Report 17.06.20

17/06/2020

Matteo Perotti

Luca Bertaccini

Pasquale Davide Schiavone

Stefan Mach

Professor Luca Benini

Integrated Systems Laboratory

ETH Zürich

Summary

- **Update** on "*long double*" problem
- Separate **Source code** and **Lib code**
- **Code size:** when **HCC can't solve** the **problem**
- **Tested** and debugged FP **__addsf3** and **__subsf3**

long double in Embench

- cubic (from Embench) uses the *long double* type
- ***long double*** is double-precision for **ARM** (**64-bit**), quad-precision for **RISC-V** (**128-bit**)
- GCC links **quad-precision FP support** functions only for **RISC-V** code (code size explosion)
- **Without** considering the **libraries**, **this discrepancy modifies** the **code size as well** because RISC-V handles "larger" values than ARM, and values larger than 2*XLEN are passed by reference to the functions (increased stack usage)

long double in Embench (issue on Embench GitHub page)

- It is a **known issue**
- **long double is** occasionally **used**, so **it's right to benchmark it**
- It is one of the two programs where RISC-V does the worst compared to ARM

  *"One consideration was that with the **originators** of **Embench all having** a **RISC-V background**, we were **reluctant** to **drop** the **worst benchmark** for **RISC-V**, since it could seem to **undermine** the **independence** of the **benchmark**."*
- Embench programs are renewed every 2 years -> opportunity to substitute them

For our size analysis, we can drop *cubic* because we have understood the main related issue

Library code - the problem

- The program is composed of **source code** and **libraries**

- **Libraries skew** the **code size inflation** in an **uncontrolled** way:
  - Different generality (e.g. startup code, printf functions)
  - Different optimizations (FP libraries)
  - Different functionality (two functions can behave differently, e.g. denormals flushed to zero or not)
  - Different inter-dependencies (one binary can have different/more functions than the other)

- We know that an **optimized library** support **makes the difference**

Dividi et impera

- **Separate** the **two problems** and analyze them separately
- Separate the **library code** from the **compiled source code**
- **Analyze** code size issues related to ISA differences only from the **source code**
- Knowing that **library support can be tailored** and **optimized** for the specific needs

When can HCC solve a size problem?

No initial problem
No big difference

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

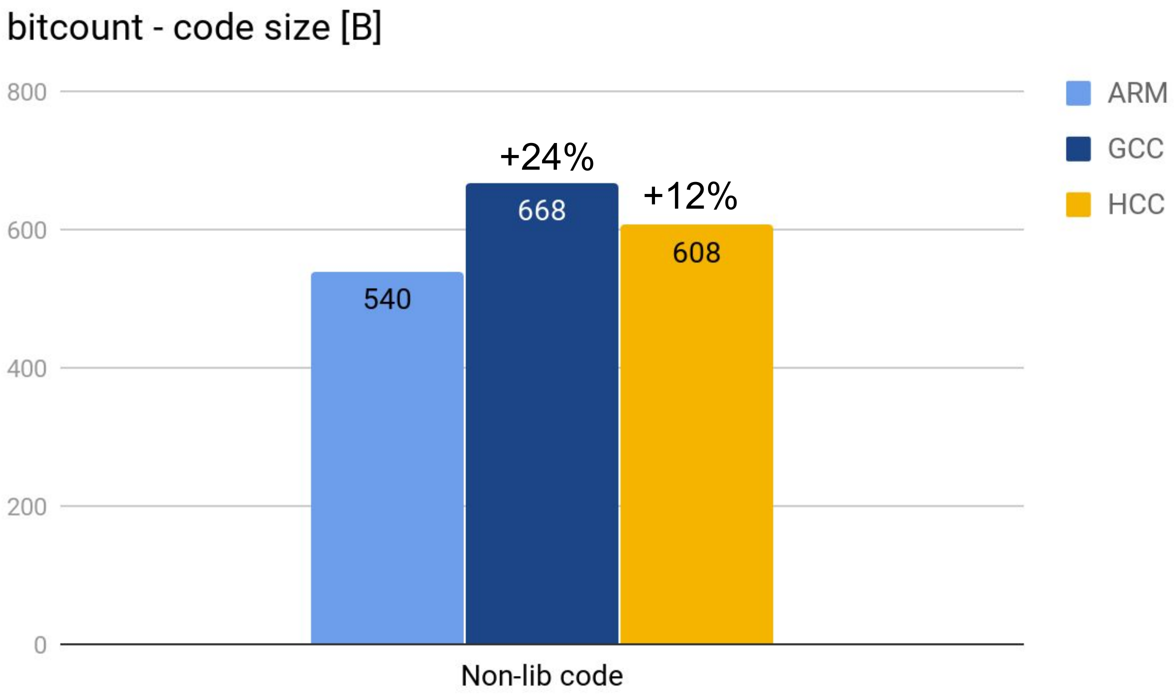
HCC makes the difference

- 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

bitcount


bitcount - code size [B]
800
600
400
200
0
+24%
+12%
540
668
608
Non-lib code
ARM
GCC
HCC

bitcount

```
000082b0 <AR_btbl_bitcount>:
    82b0: 4b07          ldr r3, [pc, #28] ; (82d0 <AR_btbl_bitcount+0x20>)
    82b2: b2c2          uxtb r2, r0
    82b4: 5c99          ldrb r1, [r3, r2]
    82b6: f3c0 2207     ubfx r2, r0, #8, #8
    82ba: 5c9a          ldrb r2, [r3, r2]
    82bc: 4411          add r1, r2
    82be: f3c0 4207     ubfx r2, r0, #16, #8
    82c2: 5c9a          ldrb r2, [r3, r2]
    82c4: 0e00          lsrs r0, r0, #24
    82c6: 5c18          ldrb r0, [r3, r0]
    82c8: 440a          add r2, r1
    82ca: 4410          add r0, r2
    82cc: 4770          bx lr
    82ce: bf00          nop
    82d0: 00009ebc      ; <UNDEFINED> instruction: 0x00009ebc
```

ARM

(3 + 4 + 6 + 8 + 4 + 4 + 2 + 2) B

```
00010384 <AR_btbl_bitcount>:
   10384: 0001 2050 079f     l.li    a5,0x12050
   1038a: 0ff57713           andi    a4,a0,255
   1038e: 973e               add     a4,a4,a5
   10390: 2314               lbu     a3,0(a4)
   10392: 00855713           srli    a4,a0,0x8
   10396: 9f01               uxtb    a4
   10398: 973e               add     a4,a4,a5
   1039a: 2318               lbu     a4,0(a4)
   1039c: 96ba               add     a3,a3,a4
   1039e: 01055713           srli    a4,a0,0x10
   103a2: 9f01               uxtb    a4
   103a4: 973e               add     a4,a4,a5
   103a6: 2318               lbu     a4,0(a4)
   103a8: 70a7879b           addshf  a5,a5,a0,srl,24
   103ac: 2388               lbu     a0,0(a5)
   103ae: 9736               add     a4,a4,a3
   103b0: 953a               add     a0,a0,a4
   103b2: 8082               ret
```

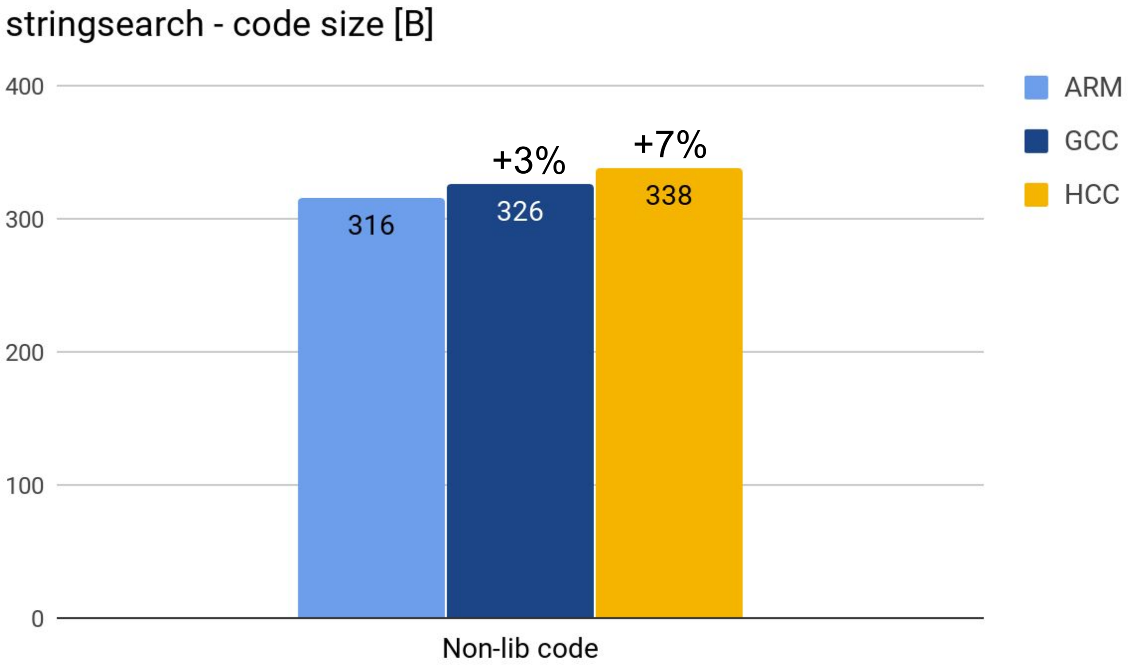
RISC-V HCC

(3 + 8 + 10 + 12 + 6 + 4 + 2) B

ARM has **16-bit memory operations** with **address** produced adding **two register values**
ARM has **ubfx, HCC implements it using 2 Bytes more**

stringsearch

- Very **small program**
- Interesting because **HCC worsens** the **code size**
- This was compiled with **all HCC instructions**, **except** from **l.li**
- **l.li** additionally **increases** the code size



stringsearch

- There are **3 non-lib functions**. The code size worsens only in <***main***>
- It **seems** that **HCC cannot compress** some **lui** instructions
- Different choices of registers, higher use of the stack (**the compilers are different**)
- The **difference** in **size** is **small**, anyway

```
69d5                lui   s3,0x15
6a49                lui   s4,0x12
```

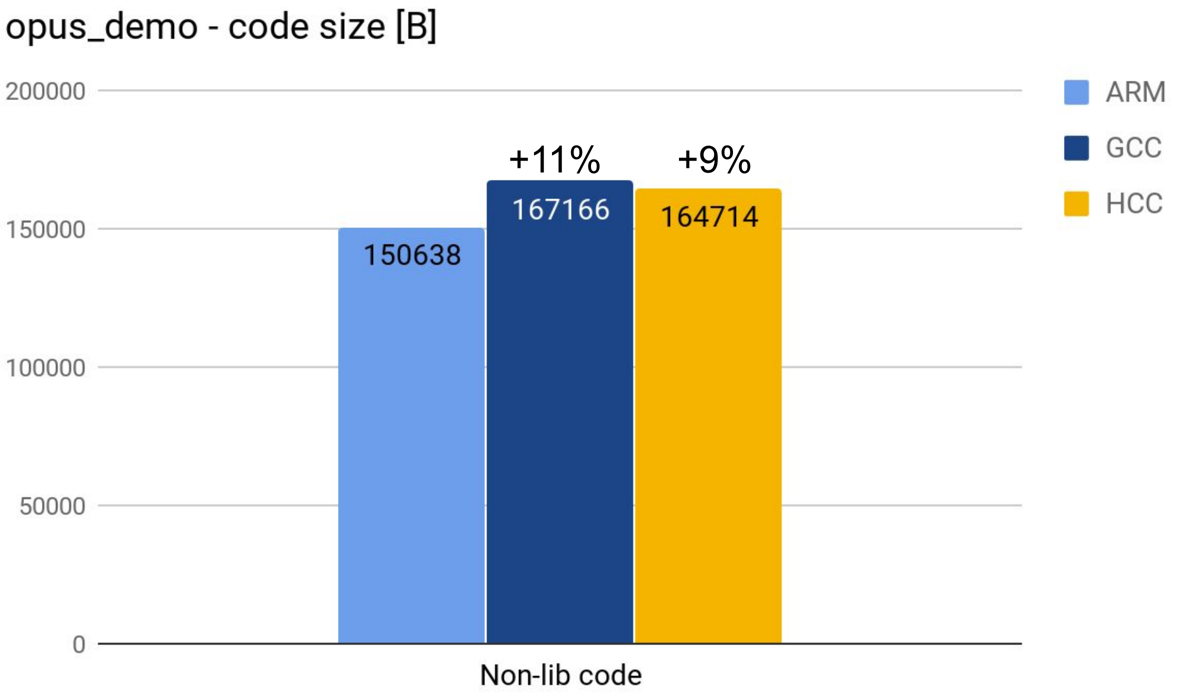
part of <*main*> from *stringsearch*,
GCC

```
          65c5                lui   a1,0x11
1010e:    00014ab7            lui   s5,0x14
10112:    00014b37            lui   s6,0x14
```

part of <*main*> from *stringsearch*,
HCC

opus

- **Approximate results** (hard to separate all the library functions)
- The **size difference** is related to **non-lib functions**
- The problem seems **extended** to many functions



Tested __addsf3 and __subsf3

- **Tested** and **debugged**

- __addsf3 (414 B), __subsf3(8 B)

```
Testing f32_add, rounding near_even.
7496193 tests performed.
In 7496193 tests, no errors found in f32_add, rounding near_even.
```

```
Testing f32_sub, rounding near_even.
7496193 tests performed.
In 7496193 tests, no errors found in f32_sub, rounding near_even.
```

Further

- Single precision FP **multiplication**
- **Analyze** the code **uncompressed** by **HCC**

Tiny Floating-Point Unit

Tiny FPU:

- **FMA** optimization
- **Reuse** of input registers

FMA HW optimizations - Single Precision

FP32	fpnew_fma	tiny_fma	tiny_fma (input regs)	tiny_fma (re-used input regs)
Overall Area	100%	51.2%	60.9%	59.8%
Comb Area	100%	38.8%	41.0%	42.1%
Non-comb	0%	12.4%	19.9%	17.8%
Latency	1 cycle (ADD/MUL/FMADD)	9 cycles (ADD) (11 cycles when -sum) 21 cycles (FMADD/MUL) (23 cycles when -sum)	9 cycles (ADD) (11 cycles when -sum) 21 cycles (FMADD/MUL) (23 cycles when -sum)	9 cycles (ADD) (11 cycles when -sum) 21 cycles (FMADD/MUL) (23 cycles when -sum)
Optimization	-	**~48.8%**	**~39.1%**	**~40.2%**

FMA HW optimizations - Double Precision

FP64	fpnew_fma	tiny_fma	tiny_fma (input regs)	tiny_fma (re-used input regs)
Overall Area	100%	33.6%	39.8%	39.3%
Comb Area	100%	25.4%	27.2%	28.2%
Non-comb	0%	8.2%	12.6%	11.1%
Latency	1 cycle (ADD/MUL/FMADD)	9 cycles (ADD) (11 cycles when -sum) 35 cycles (FMADD/MUL) (37 cycles when -sum)	9 cycles (ADD) (11 cycles when -sum) 35 cycles (FMADD/MUL) (37 cycles when -sum)	9 cycles (ADD) (11 cycles when -sum) 35 cycles (FMADD/MUL) (37 cycles when -sum)
Optimization	-	**~66.4%**	**~60.2%**	**~60.7%**

Hardware optimizations

Next steps:

- Complete **re-use** of input **registers**
- FSM-based reuse of the int datapath + potential extensions