Design and Synthesis of RISC-V Bit Manipulation Extensions

Kevin Kim
Department of Engineering
Harvey Mudd College
Claremont, CA 91711, USA
kekim@hmc.edu

David Harris
Department of Engineering
Harvey Mudd College
Claremont, CA 91711, USA
david_harris@hmc.edu

Kip Macsai-Goren
Department of Engineering
Harvey Mudd College
Claremont, CA 91711, USA
kmacsaigoren@hmc.edu

2023 57th Asilomar Conference on Signals, Systems, and Computers | 979-8-3503-2574-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/IEEECONF59524.2023.10477073

***Abstract*—This paper presents the design, verification, and synthesis results of a bit manipulation unit on a RISC-V processor. We assess area and timing tradeoffs of the additional hardware on a configurable, open-source processor. Supporting bit manipulation has no impact on the critical path of an application processor. It adds 1-2.5% to total area, or less than 1% without carry-less multiplication.**

Keywords—RISC-V, Bit Manipulation

I. Introduction

Most computer architectures offer bit manipulation instructions (BMI). RISC-V recently ratified BMI extensions for address generation (Zba), basic bit manipulation (Zbb), carry-less multiplication (Zbc), and single bit operations (Zbs) [1] to increase performance and reduce code size. For example, Babu [2] found the BMI extensions speed up Embench 4-8% while reducing code size by 1.8%. Table I summarizes the RISC-V BMI extensions.

Several teams have published bit manipulation implementations. Koppelman [3] describes adding early draft BMI support to the Rocket CPU. Many of these instructions are not in the ratified spec. Meng [4] describes a FPGA implementation of adding BMI support to a RV64IMC processor at the cost of 30 kGE in area. The implementation details are sparse. Markov [5] finds that adding BMI to the schoolRISCV core increases FPGA logic cell usage by 10x, which is not representative of an efficient implementation. Jain [6] describes another FPGA implementation but does not provide details or quantify the extra cost of BMI support.

This paper provides the first detailed implementation of the ratified RISC-V BMI extensions and quantifies the area and speed impacts in a 28 nm technology. The extensions are added to the configurable CORE-V Wally open-source processor [7]. The processor passes the RISC-V architecture compatibility test suite [8] and has 94% functional test coverage.

II. Design

Fig. 1 shows a block diagram of CORE-V Wally using 5 pipeline stages. The BMI are handled in the integer ALU during the Execution stage. Fig. 2 shows a diagram of the ALU with the optional enhancements for bit manipulation highlighted in different shades of blue and grey. The ALU operates on SrcA and SrcB to produce ALUResult. The major

TABLE I. RISC-V Bit Manipulation Instructions

Instruction	Description
Zba: Address Generation	
`sh1add`	shift left by 1 and add
`sh2add`	shift left by 2 and add
`sh3add`	shift left by 3 and add
Zba (RV64 only): Shift and Add 32-bit Word	
`sh1add.uw`	shift unsigned word left 1, add
`sh2add.uw`	shift unsigned word left 2, add
`sh3add.uw`	shift unsigned word left 3, add
`add.uw`	add unsigned word
`slli.uw`	shift left logic immediate unsigned word
Zbb: Basic Bit Manipulation	
`min`	minimum
`minu`	unsigned minimum
`max`	maximum
`maxu`	unsigned maximum
`orn`	or inverted operand
`andn`	and inverted operand
`xnor`	xor inverted operand
`rol`	rotate left
`ror`	rotate right
`orc.b`	bitwise OR combine on bytes
`zext.h`	zero-extend halfword
`clz`	count leading zeros
`cpop`	population count
`ctz`	count trailing zeros
`sext.b`	sign-extend byte
`sext.h`	sign-extend halfword
`rev8`	byte-wise reverse
`rori`	rotate right immediate
Zbb (RV64 only): Basic Bit Manipulation on 32-bit words	
`rolw`	rotate left word
`rorw`	rotate right word
`roriw`	rotate right immediate word
`clzw`	count leading zeros word
`cpopw`	population count word
`ctzw`	count trailing zeros word
Zbc: Carry-less Multiplication	
`clmul`	carry-less multiply (lower half)
`clmulh`	carry-less multiply (upper half)
`clmulr`	carry-less multiply reversed
Zbs: Single-Bit Operations	
`bclr`	bit clear
`binv`	bit invert
`bset`	bit set
`bext`	bit extract
`bclri`	bit clear immediate
`binvi`	bit invert immediate
`bseti`	bit set immediate
`bexti`	bit extract immediate

changes within the ALU for each bit manipulation extension are given below:

- Zba adds a preshifter to shift A left by 0 to 3 bits. RV64 (cross-hatched in Fig. 2) conditionally zero-extends SrcA for unsigned-word instructions.
- Zbb adds a block for min/max, counting, extend, combine, and byte reversal instructions. The shifter is also enhanced to support rotations.
- Zbc adds a carry-less multiplication block.
- Zbs adds a decoder to produce a one-hot bit mask from SrcB and a reduction OR tree for bit extract.

When any of the bit manipulation extensions are supported, the ALU uses a BSelect multiplexer to choose ALUResult from the ALUSelect output or the extra bit manipulation units. The remaining subsections describe the implementation of each of these extensions in detail.

A. Zba: Address Generation Design

The Zba extension adds a preshifter on SrcA to support shift-and-add instructions (`sh1add, sh2add, sh3add`). RV64 Zba also adds W64-type instructions that require the unsigned-word of SrcA (`sh1add.uw, sh2add.uw, sh3add.uw, slli.uw, add.uw`). These instructions are handled by adding a zero extender to SrcA and a two-input select mux.

B. Zbb: Basic Bit Manipulation Design

The Zbb extension adds a *zbb* block (Fig. 3) to perform min/max (`min, minu, max, maxu`), extension (`sext.b, sext.h, zext.h`), counting, (`cpop/cpopw, clz/clzw, ctz/ctzw`), combining (`orc.b`), and byte reversal (`rev8`). It enhances the shifter to support rotates (`rol/rolw, ror/rorw, rori/roriw`). It uses the existing inverter on the B input to support inverted logical operations (`orn, andn, xnor`).

The existing funnel shifter forms a 2XLEN-1 bit word Z and right-shifts it by k [9]. The Z source generation mux is enhanced to support rotate instructions as shown in Tables II and III.

TABLE II. ENHANCED 64-BIT SHIFTER SOURCE GENERATION LOGIC

Shift Type	*Z*	*Shift Amount*
`srl`	{63'b0, $A_{63:0}$}	k
`sra`	{{63{A_{63}}}, $A_{63:0}$}	k
`sll`	{$A_{63:0}$, 63'b0}	$\sim k$
`srlw`	{95'b0, $A_{31:0}$}	k
`sraw`	{64'b0, {31{A_{31}}}, $A_{31:0}$}	k
`sllw`	{32'b0, $A_{31:0}$, 63'b0}	$\sim k$
`ror / rorw`	{$RotA_{62:0}$, $RotA_{63:0}$}	k
`rol / rolw`	{$RotA_{63:0}$, $RotA_{63:1}$}	$\sim k$

TABLE III. ENHANCED 32-BIT SHIFTER SOURCE GENERATION LOGIC

Shift Type	*Z*	*Shift Amount*
`srl`	{31'b0, $A_{31:0}$}	k
`sra`	{{31{A_{31}}}, $A_{31:0}$}	k
`sll`	{$A_{31:0}$, 31'b0}	$\sim k$
`ror`	{$A_{30:0}$, $A_{31:0}$}	k
`rol`	{$A_{31:0}$, $A_{31:1}$}	$\sim k$

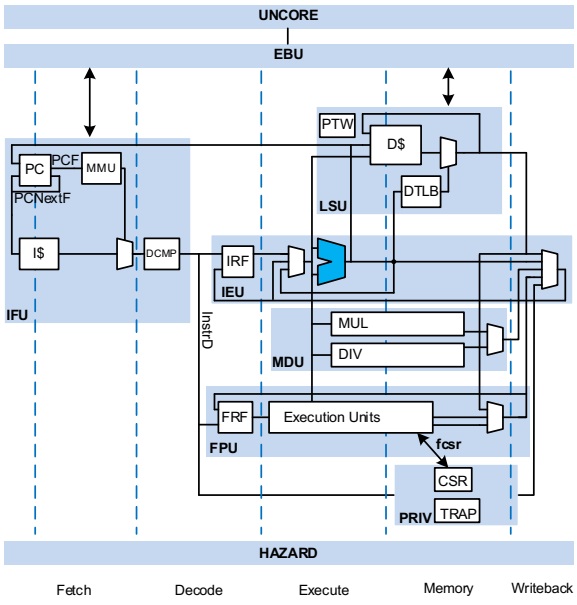

Fig. 1. RISC-V Wally high level block diagram emphasizing the ALU with bit manipulation extensions

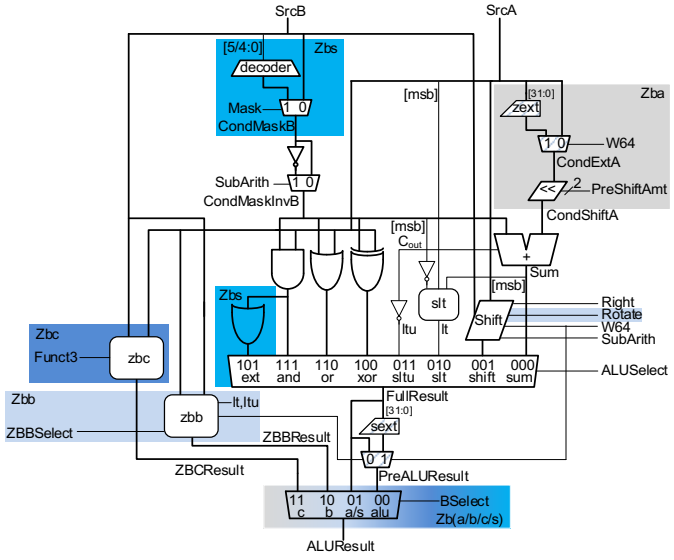

Fig. 2. ALU showing optional enhancements for bit manipulation

For inverted operations (`orn, andn, xnor`), the ALU controller drives SubArith = 1 to perform A op $\bar{B}$. No new datapath hardware is required.

Fig. 3 shows the *zbb* module that handles the other basic BMIs. Note that for I-type instructions, SrcB contains the immediate, whose lsbs may be used as control signals. The min/max (`min, minu, max, maxu`) operations select either SrcA or SrcB depending on either the *less than* (lt) or *less than unsigned* (ltu) flags from the ALU. Sign and zero extension select among the three possible values for `sext.b`, `sext.h`, and `zext.h` using the *ext* module.

Fig. 4 shows the *cnt* module handling count instructions (`cpop, clz, ctz`) and their respective W64 variants. It contains *popcnt* and *lzc* modules for counting population (total number of ones) and leading zeros. It counts trailing zeros by reversing the input; note that *rev* only involves wires, not gates. Fig. 5 shows a simplified 8-bit version of the *popcnt* module for `cpop`. In general, the module uses an XLEN:2 tree of CSAs, followed by a carry propagate adder. The leading zero counter is simply a priority encoder with a binary output.

Fig. 6 shows the *byte* module that handles `orc.b` and `rev8`. It contains one 8:1 OR gate for each byte to do combining, and a network of wires to reverse bytes.

C. Zbc: Carry-less Multiplication Design

The existing ALU does not support carry-less multiplies, so a *zbc* unit is developed to perform the three flavors of carryless multiplication: `clmul`, `clmulh`, and `clmulr`. Figure 5 shows the *zbc* module, which uses a single *clmul* carry-free multiplication array, and conditionally reverses the inputs and outputs.

Carryless multiplication is similar to ordinary multiplication [9] but does not propagate carries between columns. Hence, the full adders reduce to simple XOR gates to sum the carries within a column modulo 2. Only the least significant half of the multiplier is necessary to obtain an XLEN-sized result. Fig. 8 shows a 4-bit version of the *clmul* carryless multiplication module. In this section, the @ operator will refer to this carryless multiplication.

The `clmulh` instruction computes the upper XLEN bits of a carry-free multiplication. Fig. 9(a) shows a complete 2XLEN-sized carryless multiplier with the upper half highlighted in blue. If the blue bits are flipped horizontally and vertically as shown in Fig. 9(b), we see they have the same structure as Fig. 8 except that the inputs are reversed, Y is shifted left by 1, and then the outputs are reversed: clmulh = rev(rev(X) @ (rev(Y) << 1)). Hence, `clmulh` reuses the same *clmul* module by bit-reversing and shifting the inputs.

The `clmulr` instruction is similar to `clmulh` but omits the shift on Y: clmulr = rev(rev(X) @ rev(Y)). Hence, clmulh = clmulr >> 1.

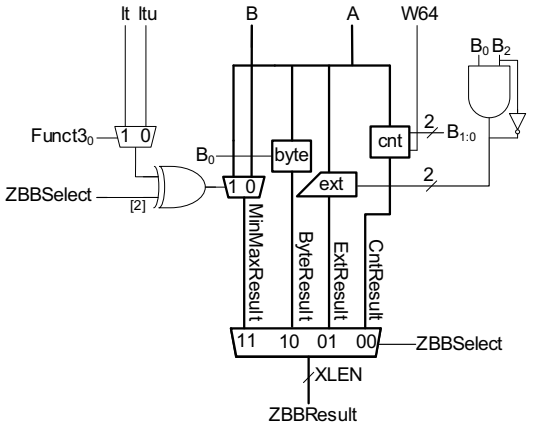

Fig. 3. *zbb* module

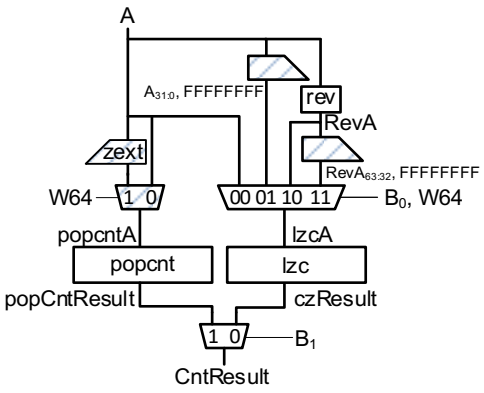

Fig. 4. *cnt* module

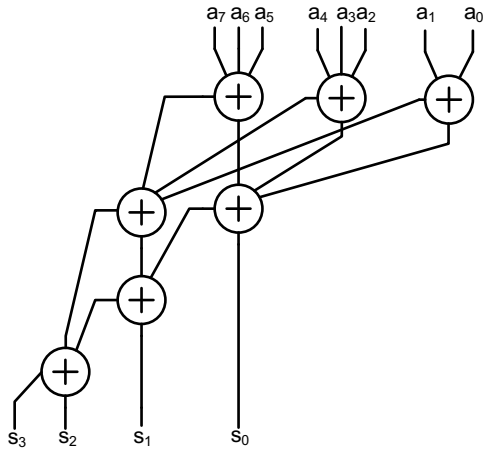

Fig. 5. 8-bit *popcnt* module

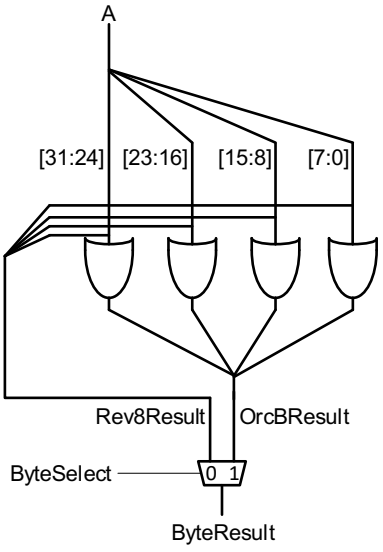

Fig. 6. 32-bit *byte* module

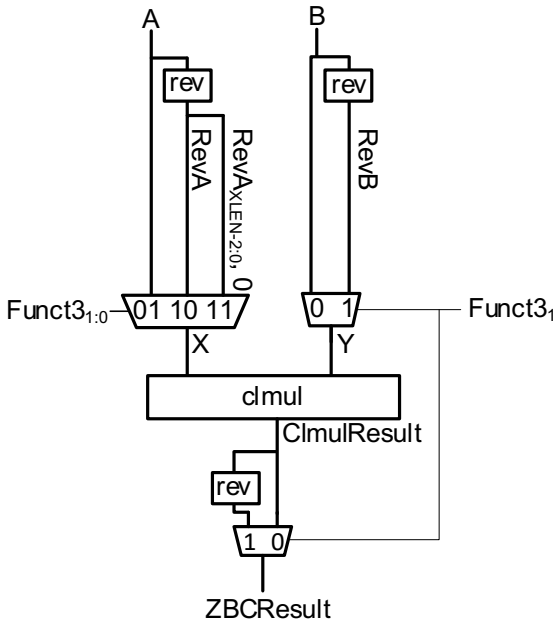

Fig. 7. *zbc* module

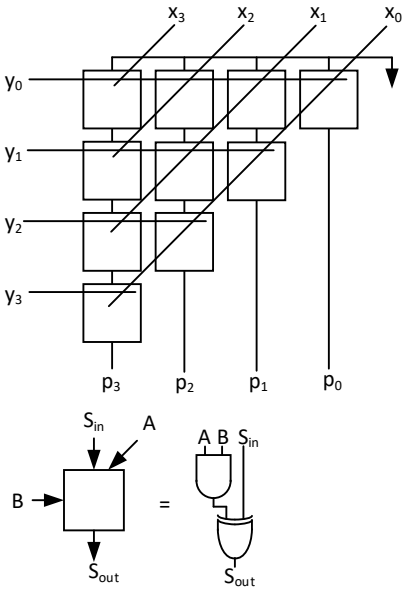

Fig. 8. 4-bit *clmul* carryless multiplication array

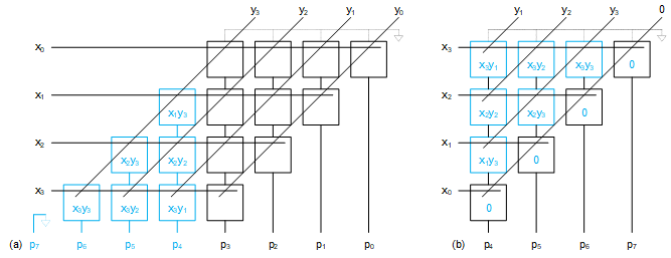

Fig. 9. 4-bit `clmulh` operation using the carryless multiplication array

D. Zbs: Single Bit Operation Design

The Zbs instructions operate on single bits by forming a one-hot mask and applying a logical operation. Fig. 2 shows the mask generation decoder: CondMaskB = 1 << SrcB. The ALU then performs logical operations:

- `bset` rd = SrcA | CondMaskB
- `bclr` rd = SrcA & ~CondMaskB
- `binv` rd = SrcA ^ CondMaskB
- `bext` rd = |(SrcA & CondMaskB)

The ALU uses a XLEN-input OR gate for `bext` to determine if the extracted bit is true.

III. Synthesis Results

The ALU and CORE-V Wally processor were synthesized hierarchically in the TSMC 28hpc+ process using Design Compiler topological synthesis. Measurements were taken at typical process, nominal voltage of 0.9 V, and at 25 °C. For reference, the FO4 inverter delay is 12.2 ps, and the NAND2 area is 0.378 μm^2. We considered 32 and 64 bit flavors of RVI (simple core with only integer instructions) and RVGC (application processor with floating point, virtual memory, caches, and branch prediction) configurations.

Table IV shows synthesis results targeting minimum delay. Adding full BMI support increases the ALU delay by approximately 10 FO4 delays. The new critical path involves bitmask generation, computing the `lt` flag with the adder, and using the flag to select min/max in the *zbb* block. The RV(32/64)I configurations slow down by 4-5.5 FO4 delays because the baseline RVI critical paths did not run through the ALU. The RVGC configuration does not slow down because the critical paths involve the FPU, PMP, and trap logic but not the ALU.

TABLE IV. Synthesis Results at Minimum Delay

Component	BMI	RV32		RV64	
		Area (μm^2)	Delay (ns)	Area (μm^2)	Delay (ns)
ALU	None	1140	0.179	2819	0.232
	Zba	1413	0.223	3374	0.269
	Zbb	1499	0.273	3912	0.333
	Zbc	2500	0.245	7317	0.299
	Zbs	1466	0.234	2396	0.281
	Zb(a,b,s)	1906	0.301	4520	0.345
	All	3144	0.300	9030	0.347
RVI	None	23971	0.422	31882	0.436
	Zb(a,b,s)	25263	0.466	35257	0.502
RVI Optimized	None	24026	0.422	31882	0.436
	Zb(a,b,s)	25933	0.445	35374	0.494
RVGC	None	144349	1.318	193323	1.618
	All	146472	1.325	198388	1.606

The ALU critical path with full BMI support is a false path in that the bitmask and `lt/ltu` flag generation never occur in the same instruction. Adding a dedicated comparator to decouple mask and flag generation slows down the RV(32/64)I critical path by 2-4.8 FO4 delays and adds an area overhead of 116-196 kGE. The new critical path for the RV32I Zb(a,b,s) core involves mask generation logic and the adder. The new critical path for the RV64I Zb(a,b,s) core involves the population count unit. Synthesis results for the RV core with the optimized ALU is denoted in Table IV and V as RVI Optimized.

Table V shows synthesis results with relaxed delay constraint. The area overhead is approximately 3.5 and 10.5 kGE for full BMI support in RV32GC and RV64GC respectively, or 1-2.5% of the total area. Note that this includes the instruction decoder as well as the ALU. 50-60% of the BMI area is for the carry-less multiplier, which is unnecessary for some applications. Also note that the area increases in Table IV are more difficult to interpret because the longer cycle time allows synthesis to downsize some non-critical logic.

TABLE V. SYNTHESIS RESULTS AT MINIMUM AREA

Component	BMI	RV32	RV64
		Area (μm^2)	Area (μm^2)
ALU	None	441	1077
	Zba	549	1338
	Zbb	862	2104
	Zbc	1270	3972
	Zbs	554	1308
	Zb(a,b,s)	942	2307
	All	1693	5021
RVI	None	19827	25943
	Zb(a,b,s)	20385	27219
RVI Optimized	None	19827	25943
	Zb(a,b,s)	20430	27295
RVGC	None	115237	156322
	All	116572	160296

IV. CONCLUSION

This paper has presented the design and synthesis results of adding bit-manipulation support to a configurable RISC-V processor. Supporting BMIs has no impact on the critical path of an application processor. It adds 1-2.5% to total area, or less than a half of that without carry-less multiplication.

ACKNOWLEDGMENT

This work was supported by the Clay-Wolkin Fellowship.

REFERENCES

[1] "RISC-V Bit-Manipulation ISA-extensions," *RISC-V International*, June 2021.

[2] P. Babu, S. Sivaraman, D. Sarma, and T. Warrier, "Evaluation of Bit Manipulation Instructions in Optimization of Size and Speed in RISC-V," *Int. Conf. VLSI Design*, February 2021.

[3] B. Koppelmann, P. Adelt, W. Mueller, and C. Scheytt, "RISC-V Extensions for Bit Manipulation Instructions," *Int. Symp. Power and Timing Modeling, Optimization and Simulation,* July 2019.

[4] Z. Meng, Y. Zhang, J. Zhou, and Z. Guo, "Design of 64-Bit High-Performance Embedded Processor Supporting RISC-V B-Extension," *Int. Conf. on Anti-counterfeting, Security, and Identification*, December 2022.

[5] D. Markov and A. Romanov, "Implementation of the RISC-V Architecture with the Extended Zbb Instruction Set," *Int. Ural Conf. on Electrical Power Engineering*, September 2022.

[6] V. Jain, A. Sharma, and E. Bezerra, "Implementation and Extension of Bit Manipulation Instruction on RISC-V Architecture using FPGA," *Int. Conf. Communication Systems and Network Technologies*, April 2020.

[7] "CORE-V Wally," *OpenHW Group*, April 2023. github.com/openhwgroup/cvw

[8] *RISC-V Architecture Test SIG*, April 2023. github.com/riscv-non-isa/riscv-arch-test

[9] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective, 4th Ed*. Pearson, 2010.