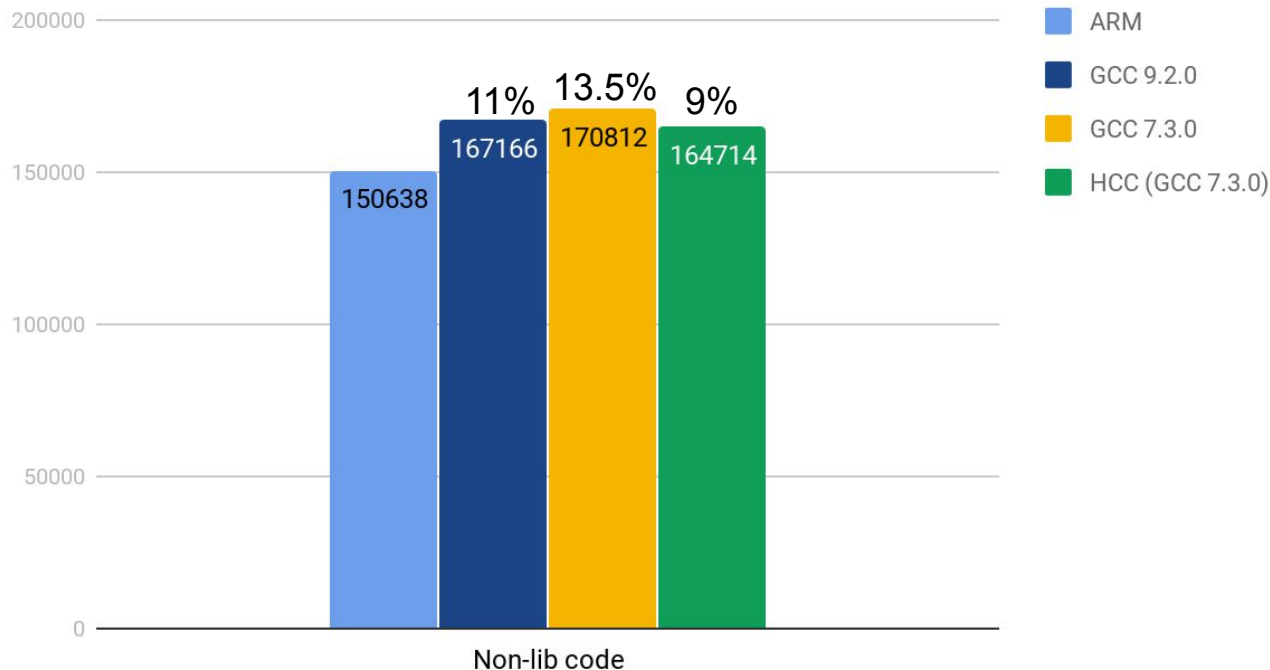
- xxxx li a5, 15
- xxxx slli s1, s1, 1
- 65c1 lui a1,0x10

- It would save 6 B in this sequence (remove **addi** and compress **slli**)

Recompile opus with gcc 7.3.0 (same GCC of HCC)

- Recompiled with GCC 7.3.0 and binutils 2.32
- 2.5% of code size improvement from GCC 7.3.0 to GCC 9.2.0
- Something has been improved, but not everything...

opus_demo - Code Size [B]



Recompile OPUS with GCC 7.3.0

Code Size [B] (Inflation %)	ARM	GCC 9.2.0	GCC 7.3.0	HCC (GCC 7.3.0)
celt_decode_lost	1796	2992 (67%)	3126 (74%)	2986 (66%)
op_pvq_search_c	660	796 (21%)	936 (42%)	900 (36%)
validate_celt_decoder	388	616 (59%)	616 (59%)	596 (54%)
silk_resampler_init	584	748 (28%)	756 (29%)	734 (26%)

From previous report: OPUS analysis (<op_pvq_search_c>)

- >50 memory operations with **s0** as a **base register**
- **s0** is a **fixed value** loaded in the prologue of the function (sp + offset)
- This fixed value **imposes** a **negative offset** in the memory ops
- The negative offset **prevents** the **compression** of the **memory ops**
- Tried with **GCC 9.2**, this problem seems **solved!**

```

·23e74:-fac42823.....sw—a2,-80(s0)
·23e78:-fad42a23.....sw—a3,-76(s0)
·23e7c:-fae42c23.....sw—a4,-72(s0)
·23e80:-faf42e23.....sw—a5,-68(s0)
·23e84:-766180ef.....jal—ra,3c5ea<
·23e88:-fb842703.....lw—a4,-72(s0)
·23e8c:-fac42803.....lw—a6,-84(s0)
·23e90:-00000793.....li—a5,0
·23e94:-fb042603.....lw—a2,-80(s0)
·23e98:-fb442683.....lw—a3,-76(s0)

```

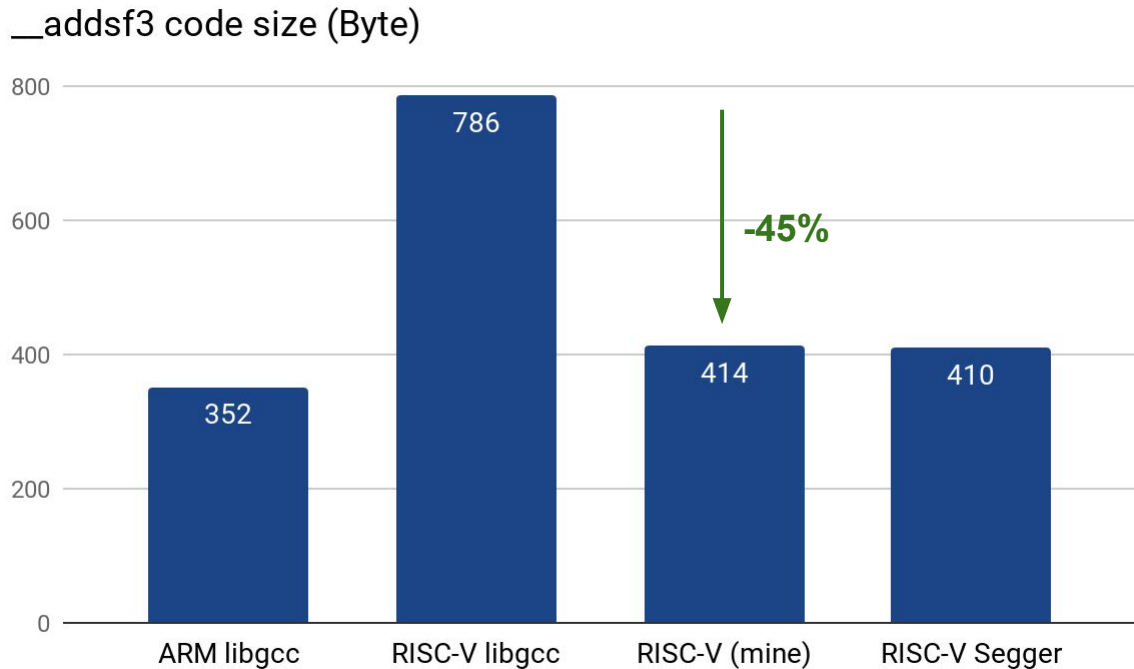
Fragment of HCC code (<op_pvq_search_c>)

Summary from the analysis so far

- There are some situations ARM can handle better (load small and mid immediates in registers, repeated loads from the constant pool of large immediates, comparison and branch)
- But it seems RISC-V GCC does not always produce an optimal code
- Some of these inefficiencies can be corrected also late in the compilation

FP Add

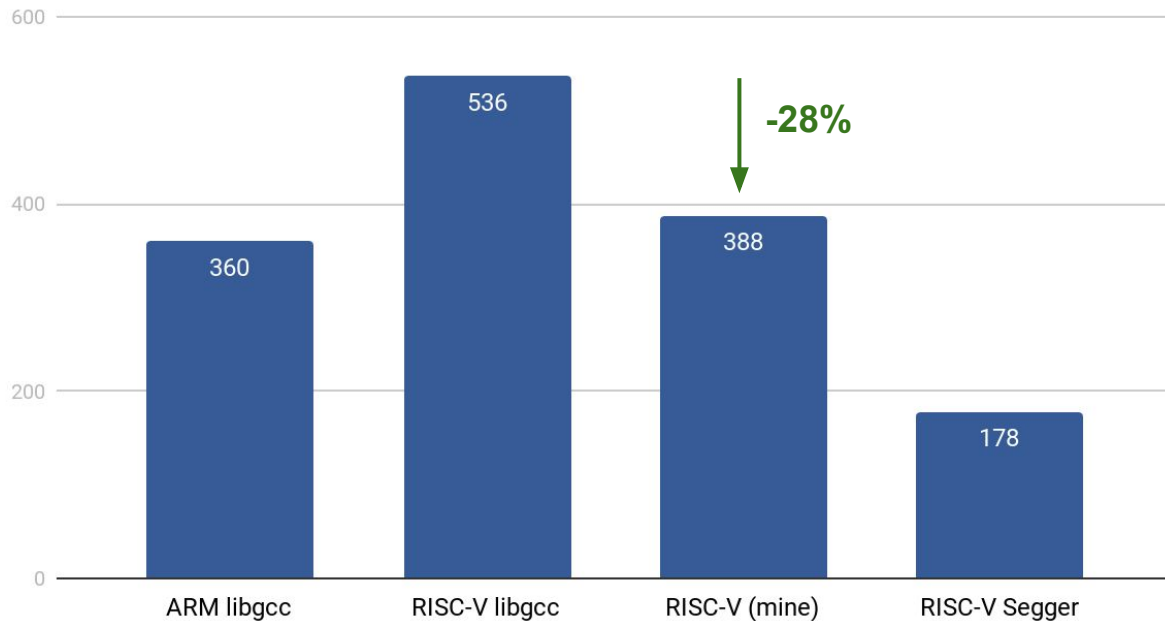
- Near to SEGGER, but it depends on clz
- Handle denormals
- PULP implementation would shorten the distance with ARM
- We don't have performance metrics



FP Multiplication

- Near to ARM code size
- SEGGER implementation does not handle denormals
- We still miss performance metrics
- The function can be optimized

__mulsf3 code size (Byte)



Further

- Test the performance of both **FADD** and **FMUL**
- Collect **performance** and **code size**, and compare them with the other available implementations

Tiny Floating-Point Unit

- **Tiny FPU** (FMA + COMP + CAST)
- **Comparison** with **SW** emulation

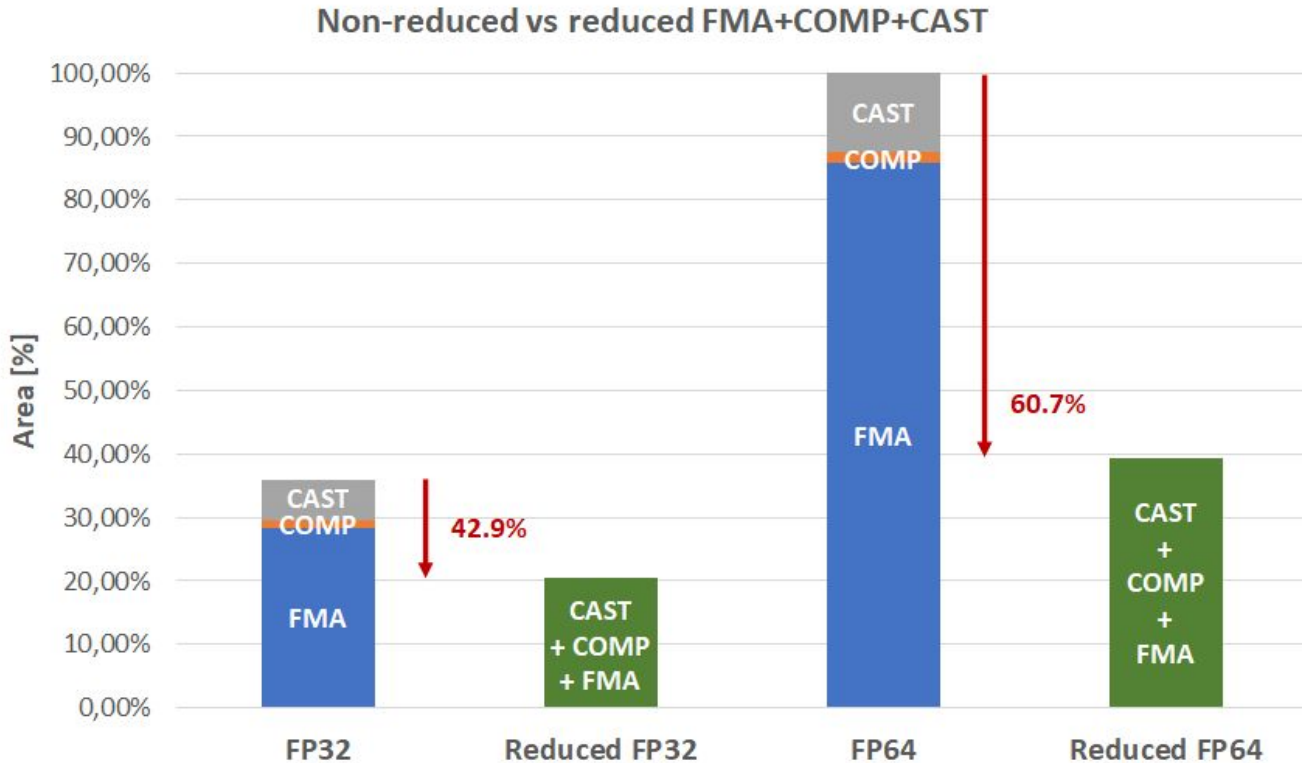
FMA + COMP + CAST HW optimizations - Single Precision

FP32	fpnew_fma + fpnew_noncomp + fpnew_multi_cast (NON-REDUCED)	fma_reduced comp_cast_idle	reduced_fma_comp_cast
Overall Area	100%	~59.5%	~57.1%
Latency	1 cycle (ADD/MUL/FMADD/COMP)	2 cycles COMP/CAST 10-12 cycles (ADD) 22-24 cycles (FMADD/MUL)	2 cycles COMP 9 cycles CAST 10-12 cycles (ADD) 22-24 cycles (FMADD/MUL)
Optimization	-	~40.5%	~42.9%

FMA + COMP + CAST HW optimizations - Double Precision

FP64	fpnew_fma + fpnew_noncomp + fpnew_multi_cast (NON-REDUCED)	fma_reduced comp_cast_idle	reduced_fma_comp_cast
Overall Area	100%	~43.0%	~39.3%
Latency	1 cycle (ADD/MUL/FMADD/COMP)	2 cycle COMP/CAST 10-12 cycles (ADD) 36-38 cycles (FMADD/MUL)	2 cycles COMP 9 cycles CAST 10-12 cycles (ADD) 36-38 cycles (FMADD/MUL)
Optimization	-	~57.0%	~60.7%

FMA + COMP HW optimizations



FMA - Single Precision

FP32	Latency reduced FPU	Average Latency SW emulation (SEGGER)	Code size (SEGGER)	Speed-up
ADD	10-12 cycles	49.5 cycles	410 B	~4.5x
SUB	10-12 cycles	62.2 cycles	10 B	~5.5x
MUL	22-24 cycles	39.3 cycles	178 B	~1.7x
FMADD	22-24 cycles	$49.5 + 39.3 = 88.8$ cycles		~3.9x

<https://www.segger.com/products/development-tools/runtime-library/technology/floating-point-library/>

COMP - Single Precision

FP32	Latency reduced FPU	Average Latency SW emulation (SEGGER)	Code size (SEGGER)	Speed-up
<	2 cycles	11 cycles	58 B	5.5x
<=	2 cycles	10 cycles	54 B	5x
>	2 cycles	10 cycles	50 B	5x
>=	2 cycles	11 cycles	62 B	5.5x
==	2 cycles	10 cycles	44 B	5x
!=	2 cycles	10 cycles	--	5x

<https://www.segger.com/products/development-tools/runtime-library/technology/floating-point-library/>

CAST - Single Precision

FP32	Latency reduced FPU	Average Latency SW emulation (SEGGER)	Code size (SEGGER)	Speed-up
INT32 -> FP32	9 cycles	32.6 cycles	66 B	3.6x
FP32 -> INT32	9 cycles	14 cycles	74 B	1.5x

Overall code size - SW emulation (FMA+COMP+CAST) = **1006 B**

<https://www.segger.com/products/development-tools/runtime-library/technology/floating-point-library/>

FMA - Double Precision

FP64	Latency reduced FPU	Average Latency SW emulation (SEGGER)	Code size (SEGGER)	Speed-up
ADD	10-12 cycles	62.8 cycles	724 B	~5.7x
SUB	10-12 cycles	82.8 cycles	10 B	~7.5x
MUL	36-38 cycles	75.0 cycles	286 B	~2x
FMADD	36-38 cycles	62.8 + 75 = 137.8 cycles	-	~3.7x

Overall code size - SW emulation (FMA+COMP+CAST) = **2332 B**

<https://www.segger.com/products/development-tools/runtime-library/technology/floating-point-library/>

COMP - Double Precision

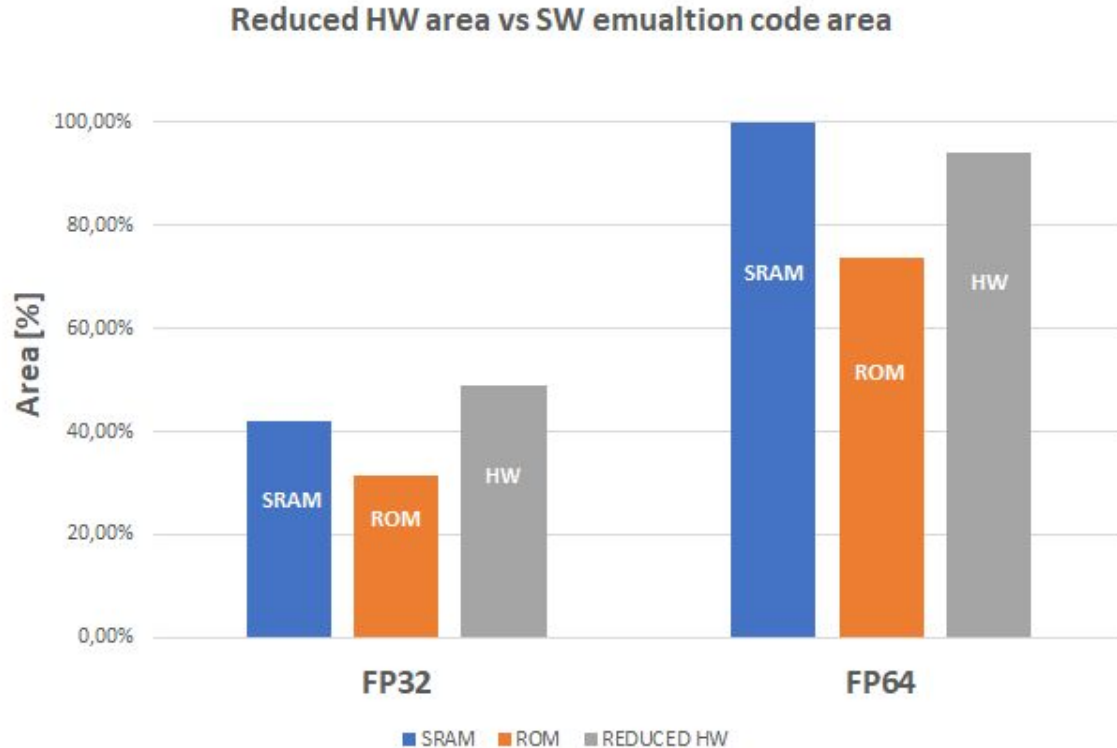
FP64	Latency reduced FPU	Average Latency SW emulation (SEGGER)	Code size (SEGGER)	Speed-up
<	2 cycles	16 cycles	70 B	~8x
<=	2 cycles	16 cycles	70 B	~8x
>	2 cycles	16.1 cycles	70 B	~8x
>=	2 cycles	16.1 cycles	70 B	~8x
==	2 cycles	14 cycles	52 B	~7x
!=	2 cycles	14 cycles	--	~7x

<https://www.segger.com/products/development-tools/runtime-library/technology/floating-point-library/>

CAST - Double Precision

FP64	Latency reduced FPU	Average Latency SW emulation (SEGGER)	Code size (SEGGER)	Speed-up
INT32 -> FP32	9 cycles	32.6 cycles	66 B	~3.6x
FP32 -> INT32	9 cycles	14 cycles	74 B	~1.5x
INT64 -> FP32	9 cycles	49.1 cycles	96 B	~5.5x
FP32 -> INT64	9 cycles	23.2 cycles	146 B	~2.6x
FP32 -> FP64	9 cycles	14.1 cycles	64 B	~1.5x
FP64 -> INT64	9 cycles	26.9 cycles	146 B	~3.0x
FP64 -> INT32	9 cycles	16.8 cycles	84 B	~3.6x
INT32 -> FP64	9 cycles	31.6 cycles	46 B	~1.9x
INT64 -> FP64	9 cycles	45.1 cycles	128 B	~5x
FP64 -> FP32	9 cycles	25.1 cycles	130 B	~2.8x

Reduced FPU vs SW emulation - estimates



Next steps

- Test the FPU for different input cases:
 - normals
 - denormals
 - inf
 - NaN
 - ...
- Performance evaluation