2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID) | 978-1-6654-4087-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/VLSID51830.2021.00014

Evaluation of Bit Manipulation Instructions in Optimization of Size and Speed in RISC-V

Babu P S[†], Snehashri Sivaraman[*], Deepa N Sarma[§] and Tripti S Warrier[†]
[†]*Department of Electronics*, *Cochin University of Science and Technology*, Kochi, Kerala, India
[*]Department of ECE, Meenakshi Sundararajan Engineering College, Chennai, India
[§]*Department of Computer Science & Engineering*, *Indian Institute of Technology Madras*, Chennai, India
[†]Corresponding Author: babu.ps@cusat.ac.in

***Abstract*—With an ever-increasing usage of electronic controllers in various disciplines that could be attributed to Industry 4.0, Internet of Things (IoT) and quick shift in computational paradigms, a demand for high code density and faster controllers are expected at the diversified nodes that improve energy efficiency without performance penalty. RISC-V is an open-source Instruction Set Architecture (ISA) which is designed with modularized extensions, that enables to design processors with a provision of individual extension evaluation helping in the design of low-power and secure embedded controllers. Bit manipulation is one of the key operations performed in domains such as Cryptography, Communication and Networking protocols, Digital Signal Processing, Bioinformatics etc., which are currently implemented using RISC-V standard instruction set. This paper implements the 'B' extension of RISC-V that hosts instructions specific to operate at bit-level manipulations, which is absent in ratified unprivileged ISA manual. A quantitative analysis is performed to assess the impact of Bit Manipulative Instructions (BMI) in size and speed improvements using the Embench™ benchmarks against the standard instruction set 'IMAC' under the RV32 configuration. The results show significant improvements, with some programs achieving a speedup of 28% and size reduction of 20%.**

Index Terms*—RISC-V 'B' Extension, Bit Manipulation Instructions (BMI), Embench, Size & speed, SoC performance, *bSoC

I. Introduction

Embedded systems drive the day-to-day activities in the modern age, which is evident from its consumption pattern fuelling the digital revolution. Highly energy-efficient designs in smaller packages are the need of the hour to sustain the ever-increasing demand for electronic controllers. Across the computational paradigms used in IoT [1], edge devices are used to sense, measure, intelligently interpret and transport information to the higher nodes as part of its fundamental service. The controllers used in these devices are usually the entry point for the physical world parameters to get converted into computational data which is used in making informational decisions. The current computational architectures are designed to enforce strong security at edge device to maintain data integrity, meanwhile avoiding nefarious activities at these entry-points.

Logical and bit manipulative operations are needed to improve the performance of edge device without a bigger penalty on code size. Various instances where BMI can be harnessed to improve performance of an edge device are as follows: Starting with interfacing sensor(s), whose input is predominantly analog that needs to be sampled & digitally filtered (e.g. FFT filtering techniques [2], state-space modelled techniques) and stored digitally without corruption using checksum (e.g. CRC, fletcher [3]). These devices reduce the data transmitted to higher nodes by preferring mechanisms like sorting [4] or interpretation which are usually computationally intensive algorithms [5] . Such a data transmission is performed in encrypted format [6] (e.g. AES [2] , Blowfish [7]) with the secure hash (e.g. SHA, BLAKE [8]) to ensure integrity in secured data transmission. The embedded controllers used at the edge devices need to perform these operations iteratively. Moreover examples used for the indicated operations gains from bit-level manipulations as in [9] which is currently achieved using standard instruction set.

RISC-V is an open-source RISC ISA driving innovation by providing freedom on customizing architecture. Typical instructions in RISC-V were optimized to process word-sized or half-word sized operations, which can't support bit level operations natively. In such a scenario, for instance, the skeletal code of CRC32 can be represented as shown in Listing 1, whose RISC-V assembly is shown in Listing 2.

Listing 1: CRC32 Pseudo code

```
x = (x >> 1) ^ (0xEDB88320 & ~((x&1)-1));
```

Listing 2: Base RISC-V

```
li   x6 , 0xEDB88320
li   x7 ,  1
SRLI x10, x5 ,  1
ANDI x5 , x5 ,  1
SUB  x5 , x5 , x7
XORI x5 , x5 , -1
AND  x5 , x5 , x6
XOR  x5 , x10, x5
```

Listing 3: Modified RISC-V

```
li   x6 , 0xEDB88320
li   x7 ,  1
SRLI x10, x5 ,  1
ANDI x5 , x5 ,  1
SUB  x5 , x5 , x7
ANDN x5 , x6 , x5
XOR  x5 , x10, x5
```

Listing 3 uses logical + negate operation ***ANDN*** to fuse two instructions. Since this snippet runs for each bit of the payload generated at the edge device it's worth saving one cycle in processing each bit, meanwhile saving memory by one instruction. On a similar instance, single bit set ***SBSET*** instruction is proven to be useful in applications like cubic root solver, N-body simulation, statistics & sorting. This is shown in Table. I, which is an excerpt from the Embench™ benchmark. The Assembly file is obtained with GCC 10.0 compiler at -Os optimization and trace is obtained by running the executable on verilator simulated *'bSoC'*.

TABLE I: ***SBSET*** instruction usage in various applications

Appl. / File	sorting	N-body	statistics	cubic root solver
Assembly	12	12	12	26
Trace	5595	20861	33358	37690

The performance equation shown in (1), depends on Instruction Count (IC), Cycles per instruction (CPI) and Clock Time (T). By making changes at ISA, all three factors get affected resulting in improvement of Execution time.

$$CPU\ time = IC \times CPI \times T \tag{1}$$

Popular ISAs like x86 and ARM has BMI in their instruction set spectrum, as usage of BMI can be found in literature for square root generation [5], communication systems [10], searching & sorting algorithms [4], LSB image steganography [11], Neural Network based Image classification [12] etc. Emphasizing the need of BMI, RISC-V Foundations Bitmanip Extension working group, had proposed a set of bit manipulative instructions and it's draft specifications, which was at v0.93 as of the commit *cb7a84eb* [13]. The previous work includes Bit-level permutation operations in [9], IP-XACT modelled Bit-level counting operation in [14] & BMI impact on code size in [15]. This paper differs from those by:

- Implementing BMI as per draft [13] and emulating with a standard core on FPGA.
- Avoids assembly in-lining in source code to assess the compiler efficiency in generating BMI.
- Assessing both size & speed impacts due to BMI at various levels of compiler optimizations.

Section II focuses on the design & implementation of **b**it manipulation enabled **SoC** *"bSoC"* that includes development of BMI as a Bit Manipulation Unit (BMU) & interfacing with a RISC-V standard core [16] using the **S**hakti **Cop**rocessor (SCop) Interface. Section III evaluates *bSoC* against 'IMAC' *standard SoC* using Dhrystone & Embench™ benchmarks [17] with possible improvements in near-future.

II. Design & Implementation of bSoC

Design of *bSoC* involves three key sections:

1) Design of BMU as coprocessor
2) Interfacing BMU to core using SCop Interface.
3) Modification to core pipeline

From the draft [13], 58 instructions qualify for 32-bit configuration of BMI as shown in Fig. 1. The design of Bit Manipulation Unit (BMU) consists of decoder and computational unit as shown in Fig. 2. BMU provides decoupled *Request* interface as input for instruction & operands and

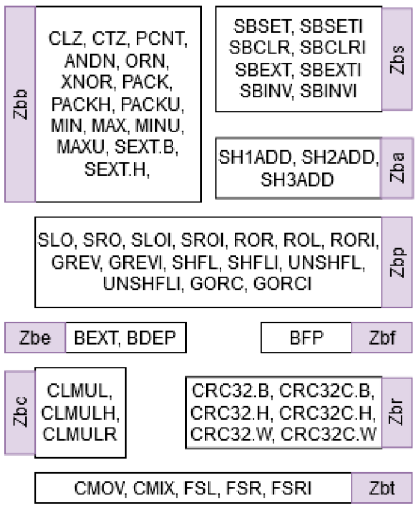

Zbb
CLZ, CTZ, PCNT, ANDN, ORN, XNOR, PACK, PACKH, PACKU, MIN, MAX, MINU, MAXU, SEXT.B, SEXT.H,
SBSET, SBSETI SBCLR, SBCLRI SBEXT, SBEXTI SBINV, SBINVI
Zbs
SH1ADD, SH2ADD, SH3ADD
Zba
SLO, SRO, SLOI, SROI, ROR, ROL, RORI, GREV, GREVI, SHFL, SHFLI, UNSHFL, UNSHFLI, GORC, GORCI
Zbp
Zbe
BEXT, BDEP
BFP
Zbf
Zbc
CLMUL, CLMULH, CLMULR
CRC32.B, CRC32C.B, CRC32.H, CRC32C.H, CRC32.W, CRC32C.W
Zbr
CMOV, CMIX, FSL, FSR, FSRI
Zbt

Fig. 1: 32-bit BMI set reproduced from [13]

Response interface for the output as shown in inset of Fig. 5. Decoder unit distinguishes between various instruction, generating control signal that is passed to computational unit along with the operands. Instructions are grouped into sub-extensions and is implemented as compile-time reconfigurable blocks using Bluespec System Verilog (BSV). The validity of instruction is indicated by *valid* bit prefixed to the computed result. This computational unit is designed to compute in single cycle for all BMI. BMU consumes 3525 slice LUTs during synthesis phase as reported by Xilinx Vivado®.

An embedded class processor *E-Class* [16] from Shakti family is chosen as RISC-V core, configured with **IMAC** extensions of RISC-V ISA, implemented as 32-bit 3-stage in-order pipeline architecture. Besides the core, this '*standard SoC*' has a UART controller, Core-Level Interrupt Controller & Debug Module as minimal peripherals shown in Fig. 3. Ideally, BMU should be placed alongside the *Execute* stage of Fig. 3, but for flexibility of interfacing to any RISC-V core, BMU is designed as standalone module. This is interfaced to *standard SoC* using the *SCop* Interface, which from here-on will be referred to as *'bSoC'*, whose architectural changes are depicted in Fig. 5.

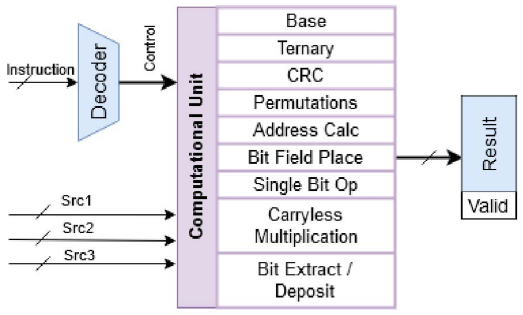


Fig. 2: 32-bit Bit Manipulation Unit

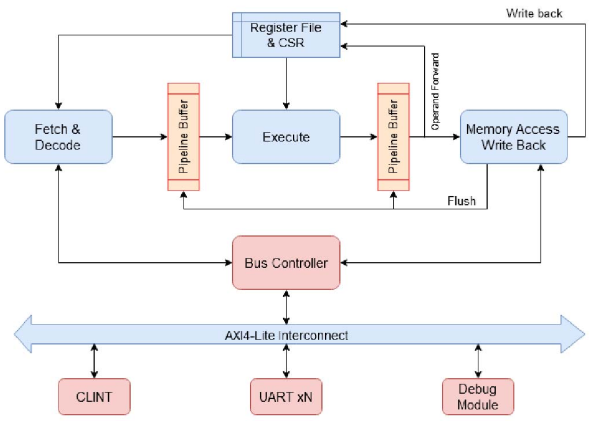


Fig. 3: Shakti E-Class Architecture (Modified [16])

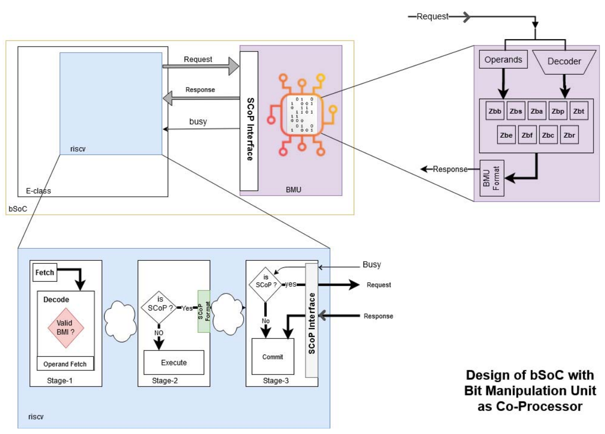


Fig. 5: Architecture of *bSoC*

A high-level overview of SCop interface is shown in Fig. 4a, where the core and BMU communicates on *Request-Response* sub-interfaces based on *Busy* indicator. When *Valid* and *Ready* signals are high in the same clock cycle, data transfer takes place. *Instruction* & *operands* in *Request* sub-interface are 32-bit wide bus. *User* in both sub-interfaces is used for communicating custom data which is not relevant in *bSoC* context. *Response* sub-interface shown in Fig. 4c, consists of signals including 32-bit wide output *Data*, 1 bit for *Trap* to indicate the validity of *Data* and *Trap info* for the cause of invalidity.

As BMU completes the operation in a single-cycle, a Tightly Coupled Coprocessor (TCC) mechanism was chosen, yet the core register file control is not provided to the coprocessor. Ideally SCop interface at *Execute* in Fig. 3 is optimal to feed the *Request* to BMU. Instead *Write back* stage was chosen to interface SCop that could be justified for the following reasons:

- Guaranteed Execution of previous instruction
- Complications in dealing with exceptions at *execute* stage can be avoided
- Decoupled Interface

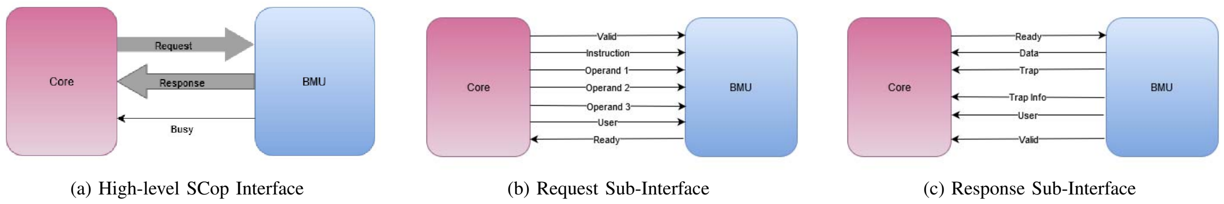
The core is modified as shown in the inset of Fig. 5, where the fetched instruction is checked for BMI, if so, the execution order would go through the process of fetching operands and passing to stage-3. Else, standard execution would follow through. In order to work with BMU as coprocessor, individual stages of the core had been modified.

1) Modifications to stage-1: After fetch & decode of instruction, as indicated in Fig. 6a, a register named *csr.misa*, is checked whether 'B' extension is enabled by SoC to process BMI. Later, the generated instruction is checked, whether it qualifies to be BMI. If the instruction passes both checks, the instruction is marked as 'SCop' type indicating other stages that instruction will be executed in BMU. Operands are fetched from the registers to prepare the stage-1 meta-info which is passed as input to stage-2.

2) Modifications to stage-2: As shown in Fig. 6b, based on the stage-1 meta-info, if the instruction is marked as SCop type, latest operands are fetched and arranged in the 'SCop format' required for the *Request sub-interface* as shown in Fig. 4b. This stage generates meta-info that is passed to stage-3.

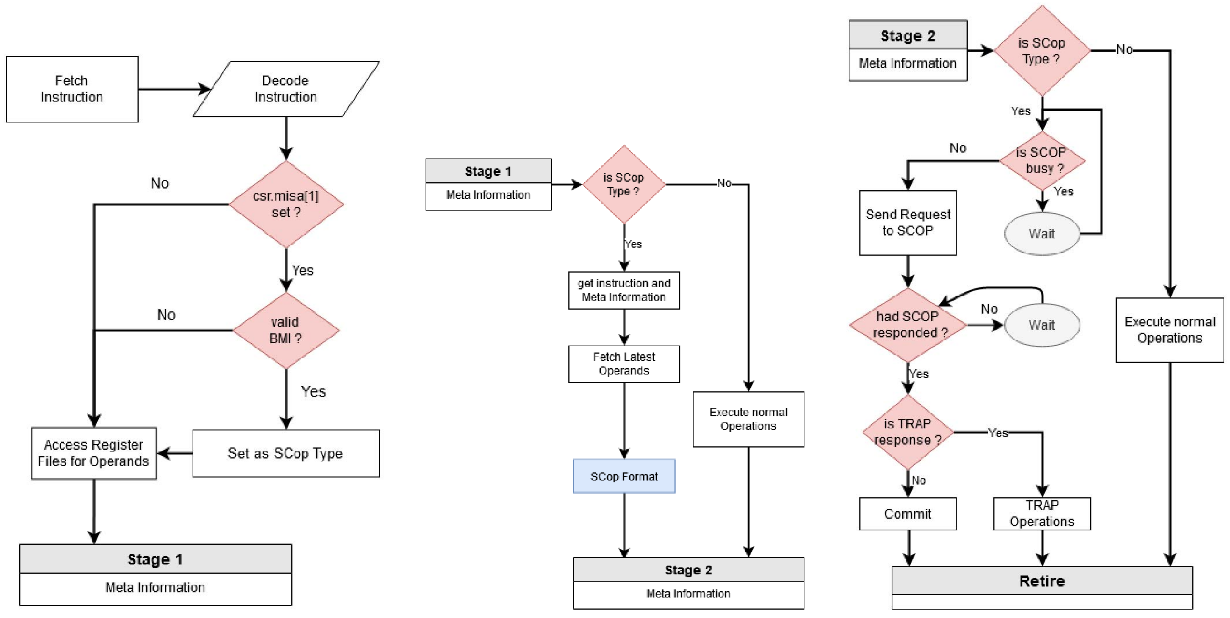
3) Modifications to stage-3: Fig. 6c depicts the Stage-3, that handles the crucial part of interaction with BMU using the *busy* signal shown in Fig. 4a. Stage-3 waits until the BMU is ready to receive the *Request*. Once the *Request* transaction is performed, stage-3 should wait until the BMU responds. Within a single clock cycle BMU provides *Response* to stage-3 as shown in Fig. 4c. Based on the 'Trap' field of *Response*, the instruction will retire either by commit or by executing trap operations.

All the design and modifications are performed in BSV and is compiled to generate Verilog files, which is simulated on cycle-accurate Verilog simulator, Verilator® and synthesized



(a) High-level SCop Interface (b) Request Sub-Interface (c) Response Sub-Interface

Fig. 4: SCop interface



(a) Modified Stage 1 (b) Modified Stage 2 (c) Modified Stage 3

Fig. 6: Modification of core pipeline

for xc7a100tcsg324-1 FPGA device as softcore CPU using Xilinx™ Vivado®. The *bSoC* is validated with compliance tests and is built as softcore. The *bSoC* consumes 6% more LUTs than *standard SoC* for the same FPGA device. When *bSoC* is run at 50 MHz, the slack margin was reduced by 50% compared to *standard SoC* and the power consumption increased by 9.3%, as reported by Vivado Power Analysis Tool.

III. Result Analysis

This section attempts to compare the *bSoC* against the *standard SoC* using a synthetic benchmark Dhrystone and an open-source benchmark suite Embench™ to evaluate the effectiveness of BMI. All the programs in this section were compiled using GNU C Compiler 10.0.0 (experimental version) and the compiler options are specified along with the benchmarks.

A. Dhrystone

Dhrystone is one amongst the popularly criticized yet widely published benchmark in terms of *DMIPS/MHz* speed. Following suit, Dhrystone 2.1 is compiled using the compiler options as tabulated in Table. II. The *bSoC* uses *arch* option, different from the *standard SoC*. The results are tabulated in Table II for the chosen compiler options. *bSoC* reported 0.226 DMIPS/MHz over the 0.215 DMIPS/MHz of *standard SoC*. The average CPI reduced from 8.08 to 7.55 in *bSoC*, resulting in ≈5% speedup. Despite the arguments of improving dhrystone using compiler options [18], introduction of BMU had improved the performance.

TABLE II: Dhrystone options and results

version	2.1	
Compiler Options	-mcmodel=medany -std=gnu99 -O2 -ffast-math -fno-common -nostartfiles -fno-builtin-printf -mabi=ilp32 -w -static -lgcc	
	Standard Soc	***bSoC***
	-march=rv32imac	*-march=rv32imacb*
Iterations	10000	
Frequency	32MHz	
Results		
Standard SoC	μS for one run through Dhrystone	82
	Dhrystones per second	12134
bSoC	μS for one run through Dhrystone	78
	Dhrystones per second	12764

B. Embench™

Embench v0.5 is a set of 19 programs that has widely varying branch, computational and memory intensive operations reported in terms of size and speed as geometric mean score. To better understand the impact of BMI, benchmarks were run on Verilator® simulated Executable (cycle-accurate as per benchmark guidelines) for various optimizations provided by compiler ranging from Size Optimized (-Os -msaverestore) to Speed Optimized (-O3) and scores are reported in *Absolute* numbers along with the std. deviation (error bars) around the Geometric mean. Size and speed of each application in the benchmark was calculated using (2) and (3) respectively. As per benchmark guidelines, record of details are tabulated in Table III.

$$size = .text + .data + .bss + .rodata \tag{2}$$

$$CPU_{\text{Exec. time}} = cyclecount \times \frac{1}{operating frequency} \tag{3}$$

$$Speedup = \frac{CPU_{\text{Exec. time}} of\, Std.\ SoC}{CPU_{\text{Exec. time}} of\, bSoC} \tag{4}$$

TABLE III: Embench record of details

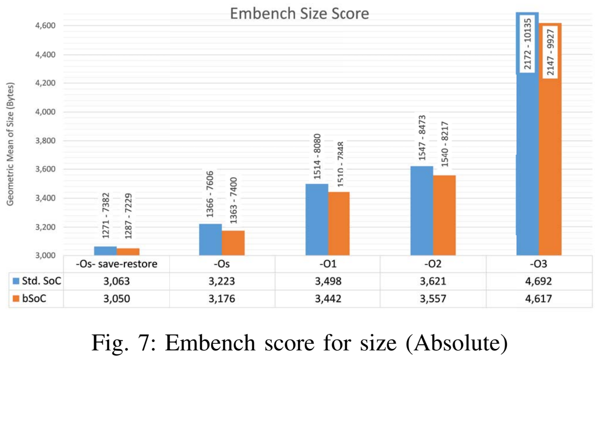
Version	0.5				
Platform	Verilator Executable (Cycle Accurate)				
Frequency	1MHz				
Toolchain	32-bit RISC-V Toolchain				
	riscv-binutils 2.33.5	*c8704188*			
	riscv-gcc 10.0.0	*d69e3efd*			
	riscv-newlib-3.2	*f289cef6*			
Compiler Flags	-mabi=ilp32, -mcmodel=medany, -c, -fdata-sections, -ffunction-sections				
	Standard SoC	***bSoC***			
	'-march=rv32imac'	'-march=rv32imacb'			
Linker Flags	-Wl,-gc-sections, -nostartfiles, -nostdlib				
Varying Options	-Os -msaverestore	-Os	-O1	-O2	-O3

1) Benchmarking for Size: The variables being summed up in (2) are sections from Executable and Linkable Format (ELF) compiled with libraries provided by benchmarks suite to get the size score whose absolute values are plotted in Fig. 7. As shown in Table. IV, BMI improved code density by 1.8% when compiled for (-O2) speed optimization.

2) Benchmarking for Speed: Libraries '-lm, -lc, -lgcc' were used by benchmarks for running on the platform. The *'mcycle'* Control Status Register (CSR) is monitored to get the execution time. The results were plotted as shown in Fig. 8 and improvements can be observed in Table. IV. 8% speedup is observed on running the (-Os) optimized executable.

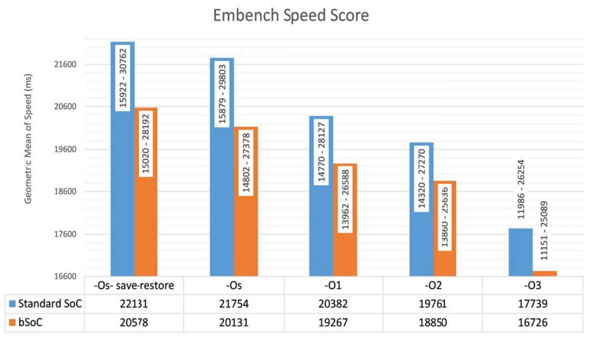
TABLE IV: Size and Speed Improvements due to BMI

	-Os -msaverestore	-Os	-O1	-O2	-O3
Size	1.0043	1.0148	1.0163	1.0180	1.0162
Speed	1.0755	1.0806	1.0579	1.0483	1.0606



	-Os- save-restore	-Os	-O1	-O2	-O3
Std. SoC	3,063	3,223	3,498	3,621	4,692
bSoC	3,050	3,176	3,442	3,557	4,617

Fig. 7: Embench score for size (Absolute)



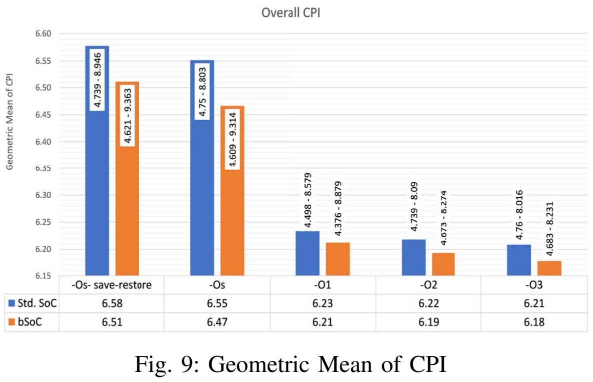
	-Os- save-restore	-Os	-O1	-O2	-O3
Standard SoC	22131	21754	20382	19761	17739
bSoC	20578	20131	19267	18850	16726

Fig. 8: Embench score for speed (Absolute)

Fig. 7,8 & 9 represents geometric mean score for each optimization, along with the std. deviation represented as numerical range. These charts while representing trend-line amongst optimizations also contrasts the difference between SoC's and, std. deviation represented as numeric range gives the spread of data around geometric mean.

The results in the previous section is represented as geometric mean score and a primary inspection would reveal that *bSoC* outperforms the *standard SoC* in terms of the overhead incurred with the reduced mean value. Scrutinizing each application in the benchmark per optimization, reveals that speed improves for the computationally intensive operations like matrix calculations, cryptography algorithms, sorting algorithms etc. Memory intensive operations can gain from ***PACK*** instruction. The CPI chart for each optimization is plotted in Fig. 9. Some notable mentions of the results are:

- *nettle-sha256* application performed at its best using *bSoC* at '-O3' optimization with reduction in size by 20% and speedup ratio of 1.42. Six BMI {rori, rol, pack, grevi, cmix & max} of 247 instances were found in disassembly that accounted to five BMI of 576732 instances in run-trace.
- *crc32* program can be run at its best using *Zbr* of BMI, if assembly in-lining is considered. Despite that, compiler used Listing 3, where single ***ANDN*** accounted for 174182



	-Os- save-restore	-Os	-O1	-O2	-O3
Std. SoC	6.58	6.55	6.23	6.22	6.21
bSoC	6.51	6.47	6.21	6.19	6.18

Fig. 9: Geometric Mean of CPI

instances in run-trace resulting in poor speedup ratio of 0.96 which could be attributed to the BMU outside the core. *PicoJPEG* followed suit, despite a high presence of BMI instance in run-trace.

- *minver & st* are programs that have lost around 30 bytes yet gained in speedup ratios of $>1.23+$ with significant account of BMI presence in run-trace.
- *nettle-aes* is another program that could be assembly in-lined for its bit permutative operations. Compiler efficacy can be underpinned for the 'no-gain' in size as well as speed.
- Out of 95 (19 application $\times$ 5 optimization) run instances
  - 26 instances showed >1.10 speedup ratio.
  - 52 instances showed reduction in CPI due to BMI.
  - 25 instances showed <1.0 speedup ratio.
- Of all the 95 run-traces only 15 out of 58 BMI set were generated by the compiler. As worst-case scenario is approached, and the benchmark programs are not supposed to be assembly in-lined, the compiler needs to be effective in generating more BMI.

IV. Conclusion

This paper tries to evaluate BMI impact in embedded applications by designing *bSoC*, where BMI is designed as a single-cycle executable and interfacing it as coprocessor to RISC-V core. Besides the overhead in area and power, *bSoC* is analysed to interpret the effectiveness in improving performance. The results proved that certain benchmark applications improved in speedup ratio for a negligible loss in size. This also emphasizes the need for efficient compilers that generate assembly and complements the hardware, without which the efforts on development in hardware will prove to be futile. With assembly in-lining techniques, BMI can help in applications like cipher operations in cryptographic algorithms, binary compression & decompression, binary image morphological processing, DNA sequence compression, alignment & translation, least significant bit steganography, transfer-coding, Error Correcting Codes (ECC) in digital communications, puncturing techniques etc. The further developments to this paper include reducing the area overhead by making instructions multi-cycle or pipelined without loss on performance, keeping BMU along with ALU to avoid extra latency for handling BMI, Building a framework that could replace standard instructions with equivalent BMI would improve byte efficiency in size.

Acknowledgment

This paper is a derived work from various repositories. Section II was derived from Shakti Processors licensed under *BSD 3-Clause*. Section III is derived from Embench™ licensed under *GPLv3*. The authors would like to thank and acknowledge the contribution of resource persons at Shakti Processors in development of section II.

References

[1] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the internet of things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

[2] Y. Hilewitz, C. Lauradoux, and R. B. Lee, "Bit matrix multiplication in commodity processors," in *2008 International Conference on Application-Specific Systems, Architectures and Processors*, 2008, pp. 7–12.

[3] R. Jain, "A comparison of hashing schemes for address lookup in computer networks," *IEEE Transactions on Communications*, vol. 40, no. 10, pp. 1570–1573, 1992.

[4] J. H. Reynolds, "Using bit manipulation to reduce sequential search times," *Journal of Computing Sciences in Colleges*, vol. 17, no. 2, pp. 263–270, 2001.

[5] V. Kotlov, "Method and apparatus for square root generation using bit manipulation and instruction interleaving," U.S. Patent 6,625,632, Sep. 23, 2003.

[6] S. Dey, J. Nath, and A. Nath, "An advanced combined symmetric key cryptographic method using bit manipulation, bit reversal, modified caesar cipher (sd-ree), djsa method, ttjsa method: Sja-i algorithm," *International Journal of Computer Applications*, vol. 46, no. 20, pp. 46–53, 2012.

[7] L. Christina and V. Joe Irudayaraj, "Optimized blowfish encryption technique," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 2, no. 7, pp. 5009–5015, 2014.

[8] M. Arsalan, M. Ata-ur Rehman, N. Mehmood, and A. Aziz, "Compact hardware implementation of sha-3 finalist blake on fpga," in *2013 IEEE 9th International Conference on Emerging Technologies (ICET)*. IEEE, 2013, pp. 1–5.

[9] B. Koppelmann, P. Adelt, W. Mueller, and C. Scheytt, "Risc-v extensions for bit manipulation instructions," in *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, July 2019, pp. 41–48.

[10] S. H. Jeong, M. H. Sunwoo, and S. K. Oh, "Bit manipulation accelerator for communication systems digital signal processor," *EURASIP Journal on Advances in Signal Processing*, vol. 2005, no. 16, p. 793614, 2005.

[11] M. R. Islam, A. Siddiqa, M. P. Uddin, A. K. Mandal, and M. D. Hossain, "An efficient filtering based approach improving lsb image steganography using status bit along with aes cryptography," in *2014 International Conference on Informatics, Electronics & Vision (ICIEV)*. IEEE, 2014, pp. 1–6.

[12] S. Payvar, M. Khan, R. Stahl, D. Mueller-Gritschneder, and J. Boutellier, "Neural network-based vehicle image classification for iot devices," in *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2019, pp. 148–153.

[13] *RISC-V Bit manipulation extension*, RISC-V Foundations Bitmanip Extension working group Working Draft of proposed standard, Rev. 0.93, Jan. 29 2020. [Online]. Available: https://github.com/riscv/riscv-bitmanip/blob/master/bitmanip-draft.pdf

[14] S. Payvar, E. Pekkarinen, R. Stahl, D. Mueller-Gritschneder, and T. D. Hämäläinen, "Instruction extension of a risc-v processor modeled with ip-xact," in *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. IEEE, 2019, pp. 1–5.

[15] M. Perotti, P. Davide, G. Tagliavini, D. Rossi, T. Kurd, M. Hill, L. Yingying, and L. Benini, "Hw/sw approaches for risc-v code size reduction," *CARRV 2020: Workshop on Computer Architecture Research with RISC-V*, May 2020.

[16] E-class: An embedded class processor. Internet draft. Department of Computer Science and Engineering, Indian Institute of Technology, Madras. Chennai, India. [Online]. Available: http://shakti.org.in/processors.html

[17] D. Patterson, J. Bennett, P. Dabbelt, C. Garlati, G. S. Madhusudan, and T. Mudge, "Embench™: An evolving benchmark suite for embedded iot computers from an academic-industrial cooperative(towards the long overdue and deserved demise of dhrystone)," presented at the RISC-V Workshop Zurich Proceedings, ETH Zurich, Zurich, Switzerland., Jun. 12 2019. [Online]. Available: https://youtu.be/VfWSGewNsT8

[18] R. York, "Benchmarking in context: Dhrystone," *ARM, March*, 2002.