# HW/SW approaches for RISC-V code size reduction

Matteo Perotti
ETH Zurich
Zurich, Switzerland
mperotti@iis.ee.ethz.ch

Pasquale Davide Schiavone
ETH Zurich
Zurich, Switzerland
pschiavo@iis.ee.ethz.ch

Giuseppe Tagliavini
University of Bologna
Bologna, Italy
giuseppe.tagliavini@unibo.it

Davide Rossi
University of Bologna
Bologna, Italy
davide.rossi@unibo.it

Tariq Kurd
Huawei
Bristol, United Kingdom
tariq.kurd@huawei.com

Mark Hill
Huawei
Bristol, United Kingdom
mark.hill@huawei.com

Liu Yingying
Huawei
Hangzhou, China
liuyingying19@huawei.com

Luca Benini
ETH Zurich and University of Bologna
Zurich, Switzerland
lbenini@iis.ee.ethz.ch

## ABSTRACT

Code density is a significant concern for low-cost IoT MCUs, as it directly impacts the on-chip memory area (and cost) and indirectly influences power and performance. Even though RISC-V features a compressed ISA, the size of RISC-V binaries is often larger, on several benchmarks, than that of the incumbent MCU ISA. In this paper, we provide a deeper insight into how the ISA design, together with the SW environment, influences RISC-V code density, and we explore several approaches to improve it. First, we explain how to tune the toolchain at compile- and link-time with optimal settings, including those needed for including libraries and using the linker script. Second, we demonstrate that RISC-V non-standard extensions, such as the Xpulp extension, can boost performance without any code size penalty. Finally, we propose a new RISC-V ISA extension that explicitly targets improved code density, with a special focus on the push/pop instructions, needed to handle multiple stack memory operations. The extension effectively reduces the code density gap, both on the Embench open benchmark (from 11.8% to 7.7%) and also on 3GPP industry-strength code aimed at enabling low power, low data rate machine-to-machine communication (from 11.5% to 5.8%). Finally, we provide an implementation of the ISA extension on the open CV32E40P core to evaluate the impact on the core area and operating frequency. Results show a minimal increase of 2.5% of the core area and no impact on the maximum frequency.

## KEYWORDS

Code size density, RISC-V, ARM, Push Pop

## 1 INTRODUCTION

The market for embedded systems is thriving today thanks to the pervasive adoption of Internet of Things (IoT) devices. Compared to other domains, the design and deployment of solutions based on embedded systems must take into account tight resource, power, and energy constraints. The memory becomes a fundamental unit to use with care as it is usually one of the most expensive parts of chips [15], and it biases the performance, power, and area (PPA) results [3, 10].

*CARRV 2020, May 2020, Virtual Workshop*
.

In software-programmable edge-computing devices, memories are designed to be just big enough to host data and code with some margin size. Thus, the code density of a program becomes an important knob to tune PPA results. While providing shorter access time, a smaller memory also consumes less dynamic and static power [8] and subsumes the reduction of the code size of the stored applications. In addition, a denser code increases the energy efficiency thanks to fewer memory accesses [12].

One strategy to achieve a higher code density is to exploit variable-length Instruction Set Architectures (ISAs), which assign smaller encodings to the most common instructions. Complex Instruction Set Computers (CISCs) like Intel x86 are intrinsically variable-length. On the other hand, Reduced Instruction Set Computers (RISCs) provide this feature as an option. For example, the ARM and RISC-V ISAs offer Thumb2 and RVC extensions respectively to encode some common 32-bit or 64-bit instructions in only 16 bits, thus reducing size [1] [14]. In [8], an 8-bit compressed instruction extension has been proposed showing 30% of code size reduction, without lowering performance at the cost of <5% of core area increase. Another strategy is to leverage the optimizations provided by the compilation toolchain, which can provide specific knobs for different ISA targets.

In this paper, we provide: a) an analysis of the main compiler and linker options to reduce code size; b) a code size comparison of the two ARM and RISC-V ISAs; c) we evaluate the impact on code size of the custom RISC-V Xpulp extension that has been presented in [6] and originally designed for pushing energy efficiency; d) we propose a new RISC-V extension that targets an increased code size density as a possible solution to decrease the density gap between the two ISAs; e) and finally we evaluate the impact on core logic area of the HW implementation of compressed push/pop/popret on an open-source core.

## 2 RELATED WORK

The interest in reducing the code size has remained high over time to reduce flash-related operations, memory accesses, and memory occupation. In 2003, Árpád Beszédes et al. wrote a survey on 12 representative methods to compress the code size [2] using Huffman, arithmetic and dictionary-based coding, pointing out the difficulty of a fair comparison among them. Compression methods are not only characterized by their ability to compress the code size, but also by their implementation complexity, application domain, etc. For this reason, they should be carefully evaluated depending on the custom needs.

Optimization techniques in the compiler domain also play an important role to minimize the final binary size as shown in [5].

Another way to achieve lower code sizes is to modify the ISA itself. For example, to lower the memory footprint, ARM introduced 16-bit Thumb ISA, and then Thumb2, a variable-length ISA with both 16- and 32-bit instructions with improvements on both code size and performance [11]. As noted in [13], Thumb and Thumb2 need a complex decoder aware of three different ISAs. The RISC-V ISA instead uses the same opcode space as the base ISA to incorporate the official RVC extension for low code sizes. In designing the RVC RISC-V extension, the exclusion of *load-/store- multiple* instructions was the hardest decision the designers took [12]. They excluded them mainly to not complicate the compiler/processor design and because there were no RISC-V 32-bit corresponding instructions [14].

In 2019, Peijie Li analyzed the RVC extension and proposed some modifications to further reduce code footprint, improving the compression ratio of non floating-point intensive programs of 5% without hurting execution time. The idea is to drop some floating-point compressed instructions, recycling their encodings to compress other more common uncompressed instructions. In the same year, David Patterson introduced the Embench Benchmark suite and reported results showing that ARM Thumb2 still generates smaller code than RV32IMC on average [4]. To the best of our knowledge, there are no other proposals to further reduce RISC-V program code size extending the ISA.

In this work, we propose a new RISC-V ISA extension to reduce the code size, we evaluate it on a set of benchmarks, and finally, we implement the instructions that impact the most the code size (compressed push/pop/popret) on an open-source RISC-V 32-bit core to evaluate the impact in terms of core area and frequency.

## 3 BACKGROUND

In this work, we refer to the *code size* of a program as the size (in bytes) of the code part of a binary. Where we refer to *code density*, we refer to the average fraction of the program completed in a size unit, i.e. the reciprocal of the code size. In the next subsections, we review the baseline ISA and the compilation and linking process with a focus on code density.

### 3.1 ISA

ARM and RISC-V ISAs are two popular choices in the Embedded domain. Throughout this paper, we use the RISC-V ISA with HW multiplier support and Compressed instructions for RISC-V (RV32IMC), and ARM Thumb2 ISA for a Cortex-M3 target. These are realistic configurations for simple embedded processors. In addition to the RVC ISA extension, we evaluate the RISC-V Xpulp extension presented in [6]. The goal of such an extension is to boost the energy efficiency of data processing applications in the context of edge-computing. Such extension includes: load and store instructions with an automatic increment of the memory pointer, which allows saving further explicit addition instructions; zero-overhead hardware-loops, which remove the branch penalties and the extra instructions to control the loops; bit manipulation instructions, that fuse in single instructions clear, set and extract operations on sets of bits; fixed-point arithmetic support to round and normalize in a single instruction fixed-point operations like additions and multiplications; and packed-SIMD extensions, which enables to process multiple data with single instructions, as for example vectorial additions, comparisons, or dot products. These instructions have been designed mainly to increase performance at the price of a small increase in the core area and power consumption for a major gain in energy efficiency (up to 10x [6]).

### 3.2 Tuning Compiler and Linker

We summarize below the main compiler and linker options, the needed libraries, and the *linker script* settings to obtain a binary with low code size.

#### 3.2.1 *Toolchain options.*

- *Compile for size* (*-Os*): to enable optimizations to reduce the code size of the program.
- *SW stack-manipulation routines* (*-msave-restore*): to introduce special prologue and epilogue SW routines to handle stack operations and avoid the repetition of stack-memory instructions at the beginning and at the end of the functions. This option is RISC-V specific.
- *Garbage collection* (*-gc-sections*): to eliminate the unreferenced functions and data from the final code to further reduce its size. To achieve this, pass *-ffunction-sections* and *-fdata-sections* to the compiler and *-gc-sections* to the linker.
- *Link Time Optimization* (*-flto*): to allow optimizations among different translation units. Without this option, the compiler acts on each source file without assuming anything about the external referenced code. Thus, compiler optimizations are performed only within the single translation unit. *-flto* is passed to the compiler to save additional information in the intermediate binary files to allow a more comprehensive optimization at link time and then is also passed to the linker to enable the optimization itself. When compiling for code size with LTO, enable *-Os* also at link-time.

3.2.2 *Software libraries.* Embedded applications make use of specific software libraries targeted at embedded use-cases. These include Newlib and the even simpler and smaller Newlib-nano which further reduces the memory footprint.

3.2.3 *Linker Script.* The linker creates the memory image of a program under the directives of a configuration file called *linker script.* Each input file to the linker is a binary code composed of different sections. The code is usually kept together in the *.text* section, the read-only data in the *.rodata* section, the initialized data in *.data*, etc.

Some ISAs, like RISC-V and MIPS, instantiate a special *Global Pointer* (GP) register containing a fixed memory address used as a base address for memory operations. This allows accessing addresses within ±2 KiB from the GP with a single instruction, as RISC-V *load*/*store* immediates are 12-bit long. The relative position of the GP with respect to the data sections influences both code size and performances, thus designers should place the most frequently referenced data sections (like .sbss and .sdata) within a ±2

KiB range from the GP to maximize the code size density when writing a custom linker script.

## 4 THE IMPACT OF ISA EXTENSIONS

To assess the impact of ISA extensions, we first built an industry-standard program and a set of benchmarks using the RISC-V and ARM toolchains for a fair comparison. Then, we evaluated both the Xpulp RISC-V extension, developed for boosting the performance of DSP processing code and our new RISC-V extension designed to decrease the code size. For this purpose, we built the same program and benchmarks for both the extensions, enabling one set of HW instructions at a time and comparing it with the reference code with no added instructions, to evaluate their impact on the code size. We provided also a code density comparison with ARM Thumb2.

## 4.1 Experiment Setup

*4.1.1 Toolchains.* For the analysis, we used the ARM, PULP and HCC toolchain to build the source code, then we measured the byte size of the code-related sections (*.text* and related, if present). Since both PULP and HCC toolchains are based on GCC 7 [9], we installed the ARM compiler based on the same GCC version.

- ARM Toolchain (GCC 7.2): used to build ARM-Thumb2 binaries, targeting *Cortex-M3* and *armv7-m*.
- PULP Toolchain (GCC 7.1.1): modified GCC compiler supporting the Xpulp extension. used to build pure RISC-V and Xpulp extended binaries.
- HCC Toolchain (GCC 7.3): modified GCC compiler with our new RISC-V extension. Used to build HCC binaries.

To collect data relative to the base RISC-V compiler, we used the PULP compiler without enabling the support for the Xpulp extension; with this configuration, it is equivalent to a standard RISC-V compiler.

*HCC extension.* The compiler provides a different command-line option for every single set of new assembly instructions, e.g. *-mpush-pop* enables the support for push/pop/popret instructions. Passing all the options related to the new instructions enables the whole HCC RISC-V extension.

*Xpulp extension.* The PULP toolchain provides command-line options to disable part of the Xpulp instructions, e.g. *-mnohwloop* disables the support for HW-loops instructions. To assess the code size impact of each PULP instruction, we use as reference the RV32IMC code with only the Xpulp generic instructions like *branch against immediate* or *average* enabled (PULP reference).

*Libraries.* To avoid recompiling the libraries for each set of instructions, we used RV32IMC libraries when evaluating the impact of both PULP and HCC single instructions. However, we linked fully coherent libraries to analyze the whole Xpulp and HCC extensions, i.e. when we compiled the code using the full Xpulp or HCC extensions, the libraries included all the Xpulp or HCC instructions. Since Embench programs were linked with dummy libraries, this choice impacted only on the IoT code results related to the effect of the single instructions. Further details about the used Benchmarks are discussed in Section 4.1.2.

**Table 1: Size of RISC-V and ARM compiled programs.**

| ISA condition | IoT [B] | Embench [B] |
|---|---|---|
| ARM-Thumb2 | 209696 | 1766 |
| RV32IMC | 233628 (+11.41%) | 1966 (+11.33%) |

*Toolchain options.* To compile ARM code, we passed to the compiler *-mcpu=cortex-m3 -march=armv7-m -mthumb -Os -ffunction-sections -fdata-sections*. For RISC-V, we used *-march=rv32imc -Os -msave-restore*. For both the ISAs, we provided the linker with *-gc-section*.

*LTO.* The compiler returned unexpected errors during the link-phase of IoT code when using ARM GCC 7.2 and LTO. More generally, all GCC versions returned errors during the link-phase of Embench linked with dummy libraries (see 4.1.2). Therefore, in our first analyses, we used all the optimization flags to reduce the code size, except for *-flto*. Then, we provided data related to LTO in Section 4.5 only for IoT code compiled using ARM GCC 8.2, which did not throw any unexpected error.

*4.1.2 Programs and Benchmarks.* To evaluate the code size reduction by the different applied techniques, we use:

- *IoT*: an industrial use case software developed by Huawei to implement a 3GPP standard aimed at enabling low power and low data rate machine-to-machine communication. Its ARM Binary compiled with Thumb2 instructions and static libraries is more than 200 KiB.
- *Embench 0.5*: a recent [1] open-source set of 19 programs developed to give real feedback on the world of Embedded applications, providing a valid representative set and a reproducible metric to express code size and performance results. In our analysis, we compiled Embench programs with the Embench dummy libraries as suggested in the related documentation, to remove the bias of heavy external code that can be differently optimized for different ISA. The code size of Embench programs is very different with respect to the IoT code size, spacing from 212 B (*crc32*) to 15.42 KiB (*nsichneu*), compiled with ARM Thumb2. To favour coherence of measurement with IoT, instead of the official score format we used the geometric mean of the absolute sizes of the Embench programs and calculated relative results on our references.

## 4.2 RISC-V vs. ARM

We compiled both IoT and Embench code targeting RV32IMC and ARM Thumb2 ISAs. Table 1 shows that the baseline RISC-V code is more than 11% bigger than the ARM counterpart in both IoT code and Embench.

## 4.3 RISC-V Xpulp extension

We analyze here the code size impact of the Xpulp RISC-V extension. Table 2 shows that even though some instructions slightly increase the code size, the overall effect is positive even on this metric, with

[1]announced in June 2019 during the RISC-V Workshop

**Table 2: Size effect of PULP instructions on PULP reference, i.e. RV32IMCXpulp with all the adjustable PULP instructions disabled (see Section 4.1.1). The relative results are calculated over PULP reference. The relative results of *Xpulp* in the last row are calculated over the RV32IMC reference.**

| ISA condition | IoT [B] | Embench [B] |
|---|---|---|
| `PULP reference` | 231900 | 1956 |
| `HW-loops` | 231896 (0.00%) | 1956 (0,00%) |
| `MAC` | 231820 (-0.03%) | 1948 (-0,41%) |
| `Bit Manip.` | 231980 (0.03%) | 1923 (-1,69%) |
| `ALU` | 231240 (-0.28%) | 1950 (-0,31%) |
| `SIMD` | 231900 (0.00%) | 1956 (0,00%) |
| `Mem post incr.` | 232076 (0.08%) | 1931 (-1,28%) |
| `Mem reg. offset` | 231620 (-0.12%) | 1945 (-0,56%) |
| `Xpulp` | 230264 (-1.44%*) | 1896 (-3,56%*) |

a final size improvement of -1.44% for IoT code and -3.56% for Embench, with respect to their RV32IMC binaries.

Some extensions like HW-loops have a negligible impact on the total code size, even if single functions change size. Sometimes the code size increases as adding newly defined instructions can change the register allocation during the compilation phase. In RISC-V, many instructions can be compressed only if their source/destination registers are part of a small subset, and so any change to the register allocation may change whether compressed versions of instructions are available. Therefore, adding new instructions may indirectly cause some code size to increase if the compiler assigns strategic registers to the new instructions without successfully targeting the optimization of the whole code size.

Bit manipulation instructions slightly increase the IoT code size. Throughout the code, some 16-bit compressed *andi* instructions are replaced by 32-bit *p.bclr* Xpulp instructions that increase the code size. The important size reduction of Embench is almost completely caused by one of its 19 programs. Some of them slightly increase or decrease their size, but the code of Nettle-sha256 reduces from 5490 Bytes to 4124 Bytes. This reduction is due to the intensive use of the PULP rotate instruction *p.ror*.

The IoT code size increasing related to the post-increment memory operations is also related to a compiler issue. Some sequences of 16-bit compressed memory operations are replaced by the respective 32-bit post-increment load/store instructions. This behaviour is sub-optimal when compiling for code size. Embench presents more loops with memory operations that use a base address iteratively modified by *add* instructions. These sequences are effectively compressed into post-increment loads/stores, reducing the overall code size.

SIMD instructions are not used in the code and therefore have no measurable effect on it.

Overall, the Xpulp extension reduces the code size of RISC-V programs and boosts their performance, and with some optimization to the PULP toolchain for better placement of the custom instructions the code density can even be improved.

## 4.4 RISC-V HCC extension

We propose a new RISC-V extension that explicitly targets the reduction of the code size of a program, with results summarized in Table 3. The new instructions introduced are:

- 16-bit push/pop/popret (*-mpush-pop*): the new *push* can save onto the stack pre-defined sequences of the callee-saved registers (s0-s11) and the most frequently spilled ones (ra, a0-a1). Moreover, it automatically adjusts the stack pointer taking into account additional stack space for automatic variables. *pop* loads the same sequence into the registers and, if it is a *popret*, returns to the caller.
- 48-bit long load immediate (*-femit-lli*): when RISC-V code loads into a register an immediate outside the range $\pm 128Ki$, it needs 8 Bytes. This instruction allows using 6 Bytes only. The HW implementation cost is not negligible if the processor does not already support 48-bit instructions.
- Compressed load-byte-unsigned/store-byte and load-halfword-unsigned/store-halfword (*-Wa,-enable-c-lbu-sb* and *-Wa,-enable-c-lhu-sh*): these memory operations are common in RISC-V programs, as witnessed also by other works [7]. HCC extension allows compressing them, with limitations on the choice of the registers and the immediate. If the processor already supports 16-bit instructions, the HW implementation cost is not high: there's only the need for modifying the actual decoder to recognize these compressed instructions because they are already supported by the back-end.
- Branch Immediate (*-fimm-compare*): RISC-V does not give the possibility to conditionally branch on the comparison between a register value and an immediate in a single instruction. Therefore, the compiler puts a 2-Byte LI and a 4-Byte branch instruction. ARM does the same using 4 Bytes only, and this instruction provides the same feature also for RISC-V. Xpulp already implements a reduced form of this instruction.
- Muliadd (*-femit-muliadd*): instruction to multiply a register value by a constant, and then add the value from another register. This appears when the code indexes the value from an array. Muliadd can save at least 2 Bytes each sequence.
- Enjal16 (*-Wl,-enjal16*): particular instruction to support jumps to ±16MiB of code from the actual program counter, breaking the limitation of the standard ±1MiB. This is both useful when the code is very sparse and large, and when sections of memory (e.g. internal RAM / external RAM) are far apart in the memory map. This sometimes happens in some embedded applications, and this instruction is used to jump between these distant sections.
- Immediate shift (*-fmerge-immshf*): 32-bit instruction that shifts one of the operands and then performs an arithmetical/logical operation. For example, RISC-V uses at least 6 Bytes to shift a register operand and then add it to another register operand. With this instruction, it uses 4 Bytes only like ARM.
- Unsigned-extend byte/halfword (*-femit-uxtb-uxth*): instruction to unsigned-extend byte/halfword stored in a register.

**Table 3: Size effect of HCC instructions on RV32IMC HCC reference. The relative results are calculated over HCC reference.**

| ISA condition | IoT [B] | Embench [B] |
|---|---|---|
| `HCC reference` | 233804 | 1975 |
| `-mpush-pop` | 221860 (-5.11%) | 1902 (-3.70%) |
| `-Wa,-enable-c-lbu-sb` | 230328 (-1.49%) | 1960 (-0.76%) |
| `-Wa,-enable-c-lhu-sh` | 232084 (-0.74%) | 1971 (-0.20%) |
| `-femit-lli` | 232556 (-0.53%) | 1988 (0.66%) |
| `-fimm-compare` | 230636 (-1.35%) | 1967 (-0.41%) |
| `-Wl,-enjal16` | 233804 ( 0.00%) | 1975 (0.00%) |
| `-femit-muliadd` | 233396 (-0.17%) | 1975 (0.00%) |
| `-fmerge-immshf` | 232992 (-0.35%) | 1915 (-3.04%) |
| `-femit-uxtb-uxth` | 232864 (-0.40%) | 1970 (-0.25%) |
| `-fldm-stm-optimize` | 233796 ( 0.00%) | 1964 (-0.56%) |
| `HCC extension` | 206020 (-11.88%) | 1805 (-8.61%) |

- Load/Store-multiple (*-fldm-stm-optimize*): merge multiple load and store instructions into *load multiple* (*ldm*) and *store multiple* (*stm*) instructions.

Note that when using *push/pop/popret* instructions, it is important to disable the *-msave-restore* flag. Data suggests that the set of instructions that gives the best result on different types of codes is the push/pop/popret with a code size decrease of 5.11% and 3.7%. This is because almost all the functions manipulate the stack. Embench code shows a more limited benefit because some of the internal functions push to the stack (and then pop from it) sequences of registers that are unsupported by the new instructions. These uncompressed load/store sequences are merged into *ldm*/*stm* when the corresponding support is enabled, leading to 0.56% of size improvement, but cannot be converted into push/pop instructions. The support for *ldm*/*stm* does not bring to appreciable benefits into the IoT code if used in combination with either *-msave-restore* or *-mpush-pop*. In this case, these instructions are only used by a single function. Even without both the options, the IoT binary produced with only *-fldm-stm-optimize* enabled is worse than the HCC reference (compiled with *-msave-restore*) by 2.6%. Therefore, data suggests implementing compressed push/pop/popret to achieve a better code density.

Enjal16 instructions do not show any measurable effect in these codes, maybe because they are designed to satisfy special programs that are either large or implement a custom memory map with areas mapped to widely spaced addresses.

The impact of other instructions also depends on the source code. The advantage of compressing memory operations on bytes and halfwords is dependant on how many of these operations are originally present in the code, but our data confirms the trend highlighted also in [7]: load/store on bytes are more frequent than on halfwords. The same dependence is shown by the arithmetical/logical operations: *muliadd* is not used in Embench programs and the *immshf* instructions show different reductions on IoT (-0.35%) and Embench (-3.04%).

The Embench code size increment related to *l.li* instructions is due to issues in implementing the instruction support in the compiler, which modifies also sequences of memory operations not directly linked to the replacement of longer 8-Byte *lui+addi* sequences with *l.li*.

**Table 4: Size effect of push/pop/popret instructions on RISC-V PULP ISA. The relative results are calculated over RV32IMC Xpulp.**

| ISA condition | IoT [B] | Embench [B] |
|---|---|---|
| `RV32IMC Xpulp` | 230264 | 1896 |
| `RV32IMC Xpulp push/pop` | 220956 (-4.04%) | 1860 (-1.89%) |

Enabling all the new HCC instructions brings to a considerable code size reduction on both IoT and Embench. RISC-V IoT code with HCC extension becomes smaller than the ARM counterpart by 1.75% (Table 5). Moreover, these results are worsened by the suboptimal compiler support for the *l.li* instruction, which in principle should not be able to increase the code size of a program.

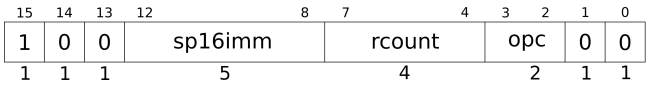

**Figure 1: Push/pop/popret instructions encoding.**

*4.4.1 16-bit push/pop/popret.* Data suggests that *push/pop/popret* instructions are the ones having the most significant effect on code size reduction.

Push/pop/popret HW instructions are better than save/restore routines because save/restore routines have their memory footprint and every time a function jumps to them, it uses either a 32-bit jump (when t0 is used as link register or when the functions are out of ±2 KiB range) or a 16-bit jump (more frequent for unconditional jumps to restore because t0 cannot be used). If the function is very far, it can use also more than 32 bits. In the worst-case scenario, when the jump to the restore routine is not the last instruction of the function, save/restore routines add 4 jumps per function: to/from save, to/from restore. Moreover, save/restore routines can re-adjust the stack pointer after the call and perform redundant operations. On the contrary, 16-bit push and pop/popret provide a small and controlled prologue and epilogue to the functions, without unnecessary jumps. This instruction does not completely adhere to the RISC-V philosophy of simplicity: it is a multi-cycle instruction that injects sequences of memory operations, an addition, and, possibly, a jump. In Figure 1, we report the encoding of the instructions. The field *opc* stores the instruction type (*push*, *pop* or *popret*), *rcount* specifies one of the fixed sequences of registers to be pushed/popped, and the unsigned 5-bit value encoded in *sp16imm* tells the processor to adjust the stack pointer by additional 16-Byte blocks, space that can be used for automatic variables by the function. We added the push/pop/popret support to the PULP toolchain with the results shown in Table 4. The summary of the code sizes of RISC-V with respect to ARM Thumb2 is presented in Table 5.

*4.4.2 Implementation on CV32E40P.* To evaluate the impact in area of the proposed push, pop, and popret instructions, we implemented

**Table 5: Code size inflation of different ISA and instructions w.r.t. ARM-Thumb2.**

| ISA condition | IoT [B] | Embench [B] |
|---|---|---|
| ARM-Thumb2 | 209696 | 1766 |
| RV32IMC HCC | 206020 (-1.75%) | 1805 (2.21%) |
| RV32IMC | 233628 (11.41%) | 1966 (11.33%) |
| RV32IMC push/pop | 221860 (5.80%) | 1902 (7.70%) |
| RV32IMC Xpulp | 230264 (9.81%) | 1896 (7.36%) |
| RV32IMC Xpulp push/pop | 220956 (5.37%) | 1860 (5.32%) |

them on the open-source CV32E40P core[2]. The CV32E40P core (Figure 2) implements the RISC-V RV32IMCXpulp ISA [6] and it has been designed for energy-efficient execution of data processing algorithms on edge-devices. These instructions have been implemented leveraging the existing datapath to perform load and store operations, updates to the stack pointer, and jump operations to return from functions. A finite-state-machine has been implemented in the core to first emit the memory operations in sequence, then it schedules an addition operation to update the stack pointer, and eventually jumps to the return address. To simplify the HW implementation, the execution of push and pop instructions cannot be interrupted by interrupts. Also, as the core does not support memory bus errors, no exception can be triggered. Under the same timing constraints, the original CV32E40P core and the one extended with the proposed instructions have been synthesized targeting the 22nm GLOBALFOUNDRIES FDX technology node. Results show that the new instructions do not impact the maximum frequency, and add only 2.5% of core area (~335 nand2-equivalent gates), providing improvements in stack handling operations in performance, power, and code-size.

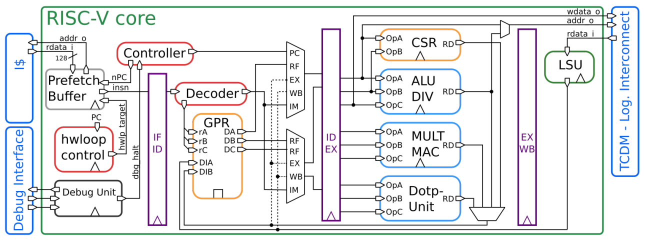

**Figure 2: CV32E40P block diagram.**

## 4.5 LTO

At the end of the analysis, we applied Link Time Optimization to the IoT code, both for ARM and HCC. The results are reported in Table 6.

Link Time Optimization seems to be a powerful tool, but also highly code-dependant: other non-reported analyses showed lower code size benefit (a different Huawei program code is reduced only by 2% in size). This optimization greatly reduces IoT code size, with a shrinking of almost 11.18% for the Xpulp-push/pop code.

[2]The OpenHW Group CV32E40P core is freely downloadable at https://github.com/openhwgroup/cv32e40p/ under the SolderPad licence

**Table 6: LTO results (*-flto -Os*) for ARM and HCC, PULP RISC-V extensions.**

| ISA condition (with LTO) | IoT size [B] | Inflation over ARM |
|---|---|---|
| ARM-Thumb2 | 184184 | 0.00% |
| RV32IMC push/pop | 197156 | 7.04% |
| RV32IMC HCC | 190136 | 3.23% |
| RV32IMC Xpulp push/pop | 196236 | 6.54% |

However, the code size gap with ARM is slightly increased, because LTO works better on ARM. LTO can also increase performance, therefore if the binary is not supposed to be debugged later, it is a good choice to enable this option. As we already said, for this comparison we used ARM GCC 8.2, because the previous version raised errors during the compilation.

## 5 CONCLUSION

Despite the RISC-V RVC extension and compiler optimizations, RISC-V code can be more than 11% larger than the ARM counterpart on embedded-domain benchmarks. In this paper, we analyzed how the Xpulp ISA extensions affect the code size, showing their benefit on code footprint while increasing performance at the same time. Then we proposed a RISC-V extension, specifically designed to decrease the code size. This extension targets a different range of applications, and our experimental results highlight that it closes the code density gap with ARM to almost zero. Specifically, RISC-V HCC IoT code is smaller than ARM IoT code by 1.75%. We implemented compressed push/pop/popret, the extension giving the largest single-handed code density improvement, on the CV32E40P core to provide an evaluation of its impact on core area and frequency. The enhanced CV32E40P core hits a sweet spot for code density as well as boosted performance, with a code size just 5.37% larger than the ARM Thumb2 code, a core area increasing of only 2.5%, and no impact on the operating frequency, therefore maintaining the 10x boost for DSP applications.

## 6 ACKNOWLEDGEMENTS

This work is sponsored by Computer Architecture Research Lab in Huawei Zurich Research Center.

## REFERENCES

[1] ARM 2014. *ARM® v7-M Architecture Reference Manual* (6 ed.). ARM.
[2] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. 2003. Survey of Code-Size Reduction Methods. *ACM Comput. Surv.* 35, 3 (Sept. 2003), 223–267. https://doi.org/10.1145/937503.937504
[3] D. Bol, M. Schramme, L. Moreau, T. Haine, P. Xu, C. Frenkel, R. Dekimpe, F. Stas, and D. Flandre. 2019. 19.6 A 40-to-80MHz Sub-4W/MHz ULV Cortex-M0 MCU SoC in 28nm FDSOI With Dual-Loop Adaptive Back-Bias Generator for 20s Wake-Up From Deep Fully Retentive Sleep Mode. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*. 322–324.
[4] Palmer Dabbelt Cesare Garlati Ofer Shinaar David Patterson, Jeremy Bennett. 2019. *Initial Evaluation of Multiple RISC ISAs using the Embench™ Benchmark Suite. What is the Cost of Simplicity?* Retrieved April 24, 2020 from https://content.riscv.org/wp-content/uploads/2019/12/12.10-12.50a-Code-Size-of-RISC-V-versus-ARM-using-the-Embench%E2%84%A2-0.5-Benchmark-Suite-What-is-the-Cost-of-ISA-Simplicity.pdf
[5] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler Techniques for Code Compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 378–415. https://doi.org/10.1145/349214.349233

[6] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. 2017. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (2017), 2700–2713.

[7] Peijie Li. 2019. *Reduce Static Code Size and Improve RISC-V Compression*. Master's thesis. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-107.html

[8] H. Lozano and M. Ito. 2016. Increasing the Code Density of Embedded RISC Applications. In *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*. 182–189.

[9] GNU Project. 2020. *GNU GCC*. Retrieved April 24, 2020 from https://gcc.gnu.org/

[10] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini. 2018. Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 1–3.

[11] C. Sudanthi, M. Ghosh, K. Welton, and N. Paver. 2009. Performance analysis of compressed instruction sets on workloads targeted at mobile internet devices. In *2009 IEEE International SOC Conference (SOCC)*. 215–218.

[12] Andrew Waterman. 2011. *Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed*. Master's thesis. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-63.html

[13] Andrew Waterman. 2016. *Design of the RISC-V Instruction Set Architecture*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html

[14] Editors Andrew Waterman and Krste Asanovic. 2019. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation.

[15] X. H. Xu, S. R. Jones, and C. T. Clarke. 2003. ARM/THUMB code compression for embedded systems. In *Proceedings of the 12th IEEE International Conference on Fuzzy Systems (Cat. No.03CH37442)*. 32–35.