Report 01.07.20

01/07/2020

Matteo Perotti

Luca Bertaccini

Pasquale Davide Schiavone

Stefan Mach

Professor Luca Benini

Integrated Systems Laboratory

ETH Zürich

Summary

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

When can HCC solve a size problem?

No initial problem
No big difference

- dijkstra
- fft
- aha-mont64
- minver
- nbody
- nsichneu
- st
- statemate
- wikisort

HCC makes the difference

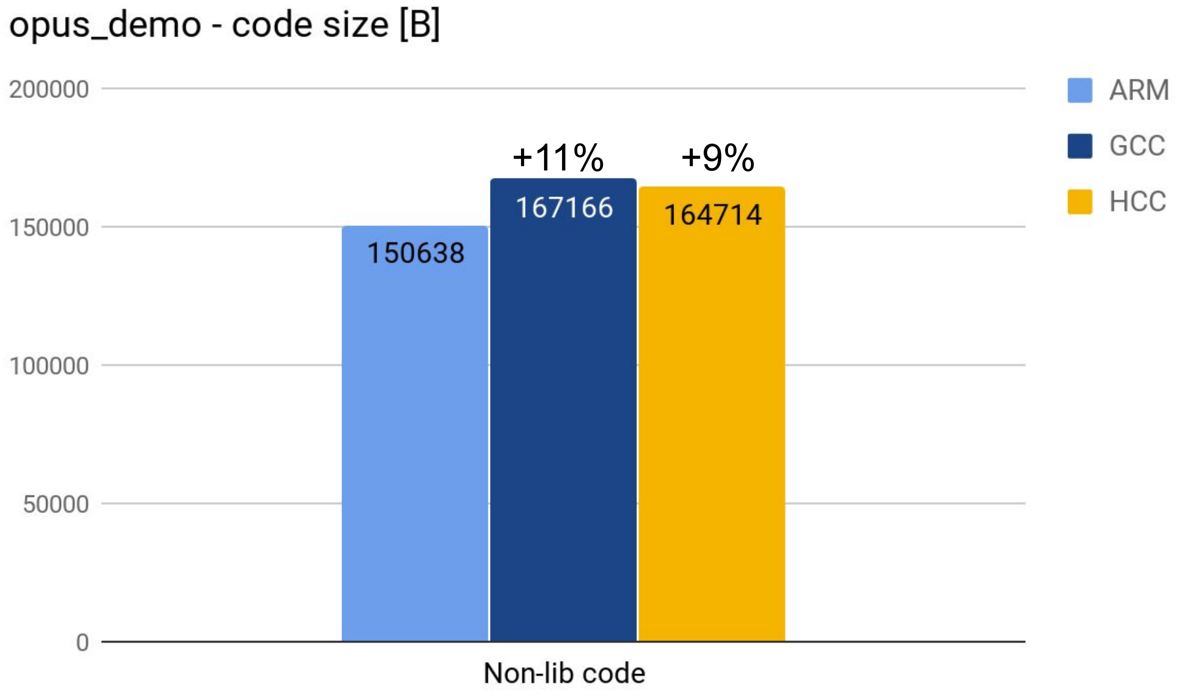
- NB-IoT
- sha
- qsort
- crc32
- edn
- matmult-int
- nettle-sha256
- qrduino
- sglib-combined
- ud

HCC does not solve the problem

- **opus**
- bitcount
- qsort
- stringsearch
- susan
- huffbench
- nettle-aes
- picojpeg
- slre

opus

- Approximate results (hard to separate all the library functions)
- The size difference is related to non-lib functions
- Analyzed the most inflated functions



opus

OPUS function	**ARM size [B]**	**HCC size [B]**	**Size diff [B]**	**Inflation**
celt_decode_lost	1796	2986	1190	66%
op_pvq_search_c	660	900	240	36%
validate_celt_decoder	388	596	208	53%

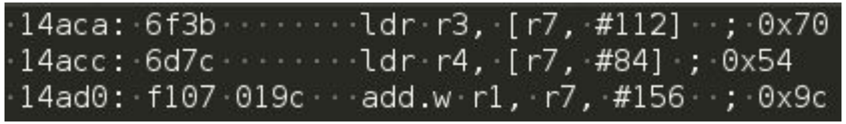
OPUS analysis (<celt_decode_lost>)

- The **function** is **big** and performs a lot of operations
- Hard to precisely make a comparison, but… we can note something!

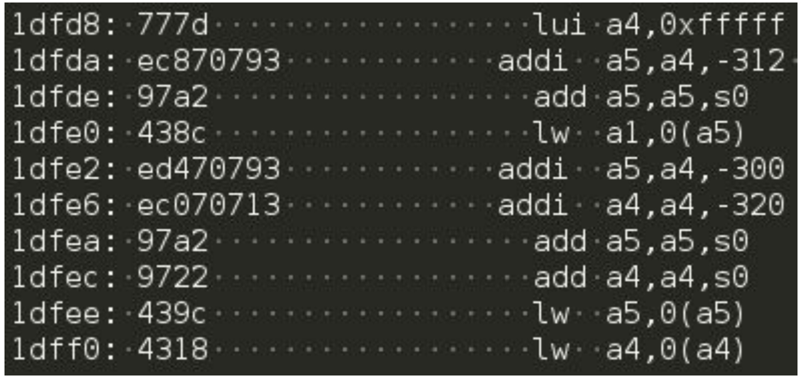
- **ARM** and **HCC** code have approx the **same number** of **mem ops** (250)
- The **ratio** of **compressed/uncompressed memory ops** is the **same**

- **ARM** always **keeps** the **same register** as **base address** for **+65%** of **memory ops**. This **reg** is **never modified**
- **RISC-V** uses **multiple regs**, every time **modifying them**

OPUS analysis (<celt_decode_lost>)



```
14aca: 6f3b        ldr r3, [r7, #112]  ; 0x70
14acc: 6d7c        ldr r4, [r7, #84] ; 0x54
14ad0: f107 019c   add.w r1, r7, #156  ; 0x9c
```



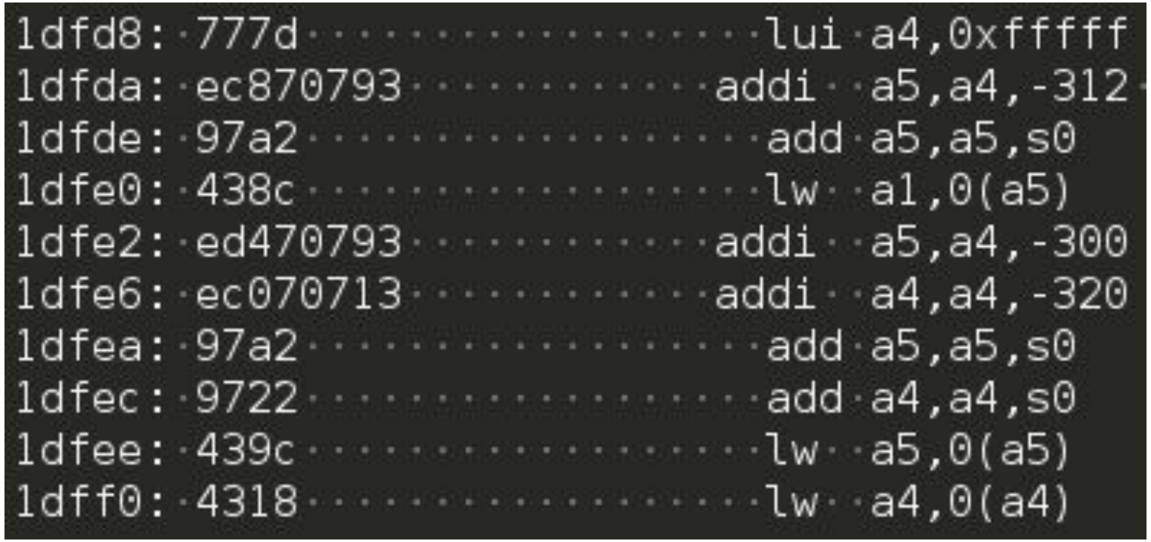
```
1dfd8: 777d                  lui a4,0xfffff
1dfda: ec870793            addi  a5,a4,-312
1dfde: 97a2                  add a5,a5,s0
1dfe0: 438c                  lw  a1,0(a5)
1dfe2: ed470793            addi  a5,a4,-300
1dfe6: ec070713            addi  a4,a4,-320
1dfea: 97a2                  add a5,a5,s0
1dfec: 9722                  add a4,a4,s0
1dfee: 439c                  lw  a5,0(a5)
1dff0: 4318                  lw  a4,0(a4)
```

ARM code: 173 mem ops on 256 use r7 as base reg. r7 is loaded in the beginning and never modified.

HCC code: before a memory operation, the code loads a value into a reg and adds an offset.

OPUS analysis (<celt_decode_lost>)

- The way **HCC schedules** the **operations** seems **redundant**.
- In this fragment of code, the **net effect** is to **load** values in **a1, a4, a5**.



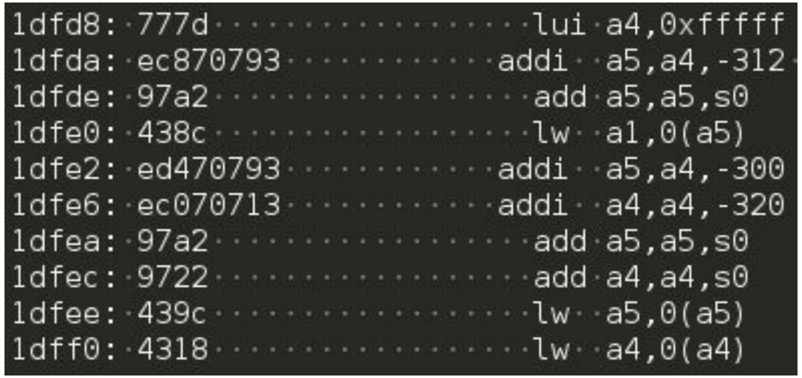
```
1dfd8: 777d                 lui  a4,0xfffff
1dfda: ec870793             addi  a5,a4,-312
1dfde: 97a2                 add  a5,a5,s0
1dfe0: 438c                 lw   a1,0(a5)
1dfe2: ed470793             addi  a5,a4,-300
1dfe6: ec070713             addi  a4,a4,-320
1dfea: 97a2                 add  a5,a5,s0
1dfec: 9722                 add  a4,a4,s0
1dfee: 439c                 lw   a5,0(a5)
1dff0: 4318                 lw   a4,0(a4)
```

Fragment of <celt_decode_lost> function (compiled with HCC)

ETH zürich

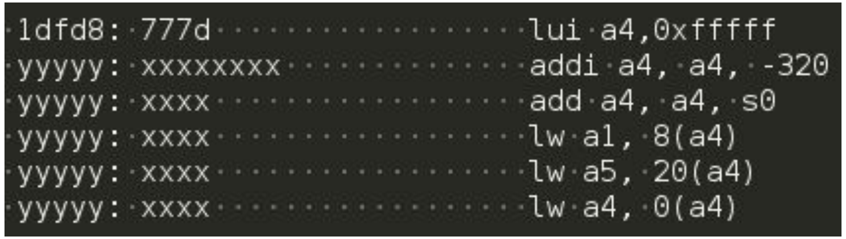
OPUS analysis (<celt_decode_lost>)

- The **addresses** are **similar**: we **can use only one register** as **base address**



```
1dfd8: 777d                  lui a4,0xffff
1dfda: ec870793              addi  a5,a4,-312
1dfde: 97a2                  add a5,a5,s0
1dfe0: 438c                  lw  a1,0(a5)
1dfe2: ed470793              addi  a5,a4,-300
1dfe6: ec070713              addi  a4,a4,-320
1dfea: 97a2                  add a5,a5,s0
1dfec: 9722                  add a4,a4,s0
1dfee: 439c                  lw  a5,0(a5)
1dff0: 4318                  lw  a4,0(a4)
```

Original HCC code

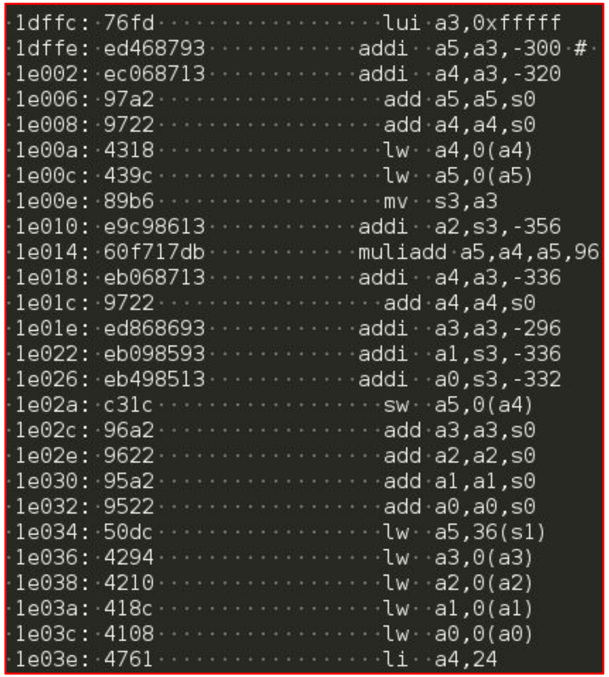


```
1dfd8: 777d                  lui a4,0xffff
yyyyy: xxxxxxxx              addi a4, a4, -320
yyyyy: xxxx                  add a4, a4, s0
yyyyy: xxxx                  lw a1, 8(a4)
yyyyy: xxxx                  lw a5, 20(a4)
yyyyy: xxxx                  lw a4, 0(a4)
```

Manually optimized code

OPUS analysis (<celt_decode_lost>)

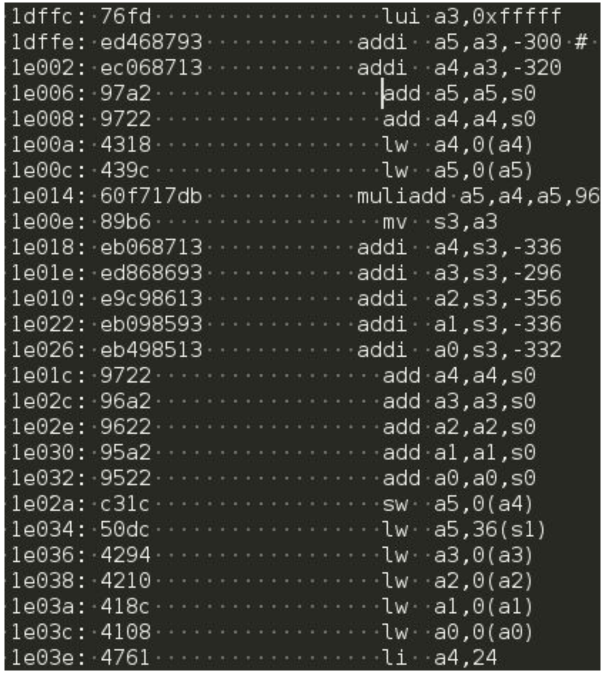
- Another portion of the code (GCC 9.2 does not solve the issue):



```
1dffc: 76fd                 lui a3,0xfffff
1dffe: ed468793           addi  a5,a3,-300 #
1e002: ec068713           addi  a4,a3,-320
1e006: 97a2                 add a5,a5,s0
1e008: 9722                 add a4,a4,s0
1e00a: 4318                 lw  a4,0(a4)
1e00c: 439c                 lw  a5,0(a5)
1e00e: 89b6                 mv  s3,a3
1e010: e9c98613           addi  a2,s3,-356
1e014: 60f717db           muliadd a5,a4,a5,96
1e018: eb068713           addi  a4,a3,-336
1e01c: 9722                 add a4,a4,s0
1e01e: ed868693           addi  a3,a3,-296
1e022: eb098593           addi  a1,s3,-336
1e026: eb498513           addi  a0,s3,-332
1e02a: c31c                 sw  a5,0(a4)
1e02c: 96a2                 add a3,a3,s0
1e02e: 9622                 add a2,a2,s0
1e030: 95a2                 add a1,a1,s0
1e032: 9522                 add a0,a0,s0
1e034: 50dc                 lw  a5,36(s1)
1e036: 4294                 lw  a3,0(a3)
1e038: 4210                 lw  a2,0(a2)
1e03a: 418c                 lw  a1,0(a1)
1e03c: 4108                 lw  a0,0(a0)
1e03e: 4761                 li  a4,24
```

Original HCC code

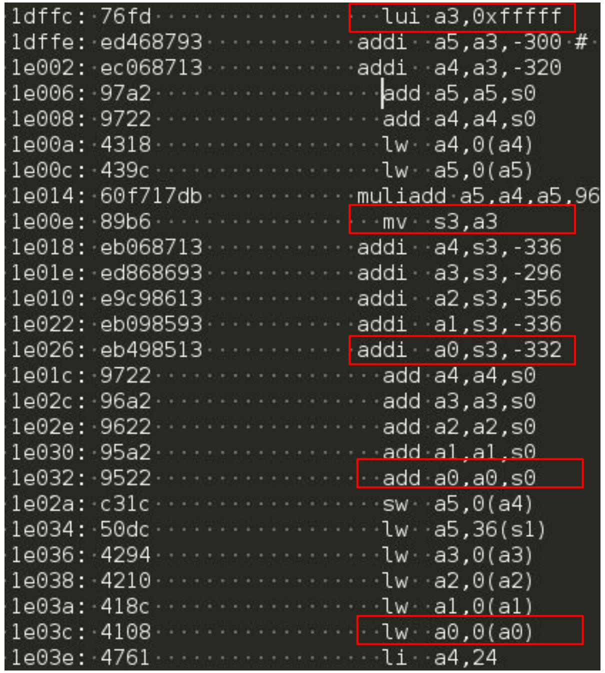
Manual reordering $\longrightarrow$



```
1dffc: 76fd                 lui a3,0xfffff
1dffe: ed468793           addi  a5,a3,-300 #
1e002: ec068713           addi  a4,a3,-320
1e006: 97a2                 add a5,a5,s0
1e008: 9722                 add a4,a4,s0
1e00a: 4318                 lw  a4,0(a4)
1e00c: 439c                 lw  a5,0(a5)
1e014: 60f717db           muliadd a5,a4,a5,96
1e00e: 89b6                 mv  s3,a3
1e018: eb068713           addi  a4,s3,-336
1e01e: ed868693           addi  a3,s3,-296
1e010: e9c98613           addi  a2,s3,-356
1e022: eb098593           addi  a1,s3,-336
1e026: eb498513           addi  a0,s3,-332
1e01c: 9722                 add a4,a4,s0
1e02c: 96a2                 add a3,a3,s0
1e02e: 9622                 add a2,a2,s0
1e030: 95a2                 add a1,a1,s0
1e032: 9522                 add a0,a0,s0
1e02a: c31c                 sw  a5,0(a4)
1e034: 50dc                 lw  a5,36(s1)
1e036: 4294                 lw  a3,0(a3)
1e038: 4210                 lw  a2,0(a2)
1e03a: 418c                 lw  a1,0(a1)
1e03c: 4108                 lw  a0,0(a0)
1e03e: 4761                 li  a4,24
```

Equivalent code

OPUS analysis (<celt_decode_lost>)



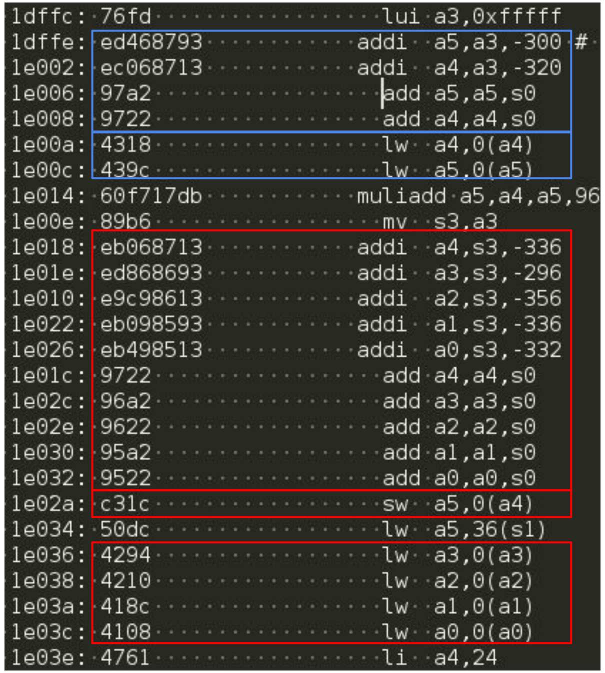
```
1dffc: 76fd                 lui a3,0xfffff
1dffe: ed468793           addi  a5,a3,-300 #
1e002: ec068713           addi  a4,a3,-320
1e006: 97a2                 add a5,a5,s0
1e008: 9722                 add a4,a4,s0
1e00a: 4318                 lw  a4,0(a4)
1e00c: 439c                 lw  a5,0(a5)
1e014: 60f717db           muliadd a5,a4,a5,96
1e00e: 89b6                 mv  s3,a3
1e018: eb068713           addi  a4,s3,-336
1e01e: ed868693           addi  a3,s3,-296
1e010: e9c98613           addi  a2,s3,-356
1e022: eb098593           addi  a1,s3,-336
1e026: eb498513           addi  a0,s3,-332
1e01c: 9722                 add a4,a4,s0
1e02c: 96a2                 add a3,a3,s0
1e02e: 9622                 add a2,a2,s0
1e030: 95a2                 add a1,a1,s0
1e032: 9522                 add a0,a0,s0
1e02a: c31c                 sw  a5,0(a4)
1e034: 50dc                 lw  a5,36(s1)
1e036: 4294                 lw  a3,0(a3)
1e038: 4210                 lw  a2,0(a2)
1e03a: 418c                 lw  a1,0(a1)
1e03c: 4108                 lw  a0,0(a0)
1e03e: 4761                 li  a4,24
```

Equivalent code

- The memory operations are:
  - lw rd, 0(0xfffff000 - **off** + s0)
- **off** is:
  - 296
  - 300
  - 320
  - 332
  - 336
  - 336
  - 356
- **off** is always in the range [296, 356]
- The **sequence** of “addi + add” is **repeated** for **each register** (a0, a1, a2, a3, a4)
- But with the **load/store** operations we can **add** an **offset in place**!!!

OPUS analysis (<celt_decode_lost>:)

- Strange compiler choices (GCC 9.2 produces similar code)



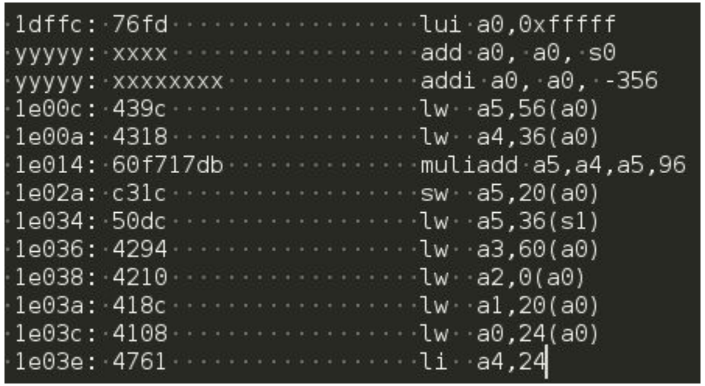
```
1dffc: 76fd                 lui  a3,0xfffff
1dffe: ed468793          addi  a5,a3,-300 #
1e002: ec068713          addi  a4,a3,-320
1e006: 97a2                 add  a5,a5,s0
1e008: 9722                 add  a4,a4,s0
1e00a: 4318                 lw   a4,0(a4)
1e00c: 439c                 lw   a5,0(a5)
1e014: 60f717db          muliadd a5,a4,a5,96
1e00e: 89b6                 mv   s3,a3
1e018: eb068713          addi  a4,s3,-336
1e01e: ed868693          addi  a3,s3,-296
1e010: e9c98613          addi  a2,s3,-356
1e022: eb098593          addi  a1,s3,-336
1e026: eb498513          addi  a0,s3,-332
1e01c: 9722                 add  a4,a4,s0
1e02c: 96a2                 add  a3,a3,s0
1e02e: 9622                 add  a2,a2,s0
1e030: 95a2                 add  a1,a1,s0
1e032: 9522                 add  a0,a0,s0
1e02a: c31c                 sw   a5,0(a4)
1e034: 50dc                 lw   a5,36(s1)
1e036: 4294                 lw   a3,0(a3)
1e038: 4210                 lw   a2,0(a2)
1e03a: 418c                 lw   a1,0(a1)
1e03c: 4108                 lw   a0,0(a0)
1e03e: 4761                 li   a4,24
```

Equivalent code

Manual optimization

$$\longrightarrow$$

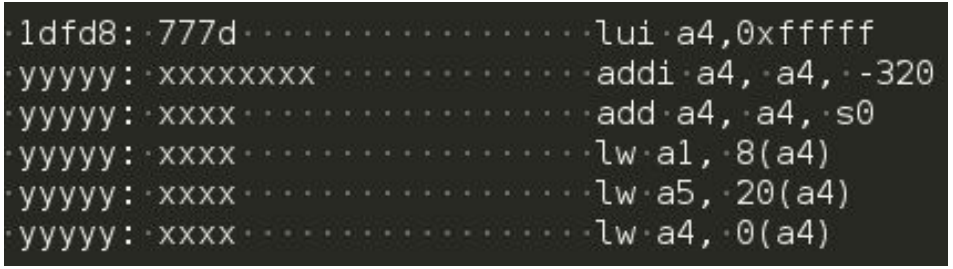
- ARM: 40 B
- HCC: **68 B**
- Manually optimized: **30 B**



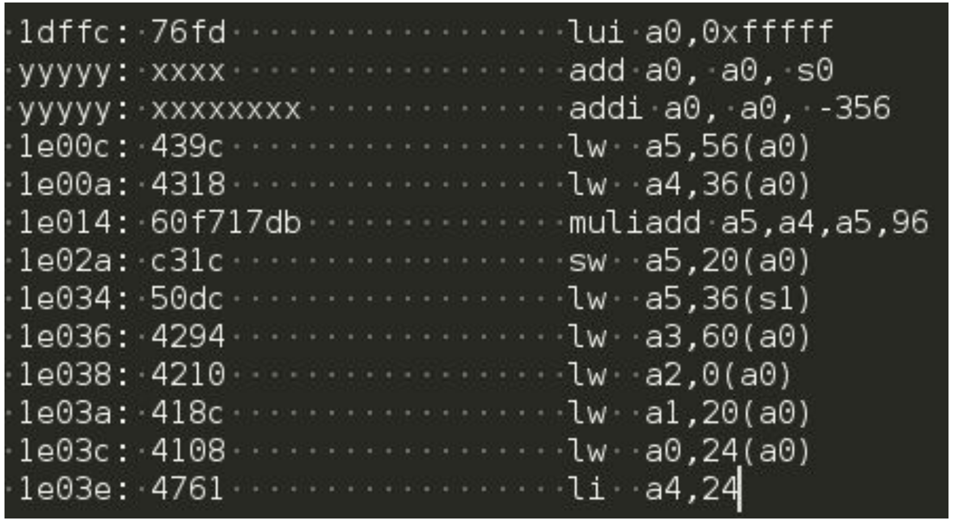
```
1dffc: 76fd                 lui  a0,0xfffff
yyyyy: xxxx                 add  a0, a0, s0
yyyyy: xxxxxxxx             addi a0, a0, -356
1e00c: 439c                 lw   a5,56(a0)
1e00a: 4318                 lw   a4,36(a0)
1e014: 60f717db             muliadd a5,a4,a5,96
1e02a: c31c                 sw   a5,20(a0)
1e034: 50dc                 lw   a5,36(s1)
1e036: 4294                 lw   a3,60(a0)
1e038: 4210                 lw   a2,0(a0)
1e03a: 418c                 lw   a1,20(a0)
1e03c: 4108                 lw   a0,24(a0)
1e03e: 4761                 li   a4,24
```

Equivalent code

OPUS analysis (<celt_decode_lost>:)



```
1dfd8: 777d                 lui a4,0xfffff
yyyyy: xxxxxxxx             addi a4, a4, -320
yyyyy: xxxx                 add a4, a4, s0
yyyyy: xxxx                 lw a1, 8(a4)
yyyyy: xxxx                 lw a5, 20(a4)
yyyyy: xxxx                 lw a4, 0(a4)
```



```
1dffc: 76fd                 lui a0,0xfffff
yyyyy: xxxx                 add a0, a0, s0
yyyyy: xxxxxxxx             addi a0, a0, -356
1e00c: 439c                 lw  a5,56(a0)
1e00a: 4318                 lw  a4,36(a0)
1e014: 60f717db             muliadd a5,a4,a5,96
1e02a: c31c                 sw  a5,20(a0)
1e034: 50dc                 lw  a5,36(s1)
1e036: 4294                 lw  a3,60(a0)
1e038: 4210                 lw  a2,0(a0)
1e03a: 418c                 lw  a1,20(a0)
1e03c: 4108                 lw  a0,24(a0)
1e03e: 4761                 li  a4,24
```

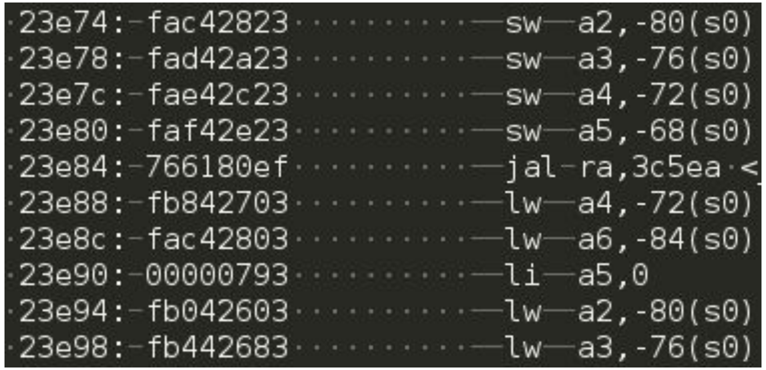
- These are the **two** manually **optimized fragments** of code.
- s0 = sp + fixed_offset
- After the initial assignment, **s0** is **never modified**.
- We always **load 0xfffff**, then **add s0** and a **variable offset**.
- This **pattern** is **frequent** into the function (it seems >60 times).
- The **variable offsets** are very **similar** also for **different fragments**.

$\downarrow$

- A **fixed register** for the **whole function** as a **base** for the memory operations (as ARM does) can be **convenient**.

OPUS analysis (<op_pvq_search_c>)

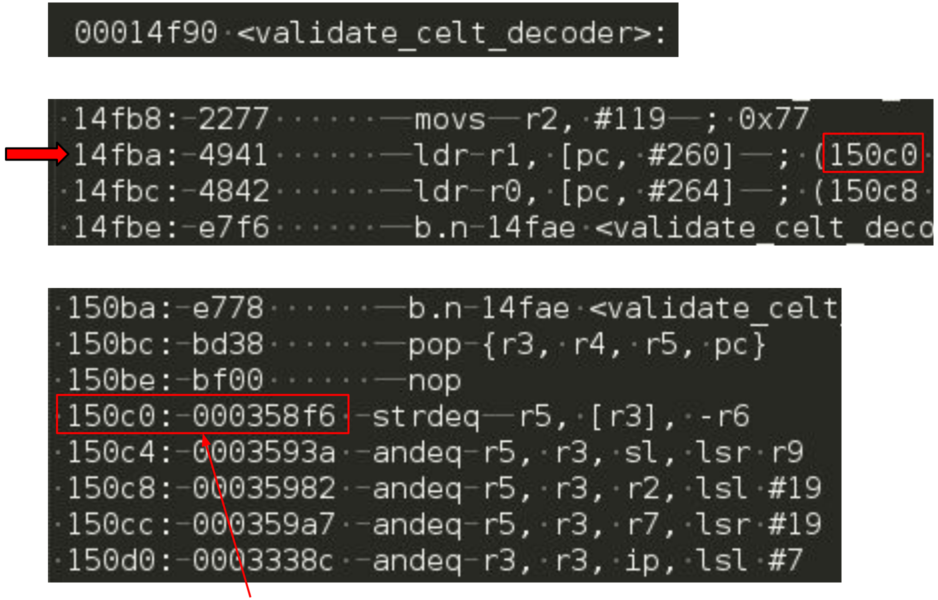
- >50 memory operations with **s0** as a **base register**
- **s0** is a **fixed value** loaded in the prologue of the function (sp + offset)
- This fixed value **imposes** a **negative offset** in the memory ops
- The negative offset **prevents** the **compression** of the **memory ops**
- Tried with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

OPUS analysis (<validate_celt_decoder>)



```
00014f90 <validate_celt_decoder>:
```

```
 14fb8: 2277        movs  r2, #119  ; 0x77
 14fba: 4941        ldr r1, [pc, #260]  ; (150c0
 14fbc: 4842        ldr r0, [pc, #264]  ; (150c8
 14fbe: e7f6        b.n 14fae <validate_celt_deco
```

```
 150ba: e778        b.n 14fae <validate_celt
 150bc: bd38        pop {r3, r4, r5, pc}
 150be: bf00        nop
 150c0: 000358f6  strdeq  r5, [r3], -r6
 150c4: 0003593a  andeq r5, r3, sl, lsr r9
 150c8: 00035982  andeq r5, r3, r2, lsl #19
 150cc: 000359a7  andeq r5, r3, r7, lsr #19
 150d0: 0003338c  andeq r3, r3, ip, lsl #7
```

This value is loaded 20 times

- Alternative to 48-bit l.li: **constant pool** after the function code
- Byte cost per mem op, on N mem ops:
  - *PC load:* 2 B + (4/N) B ≈ **2 B** if N is high
  - *l.li:* **6 B**
- Implementation cost: ***PC relative loads*** (almost for free) vs. ***48-bit support*** (mod. aligner + ID)
- Drawback: increased **D-memory traffic** (but high locality)

Further

- Go on with the **analysis** of **OPUS**
- Finish the **FP multiplication** (almost done)
- Start to code the **FMA** for HW comparison

Tiny Floating-Point Unit

Tiny FPU:

- Final **FMA** design
- Added **COMPARISON** instructions + optimization
- Added **CAST** instructions (to be optimized)

FMA HW optimizations - Single Precision

FP32	fpnew_fma (NON-REDUCED)	floatli_fma (no_input regs)	floatli_fma_input_regs	floatli_fma_input_regs_ reused
Overall Area	100%	51.2%	60.9%	58.2%
Comb Area	100%	38.8%	41.0%	44.2%
Non-comb	0%	12.4%	19.9%	14.0%
Latency	1 cycle (ADD/MUL/FMADD)	9-11 cycles (ADD) 21-23 cycles (FMADD/MUL)	9-11 cycles (ADD) 21-23 cycles (FMADD/MUL)	9-11 cycles (ADD) 21-23 cycles (FMADD/MUL)
Optimization	-	**~48.8%**	**~39.1%**	**~41.8%**

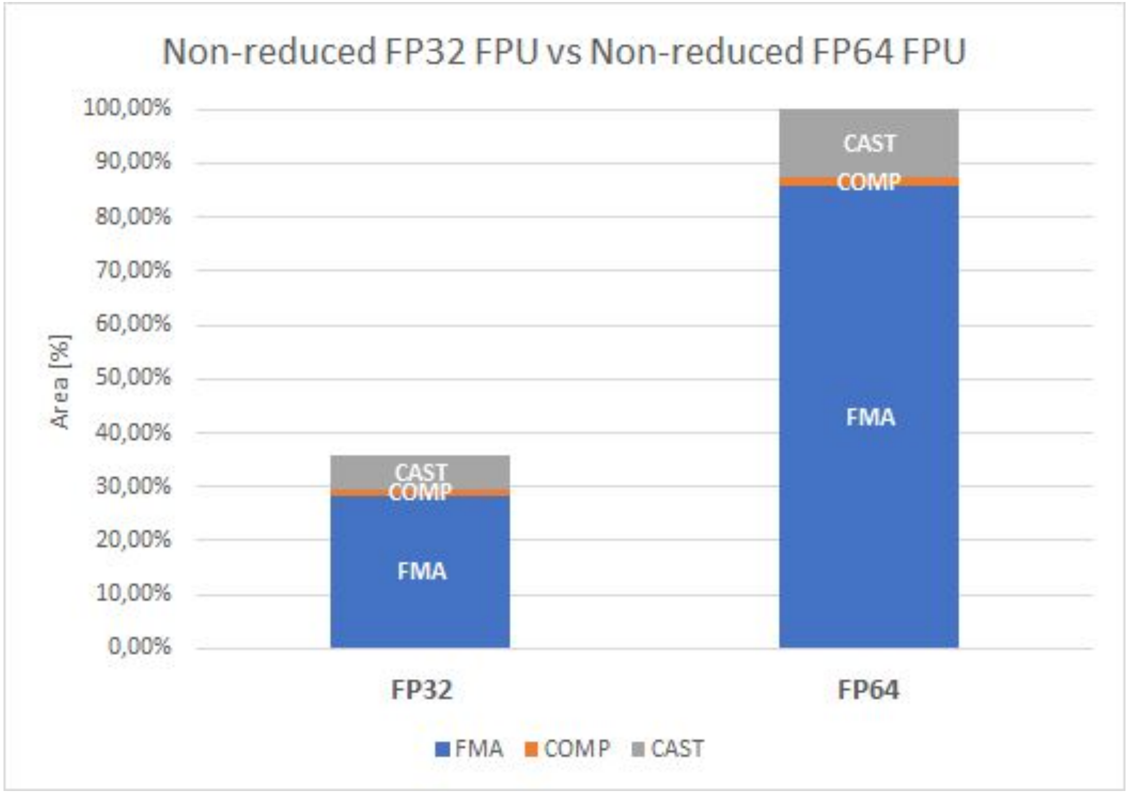
FMA HW optimizations - Double Precision

FP64	fpnew_fma (NON-REDUCED)	floatli_fma (no_input regs)	floatli_fma_input_regs	floatli_fma_input_regs_ reused
Overall Area	100%	33.6%	39.8%	37.9%
Comb Area	100%	25.4%	27.2%	29.5%
Non-comb	0%	8.2%	12.6%	8.4%
Latency	1 cycle (ADD/MUL/FMADD)	9-11 cycles (ADD) 35-37 cycles (FMADD/MUL)	9-11 cycles (ADD) 35-37 cycles (FMADD/MUL)	9-11 cycles (ADD) 35-37 cycles (FMADD/MUL)
Optimization	-	**~66.4%**	**~60.2%**	**~62.1%**

Non-reduced Floating-Point Unit modules

- **FMA** (FMADD, FNMSUB, ADD, SUB, MUL)
- **COMPARISON** (SGNJ, MINMAX, CMP, CLASSIFY)
- **CAST** :
  - Single-precision: FP32 ← → INT32
  - Double-precision:
    - FP64 ←→ INT64
    - FP64 ←→ INT32
    - FP32 ←→ INT64
    - FP32 ←→ INT32

Non-reduced Floating-Point Unit



FMA + COMP HW optimizations - Single Precision

FP32	fpnew_fma + fpnew_noncomp (NON-REDUCED)	reduced_fma_comp	reduced_fma_reduced _comp	reduced_fma_reduced _comp_idle
Overall Area	100%	59.7%	58.2%	53.7%
Latency	1 cycle (ADD/MUL/FMADD/ COMP)	1 cycle (COMP) 9-11 cycles (ADD) 21-23 cycles (FMADD/MUL)	1 cycle COMP 9-11 cycles (ADD) 21-23 cycles (FMADD/MUL)	2 cycle COMP 10-12 cycles (ADD) 22-24 cycles (FMADD/MUL)
Optimization	-	**~40.3%**	**~41.8%**	**~46.3%**

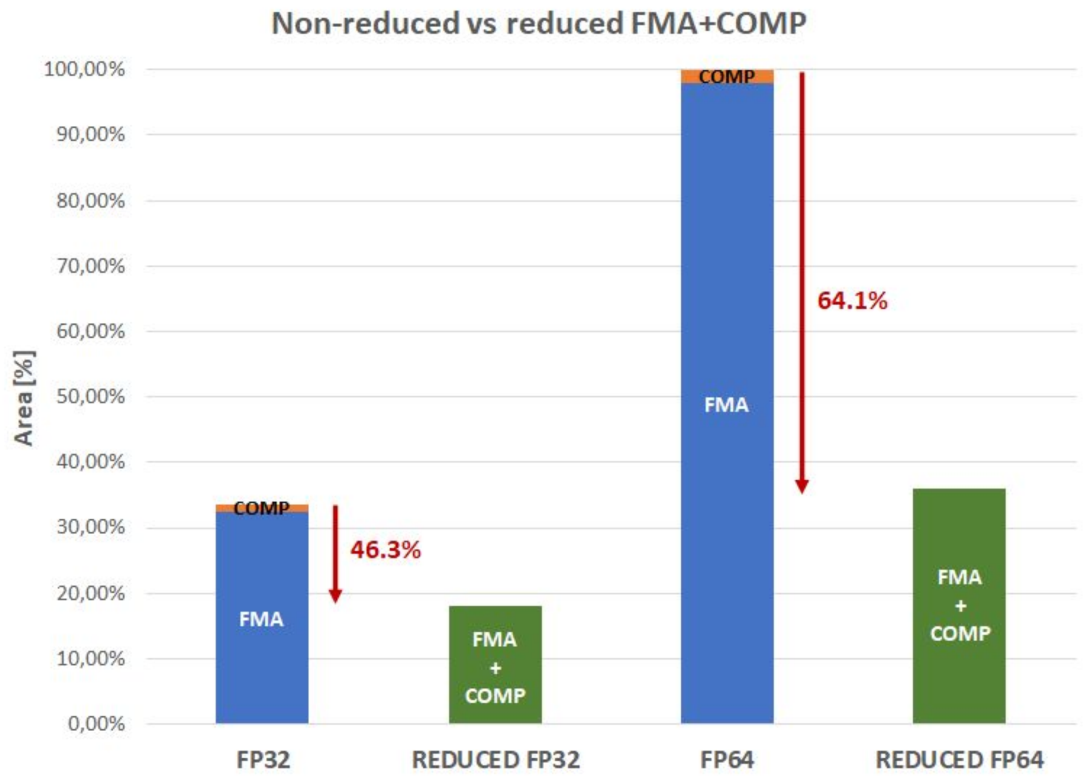
- No additional non-combinational area

FMA + COMP HW optimizations - Double Precision

FP64	fpnew_fma + fpnew_noncomp (NON-REDUCED)	reduced_fma_comp	reduced_fma_reduced _comp	reduced_fma_reduced _comp_idle
Overall Area	100%	39.2%	38.4%	35.9%
Latency	1 cycle (ADD/MUL/FMADD/ COMP)	1 cycle (COMP) 9-11 cycles (ADD) 35-37 cycles (FMADD/MUL)	1 cycle COMP 9-11 cycles (ADD) 35-37 cycles (FMADD/MUL)	2 cycle COMP 10-12 cycles (ADD) 36-38 cycles (FMADD/MUL)
Optimization	-	**~60.8%**	**~61.6%**	**~64.1%**

- No additional non-combinational area

FMA + COMP HW optimizations



FMA + COMP + CAST HW optimizations - Single Precision

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

FMA + COMP + CAST HW optimizations - Double Precision

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

Next steps

- Re-use of FMA components for multicycle CAST:
  - Shifter
  - Adder
  - Registers