HW/SW approaches for RISC-V code size reduction

CARRV Workshop
30 May 2020

Matteo Perotti
Pasquale Davide Schiavone
Giuseppe Tagliavini
Davide Rossi
Tariq Kurd
Mark Hill
Liu Yingying
Luca Benini

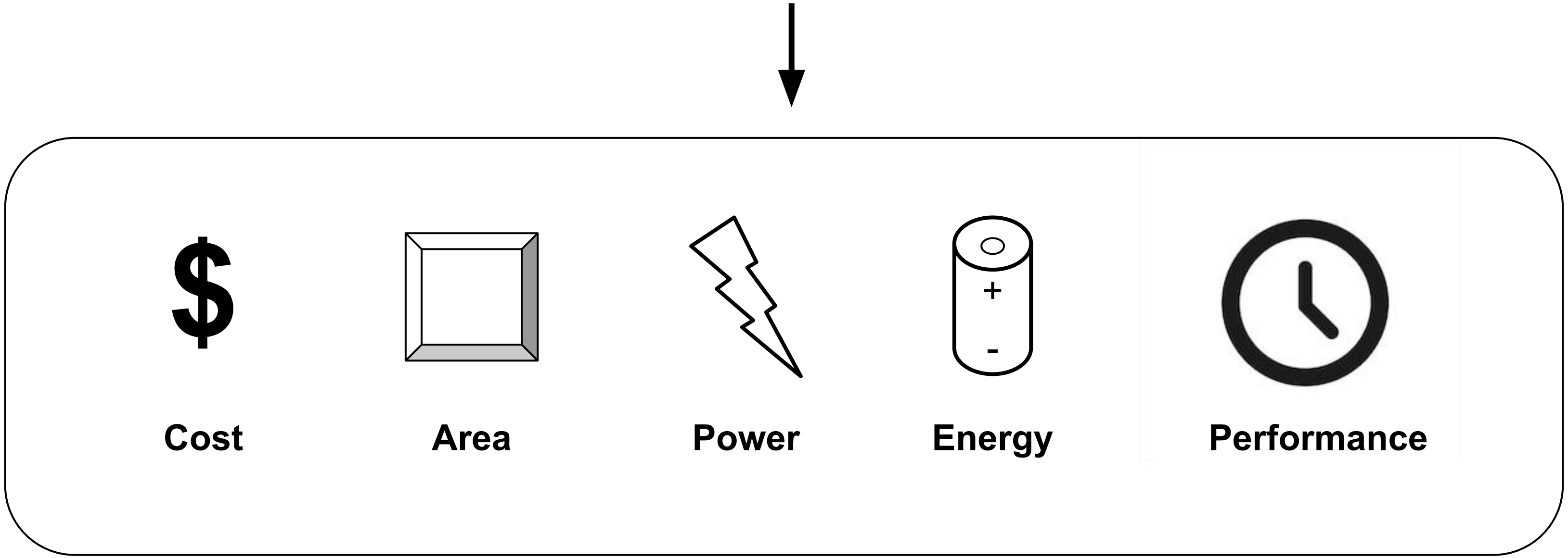
ETH zürich

HUAWEI

ETH zürich

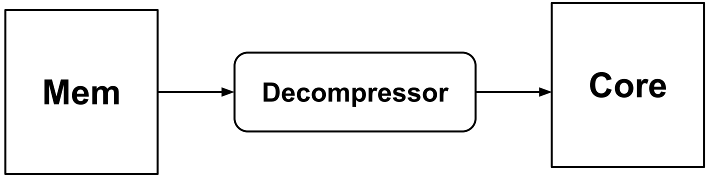
Code size in embedded systems

Code Size impacts



How to reduce the code size?

Compaction techniques



Compression techniques

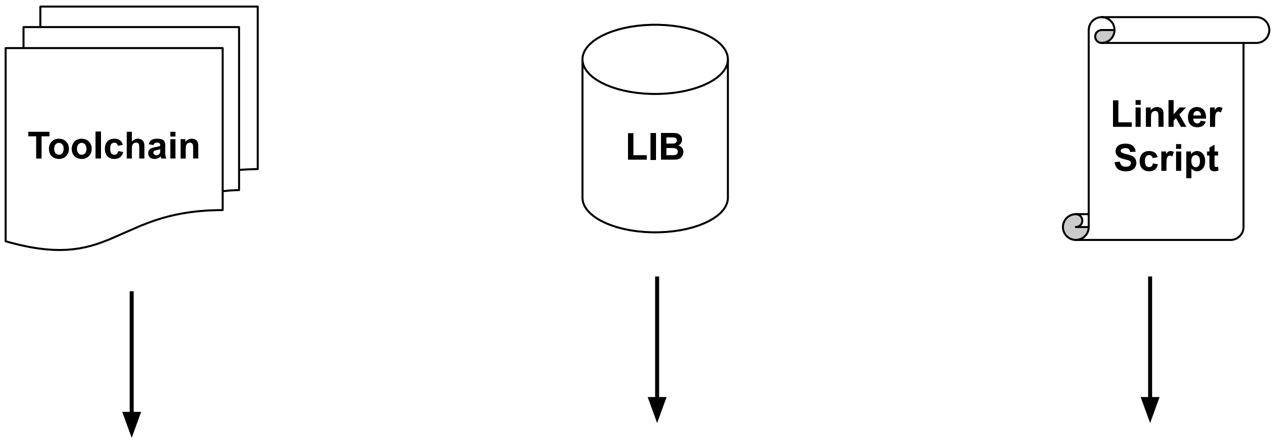
ARM Thumb2
RISC-V RVC

Modify/Extend the **ISA**

Contributions of this work

- **Evaluate** RISC-V **RVC** code size on Benchmarks
- **Effect** of **Xpulp extension** on code size
- **HCC**: **new** RISC-V **extension** for **code size** reduction
- **Implement** 16-bit ***push***/***pop***/***popret*** on the open RISC-V core **CV32E40P**
  - **CV32E40P implements RV32IM[F]CXpulp**
- **Area** and **performance costs** estimation

Tuning the toolchain environment



- **Updated toolchains** can produce smaller code
- **Generic** and **ISA-specific options** for code size

- Compiled with the **same ISA** used for the code
- **Optimized** for size (Newlib-nano, picolibc,...)

- **Collect** all the **frequently accessed** data **sections** (sbss, sdata) **near** the **Global Pointer**

Experimental setup: compilers

ISA:
ARM Thumb

Compiler:
arm-none-eabi-gcc (7.2)

Compiler flags:

- -march=armv7-m
- -mcpu=cortex-m3
- -mthumb
- -Os
- -ffunction-sections
- -fdata-sections
- -gc-section

ISA:
RISC-V RVC

Compiler:
riscv32-unknown-elf-gcc (7.X)

Compiler flags:

- -march=rv32imc
- -Os
- -msave-restore
- -ffunction-sections
- -fdata-sections
- -gc-section

Experimental setup: programs

Program:
IoT

ARM size:
~200 KiB

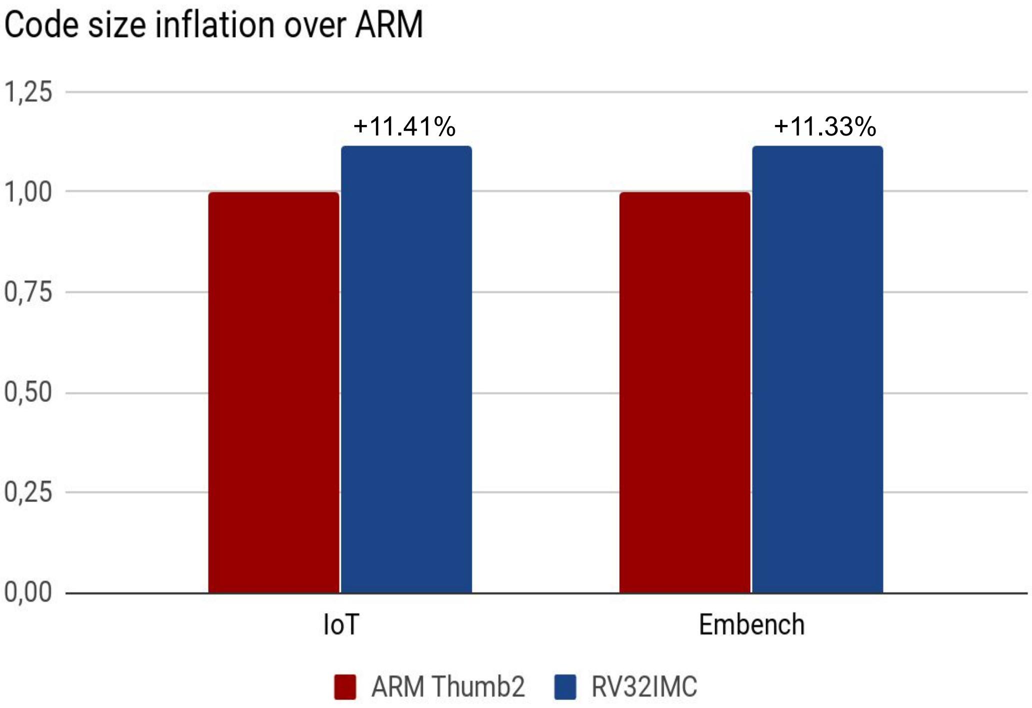
Note:
- Industry program

Benchmark suite:
Embench 0.5

ARM size:
from ~200 to ~1500 B

Note:
- Set of 19 programs
- Dummy libraries
- Geometric mean

RISC-V RVC comparison with ARM Thumb-2



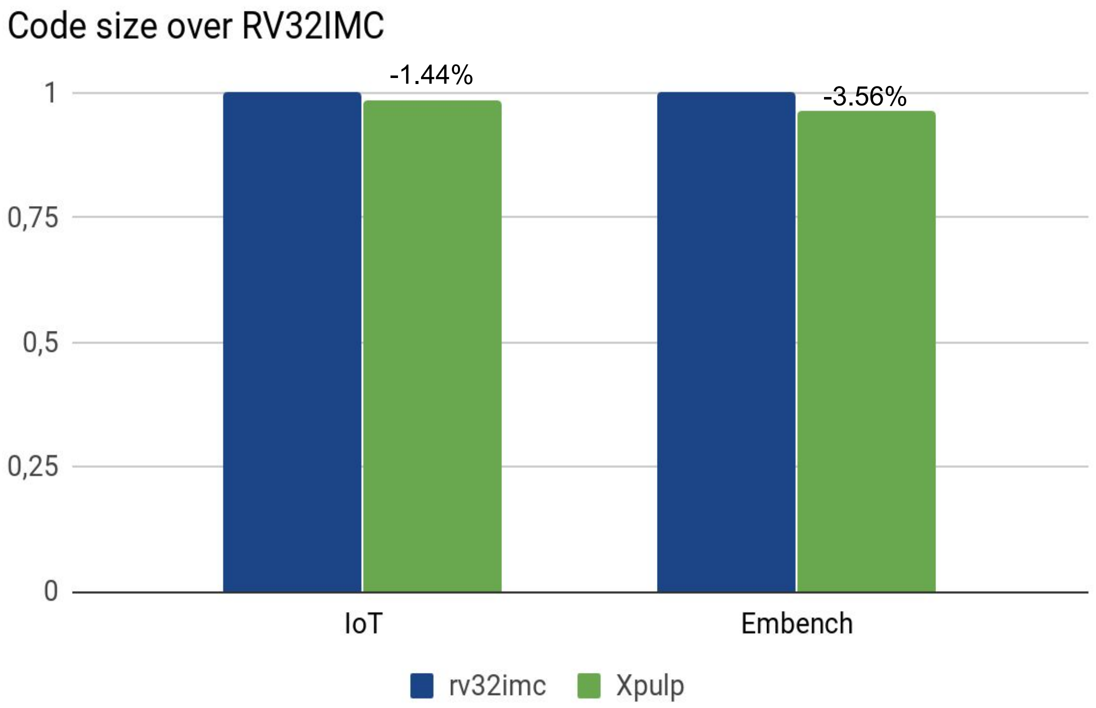
RISC-V code >11% than ARM!

Xpulp extension - Designed for performance

Xpulp new instructions:

- Branch immediate
- MAC
- Additional ALU operations
- Bit manipulation
- SIMD operations
- HW loops
- Post increment mem ops
- Immediate offset mem ops

Performance, efficiency boost
+
lower code size!



Xpulp - Originally designed for performance

Xpulp RISC-V extension can:

- **boost** DSP applications **performance** up to **10x**
- **increase** their **efficiency**
- **improve** the **code size**

Xpulp + RVC coexistence can be even improved!

↓

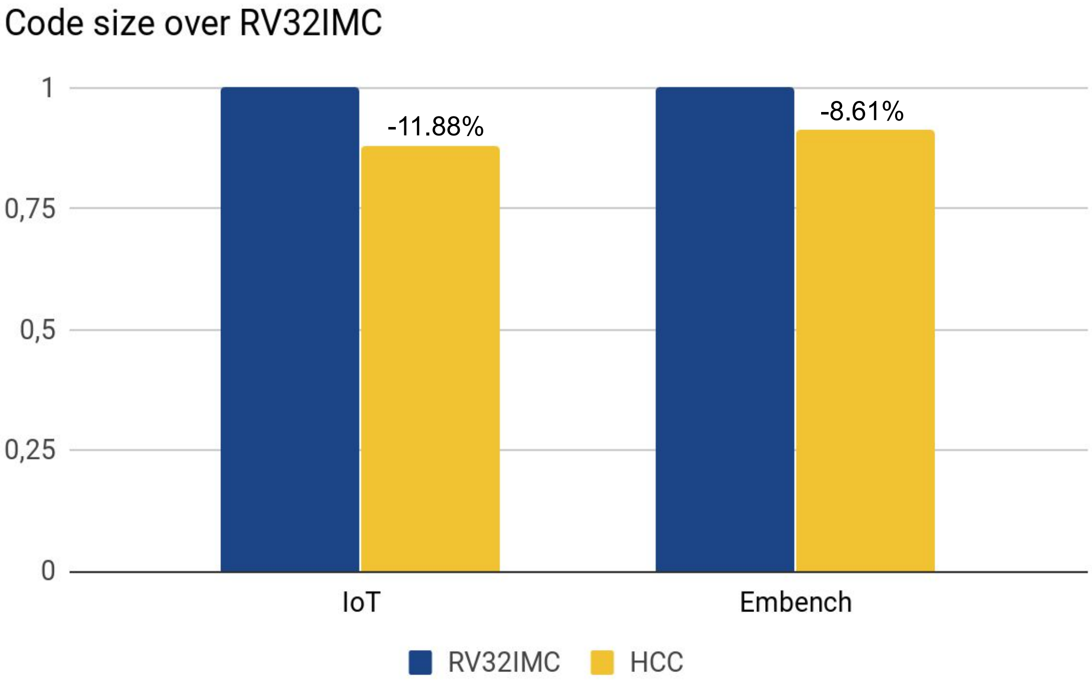
PULP toolchain sub-optimal behaviour:

- Xpulp instructions steal RVC registers to the surrounding other instructions
- 32-bit Xpulp instructions replace equivalent 16-bit RVC instructions

HCC extension - Designed for low Code Size

HCC new instructions:

- 16-bit **push/pop/popret**
- 16-bit **lbu/sb/lhu/sh**
- **48-bit load-immediate**
- **Branch** against **immediate**
- **Muliadd** (32-bit)
- En**jal16** (32-bit)
- **Immediate shift**
- Unsigned **extend byte/halfword**
- **Load/store multiple**



HCC - Single instructions

- **HCC reference** size
  - **IoT**: 233804 B
  - **Embench**: 1975 B
- Used **either *save*/*restore*** routines **or push/pop/popret**
- **48-bit load-immediate** increases the code size due to **compiler issues**
- **JAL16 not used** here. Useful with scattered code or large programs

Code size reduction	IoT	Embench
HCC reference	0%	0%
push/pop/popret	-5.11%	-3.70%
lbu/sb	-1.49%	-0.76%
lhu/sh	-0.74%	-0.20%
48-bit load-Immediate	-0.53%	0.66%
Branch Immediate	-1.35%	-0.41%
JAL16	0.00%	0.00%
MuliAdd	-0.17%	0.00%
Imm. Shift	-0.35%	-3.04%
uxtb/uxth	-0.40%	-0.25%
ldm/stm	0.00%	-0.56%
HCC extension	-11.88%	-8.61%

HCC - Single instructions

- **Some instructions** are highly **code dependant** (lbu/sb/lhu/sh, immshf, brimm)
- It's **common** for functions to **manipulate** the **stack**. 16-bit **push/pop/popret easily** return important **size reductions**
- They lead also to **performance improvements** (less jumps, improved locality)

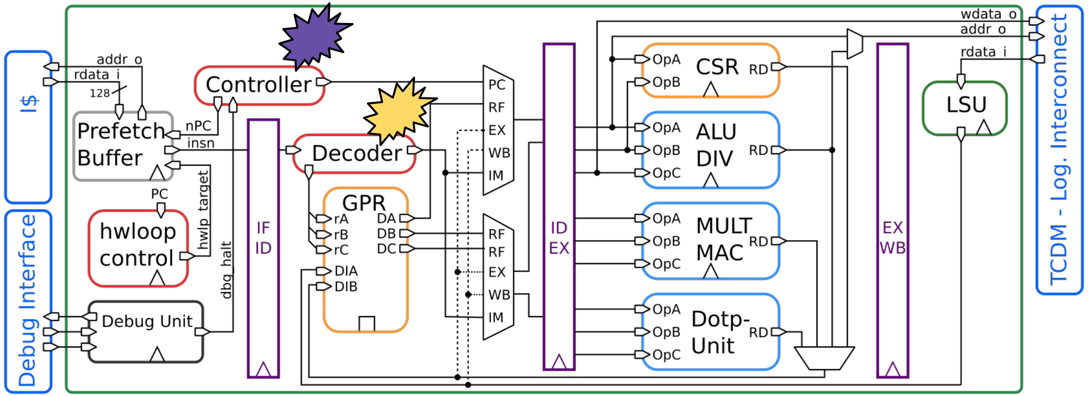
Code size reduction	IoT	Embench
HCC reference	0%	0%
push/pop/popret	-5.11%	-3.70%
lbu/sb	-1.49%	-0.76%
lhu/sh	-0.74%	-0.20%
48-bit load-Immediate	-0.53%	0.66%
Branch Immediate	-1.35%	-0.41%
JAL16	0.00%	0.00%
MuliAdd	-0.17%	0.00%
Imm. Shift	-0.35%	-3.04%
uxtb/uxth	-0.40%	-0.25%
ldm/stm	0.00%	-0.56%
HCC extension	-11.88%	-8.61%

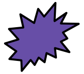
16-bit push/pop/popret

15	14	13	12 … 8	7 … 4	3 … 2	1	0
1	0	0	sp16imm	rcount	opc	0	0
1	1	1	5	4	2	1	1

- **Opcode**: Reserved but **free** space in **RVC**
- **opc**: **type** of **operation** among ***push***, ***pop***, ***popret***
- **rcount**: **register sequence** to push/pop
- **sp16imm**: **additional stack space** for automatic variables

16-bit push/pop/popret: implementation on CV32E40P





Added **FSM** to the controller to inject **memory operations,** an **addition** and possibly a **jump**

Modified decoder to recognize push/pop/popret

Costs:

- Area: +2.5%
- Frequency: not modified

Benefit on rv32imc:

- Code size: from -3.7% to -5%

Code size inflation over ARM

Code size inflation over ARM	**IoT**	**Embench**
ARM Thumb2	0%	0%
RV32IMC	11.41%	11.33%
RV32IMC Xpulp	9.81%	7.36%
RV32IMC HCC	-1.75%	2.21%
RV32IMC push/pop/popret	5.80%	7.70%

Conclusion

Code size inflation over ARM	**IoT**	**Embench**
ARM Thumb2	0%	0%
RV32IMC	11.41%	11.33%
RV32IMC Xpulp push/pop/popret	5.37%	5.32%

PULP
+
16-bit push/pop/popret

→

- **Code size <5.4%**
- Same operating frequency
- Benefits of Xpulp (up to 10x performance)

- Area >2.5%

The End

Thanks for the attention!