# RISC-V Extensions for Bit Manipulation Instructions

Bastian Koppelmann, Peer Adelt, Wolfgang Mueller, Christoph Scheytt
*Paderborn University/Heinz Nixdorf Institute*
Paderborn, Germany
{kbastian,adelt,wmueller,cscheytt}@hni.uni-paderborn.de

***Abstract*—Embedded systems require a high energy efficiency in combination with an optimized performance. As such, Bit Manipulation Instructions (BMIs) were introduced for x86 and ARMv8 to improve the runtime efficiency and power dissipation of the compiled software for various applications. Though the RISC-V platform is meanwhile widely accepted for embedded systems application, its instruction set architecture (ISA) currently still supports only two basic BMIs.**

**We introduce ten advanced BMIs for the RISC-V ISA and implemented them on Berkeley's Rocket CPU [1], which we synthesized for the Artix-7 FPGA and the TSMC 65nm cell library. Our RISC-V BMI definitions are based on an analysis and combination of existing x86 and ARMv8 BMIs. Our Rocket CPU hardware extensions show that RISC-V BMI extensions have no negative impact on the critical path of the execution pipeline. Our software evaluations show that we can, for example, expect a significant impact for time and power consuming cryptographic applications.**

***Index Terms*—bit manipulation, microprocessor, RISC-V, instruction set architecture**

## I. Introduction

Embedded systems and Internet of Things (IoT) devices in particular require a high energy efficiency and security, which are both in conflict with the performance of the system. Embedded systems are currently dominated by x86 and ARM based processors, which are in several variants highly optimized for embedded systems applications. In the past, Intel and ARM introduced several bit manipulation instructions (BMIs) for their x86 and ARMv8 instruction sets. In that context, it has been shown that BMIs are crucial for the energy efficiency and performance of embedded processors. This becomes even more relevant when targeting at IoT devices, as many of them do not integrate a Floating Point Unit (FPU). Unfortunately, typical applications still require more complex mathematical operations, like the square-root. When no FPU and no BMIs are available, those operations have to be costly emulated in software by mapping them to a set of other software instructions during compilation. Most relevant fields for advanced BMIs are machine learning and cryptographic applications as there are many algorithms that operate on binary data and often use bit rotations and shuffles. In this context, the support of more advanced BMIs can significantly reduce power dissipation of resource limited IoT devices.

As of today RISC-V [18] received a world-wide acceptance by multiple industrial and academic communities. RISC-V comes with a mature open source instruction set architecture (ISA) and several freely available synthesizable CPU implementations like the Rocket CPU [1] and PULPino [17]. Unfortunately, the official RISC-V ISA just supports two bit manipulation instructions (shift left, shift right) though the space in the ISA for the definition of more advanced BMIs has already been reserved.

This article introduces a set of ten advanced RISC-V BMIs as an extension to the current official RISC-V ISA. The BMIs are mainly derived from the existing x86 and ARMv8 BMIs. We show how to embed them into the current RISC-V ISA and present a hardware extension of the Berkeley Rocket CPU for their execution. Our logic synthesis results based on the TSMC 65nm cell library show that our BitALU seamlessly combines with the existing hardware. The software evaluations demonstrate the efficiency of our implementation for cryptographic and more complex arithmetic operations executed on a BMI extended RISC-V processor on an Artix-7 FPGA.

The article is organized as follows. Section II introduces and discusses the state of the art in related technologies. Section III compares our RISC-V BMIs with Intel's x86 instruction set. Section IV presents our RISC-V BMIs and outlines the hardware implementation before Section V presents our evaluation results. Section VI finally closes our paper with a conclusion.

## II. Related Work

Intel has introduced x86 BMIs in three subsets until 2013: ABM, BMI1, BMI2 [7]. ARMv8 BMIs were released in 2011 [15]. Our RISC-V BMIs are a selection and improvement from both. The next section has a deep analysis about the x86 BMIs and their contribution to our RISC-V BMI definitions.

Butterfly networks are an efficient way in terms of performance to implement *pext* and *pdep* as shown in [13]. Hilewitz [12] introduced the use of butterfly networks for the implementation of *rotate*, *parallel extract* (pext), and *parallel deposit* (pdep) operations. The butterfly network can also be reused in area efficient circuits for *rotate* operations as shown in [12].

Clark [5] presented an area efficient implementation for *bswap*, *brev*, *popcnt*, *ctz*, and *clz* operations. It consists of the three units *Generalized Bit Reverse*, *Set Trailing Zeros to Ones*, and *Popcount* that are shared by the five instructions.

Dimitrakopoulos et al. [8] propose an efficient *clz* implementation for floating point units.

978-1-7281-2103-1/19/$31.00 

Wolf [19] introduced several Verilog implementations for *pext*, *pdep*, and *brev* instructions together with a proposal for RISC-V BMIs [20]. For *pext* and *pdep* he introduced sequential implementations with some of them implemented by pipelining. He applied butterfly networks for his implementation. Note here that Wolf applied different names: *pext* $\rightarrow$ *bext*, *pdep* $\rightarrow$ *bdep*, *brev* $\rightarrow$ *grev*. Wolf evaluated his implementations using the Yosys [21] tool set by area and timing. He also compares his results with the Rocket CPU ALU [1].

We have defined and implemented a complete and comprehensive set of ten RISC-V BMIs. Our implementation is based on the principles of Hilewitz' butterfly network for the *pext* and *pdep* BMIs. To arrive at an optimal set, we did a comparison of x86 BMIs to RISC-V BMIs along the lines of Wolf [20].

## III. x86 BMIs

In a first step, we carefully analyzed the Intel's x86 and ARMv8 BMIs to derive a competitive RISC-V BMI set from them. Our goal was to minimize the number of target RISC-V BMIs, while still having a powerful set to express the same functionality. In that analysis we did not make any further considerations about other properties like timing or power dissipation as we were not interested in implementation details in that first step. Instead we compared the number of bytes necessary to express an x86 BMI with respect to our targeted RISC-V instruction set extension. However, to arrive at a minimal number is an important metric for energy efficiency as the fetch-and-decode logic may occupy up to 45% of the energy consumption in CPUs [10]. Our BMI definition does not include instructions that can be expressed by either less than four already defined RISC-V BMI or by regular RISC-V instructions. For example, in Table I, which gives an overview of those dependencies, the *blsi* instruction can be expressed by two regular RISC-V instructions and is thus not considered for our BMI definition.

Intel BMIs are separated into three groups [7]: ABM, BMI1, and BMI2, which we further refer to as *x86 BMIs* hereafter. In the comparison, our new RISC-V BMI extensions from Section IV are denoted as *RISC-V BMIs* in the remainder. Table I gives an overview how the x86 BMIs can be implemented in combination with existing regular and compressed RISC-V instructions where our new RISC-V BMIs are highlighted in bold. Compressed RISC-V instructions are variants of the most common RISC-V instructions only using 2 bytes for encoding. Note here that the table also includes *srl* and *sll*, which are the two already existing basic RISC-V BMIs.

### A. Advanced Bit Manipulation Instructions (ABM)

ABM consists of the two instructions *popcount* and *lzcnt*. *Popcount* counts the number of '1' bits in a register and directly maps to the RISC-V BMI *popcnt*. Likewise for *lzcnt*, which counts the leading zero bits in a register and implemented by the RISC-V BMI *clz*.

| | x86 BMI | Description | RISC-V BMI | Encoding length (byte) | |
|---|---|---|---|---|---|
| | | | | RISC-V | x86 |
| ABM | popcount | count '1' bits | **popcnt** | 4 | 5 |
| | lzcnt | count leading '0' | **clz** | 4 | 5 |
| BMI1 | andn | and not | xori, and | 6 | 5 |
| | bextr | bit extr. | **pext** | 4 | 5 |
| | blsi | extr. lowest '1' | sub, and | 4 | 5 |
| | blsmsk | mask to lowest '1' | addi, xor | 4 | 5 |
| | blsr | reset lowest '1' | addi, and | 4 | 5 |
| | tzcnt | count trailing '0' | **ctz** | 4 | 4 |
| BMI2 | bzhi | zero high bits | li, srli, **pdep** | 8 | 5 |
| | mulx | multiply | mul | 4 | 5 |
| | pdep | parallel deposit | **pdep** | 4 | 5 |
| | pext | parallel extract | **pext** | 4 | 5 |
| | rorx | rotate right | **rotr** | 4 | 6 |
| | sarx | shift right arith. | sra | 4 | 5 |
| | shrx | shift right logically | srl | 4 | 5 |
| | shlx | shift left logically | sll | 4 | 5 |

TABLE I: x86 BMIs versus RISC-V BMIs (bold).

### B. Bit Manipulation Instructions 1 (BMI1)

BMI1 defines the two instructions *bextr* and *tzcnt* that directly refer to two RISC-V BMIs. *Bextr* extracts only one bit field of a register and is a special case of the RISC-V BMI *pext*. *Tzcnt*, which counts the trailing zeros of a word directly maps to the RISC-V BMI *ctz*.

The other instructions do not directly map to our BMIs rather than to a combination of two. *Andn* implements ˜x & y for the inputs x and y. We thus need one inversion instruction and one *and* instruction, where the RISC-V *invert* is given by *xori x, x, -1*. Since *and* is also available as a compressed instruction, we can express this with macro-op fusion by a single 6 byte instruction using 4 bytes for *xori* and 2 bytes for *and*. *Blsi*, which extracts the lowest '1' bit, can be expressed by x & -x with one *negate* and one *and* instruction. *Negate* is a pseudo instruction that can be implemented by *sub x, 0, x*. This can be emulated by a single 4 byte instruction since sub and *and* are both available as compressed instructions. The remaining two BMI1 instructions, *blsmsk* and *blsr*, mask up to the lowest '1' bit and reset the lowest '1' bits. This is expressed by x ˆ (x - 1) and x & (x - 1), respectively. For their RISC-V emulation we need one *addi* instruction to implement (x - 1) for *blsmsk* and *blsr*. Additionally, we need a *xor* instruction for the former and a *and* for the latter. All three RISC-V instructions are available as compressed variants.

### C. Bit Manipulation Instructions 2 (BMI2)

BMI2 was introduced in 2013 by Intel. In their definitions, all the instructions ending by *x*, like *mulx* or *sarx* compare to regular *mul* or *sar* instructions without raising exception flags. Since RISC-V does not have exception flags, we can basically ignore them. Correspondingly, *rorx*, which rotates a word right, compares to RISC-V BMI *rotr*. The same holds for *pdep* and *pext*. *bzhi* copies the source operand into the destination and clears the highest bits of the destination starting from the index operand. We can therefore use the *pdep* instruction for its implementation as it clears all bits except the ones that are deposited. However, we have to create the mask for *pdep* that has all bits starting from the index set to '0'. We achieve this

by first applying *li tmp, -1*, which sets all bits of a temporary register to 1 and then *srli* to shift it by the index amount to the right. We need a 4 bytes encoding for both of them since both, *li* and *srli*, are available as compressed instructions. Another 4 bytes are used by the *pdep* instruction leaving a total of 8 bytes.

### D. Discussion

We can see that all the x86 BMIs can be expressed by either the basic RISC-V instructions or by our new RISC-V BMIs. They are either direct mappings or they can be emulated by at most three RISC-V instructions. Nevertheless, due to the compressed RISC-V instructions, we need only 4 bytes to express all except two x86 BMIs. However, those 4 byte instruction pairs can be efficiently implemented using macro-op fusion as shown by Ceilio et al. [4]. There are two exceptions: *andn* and *bzhi*. The former needs 6 bytes but can still be efficiently implemented by macro-op fusion. The latter needs 8 bytes which is mainly due to the need of the *pdep* instruction. In summary, the RISC-V BMIs need 4.375 bytes per instruction on average to implement x86 functionality, compared to 5 bytes per instructions of x86.

### E. Conclusion

We saw that our RISC-V BMIs can efficiently express the x86 BMIs while requiring 13.5% less bytes. With one exception, all x86 BMI can be either expressed by one or two RISC-V instruction, which supports an efficient implementation by means of macro-op fusion. In conclusion, our BMIs are as expressive as the x86 BMIs while requiring less instructions and less bytes per instruction. Our RISC-V BMIs also almost match with ARMv8 BMIs. We just preferred to separate the implementation of the *bit field move* into *pext* and *pdep*.

## IV. RISC-V Bit Manipulation Instructions

The definition of our RISC-V BMIs inherits concepts from x86 BMIs and some from ARMv8 while eliminating instructions that can be emulated by less than four instructions as it is discussed in Section III. Following those lines, we defined and implemented ten RISC-V BMIs: PARITY, BSWAP, ROTL, ROTR, POPCNT, CLZ, CTZ, BREV, PEXT, PDEP. Table II gives an overview of their encoding. All instructions were implemented using the Chisel [3] hardware generation language and combined into one BitALU. The BitALU was finally synthesized and integrated into the RISC-V Rocket CPU [1].

### A. Parity (PARITY)

The parity instruction checks if the number of '1' bits is even or odd. If the number is odd the result is '1' and '0' otherwise. For example, PARITY with 8-bit input '10101100' is '0'. We implemented *parity* by an xor-reduce on the input. The xor-reduce of an n-bit word is $x_0 \hat{} ... \hat{} x_n$.

### B. Byte Swap (BSWAP)

*BSWAP* converts data from little-endian to big-endian byte ordering or vice versa. For example, 0x89ABCDEF is composed of four bytes: 0x89, 0xAB, 0xCD, 0xEF. *BSWAP* reverses their order to 0xEFCDAB89. We implemented *byte swap* by a simple rewiring of bytes.

### C. Rotate Left and Rotate Right (ROTL, ROTR)

*Rotate left* and *right* define ring shift operations for left and right byte shifts. For example, if we rotate the 8-bit word '10110000' by two to the left, the result is '11000010'. We implemented this by a tree of multiplexers (see Figure 1 for *rotl*) where each bit of the shift amount is the select bit for the multiplexers of this stage of the tree. If the first bit of the shift amount is set, we rotate by 1, for the second bit by 2, third by 4, etc.

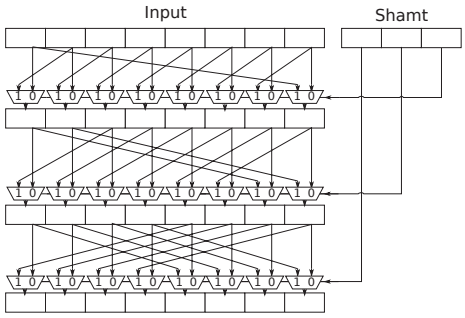



Fig. 1: An example circuit for an 8-bit left shifting barrel shifter implemented as a tree of multiplexers.

### D. Popcount (POPCNT)

*Popcount* counts the number of '1' bits. This is often known as a population and thus denoted as popcount. For example, POPCNT for the 8-bit input '10110000' is 3. We implemented *popcount* by an add-reduce on the input. The add-reduce of a n-bit word $x$ is $x_0 + ... + x_n$.

### E. Count leading and trailing zeros (CLZ, CTZ)

*Count leading zeros* counts the number of contiguous zero bits in the input word starting from the MSB. Count trailing zeros counts starting from the LSB correspondingly. For example, CLZ of the 8-bit input '00001000' is 4. We implemented *count leading zeros* by the following divide and conquer algorithm.

```
1  if (msb(left) = 0)
2      result = '0'[left]
3  else if (msb(left) = 1 and msb(right) = 1)
4     result = '1'[fill('0', len(right)]
5  else
6     result = '01'[right(msb-1, 0)]
```

The example in Figure 2, shows the operation with the 8-bit input '00000111'. The input is divided into two bit pairs. Leading zeros are then counted and encoded for each pair separately. For instance, '00' is encoded as '10' for two leading zeros. Afterwards we assemble the results of each pair.

| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type |
| 1000000 | rs2 | rs1 | 001 | rd | 1101011 | BREV |
| 0000000 | 00000 | rs1 | 110 | rd | 1101011 | BSWAP |
| 1000000 | 00000 | rs1 | 111 | rd | 1101011 | CLZ |
| 0000000 | 00000 | rs1 | 111 | rd | 1101011 | CTZ |
| 0000000 | 00000 | rs1 | 100 | rd | 1101011 | PARITY |
| 0000000 | rs2 | rs1 | 010 | rd | 1101011 | PDEP |
| 1000000 | rs2 | rs1 | 010 | rd | 1101011 | PEXT |
| 0000000 | 00000 | rs1 | 011 | rd | 1101011 | POPCNT |
| 1000000 | rs2 | rs1 | 000 | rd | 1101011 | ROTL |
| 0000000 | rs2 | rs1 | 000 | rd | 1101011 | ROTR |

TABLE II: Encoding of all RISC-V BMIs.

Considering Figure 2 on the top left as an example: ’00’ and ’00’ encodes to ’10’ and ’10’. As the MSBs of both are ’1’ we are in the case of line 3, thus result = 1’[fill(’0’, 2)] = ’100’ = 4. The leading ’1’ in both cases indicates, that both pairs have only zeros. Assembling both applies a multiplication by 2, which can also be implemented by a shift of 1 to the left.

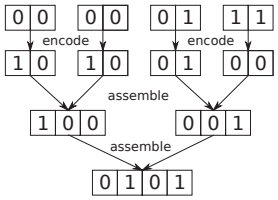



Fig. 2: CLZ/CTZ implementation

The encoded pairs ’01’ and ’00’ on the right branch match msb(left) = 0, which means that we are in the case of line 1, thus result = ’0’[left]. The leading zero in the encoding of the left side indicates that there is a ’1’ in the left pair, so the number of leading zeros of both pairs is the number of leading zeros of the left. So we change nothing on the left encoding rather bring it to the right size by adding a leading ’0’. Finally assembling the last pairs ’100’ and ’001’, we apply the last line in the above algorithm: result = ’01’[right(1,0)] = ’0101’ = 5. We know from the leading ’1’ on the encoding of the left side that the left side only contains zeros thus we need to preserve that information by keeping the leading ’1’ of the encoding at the same position and adding a leading ’0’ to that resulting in ’01’. By the leading ’0’ of the encoding of the right side, we know that there is a ’1’ somewhere in the right side. Since the leading ’0’ of the encoding holds no information, we can cut it and add the remainder of the right side to the result. Count trailing zeros applies the same algorithm with reversed input, which converts trailing zeros into leading zeros.

### *F. Bitreverse (BREV)*

*BREV* implements a generalized bit reverse instruction that allows fine-grained reverse operations on different packs of bits. The operation is controlled by the contents of a shift register. For instance, if the LSB of the shift register is set, every pair of packs of one bit of the data input register is swapped. For bitstring ’1010’ this is ’0101’, for example. If the second LSB of the shift register is set, every pair of packs of two bits is swapped. For ’11000011’ the result is ’00111100’ etc. Note here, that the conventional bitreverse, that reverses the order of all bits, is just a special case, where the shift register is equal to 31 (for 32 bit inputs). If more than one reverse bits are set, all the single bit reverse operations are applied.

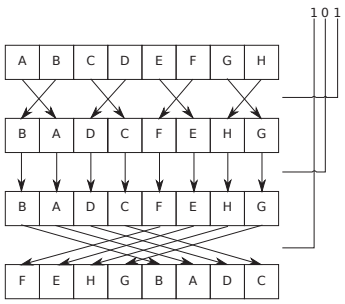
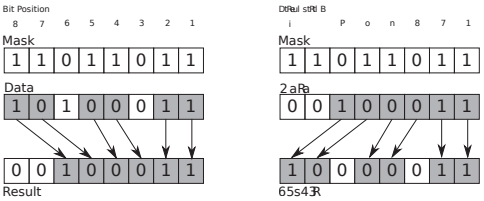



Fig. 3: Example operation of the generalized bit reverse unit for the reverse amount of 5.

The semantics of the *bitreverse* instruction matches to an inverse butterfly network, which was chosen for its implementation. Figure 3 shows an example for an 8-bit *input* and 5 for *reverse amount*. We see that each bit of the reverse amount maps to one stage of the inverse butterfly network. The LSB is mapped to the first stage, the second LSB is mapped to the second stage, and so on.

### *G. Parallel Gather and Scatter (PDEP, PEXT)*

*Parallel gather* and *scatter* was originally introduced in [13]. Parallel gather, also known as parallel extract (*PEXT*) has two inputs, a data word, and a mask. The mask defines which bits to extract from the data word. Figure 4a shows an 8-bit example, where all the ’1’ bits in the mask are selected in the data word and inserted into the corresponding position of the result. For example, the third ’1’ in the mask is at bit position 4 and the corresponding data element at that position is ’0’. Since it is the third ’1’ of the mask, the data element ’0’ is put into bit position 3 in the result. Therefore, it is also called a parallel extract because it extracts chunks of bits as specified by the ’1’ bits in the mask.




(a) 8-bit parallel gather.

(b) 8-bit parallel scatter.

Fig. 4: pdep and pext operation examples with 8-bit input.

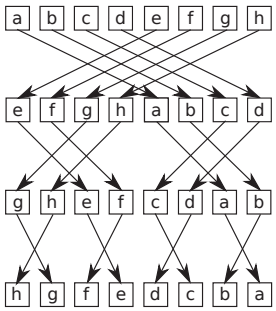
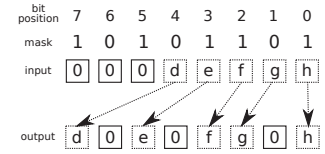



Fig. 5: 8-bit butterfly network.

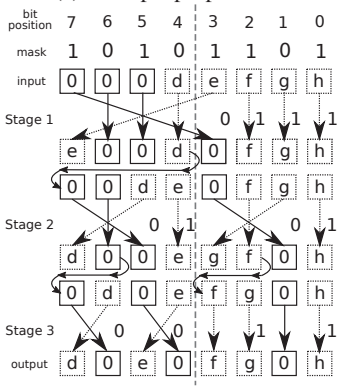
Parallel scatter, also known as parallel deposit (*PDEP*) does exactly the inverse operation. In contrast to *PEXT*, just data and the result are exchanged. Figure 4b shows an 8-bit example. Here, the '1' bits in the mask represent positions in the result word. If we consider the third '1' bit of the mask at bit position 4, we see that the third bit of the data word is deposited into position 4 of the result. Therefore, it is also called parallel deposit because it deposits chunks of bits into the result as specified by the 1's in the mask. We implemented *PEXT* and *PDEP* by butterfly networks [14]. In the remainder we focus on the *PDEP* implementation. *PEXT* is implemented correspondingly with reverse dataflow.

A butterfly network is a very powerful and highly configurable bit operation architectural structure that can be used to permute $2^n$ data bits through n stages with low latency. Figure 5 outlines the example of a 8-bit butterfly network. The top indicate the network input bits given by *a–h*. The network shuffles these bits in three different steps, called stages. In the first stage it splits the input by half exchanges their positions, i.e., *a–d* take the positions of *e–*h and vice versa. In the next stage each of the exchanged groups is again split by half and shuffled, i.e., *e–f* with *g–h* and *c–d* with *a–b*, and so on. In general, a butterfly network with an input width of $n$ requires $log_2(n)$ stages. Each bit can also either stay in its position or move to the other half in the next stage. However, if one bit changes positions, its partner in the other half also has to switch. Whether a bit switches position or stays in place can be configured on a per-bit basis in each stage separately.

Figure 6a shows an example for the *PDEP* operation on an 8-bit mask and an 8-bit input word. Since our bitmask contains five '1' bits, only the least significant 5 bits are moved by the *pdep* operation. Thus, they are enumerated using the letters *d–h* and moved to the output to their corresponding positions marked by '1' bits in the mask. Figure 6b shows how this *pdep* is executed on a butterfly network and is broken down into *stages*. In stage 1 we determine the elements in the right half of the input that are in their correct half (*f*, *g*, and *h* in Figure 6a). All others are moved to the left half and the corresponding pair in the left half is moved to the right. In our example this is *e*. We can easily determine the bits that stay in the right half by counting '1' bits in the mask corresponding to the right half. Here we count 3 bits, which means the least significant 3 bits have to stay in this half, which are indeed *f*, *g*, and *h*. Hereby, we can compute the control signals for each stage that indicate whether a bit ought to be swapped. In Figure 6b they are written next to the arrows in the right half in between the first and second stage. A control bit of '0' indicates here that this bit should be moved into the other half and a '1' bit means this bit stays in its half.




(a) 8-bit pdep operation.




(b) PDEP on a butterfly network using explicit rotations.

Fig. 6: Data flow of an 8-bit pdep operation through a butterfly network, with and without explicit rotation [13].

For stage 2, we can see each of the halves of stage 1 as their own 4 bit butterfly network and repeat this process. However, simply counting 1's in these new right halves is not sufficient anymore, since we changed the bit order when we moved *e* to the left half in stage 1. Originally the order was *defgh* (from MSB to LSB), but now it is *edfgh*. Therefore, we cannot easily determine whether a bit is in the correct half or not. For compensation, we right-rotate the left half of stage 1 by the offset it was moved.

Figure 7 illustrates this process. We have three blocks of bits, *X*, *Y*, and *Z* that are input to stage 1. *X* starts in the right half and stays in the right half, *Y* starts in the right half and moves to the left half, and Z starts in the left half and stays in the left half. In the intermediate stage (the middle) we see that *Y*'s relative position to *X* is offset to the left by the length of

*X* and thus leads the wrong order. We correct this by rotating the left half containing *Y* and *Z* to the right by the length of *X*.

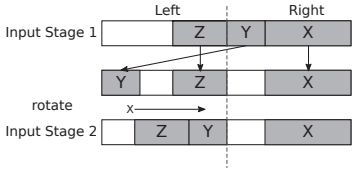



Fig. 7: An example for the rotation operation to restore the original order of bits [13].

After rotation we can repeat the process of stage 1 by looking at the right half of the right half, and the right half of the left half. For the latter case, we have the bits *de* and we count the corresponding '1' bits in the mask at bit the positions 4 and 5, which is 1. Therefore, the LSB of this half (*e*) stays in its position and the other bit is moved to the left half at the bit position 7. The former case is handled likewise, we count the mask bits at the positions 0 to 1, which is 1 and thus the LSB *h* stays in position and the other one switches position with the bit at the position 3. Afterwards we have to rotate both left halves of the 4-bit butterfly network to compensate for the order. Our offset is in both cases 1, since 1 bit stayed in position which forms our offset. We repeat this process one last time for the four 2-bit butterfly networks in stage 3. For our final implementation we used the improved version using implicit rotations as described in [13].

## V. Evaluation

We combined all hardware implementations for the ten RISC-V BMIs and integrated them into a BitALU. It was implemented by means of the hardware constructions language Chisel [3]. The BitALU was integrated into the Rocket CPU [1] for hardware synthesis and our software evaluation. The software benchmarks were executed on a BMI enhanced Rocket CPU, which was synthesized for an Xilinx Artix-7 FPGA. BitALU area consumption and timing numbers were derived from logic synthesis of the BitALU with the TSMC 65nm cell library.

### A. Software Evaluation

This section presents measurements of software runtime performance of our RISC-V BMIs. After the test setup we introduce our test benchmarks, the manual BMI replacement strategies, and the evaluation results.

*1) Test Setup:* We tested the same program with RISC-V BMIs replaced and compared it with the version before their replacement. We compiled each test in both cases using the freedom-e-sdk[1] into an "ELF" executable file. We then uploaded the compiled tests onto the Rocket CPU running on the Digilent Arty 35T board with an Xilinx Artix-7 FPGA. We compiled any input that is required for the tests to run as an array into the binary. Whenever a test required random input, we generated the input using an *input generator* once per test run and used the same random inputs for the runs with and without BMIs. We applied inline assembly to improve manually selected areas and replacing instructions with our BMIs. Additionally, we augmented each test with reads to the *mcycle* register. To measure time, we compared the number of cycles when the test started with the number after the test has completed, finally calculating the difference between both.

*2) Tests:* To evaluate the instruction performance, we applied tests from the MiBench [9] benchmark suite, which targets embedded systems. We augmented the software binary and manually replaced instructions by our RISC-V BMIs at specific locations with high potential for improvements and compared the cycles it took to complete the tests with an unmodified Rocket CPU. Table III gives an overview over which BMI is inserted in which test.

| **rotl** | **bswap** | **brev** | **clz** | **ctz** | **pext** | **pdep** | **popcnt** |
|---|---|---|---|---|---|---|---|
| sha | sha | fft | isqrt | gcd | m_dec | m_enc | bitcnt |
| norx | norx | — | — | — | — | — | — |

TABLE III: Overview of tests and applied BMIs.

The following briefly introduces the individual tests and describes how we applied our BMIs before outlining the evaluation results.

*sha* is a test from MiBench that computes the secure hash algorithm using the ASCII representation of an article that was provided by MiBench, as an input. The test expects the input data to be in the big-endian format. Thus it starts by reversing all the input data into the little-endian format which applied the *bswap* instruction. The compression function of this algorithm requires two inputs to be rotated left for which we applied the *rotl* instruction.

*norx* is a security algorithm [2] that was designed as an authenticated encryption scheme supporting associated data (AEAD). Norx uses almost byte-aligned rotation to which we applied the *rotl* instruction. This algorithm also expects the input data to be in big-endian format, therefore we used the *bswap* instruction for conversions to little-endian.

*fft* is a Fast Fourier Transformation implementation, which uses one pseudo-random wave as the input. This test is part of MiBench and uses its pseudo random generator. FFT uses bit reverse addressing to which we applied the *brev* instruction to calculate addresses efficiently.

*isqrt* implements an integer square root function based on Reza Hashemians square rooting algorithms [11]. This algorithm has two steps. Firstly, it makes an initial guess of the square root and secondly, uses an iterative procedure that improves the initial guess. The algorithm needs to determine the number of effective bits that the input to the square root function actually uses. For example, the 8-bit input '00010110' effectively uses 5 bits. This is $8 - clz(00010110)$ where we applied our *clz* instruction.

*gcd* implements the greatest common divisor for two integers $u$ and $v$ by the binary method. If $u$ and $v$ are both even, then $gcd(u, v) = 2 \cdot gcd(u/2, v/2)$ is recursively applied

[1] https://github.com/sifive/freedom-e-sdk

until $u$ and $v$ are not even anymore. Here, division by 2 is represented by a binary shift right by 1. To check if both are even, we check whether the LSBs of $u$ and $v$ are 0. After one iteration and one shift on $u$ and $v$ we determine again if the LSBs are 0 and so on. Here, we applied the *ctz* instruction of the result to obtain the overall number of shifts for $u$ and $v$. After this initial step at least one of the inputs is odd. However, the other input can still be even and we can repeat the above procedure with the single even input. In this context, we also applied this to $v$.

*morton* codes [16] map multidimensional data into a single dimension and order them at the same time. We used two tests called *morton_enc* and *morton_dec*. Once multidimensional data is encoded into morton codes and once a morton code is decoded into multidimensional data again.

To propagate the bits of each dimension to the correct position in the result, we first applied *pdep* instruction and the bitwise *or*-operation to each dimension's result. In order to decode the morton codes we applied the *pext* instruction for software acceleration.

*bitcnt* is part of MiBench and tests the ability of a processor to count the number of bits in a word and comes with six different algorithms: *bitcnt_1* to *bitcnt_6*. They range from naive loop-based approaches to sophisticated table lookups. We added another algorithm for our *popcnt* instruction. The inputs to each of the algorithms are integer numbers with an equal number of '1' and '0' bits.

*3) Results:* The summary of our software evaluation are shown in Table IV. The smallest speedup of 1.09x was for *fft* since the bit reverse addressing is only a small portion of the overall FFT. The encoding and decoding algorithms like *Sha* and *Norx* show higher speedups (1.14x and 1.43x) since every input word needs to be byte-swapped, which accounts to a larger portion of the program. This category is thus a candidate for further investigation. Arithmetic operation replacements in *Isqrt* and *gcd* show with 2.04x and 1.79x higher accelerations as we could eliminate many loops that are used to count trailing/leading zeros and individual 1-bit shifts could be combined by one shift. *Isqrt* is an adequate benchmark for embedded microcontrollers as they rarely contain a floating point unit with square root operation support.

We achieved with 4.08x and 4.16x a high speedup for the bit en/decoding intensive *morton* benchmarks as they are both massive applications for *pext*/*pdep*. Those en/decodings are especially useful for graphical applications, e.g., to construct Octrees. As expected, the *bitcnt* tests achieved the highest speedup for *bitcnt_2* and *bitcnt_3* (96.87x and 96.71) which are the "recursive-" and "non-recursive bit count by nybbles algorithm", respectively. Counting bits fast is useful for sparse matrices implementations or for Hamming distance computations, for instance.

It is important to note that we manually replaced instructions by our BMIs in a first step in all tests. By the means of a compiler, which is not available yet, we could automatically detect BMI application patterns, which certainly gives significantly better results with the same tests.

| Test | with BMIs | without BMIs | Speedup |
|---|---|---|---|
| sha | 460798 | 524619 | 1.14x |
| norx | 224312 | 320454 | 1.43x |
| fft | 6461039 | 7050396 | 1.09x |
| isqrt | 239433 | 489082 | 2.04x |
| gcd | 27082 | 48414 | 1.79x |
| morton_enc | 4453 | 18182 | 4.08x |
| morton_dec | 4434 | 18429 | 4.16x |
| bitcnt_1 | 29324 | 61562 | 2.10x |
| bitcnt_2 | 29324 | 2836002 | 96.71x |
| bitcnt_3 | 29324 | 2840620 | 96.87x |
| bitcnt_4 | 29324 | 1460955 | 49.82x |
| bitcnt_5 | 29324 | 1467306 | 50.04x |
| bitcnt_6 | 29324 | 293324 | 10.00x |

TABLE IV: Execution times (in cycles) for all tests with and without BMIs.

### B. ASIC

We evaluated the efficiency of our BitALU RISC-V extension by standard cell logic synthesis in terms of the timing and area consumption. For this we applied the Synopsys Design Compiler with the TSMC 65nm cell library.

*1) Timing:* Table V shows the logic synthesis results comparing the BitALU with the regular ALU of the Rocket CPU.

| Unit | LDP (ns) | Ratio |
|---|---|---|
| ALU(reference) | 2.17 | 1.0 |
| BitALU | 2.04 | 0.94 |

TABLE V: Critical path delay of the Rocket CPU ALU and our BitALU.

Results shows that the critical path of our BitALU is not longer than the path of the Rocket CPU. This indicates that the BitALU can be easily integrated into the Rocket CPU since the BitALU logic will not enforce a lower clock frequency. This is because both ALUs are in the same pipeline stage and the maximum clock frequency is limited by the longest critical path. This met our expectation as the critical path of the regular ALU is dominated by a carry chain of length n, while all of the implementations of the BMIs use tree structures with $log(n)$ length. The main contributor to the critical path in our BitALU are the butterfly network and the complex decoder for the *pext* and *pdep* instructions.

*2) Area:* We also compared the area of our BitALU components with the area of the Rocket CPU ALU by synthesizing both implementations using the Synopsys Design Compiler [6] with TSMC 65nm and used the estimated area results report. The numbers gives a first indication of the area allocation of our first hardware implementation in contrast to the existing Rocket CPU ALU and MulDiv unit.

The current synthesis results are given in Table VI which show that the BitALU requires 3.05x the amount of gates with 5795, compared to the reference ALU with 1898 gates. As expected, the GatherScatter logic with 3265 gates takes the majority of BitALU area. This is due to the comparably complex logic of the butterfly network, which includes many multiplexers and is optimized for timing.

However, the area of the BitALU is in the same order as the multiplication/division unit (MulDIv) of the Rocket CPU. This area allocation seems to be a reasonable price for its significantly higher performance. Nevertheless, we still see a lot of potential for further improvements during physical design.

| Unit | Gates | Ratio |
|---|---|---|
| Rocket ALU | 1898 | 1.0 |
| Rocket MulDiv | 5737 | 3.00 |
| BitALU (total) | 5795 | 3.05 |
| GatherScatter | 3265 | 1.72 |
| CountReverse | 1354 | 0.71 |
| BarrelShifter | 935 | 0.49 |

TABLE VI: Area evaluation in gates using Synopsys Design Compiler with the regular ALU as a reference.

### C. FPGA

We evaluated the BitALU using the Xilinx Vivado tool chain. Unfortunately, it was impossible to identify the area of the Rocket CPU ALU, since it was heavily optimized and did not appear in the generated area report. This also prevents us to compare the timings. The area of the BitALU turned out not to be significant with a total of 84 lookup tables (LUTs). In comparison, the complete Rocket CPU including all peripherals and buses takes 15184 LUTs, i.e., BitALU takes less than 1% of the area. Even if we just consider the Rocket CPU by itself, which allocates 3052 LUTs, the BitALU still only contributes less than 3% to the total area.

## VI. Conclusion

We extended the RISC-V ISA by ten bit manipulation instructions: *parity*, *byte swap*, *rotate right/left*, *popcount*, *bitreverse*, *count leading/trailing zeros*, and *parallel gather/scatter*. Those BMIs implement the same functionality as current x86 BMIs while requiring 13.5% less encoding bytes. To proof their efficiency, we extended the RISC-V Rocket CPU instruction decoder for their execution. We extended the GNU assembler for their manual insertion and evaluation by 13 different benchmarks. The benchmarks cover a wide range of applications from arithmetic operations to de- and encoding algorithms. The evaluation was executed on the BMI extended Rocket CPU that was synthesized for the Xilinx Arty-7 FPGA. We achieved a speedup for all the tested applications with a minimum of 1.09x for the *fft* benchmark and a maximum of 96.87x for one bitcount test of the MiBench benchmark. However, those evaluations just give a first indicator as we only manually applied instruction replacements at dedicated locations. A BMI enhanced software compiler can certainly achieve much better results. Our logic synthesis results based on the TSMC 65nm cell library show that our BitALU can be seamlessly integrated into the RISC-V Rocket CPU without any performance impact and reasonable area requirements.

## Acknowledgment

The work described herein is partly funded by the Bundesministerium für Bildung und Forschung (BMBF) through the COMPACT Project (No. 01IS17028) and the Safe4I Project (No. 01IS17032).

## References

[1] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[2] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. Norx: parallel and scalable aead. In *European Symposium on Research in Computer Security*, pages 19–36. Springer, 2014.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[4] Christopher Celio, Palmer Dabbelt, David A Patterson, and Krste Asanović. The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v. *arXiv preprint arXiv:1607.02318*, 2016.

[5] Michael J. Clark. Candidate instructions for b extension (bit manipulation). https://github.com/michaeljclark/rv8/blob/master/doc/src/bmi.md. Accessed: 2018-04-19.

[6] Design Compiler. Synopsys inc, 2000.

[7] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual*, volume 2. September 2016.

[8] Giorgos Dimitrakopoulos, Kostas Galanopoulos, Christos Mavrokefalidis, and Dimitris Nikolos. Low-power leading-zero counting and anticipation logic for high-speed floating point units. *IEEE transactions on very large scale integration (VLSI) systems*, 16(7):837–850, 2008.

[9] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[10] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 37–47. ACM, 2010.

[11] Reza Hashemian. Square rooting algorithms for integer and floating-point numbers. *IEEE Transactions on Computers*, 39(8):1025–1029, 1990.

[12] Yedidya Hilewitz. *Advanced bit manipulation instructions: Architecture, implementation and applications*. Princeton University, 2008.

[13] Yedidya Hilewitz and Ruby B Lee. Fast bit compression and expansion with parallel extract and parallel deposit instructions. In *Application-specific Systems, Architectures and Processors, 2006. ASAP'06. International Conference on*, pages 65–72. IEEE, 2006.

[14] Yedidya Hilewitz and Ruby B Lee. Fast bit gather, bit scatter and bit permutation instructions for commodity microprocessors. *Journal of Signal Processing Systems*, 53(1-2):145–169, 2008.

[15] ARM Limited. *ARM Architecture Reference Manual*. Number ARM DDI 0487C.a ID121917. December 2017.

[16] Guy M Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966.

[17] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K Gurkaynak, and Luca Benini. Pulpino: A small single-core risc-v soc. In *RISC-V Workshop*, 2016.

[18] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[19] Clifford Wolf. Reference hardware implementations of bit extract/deposit instructions. https://github.com/cliffordwolf/bextdep/. Accessed: 2018-04-19.

[20] Clifford Wolf. *RISC-V XBitmanip Extension*, volume 0.34-draft. Accessed: 2018-04-19.

[21] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.