

Report 29.07.20

29/07/2020

Matteo Perotti

Luca Bertaccini

Pasquale Davide Schiavone

Stefan Mach

Professor Luca Benini

Integrated Systems Laboratory

ETH Zürich

Summary

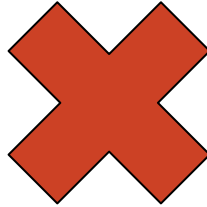
- **GCC inefficiencies**
- **FMA**
- **Soft-FP performance**

GCC inefficiencies

- Recompiled OPUS with GCC 10.2.0 to see if the inefficiencies are solved

GCC inefficiencies

- Recompiled OPUS with GCC 10.2.0 to see if the inefficiencies are solved



They are not!

Toy Example 0: stack ping-pong

```
#include <stdio.h>

#define M 2000
#define N 1000

extern void foo(int *arr);

int main(int argc, char **argv) {
    int arr_0[N]; // Loaded on the stack
    int arr_1[M]; // Loaded on the stack

    foo(arr_0);
    foo(arr_1);

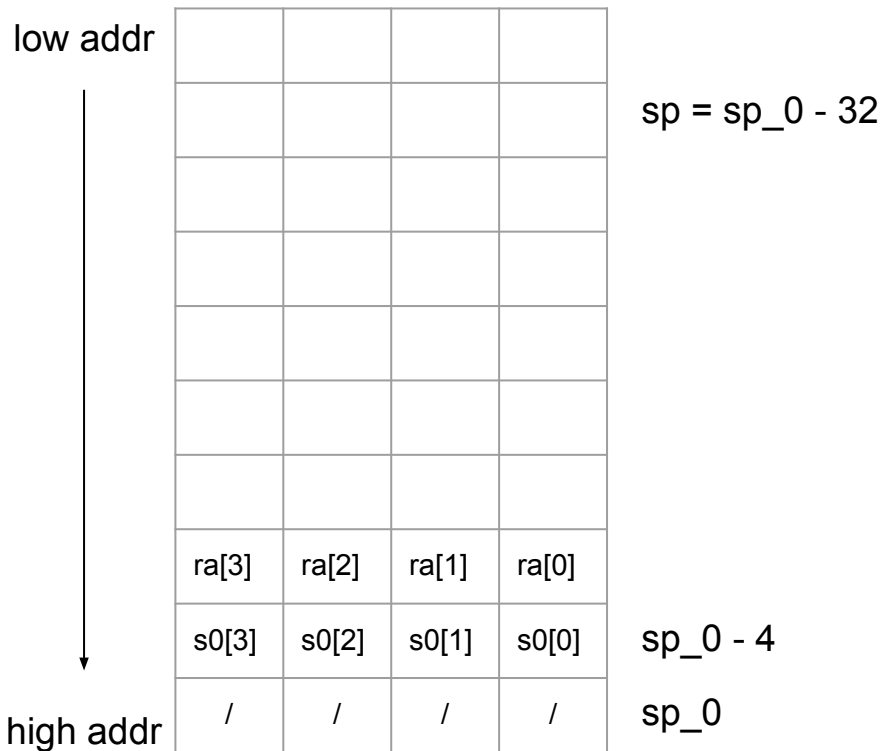
    return 0;
}
```

toy_example_0.c

```
00000000 <main>:
0: 1101 ..... addi sp, sp, -32
4: cc22 ..... sw s0, 24(sp)
c: ce06 ..... sw ra, 28(sp)
2: 7375 ..... lui t1, 0xffffd
6: 13030313 ..... addi t1, t1, 304 # fffffd130 <main+0xffffd130>
12: 911a ..... add sp, sp, t1
a: 640d ..... lui s0, 0x3
e: ee040793 ..... addi a5, s0, -288 # 2ee0 <main+0x2ee0>
16: 978a ..... add a5, a5, sp
14: 7575 ..... lui a0, 0xffffd
18: 12050513 ..... addi a0, a0, 288 # fffffd120 <main+0xffffd120>
1c: 953e ..... add a0, a0, a5
1e: 00000097 ..... auipc ra, 0x0
    1e: R_RISCV_CALL foo
    1e: R_RISCV_RELAX *ABS*
22: 000080e7 ..... jalr ra # 1e <main+0x1e>
26: ee040793 ..... addi a5, s0, -288
2c: 978a ..... add a5, a5, sp
2a: 7579 ..... lui a0, 0xfffffe
2e: 0c050513 ..... addi a0, a0, 192 # fffff0c0 <main+0xfffffe0c0>
32: 953e ..... add a0, a0, a5
34: 00000097 ..... auipc ra, 0x0
    34: R_RISCV_CALL foo
    34: R_RISCV_RELAX *ABS*
38: 000080e7 ..... jalr ra # 34 <main+0x34>
3c: 630d ..... lui t1, 0x3
3e: ed030313 ..... addi t1, t1, -304 # 2ed0 <main+0x2ed0>
42: 911a ..... add sp, sp, t1
44: 40f2 ..... lw ra, 28(sp)
46: 4462 ..... lw s0, 24(sp)
48: 4501 ..... li a0, 0
4a: 6105 ..... addi sp, sp, 32
4c: 8082 ..... ret
```

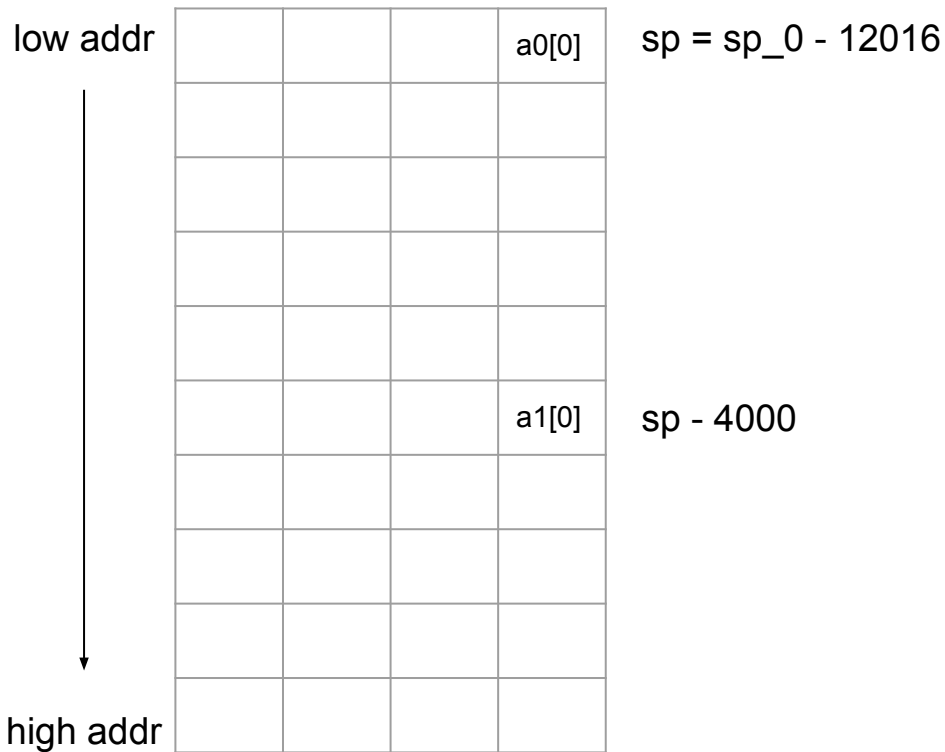
toy_example_0.s

Toy Example 0: stack ping-pong



- Push saved regs on the stack
- Align to 128-bit boundary, i.e. 32 B (ABI)


Toy Example 0: stack ping-pong



- Push saved regs on the stack
- Align to 128-bit boundary, i.e. 32 B (ABI)
- Allocate stack space for both the arrays (12000 B)

Toy Example 0: stack ping-pong

low addr



high addr

			a0[0]
			a1[0]

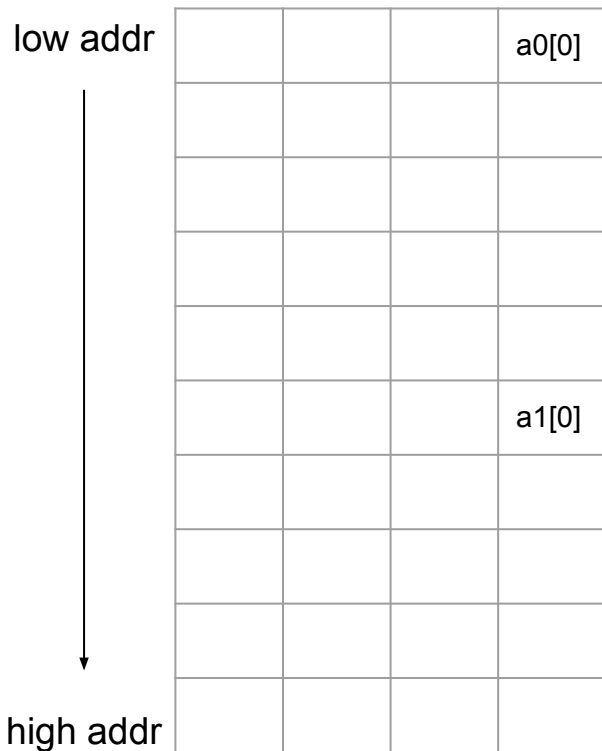
$sp = sp_0 - 12016$

- Push saved regs on the stack
- Align to 128-bit boundary, i.e. 32 B (ABI)
- Allocate stack space for both the arrays (12000 B)
- Put in a0 the pointer to the first vector:
 $a0 = sp + 12288 - 288 - 12288 + 288 = sp$

$sp - 4000$

sp_0

Toy Example 0: stack ping-pong

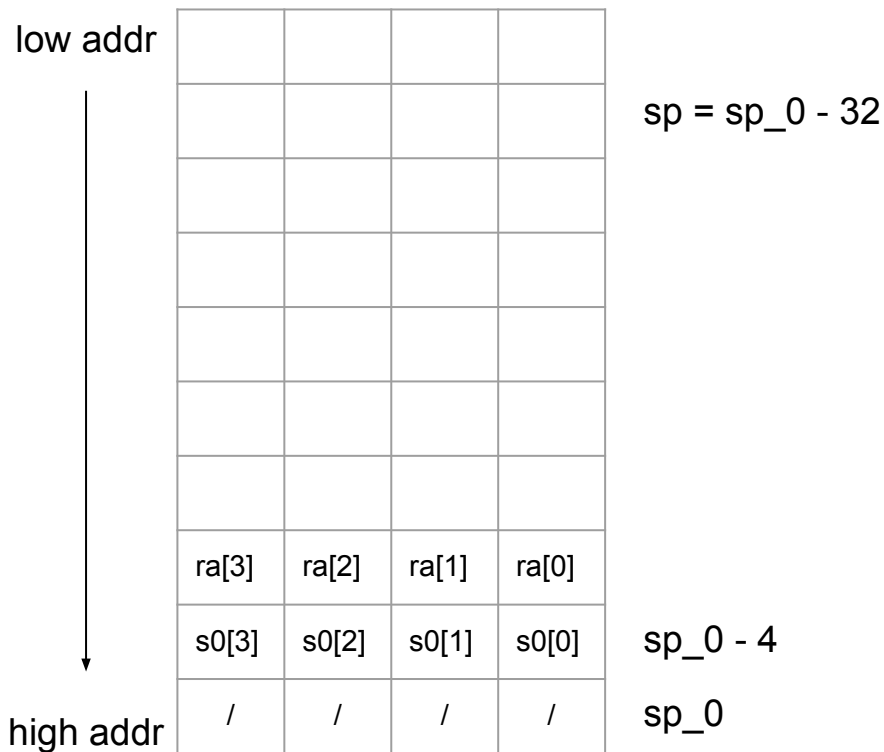


$sp = sp_0 - 12016$

$sp - 4000$

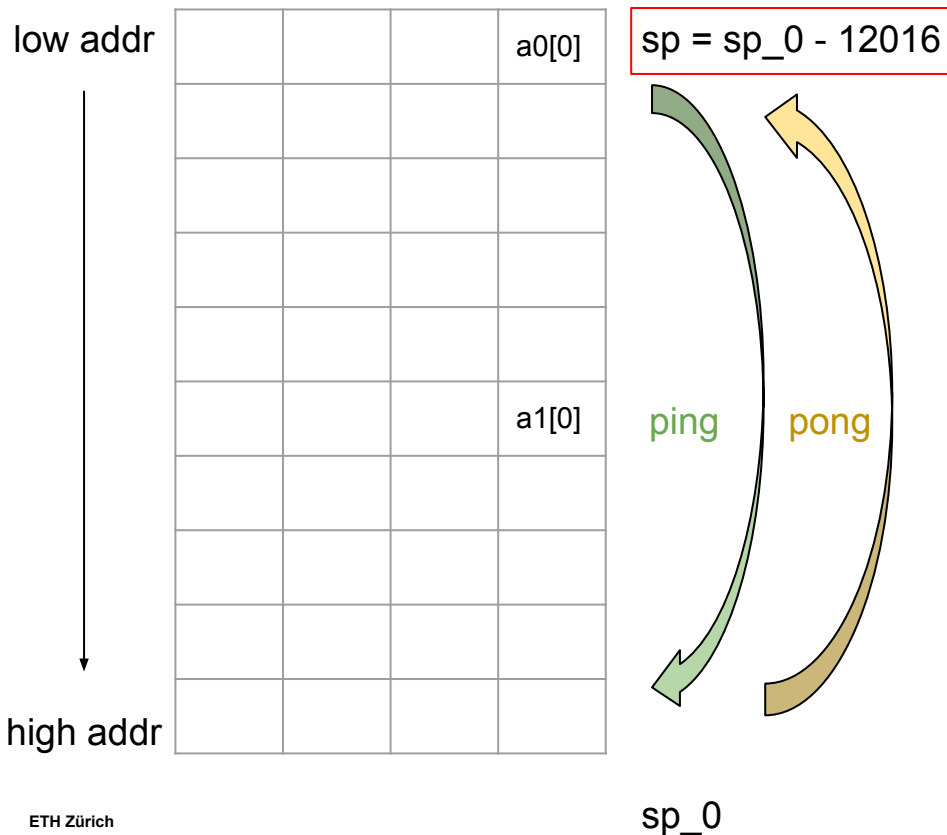
- Push saved regs on the stack
- Align to 128-bit boundary, i.e. 32 B (ABI)
- Allocate stack space for both the arrays (12000 B)
- Put in a0 the pointer to the first vector:
 $a0 = sp + 12288 - 288 - 12288 + 288 = sp$
- Jump to routine
- Put in a0 the pointer to the second vector:
 $a0 = sp + 12288 - 288 - 8192 + 192 = sp - 4000$

Toy Example 0: stack ping-pong



- Push saved regs on the stack
- Align to 128-bit boundary, i.e. 32 B (ABI)
- Allocate stack space for both the arrays (12000 B)
- Put in a0 the pointer to the first vector:
 $a0 = sp + 12288 - 288 - 12288 + 288 = sp$
- Jump to routine
- Put in a0 the pointer to the second vector:
 $a0 = sp + 12288 - 288 - 8192 + 192 = sp - 4000$
- Jump to routine
- Pop variables from the stack and return

Toy Example 0: stack ping-pong



- Push saved regs on the stack
- Align to 128-bit boundary, i.e. 32 B (ABI)
- Allocate stack space for both the arrays (12000 B)
- Put in a0 the pointer to the first vector:

$$a0 = sp + 12288 - 288 - 12288 + 288 = sp$$

$$a0 = sp + (12288 - 288) - (12288 - 288) = sp$$

$$a0 = sp + (12000) - (12000) = sp$$

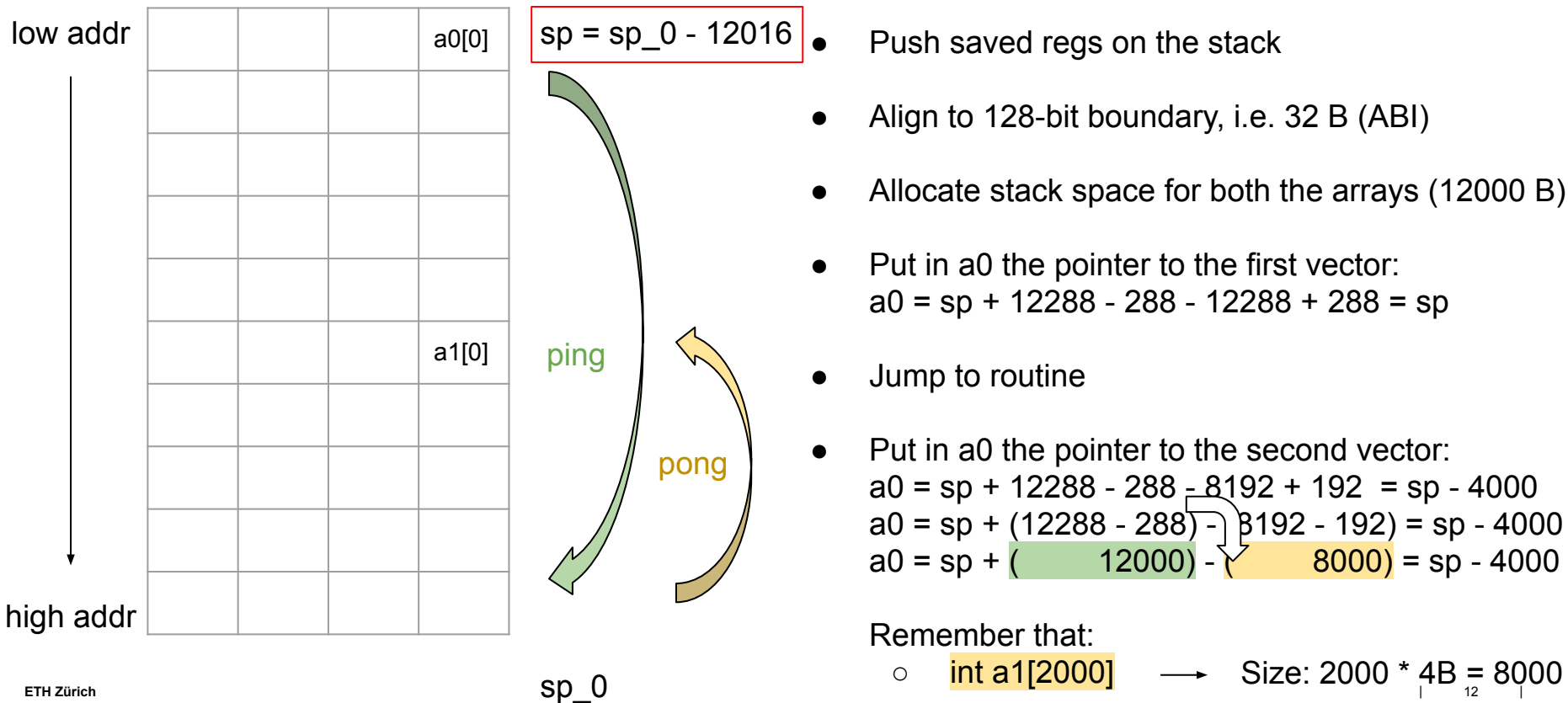
Remember that:

- int a0[1000]
- int a1[2000]



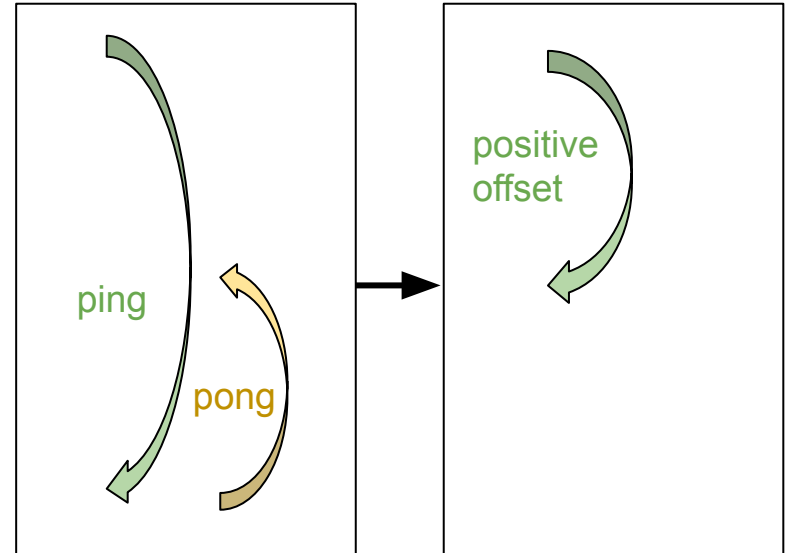
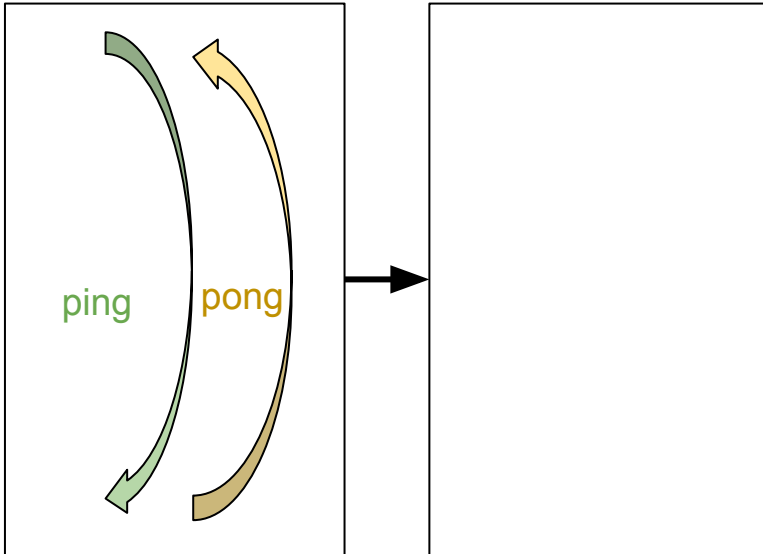
Total: $3000 * 4 \text{ B} = 12000 \text{ B}$

Toy Example 0: stack ping-pong



Toy Example 0: stack ping-pong

- The ping-pong is redundant



Toy Example 0: stack ping-pong

- The ping-pong is redundant
- When the array is large, each “ping” and “pong” value is at least 6 B
- Need to understand why this is triggered
- In the OPUS function (`celt_decode_lost`), the ping-pong seems to be repeated for each stack-related memory operation

Toy Example 1: inefficiencies + redundant epilogue

```
#include <stdio.h>

#define M 2000
#define N 1000

extern void foo(int *arr);

int main(int argc, char **argv) {
    int arr_0[argc*N]; // Loaded on the stack
    int arr_1[M]; // Loaded on the stack

    foo(arr_0);
    foo(arr_1);

    return 0;
}
```

toy_example_1.c

```
00000000 <main>:
0: 3e800793 ..... li a5,1000
4: 02f50533 ..... mul a0,a0,a5
8: 1101 ..... addi sp,sp,-32
a: 7379 ..... lui t1,0xffffe
c: ce06 ..... sw ra,28(sp)
e: cc22 ..... sw s0,24(sp)
10: 0d030313 ..... addi t1,t1,208 # fffffe0d0 <main+0xfffffe0d0>
14: 1000 ..... addi s0,sp,32
16: 911a ..... add sp,sp,t1
18: 050a ..... slli a0,a0,0x2
1a: 40a10133 ..... sub sp,sp,a0
1e: 850a ..... mv a0,sp
20: 00000097 ..... auipc ra,0x0
20: R_RISCV_CALL foo
20: R_RISCV_RELAX *ABS*
24: 000080e7 ..... jalr ra # 20 <main+0x20>
28: 7579 ..... lui a0,0xfffffe
2a: ff040793 ..... addi a5,s0,-16
2e: 0c050513 ..... addi a0,a0,192 # fffffe0c0 <main+0xfffffe0c0>
32: 953e ..... add a0,a0,a5
34: 00000097 ..... auipc ra,0x0
34: R_RISCV_CALL foo
34: R_RISCV_RELAX *ABS*
38: 000080e7 ..... jalr ra # 34 <main+0x34>
3c: 7379 ..... lui t1,0xfffffe
3e: 0b030313 ..... addi t1,t1,176 # fffffe0b0 <main+0xfffffe0b0>
42: 00640133 ..... add sp,s0,t1
46: 6309 ..... lui t1,0x2
48: f3030313 ..... addi t1,t1,-208 # 1f30 <main+0x1f30>
4c: 911a ..... add sp,sp,t1
4e: 40f2 ..... lw ra,28(sp)
50: 4462 ..... lw s0,24(sp)
52: 4501 ..... li a0,0
54: 6105 ..... addi sp,sp,32
56: 8082 ..... ret
```

toy_example_1.s

Toy Example 1: inefficiencies

- No ping-pong
- The base address is saved inside s0 (short circuited the “ping” part)
- Other inefficiencies: address formation and stack manipulation
- At the second routine call, the a0 is formed from s0 with redundant operations
- We can derive a0 from sp instead

Toy Example 1: redundant epilogue

```
#include <stdio.h>

#define M 2000
#define N 1000

extern void foo(int *arr);

int main(int argc, char **argv) {
    int arr_0[argc*N]; // Loaded on the stack
    int arr_1[M]; // Loaded on the stack

    foo(arr_0);
    foo(arr_1);

    return 0;
}
```

toy_example_1.c

```
00000000 <main>:
0: 3e800793 ..... li a5,1000
4: 02f50533 ..... mul a0,a0,a5
8: 1101 ..... addi sp,sp,-32
a: 7379 ..... lui t1,0xffffe
c: ce06 ..... sw ra,28(sp)
e: cc22 ..... sw s0,24(sp)
10: 0d030313 ..... addi t1,t1,208 # fffffe0d <main+0xfffffe0d>
14: 1000 ..... addi s0,sp,32
16: 911a ..... add sp,sp,t1
18: 050a ..... slli a0,a0,0x2
1a: 40a10133 ..... sub sp,sp,a0
1e: 850a ..... mv a0,sp
20: 00000097 ..... auipc ra,0x0
20: R_RISCV_CALL foo
20: R_RISCV_RELAX *ABS*
24: 000080e7 ..... jalr ra,#20 <main+0x20>
28: 7579 ..... lui a0,0xffffe
2a: ff040793 ..... addi a5,s0,-16
2e: 0c050513 ..... addi a0,a0,192 # fffffe0c <main+0xfffffe0c>
32: 953e ..... add a0,a0,a5
34: 00000097 ..... auipc ra,0x0
34: R_RISCV_CALL foo
34: R_RISCV_RELAX *ABS*
38: 000080e7 ..... jalr ra,#34 <main+0x34>
3c: 7379 ..... lui t1,0xffffe
3e: 0b030313 ..... addi t1,t1,176 # fffffe0b <main+0xfffffe0b>
42: 00640133 ..... add sp,s0,t1
46: 6309 ..... lui t1,0x2
48: f3030313 ..... addi t1,t1,-208 # 1f30 <main+0x1f30>
4c: 911a ..... add sp,sp,t1
4e: 40f2 ..... lw ra,28(sp)
50: 4462 ..... lw s0,24(sp)
52: 4501 ..... li a0,0
54: 6105 ..... addi sp,sp,32
56: 8082 ..... ret
```

toy_example_1.s

Toy Example 1: redundant epilogue

7379	lui	t1,0xffffe
0b030313	addi	t1,t1,176
00640133	add	sp,s0,t1
6309	lui	t1,0x2
f3030313	addi	t1,t1,-208
911a	add	sp,sp,t1

18 B

xxxxxxxx	addi	sp,s0,-32
----------	------	-----------

4 B

FMA

- Work in progress
- Trying to save code size w.r.t. add+mul
- Overhead of keeping the full precision for the partial result

Soft FP-performance

- Test and benchmarks with pulp-dsp on a Virtual Platform emulating MrWolf (<https://github.com/pulp-platform/pulp-dsp>)
- Initial benchmarks with random inputs
- Measured: total cycles, cache stalls, average IPC
- Work in progress... but some initial comparative results!

Soft FP-performance

- Benchmarked both my new implementations and the soft-float implementation from libgcc compiled with **-Os** for **rv32imc**
- The functions are put inside a wrapper. The measured cycles are not just the cycles used to execute our functions, but can give some insights

Soft FP-performance and size

	GCC Add	My Add	GCC Mul	My Mul
Cycles	451	377 (-16.4%)	487	279 (-43%)
Code size (B)	786	414 (-45%)	536	388 (-28%)

- Want to understand better what I am measuring (I am measuring more than just the cycles needed by the function)
- Measure performance also with 0-delay memories
- But... the benchmark environment is the same. Results suggest there is an improvement

Further

- Better measure the performance of the functions only
- Finish and test the FMA
- Develop more precise toy examples with the redundancy of the OPUS function
- Better understand the GCC inefficiencies

Tiny Floating-Point Unit

Tiny Floating-Point Unit

- **Speaking proposal** for RISC-V Summit 2020 **submitted**
- **Tested tiny FPU**

Tiny FPU - Test

- Designed tested for different combinations of:
 - normal
 - denormal
 - inf
 - NaN
 - zero
- Instructions:
 - FMADD, ADD, MUL
 - EQ, LT, LTE
 - F2I, I2F

Tiny FPU - Test

- From 100.000 to 5.000.000 **random test vectors** with **constraints** (normals/denormals/inf/NaN/zero)
- Some bugs due to register re-use detected and corrected:
 - MUL
 - LT, LTE
 - FMADD
- Added a couple of FFs

FMA + COMP + CAST HW optimizations - Single Precision

FP32	fpnew_fma + fpnew_noncomp + fpnew_multi_cast (NON-REDUCED)	reduced_fma_comp_cast (last presentation)	reduced_fma_comp_cast tested
Overall Area	100%	~57.1%	~58.3%
Latency	1 cycle (ADD/MUL/FMADD/COMP)	2 cycles COMP 9 cycles 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	-	~42.9%	~41.7%

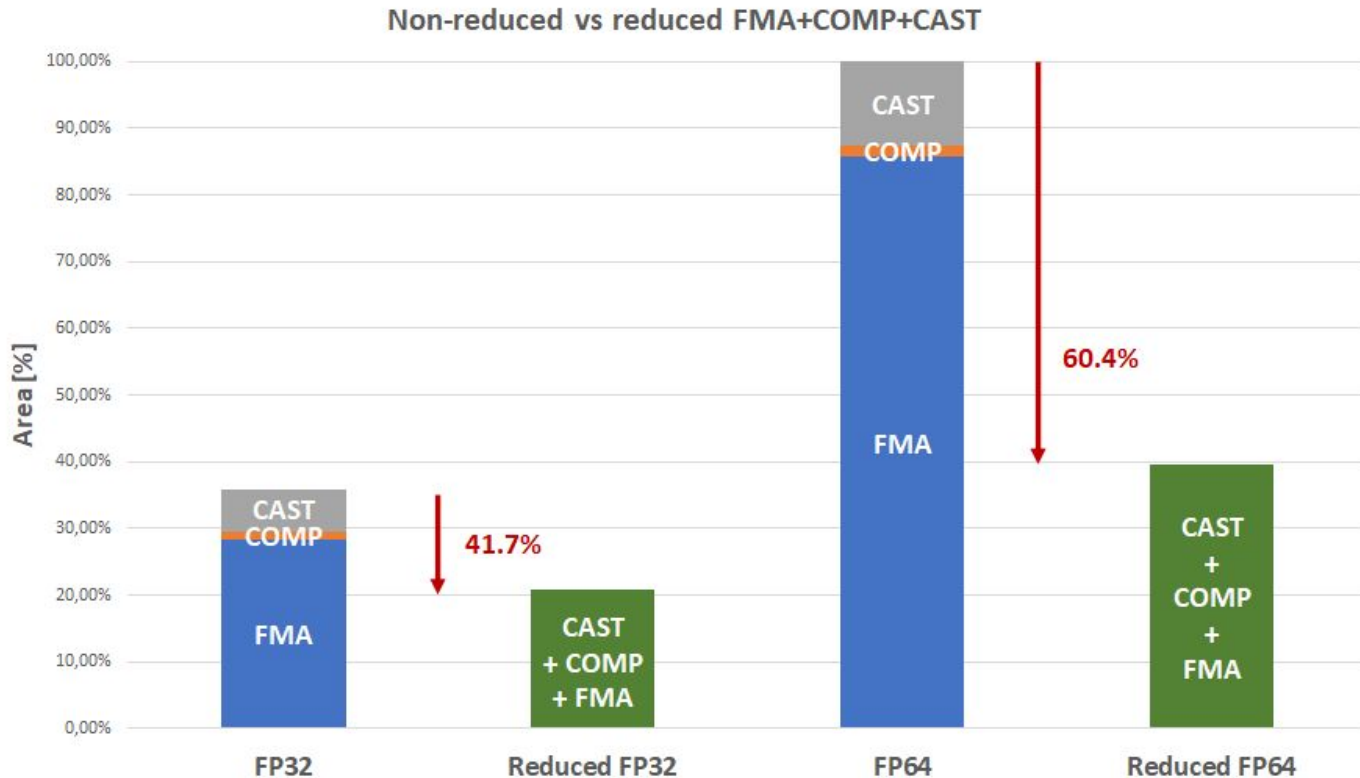
- No changes in the latencies after testing

FMA + COMP + CAST HW optimizations - Double Precision

FP64	fpnew_fma + fpnew_noncomp + fpnew_multi_cast (NON-REDUCED)	reduced_fma_comp_cast (last presentation)	reduced_fma_comp_cast tested
Overall Area	100%	~39.3%	~39.6%
Latency	1 cycle (ADD/MUL/FMADD/COMP)	2 cycles COMP 9 cycles 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	-	~60.7%	~60.4%

- No changes in the latencies after testing

FMA + COMP + CAST: HW optimization (tested)



Next steps

- Integrate the FPU into the core
 - **Fpnew**: one block instantiated per operation group (FMA, COMP, CAST)
 - **Tiny FPU**: one block handles all the operation groups
- Performance evaluation for real cases

Appendix: 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/>

Appendix: 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/>

Appendix: 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/>

Appendix: 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/>

Appendix: 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/>

Appendix: 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