Independent Submission
Internet-Draft
Intended status: Experimental
Expires: 20 September 2026

S. Bellis
Unheaded

19 March 2026

MBC Instruction Set Architecture for the Unheaded Protocol Computer
draft-bellis-unheaded-mbc-isa-00

Abstract

This document defines the MBC (Monad Bytecode) Instruction Set Architecture for the Unheaded Protocol Computer (UPC). MBC is a 48-opcode, 32-bit fixed-width instruction set designed for execution within eBPF XDP programs. It provides computational capabilities for distributed packet processing using the Monad wire format, enabling deterministic network packet classification and transformation at the network edge.

Status of This Memo

This document specifies an Experimental Internet Protocol for the Internet community. This document is a product of the Independent Submission stream. It represents the work and views of its author, who is responsible for its contents. It does not represent a consensus view of the Internet community. Readers interested in this topic should note that the content of this document may change based on deployment experience and feedback from the Internet at large. Publication of this document does not imply adoption by the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

This Internet-Draft will expire on September 18, 2026.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Unheaded Protocol Computer (UPC) is a computational engine embedded within eBPF XDP programs, executing at the network edge to provide deterministic packet processing for infrastructure automation. The MBC Instruction Set Architecture defines the low-level computation model for the UPC.

MBC is optimized for execution within eBPF verifier constraints—specifically the 256-instruction limit per eBPF program execution. This constraint drives several design decisions: fixed-width 32-bit instructions for efficient decoding, a compact 16-register set for minimal memory usage, and a carefully tuned opcode allocation that balances feature completeness with code density.

The primary use cases for MBC include:

* Packet classification based on Monad wire format headers

* Dynamic packet transformation and header rewriting

* Per-packet telemetry and flow tracking

* Distributed policy enforcement in multi-hop environments

MBC is part of the Foundation (Monad wire format specification), which defines the Monad packet header structure and flow semantics. The Monad wire format provides a 20-byte fixed header with 12 IANA-registered extensions, frozen at version 0x01. MBC executes on packets carrying Monad headers, allowing per-packet computation to be distributed across the network infrastructure.

Complementary to MBC are two other Unheaded specifications:

* Wotan: The message bus and ring buffer protocol for event distribution within the UPC ecosystem

* Sophia: The dictionary service and knowledge graph for stateful computation and policy storage

This document specifies the complete MBC ISA, including instruction encoding, register model, opcode definitions, memory addressing, interrupt handling, and BPF verifier compliance.

2. Requirements Language

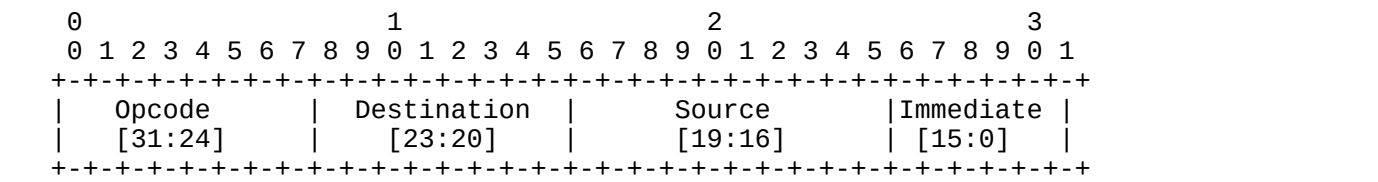
The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174].

3. Instruction Encoding

MBC uses a 32-bit fixed-width instruction format. All instructions are aligned on 4-byte boundaries and follow a uniform structure.

3.1. Instruction Format

The 32-bit instruction is laid out as follows:



Opcode [31:24]: 8-bit opcode identifier, ranging from 0x00 to 0xFF. Opcodes 0x00 and 0xFE are reserved.

Destination [23:20]: 4-bit register index (r0-r15) for the result of the operation, or flags register for compare operations.

Source [19:16]: 4-bit register index (r0-r15) for the first operand, or flags for conditional branches.

Immediate [15:0]: 16-bit signed immediate value, used for offsets, constants, or as part of extended operands.

All instructions are stored in little-endian byte order in memory. When an MBC program is loaded, the instruction stream MUST be byte-swapped from the storage format to the native CPU byte order before execution begins.

Field extraction follows standard C bitfield semantics: opcode is the high byte, destination is bits 23-20, source is bits 19-16, and immediate is the low 16 bits. All shifts and masks are performed in the native CPU word size (32-bit or larger).

3.2. Instruction Encoding Variants

Several instruction families use non-standard field mappings:

Arithmetic (ADD, SUB, MUL, DIV, MOD, NEG): Destination (dst) and Source (src) both reference GPRs. Immediate field is unused (MUST be 0).

Extended Immediate (LOAD_IMM32, ADDI): Destination is GPR. Immediate is signed 16-bit value, sign-extended to 32 bits. Immediate field is the low 16 bits of the constant for LOAD_IMM32.

Memory Operations (LD, ST, LDB, STB, LDH, STH): Destination is GPR (for loads) or ignored (for stores). Source is base address GPR. Immediate is signed byte/word offset.

Branch Operations (JZ, JNZ, JN, JP, JC, JNC): Source field encodes flags to test. Immediate is signed 16-bit word offset from next instruction.

Atomic Operations (XCHG, CAS): Destination and Source are both GPRs. Immediate specifies additional parameters.

4. Register Model

MBC provides a 16-register computational model with a separate flags register.

4.1. General-Purpose Registers

MBC defines 16 general-purpose registers, each 32 bits wide:

Register	Mnemonic	Convention
0	r0	Return value
1	r1	Argument 1, scratch
2	r2	Argument 2, scratch
3	r3	Argument 3, scratch
4	r4	Argument 4, scratch
5	r5	Argument 5, scratch
6	r6	Callee-saved
7	r7	Callee-saved
8	r8	Callee-saved
9	r9	Callee-saved
10	r10	Callee-saved
11	r11	Scratch
12	r12	Scratch
13	r13	Scratch
14	r14	Scratch
15	r15	Stack pointer (SP) convention

All registers are 32-bit unsigned values. Signed arithmetic uses two's complement representation internally; the signedness is determined by the instruction semantics, not register width.

The stack pointer (SP) convention uses r15. The stack grows downward (toward lower addresses). Stack operations are not directly supported by MBC instructions (PUSH and POP are memory operations); management of the stack is the responsibility of the MBC program.

4.2. Flags Register

MBC maintains an 8-bit flags register with the following bit assignments:

Bit	Mnemonic	Meaning
7	IF	Interrupt Enable (1 = interrupts enabled)
6	Reserved	Must be zero
5	Reserved	Must be zero
4	Reserved	Must be zero
3	Reserved	Must be zero
2	C	Carry flag (set by arithmetic, test by branch)
1	N	Negative/Sign flag (set by arithmetic)
0	Z	Zero flag (set by arithmetic, test by branch)

Flags are set by arithmetic operations (ADD, SUB, MUL, DIV, MOD, NEG) and by comparison (CMP). Flags are tested by conditional branch instructions (JZ, JNZ, JN, JP, JC, JNC) and by the interrupt control instructions (INT, CLI, STI).

The Interrupt Enable (IF) flag controls whether INT instructions will actually trigger interrupt handlers (see Section 5.6 for details).

4.3. Program Counter

The Program Counter (PC) is a 32-bit register that holds the byte address of the currently executing instruction. The PC is automatically incremented after each instruction execution. Branch and jump instructions modify the PC directly.

For the purposes of immediate field interpretation in branch instructions, the immediate value is treated as a signed 16-bit word offset (i.e., it is multiplied by 4 bytes before being added to PC). This allows branch targets to reach +/- 128 KB from the current instruction.

4.4. CPU State Structure

The MBC CPU state is organized as a single 128-byte structure:

Offset	Field	Size	Description
0-3	r0	4	General-purpose register 0
4-7	r1	4	General-purpose register 1
...	...	...	...
56-59	r14	4	General-purpose register 14
60-63	r15	4	Stack pointer (SP)
64	flags	1	Z, N, C, IF flags
65-67	reserved	3	Reserved for alignment
68-71	PC	4	Program counter
72-127	reserved	56	Reserved for future use

Total size: 128 bytes. This structure is the working memory for each MBC execution context.

5. Instruction Set

MBC provides 48 distinct opcodes, organized by functional category.

5.1. Arithmetic Instructions

Arithmetic instructions operate on two 32-bit operands (src, dst) and update the flags register.

5.1.1. ADD - Addition

```
   Opcode: 0x01
   Encoding: [0x01][dst:r0-r15][src:r0-r15][imm:unused]

   Operation: dst = dst + src
   Flags: Z, N, C set according to result

   Adds the value in src register to dst register, storing the result in
   dst.  Carry flag is set if unsigned overflow occurs.  Negative flag
   is set if result is negative (bit 31 set).  Zero flag is set if
   result is zero.
```

5.1.2. SUB - Subtraction

```
   Opcode: 0x02
   Encoding: [0x02][dst:r0-r15][src:r0-r15][imm:unused]

   Operation: dst = dst - src
   Flags: Z, N, C set according to result

   Subtracts the value in src from dst, storing the result in dst.
   Carry flag is set if unsigned underflow occurs (borrow).  Negative
   and Zero flags set as in ADD.
```

5.1.3. MUL - Multiplication (Low 32 Bits)

```
   Opcode: 0x03
   Encoding: [0x03][dst:r0-r15][src:r0-r15][imm:unused]

   Operation: dst = (dst * src) & 0xFFFFFFFF
   Flags: Z, N set; C set if high 32 bits are non-zero

   Multiplies dst by src, storing the low 32 bits in dst.  The Carry
   flag indicates whether the result overflowed 32 bits.  This
   instruction performs 32-bit multiply returning 32-bit result.
```

5.1.4. DIV - Integer Division

```
   Opcode: 0x04
   Encoding: [0x04][dst:r0-r15][src:r0-r15][imm:unused]

   Operation: dst = dst / src
   Flags: Z, N set; trap if src == 0

   Divides dst by src, storing the quotient in dst.  If src is zero,
   execution traps (see Security Considerations).  Division is unsigned
   32-bit arithmetic.  Remainder is discarded.
```

5.1.5. MOD - Integer Modulo

Opcode: 0x05
Encoding: [0x05][dst:r0-r15][src:r0-r15][imm:unused]

Operation: dst = dst % src
Flags: Z, N set; trap if src == 0

Computes dst modulo src, storing the remainder in dst. Traps on divide by zero. Modulo is unsigned 32-bit arithmetic.

5.1.6. NEG - Arithmetic Negation

Opcode: 0x06
Encoding: [0x06][dst:r0-r15][src:unused][imm:unused]

Operation: dst = -dst (two’s complement)
Flags: Z, N, C set according to result

Computes the two’s complement negation of dst. Equivalent to SUB with dst=0 and src=dst. Carry flag is set if result overflows (i.e., if dst was 0x80000000).

5.2. Logic and Bitwise Instructions

Logical and bitwise instructions operate on 32-bit values and update flags.

5.2.1. AND - Bitwise AND

Opcode: 0x07
Encoding: [0x07][dst:r0-r15][src:r0-r15][imm:unused]

Operation: dst = dst & src
Flags: Z, N set

Performs bitwise AND of dst and src, storing result in dst. Carry flag is not modified.

5.2.2. OR - Bitwise OR

Opcode: 0x08
Encoding: [0x08][dst:r0-r15][src:r0-r15][imm:unused]

Operation: dst = dst | src
Flags: Z, N set

Performs bitwise OR of dst and src, storing result in dst.

5.2.3. XOR - Bitwise Exclusive OR

Opcode: 0x09
Encoding: [0x09][dst:r0-r15][src:r0-r15][imm:unused]

Operation: dst = dst ^ src
Flags: Z, N set

Performs bitwise XOR of dst and src, storing result in dst.

5.2.4. NOT - Bitwise NOT

Opcode: 0x0A
Encoding: [0x0A][dst:r0-r15][src:unused][imm:unused]

Operation: dst = ~dst
Flags: Z, N set

Performs bitwise NOT (one’s complement) of dst, storing result in dst.

5.2.5. SHL - Shift Left (Immediate)

Opcode: 0x0B
Encoding: [0x0B][dst:r0-r15][src:unused][imm:shift amount 0-31]

Operation: dst = dst << (imm & 0x1F)
Flags: Z, N set; C set to last shifted-out bit

Shifts dst left by the amount specified in the immediate field (0-31 bits). Vacated bits are zero-filled. Carry flag is set to the last bit shifted out.

5.2.6. SHR - Shift Right Logical (Immediate)

Opcode: 0x0C
Encoding: [0x0C][dst:r0-r15][src:unused][imm:shift amount 0-31]

Operation: dst = dst >> (imm & 0x1F)
Flags: Z, N set; C set to last shifted-out bit

Shifts dst right logically by the amount in the immediate field. Vacated bits are zero-filled (unsigned shift).

5.2.7. SAR - Shift Right Arithmetic (Immediate)

Opcode: 0x0D
Encoding: [0x0D][dst:r0-r15][src:unused][imm:shift amount 0-31]

Operation: dst = ((int32_t)dst) >> (imm & 0x1F)
Flags: Z, N set; C set to last shifted-out bit

Shifts dst right arithmetically by the amount in the immediate field. Vacated bits are filled with the sign bit (bit 31). This preserves the sign of negative numbers.

5.3. Data Movement Instructions

5.3.1. MOV - Move Register

Opcode: 0x0E
Encoding: [0x0E][dst:r0-r15][src:r0-r15][imm:unused]

Operation: dst = src
Flags: Z, N set

Copies the value from src to dst. Flags are set based on the copied value.

5.3.2. MOVI - Move Immediate

Opcode: 0x0F
Encoding: [0x0F][dst:r0-r15][src:unused][imm:signed 16-bit]

Operation: dst = sign_extend(imm)
Flags: Z, N set

Loads a sign-extended 16-bit immediate value into dst. The immediate is sign-extended to 32 bits before being stored in dst.

5.4. Comparison Instruction

5.4.1. CMP - Compare

Opcode: 0x10
Encoding: [0x10][dst:r0-r15][src:r0-r15][imm:unused]

Operation: Compute (dst - src), update flags, discard result
Flags: Z, N, C set according to subtraction

Compares two values by computing dst - src and setting flags accordingly, without storing the result. The Carry flag is set if unsigned underflow occurs (src > dst in unsigned comparison). This instruction is used to prepare for conditional branches.

5.5. Extended Immediate Instructions

5.5.1. LOAD_IMM32 - Load 32-bit Immediate

Opcode: 0x1C
Encoding: [0x1C][dst:r0-r15][upper:4-bit upper word][imm:16-bit lower word]

Operation: dst = (upper << 16) | imm
Flags: Z, N set

Loads a full 32-bit immediate into dst. The upper 4 bits come from the source field, the lower 16 bits from the immediate field. The middle 12 bits are zero-filled. This allows loading of arbitrary 20-bit immediates. For full 32-bit constants, two consecutive instructions are required.

5.5.2. ADDI - Add Immediate

Opcode: 0x1D
Encoding: [0x1D][dst:r0-r15][src:unused][imm:signed 16-bit]

Operation: dst = dst + sign_extend(imm)
Flags: Z, N, C set according to result

Adds a sign-extended 16-bit immediate to dst, storing the result in dst. Flags are updated as in ADD.

5.6. Interrupt Instructions

5.6.1. INT - Trigger Interrupt

Opcode: 0x17
Encoding: [0x17][vector:r0-r15][src:unused][imm:unused]

Operation: If IF flag set, jump to IVT[vector]; save return address
Flags: IF cleared upon entry to handler

Triggers the interrupt handler at IVT entry specified by the value in the vector register. The PC+1 (next instruction address) is pushed onto the stack (via SP--). The Interrupt Enable flag is cleared upon entry to the handler, preventing nested interrupts. If IF is already clear, INT is a no-op.

5.6.2. IRET - Return from Interrupt

Opcode: 0x18
Encoding: [0x18][dst:unused][src:unused][imm:unused]

Operation: PC = pop from stack; IF = 1
Flags: IF set

Returns from an interrupt handler. Restores the saved PC from the stack and sets the Interrupt Enable flag. The stack pointer (r15) is incremented by 4.

5.7. Stack Operations

5.7.1. PUSH - Push to Stack

Opcode: 0x1A
Encoding: [0x1A][src:r0-r15][dst:unused][imm:unused]

Operation: *(--SP) = src; (SP -= 4)
Flags: None

Pushes the value in src onto the stack (using r15 as SP). The stack grows downward (toward lower addresses). SP is decremented by 4 bytes before the write.

5.7.2. POP - Pop from Stack

Opcode: 0x1B
Encoding: [0x1B][dst:r0-r15][src:unused][imm:unused]

Operation: dst = *(SP); (SP += 4)
Flags: None

Pops a value from the stack into dst. SP is incremented by 4 bytes after the read.

5.8. Branch and Jump Instructions

5.8.1. JMP - Unconditional Jump

Opcode: 0x20
Encoding: [0x20][dst:unused][src:unused][imm:signed 16-bit word offset]

Operation: PC = PC + (sign_extend(imm) * 4)
Flags: None

Unconditionally jumps to the target address specified by the sign-extended immediate. The immediate is multiplied by 4 to convert from word offset to byte offset. This allows jumps to +/- 128 KB from the current instruction.

5.8.2. JZ - Jump if Zero

Opcode: 0x21
Encoding: [0x21][dst:unused][src:flags field][imm:signed 16-bit word offset]

Operation: If Z flag set, PC = PC + (sign_extend(imm) * 4)
Flags: None

Jumps if the Zero flag is set. Typically used after a CMP or arithmetic instruction. Source field encodes which flag to test (currently only Z is supported in src=0x0).

5.8.3. JNZ - Jump if Not Zero

Opcode: 0x22
Encoding: [0x22][dst:unused][src:flags field][imm:signed 16-bit word offset]

Operation: If Z flag clear, PC = PC + (sign_extend(imm) * 4)
Flags: None

Jumps if the Zero flag is clear (i.e., result was non-zero).

5.8.4. JN - Jump if Negative

Opcode: 0x23
Encoding: [0x23][dst:unused][src:flags field][imm:signed 16-bit word offset]

Operation: If N flag set, PC = PC + (sign_extend(imm) * 4)
Flags: None

Jumps if the Negative flag is set (result bit 31 is 1).

5.8.5. JP - Jump if Positive

Opcode: 0x24
Encoding: [0x24][dst:unused][src:flags field][imm:signed 16-bit word offset]

Operation: If N flag clear, PC = PC + (sign_extend(imm) * 4)
Flags: None

Jumps if the Negative flag is clear (result is positive or zero).

5.8.6. JC - Jump if Carry

Opcode: 0x25
Encoding: [0x25][dst:unused][src:flags field][imm:signed 16-bit word offset]

Operation: If C flag set, PC = PC + (sign_extend(imm) * 4)
Flags: None

Jumps if the Carry flag is set. Used for unsigned overflow detection.

5.8.7. JNC - Jump if No Carry

Opcode: 0x26
Encoding: [0x26][dst:unused][src:flags field][imm:signed 16-bit word offset]

Operation: If C flag clear, PC = PC + (sign_extend(imm) * 4)
Flags: None

Jumps if the Carry flag is clear.

5.9. Call and Return Instructions

5.9.1. CALL - Call Subroutine

Opcode: 0x27
Encoding: [0x27][dst:unused][src:unused][imm:signed 16-bit word offset]

Operation: *(--SP) = PC + 1; PC = PC + (sign_extend(imm) * 4)
Flags: None

Calls a subroutine at the address specified by the offset. The return address (PC+1) is pushed onto the stack before jumping. Used with RET to implement subroutine calls.

5.9.2. RET - Return from Subroutine

Opcode: 0x28
Encoding: [0x28][dst:unused][src:unused][imm:unused]

Operation: PC = *(SP); SP += 4
Flags: None

Returns from a subroutine by popping the return address from the stack and jumping to it. Complements CALL.

5.9.3. JMPR - Jump Register

Opcode: 0x29
Encoding: [0x29][dst:unused][src:r0-r15][imm:unused]

Operation: PC = src_register_value
Flags: None

Performs an indirect jump to the address contained in the src register. No return address is saved.

5.9.4. CALLR - Call Register

Opcode: 0x2A
Encoding: [0x2A][dst:unused][src:r0-r15][imm:unused]

Operation: *(--SP) = PC + 1; PC = src_register_value
Flags: None

Performs an indirect call to the address in the src register. The return address (PC+1) is pushed onto the stack.

5.10. Memory Load and Store Instructions

All memory operations use indexed addressing: effective_address = base_register + sign_extend(imm). The immediate field is a signed byte offset (for byte load/store) or word offset (for word load/store).

5.10.1. LD - Load Word (32-bit)

Opcode: 0x30
Encoding: [0x30][dst:r0-r15][src:r0-r15][imm:signed 16-bit offset]

Operation: dst = *(base + sign_extend(imm))
Flags: Z, N set

Loads a 32-bit word from memory at address (src + sign_extend(imm)) into dst. The immediate is treated as a byte offset. Unaligned access behavior is undefined.

5.10.2. ST - Store Word (32-bit)

Opcode: 0x31
Encoding: [0x31][src:r0-r15 (data)][base:r0-r15][imm:signed 16-bit offset]

Operation: *(base + sign_extend(imm)) = src
Flags: None

Stores a 32-bit word from src into memory at address (base + sign_extend(imm)). The destination field is used to specify the base register. The source field specifies the data register.

5.10.3. LDB - Load Byte (8-bit)

Opcode: 0x32
Encoding: [0x32][dst:r0-r15][src:r0-r15][imm:signed 16-bit offset]

Operation: dst = *(uint8_t*)(base + sign_extend(imm)); zero-extend to 32 bits
Flags: Z, N set

Loads an 8-bit byte from memory, zero-extends it to 32 bits, and stores it in dst.

5.10.4. STB - Store Byte (8-bit)

Opcode: 0x33
Encoding: [0x33][src:r0-r15 (data, low byte used)][base:r0-r15][imm:signed 16-bit offset]

Operation: *(uint8_t*)(base + sign_extend(imm)) = (uint8_t)src
Flags: None

Stores the low byte of src into memory at address (base + sign_extend(imm)).

5.10.5. LDH - Load Half-word (16-bit)

Opcode: 0x34
Encoding: [0x34][dst:r0-r15][src:r0-r15][imm:signed 16-bit offset]

Operation: dst = *(uint16_t*)(base + sign_extend(imm)); zero-extend to 32 bits
Flags: Z, N set

Loads a 16-bit half-word from memory, zero-extends it to 32 bits, and stores it in dst.

5.10.6. STH - Store Half-word (16-bit)

Opcode: 0x35
Encoding: [0x35][src:r0-r15 (data, low half-word used)][base:r0-r15][imm:signed 16-bit offset]

Operation: *(uint16_t*)(base + sign_extend(imm)) = (uint16_t)src
Flags: None

Stores the low 16 bits of src into memory at address (base + sign_extend(imm)).

5.11. Register-Based Shifts and Extended Arithmetic

5.11.1. SHLR - Shift Left Register

Opcode: 0x36
Encoding: [0x36][dst:r0-r15][src:r0-r15 (shift count)][imm:unused]

Operation: dst = dst << (src & 0x1F)
Flags: Z, N set; C set to last shifted-out bit

Shifts dst left by the amount specified in src (masked to 5 bits). Unlike SHL, the shift amount comes from a register, allowing dynamic shift amounts.

5.11.2. SHRR - Shift Right Logical Register

Opcode: 0x37
Encoding: [0x37][dst:r0-r15][src:r0-r15 (shift count)][imm:unused]

Operation: dst = dst >> (src & 0x1F)
Flags: Z, N set; C set to last shifted-out bit

Shifts dst right logically by the amount in src (masked to 5 bits). Vacated bits are zero-filled.

5.11.3. SARR - Shift Right Arithmetic Register

Opcode: 0x38
Encoding: [0x38][dst:r0-r15][src:r0-r15 (shift count)][imm:unused]

Operation: dst = ((int32_t)dst) >> (src & 0x1F)
Flags: Z, N set; C set to last shifted-out bit

Shifts dst right arithmetically by the amount in src (masked to 5 bits). Vacated bits are filled with the sign bit.

5.11.4. MULH - Multiply High (Signed)

Opcode: 0x39
Encoding: [0x39][dst:r0-r15][src:r0-r15][imm:unused]

Operation: dst = (((int64_t)(int32_t)dst) * ((int64_t)(int32_t)src)) >> 32
Flags: Z, N set

Performs a signed 32x32->64 bit multiply and returns the high 32 bits. Both operands are sign-extended from 32 bits to 64 bits before multiplication. Useful for scaling and fixed-point arithmetic.

5.12. Extended Multiply

5.12.1. MULHU - Multiply High Unsigned

Opcode: 0x3A
Encoding: [0x3A][dst:r0-r15][src:r0-r15][imm:unused]

Operation: dst = (((uint64_t)dst) * ((uint64_t)src)) >> 32
Flags: Z, N set

Performs an unsigned 32x32->64 bit multiply and returns the high 32 bits. Both operands are treated as unsigned 32-bit values.

5.13. Atomic Operations

5.13.1. CLI - Clear Interrupt Enable

Opcode: 0x3B
Encoding: [0x3B][dst:unused][src:unused][imm:unused]

Operation: IF = 0
Flags: IF cleared

Clears the Interrupt Enable flag, disabling interrupt processing. All INT instructions become no-ops while IF is clear. Used to protect critical sections from interrupt preemption.

5.13.2. STI - Set Interrupt Enable

Opcode: 0x3C
Encoding: [0x3C][dst:unused][src:unused][imm:unused]

Operation: IF = 1
Flags: IF set

Sets the Interrupt Enable flag, re-enabling interrupt processing. Typically used after a CLI to re-enable interrupts at the end of a critical section.

5.13.3. XCHG - Atomic Exchange

Opcode: 0x3D
Encoding: [0x3D][dst:r0-r15 (address)][src:r0-r15 (new value)][imm:offset]

Operation: old = *dst; *dst = src; dst = old
Flags: Z, N set based on old value

Atomically exchanges the value at the memory address in dst with the value in src. The old value is returned in dst. The immediate field specifies an optional offset to add to the address. This operation is atomic with respect to interrupts (guaranteed to complete without interruption even if IF is set).

5.13.4. CAS - Compare and Swap

Opcode: 0x3E
Encoding: [0x3E][dst:r0-r15 (address)][src:r0-r15 (new value)][imm:compare value high 16 bits]

Operation: if (*dst == compare_value) { *dst = src; Z=1 } else { Z=0 }
Flags: Z set if swap succeeded, clear if failed

Atomically compares the value at address dst with a compare value (32-bit, specified via immediate and additional register). If equal, stores the new value from src and sets Z flag. If not equal, leaves memory unchanged and clears Z flag. Used for lock-free synchronization. The full 32-bit compare value is formed from the immediate field (lower 16 bits) and a second immediate encoded in the src field (upper 16 bits).

5.14. System Instructions

5.14.1. SYSCALL - System Call

Opcode: 0x40
Encoding: [0x40][syscall_num:r0-r15][src:unused][imm:unused]

Operation: Invoke system call (syscall number in dst register)
Flags: Depends on syscall

Invokes a system call. The syscall number is specified in the dst register. System calls provide access to eBPF helper functions and privileged operations. Not all syscalls are available in XDP context; availability depends on the eBPF verifier and kernel version.

5.14.2. HALT - Halt Execution

Opcode: 0xFF
Encoding: [0xFF][exit_code:r0-r15][src:unused][imm:unused]

Operation: Stop execution; return exit_code to caller
Flags: None (execution halts)

Terminates execution of the MBC program. The value in the dst register is returned as the exit code to the caller. This is used to signal completion or error conditions back to the eBPF XDP program.

6. Memory Addressing

MBC uses a single addressing mode: indexed addressing.

6.1. Indexed Addressing Mode

Effective Address = Base Register + Signed Offset

For load/store instructions, the effective address is computed by adding the sign-extended immediate field to a base register. The immediate field width determines the range:

* Byte instructions (LDB, STB): 16-bit signed offset = -32768 to +32767 bytes

* Half-word instructions (LDH, STH): 16-bit signed offset = -32768 to +32767 bytes (each byte)

* Word instructions (LD, ST): 16-bit signed offset = -32768 to +32767 bytes (each byte)

All memory accesses use byte offsets in the immediate field. The eBPF verifier enforces that all memory accesses are within bounds of allocated eBPF maps or kernel memory buffers.

6.2. Alignment Requirements

MBC does not require strict alignment for memory accesses, but performance is improved for naturally-aligned accesses (4-byte words should be 4-byte aligned, etc.). Unaligned access behavior is defined by the underlying eBPF map type and kernel architecture.

6.3. Address Space Layout

MBC address space is not directly visible to MBC programs. All memory access goes through eBPF map keys, and the base register values are validated by the eBPF verifier. Typical layout in an eBPF program:

* 0x00000000 - 0x0FFFFFFF: Stack memory (provided by eBPF runtime, typically 512 bytes per context)

* 0x10000000+: eBPF map memory (accessed via helper functions, validated by verifier)

7. BPF Verifier Compliance

MBC execution must comply with eBPF verifier constraints to be JIT-compiled for high performance.

7.1. Instruction Limit

An MBC program MUST NOT exceed 256 instructions per eBPF program unit ("per tick" of the XDP handler). This is the eBPF verifier's limit on linear instruction sequences without loops. Loops within MBC programs must follow bounded iteration patterns (see below).

7.2. Bounded Loop Pattern

MBC programs that use loops (via jump instructions) must ensure loops terminate in bounded time. The standard pattern is:

1. Initialize a loop counter (e.g., r1 = 0)

2. Increment counter in loop body (e.g., r1 = r1 + 1)

3. Test counter against limit (e.g., CMP r1, 256)

4. Conditional branch back to loop start (e.g., JC loop_start)

The eBPF verifier performs static analysis to verify that the loop counter is bounded and the loop will terminate. If a loop is unbounded, the program will fail verifier validation and will not be loaded.

7.3. Map Access Null Checks

Any access to eBPF map data via LD/ST instructions MUST be preceded by a bounds check or the eBPF verifier will reject the program. In practice, this means:

* The base register for a memory access must either be the stack pointer (r15) or a validated map data pointer

* The offset must be within the known bounds of the allocated region

* If accessing map data, the map lookup MUST have been performed immediately before the access

The eBPF verifier tracks data flow to ensure these properties are maintained.

7.4. Memory Safety Guarantees

The eBPF verifier enforces memory safety:

* All memory addresses are validated before access
* Stack accesses are bounds-checked
* Map accesses are validated against allocated map size
* Uninitialized reads are prevented by data-flow analysis
* Out-of-bounds writes are prevented by bounds checking

These guarantees are inherent to the eBPF execution model and are not specific to MBC. However, MBC programs must be structured to respect these constraints.

8. Interrupt Model

MBC provides a simple interrupt mechanism for synchronous events and inter-process communication.

8.1. Interrupt Vector Table (IVT)

The IVT is a 256-entry table of interrupt handler addresses, located at fixed word addresses 0x00 to 0xFF. Each entry is a 32-bit word address (4 bytes apart). The total IVT size is 1024 bytes (256 entries × 4 bytes).

Address	Vector	Purpose
0x000	0x00	Reserved (division by zero trap)
0x004	0x01	Reserved
...	...	...
0x080	0x20	Timer interrupt
0x084	0x21	Keyboard interrupt
...	...	...
0x200	0x80	System call (SYSCALL instruction)
...	...	(Reserved for user-defined handlers)

The IVT is initialized by the MBC loader before program execution begins. Entries not filled by the loader contain zero (trap on execution).

8.2. Interrupt Handling Sequence

When an interrupt is triggered (via INT instruction or external hardware event):

1. The CPU saves the current PC+1 (return address) onto the stack (at SP-4, then decrements SP)
2. The Interrupt Enable (IF) flag is cleared
3. The PC is loaded with the value from IVT[vector_number]
4. Execution continues at the interrupt handler
5. The handler executes IRET to return: restores PC from stack, sets IF flag

Interrupt handlers run with IF clear, preventing nested interrupts (re-entrancy).

8.3. Standard Interrupt Sources

0x20 (Timer): Periodic timer interrupt, used for time-based scheduling and watchdog timers.

0x21 (Keyboard): Keyboard input interrupt. Not typically used in XDP context.

0x80 (Syscall): Invoked by the SYSCALL instruction, used to access privileged operations.

Vector 0x00 is reserved for division-by-zero exceptions. Additional vectors are reserved for user-defined purposes.

9. Safety Constraints

9.1. Opcode Whitelist

Only the 48 defined opcodes are valid in MBC programs. Opcodes 0x00, 0xFE are reserved and MUST NOT be used. All other opcode values are undefined and will cause a trap if executed. An MBC loader MUST validate that all instructions in a program use only defined opcodes before loading.

9.2. Division by Zero Behavior

If a DIV or MOD instruction is executed with src = 0, the execution traps (i.e., the eBPF program is terminated and an error is returned to the kernel). This is consistent with hardware division-by-zero behavior. The trap raises an exception that is caught by the eBPF verifier.

9.3. Stack Bounds

The stack pointer (r15) starts at a predetermined address (typically the end of the eBPF stack frame, around 0x1000 in most implementations). The MBC program is responsible for managing stack bounds. If SP underflows or overflows the allocated stack space, memory corruption results. The eBPF verifier does not automatically check stack bounds for MBC programs; this is the responsibility of the programmer.

9.4. Privilege Model

MBC has a flat privilege model with no distinction between user and kernel mode. All instructions execute with the same privilege level. The eBPF verifier is the trust boundary; any MBC program that passes verification is assumed safe to execute.

10. IANA Considerations

This document requests the creation of a new IANA registry for MBC Opcodes.

10.1. MBC Opcode Registry

Registry Name: MBC Instruction Set Architecture Opcode Values

Registry Range: 0x00 - 0xFF (256 values)

Registration Procedure: Standards Action [RFC8126]

Initial Assignments:

* 0x00: RESERVED
* 0x01: ADD - Addition
* 0x02: SUB - Subtraction
* 0x03: MUL - Multiply (low 32 bits)

* 0x04: DIV - Integer Division
* 0x05: MOD - Integer Modulo
* 0x06: NEG - Negate
* 0x07: AND - Bitwise AND
* 0x08: OR - Bitwise OR
* 0x09: XOR - Bitwise XOR
* 0x0A: NOT - Bitwise NOT
* 0x0B: SHL - Shift Left (Immediate)
* 0x0C: SHR - Shift Right Logical (Immediate)
* 0x0D: SAR - Shift Right Arithmetic (Immediate)
* 0x0E: MOV - Move Register
* 0x0F: MOVI - Move Immediate (16-bit sign-extended)
* 0x10: CMP - Compare
* 0x11-0x1B: RESERVED
* 0x1C: LOAD_IMM32 - Load 32-bit Immediate
* 0x1D: ADDI - Add Immediate (16-bit sign-extended)
* 0x1E-0x1F: RESERVED
* 0x17: INT - Trigger Interrupt
* 0x18: IRET - Return from Interrupt
* 0x19: RESERVED
* 0x1A: PUSH - Push to Stack
* 0x1B: POP - Pop from Stack
* 0x20: JMP - Unconditional Jump
* 0x21: JZ - Jump if Zero

* 0x22: JNZ - Jump if Not Zero
* 0x23: JN - Jump if Negative
* 0x24: JP - Jump if Positive
* 0x25: JC - Jump if Carry
* 0x26: JNC - Jump if No Carry
* 0x27: CALL - Call Subroutine
* 0x28: RET - Return from Subroutine
* 0x29: JMPR - Jump Register
* 0x2A: CALLR - Call Register
* 0x2B-0x2F: RESERVED
* 0x30: LD - Load Word (32-bit)
* 0x31: ST - Store Word (32-bit)
* 0x32: LDB - Load Byte (8-bit)
* 0x33: STB - Store Byte (8-bit)
* 0x34: LDH - Load Half-word (16-bit)
* 0x35: STH - Store Half-word (16-bit)
* 0x36: SHLR - Shift Left Register
* 0x37: SHRR - Shift Right Logical Register
* 0x38: SARR - Shift Right Arithmetic Register
* 0x39: MULH - Multiply High (Signed)
* 0x3A: MULHU - Multiply High Unsigned
* 0x3B: CLI - Clear Interrupt Enable
* 0x3C: STI - Set Interrupt Enable
* 0x3D: XCHG - Atomic Exchange

* 0x3E: CAS - Compare and Swap
* 0x3F: RESERVED
* 0x40: SYSCALL - System Call
* 0x41-0xFD: RESERVED for future standardization
* 0xFE: RESERVED
* 0xFF: HALT - Halt Execution

Reserved ranges are allocated for future expansion. Any assignment of previously reserved opcodes requires standards action through the IETF.

11. Security Considerations

11.1. BPF Verifier as Trust Boundary

MBC security relies entirely on the eBPF verifier. An MBC program that fails verifier checks will not be loaded. The verifier ensures memory safety, bounds checking, and termination. An attacker cannot directly execute unverified MBC bytecode; all MBC programs must pass the eBPF verifier before execution.

11.2. Instruction Limit Prevents Infinite Loops

The 256-instruction limit per eBPF program guarantees that no single MBC execution can consume arbitrary CPU time. Loops must be bounded, enforced by verifier static analysis. This prevents denial-of-service attacks based on infinite loops within an MBC program.

11.3. No Privilege Separation

MBC has no user/kernel mode distinction. All MBC instructions execute at the same privilege level. This simplifies the ISA but means that any MBC code can access any memory location accessible to the eBPF program itself. Isolation between different tenants must be provided by the eBPF XDP program and kernel, not by MBC itself.

11.4. CAS Atomicity Limitations

The CAS (Compare and Swap) instruction is atomic with respect to eBPF interrupts on a single core. However, it does not provide atomicity across multiple cores or sockets. For multicore scenarios, external synchronization (e.g., distributed locks) is required. MBC programs MUST NOT assume that CAS provides multicore atomicity.

11.5. Memory Bounds Enforced by eBPF Maps

All memory access in MBC programs is validated by the eBPF verifier against allocated eBPF map sizes. Out-of-bounds accesses are prevented by the verifier. This guarantee is inherited from the eBPF execution model and does not require additional MBC-level validation.

11.6. CRC Validation Before MBC Execution

Before executing an MBC program on a Monad-framed packet, the eBPF XDP program MUST verify the packet's CRC (if present in the Monad header). Execution of MBC bytecode on corrupted packets can result in undefined behavior. The CRC check MUST occur before the first MBC instruction is executed.

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8200] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

[RFC9669] Thaler, D., Ed., "BPF Instruction Set Architecture (ISA)", RFC 9669, DOI 10.17487/RFC9669, October 2024, <https://www.rfc-editor.org/rfc/rfc9669>.

[FOUNDATION] Bellis, S., "The Monad Protocol Foundation Specification", Work in Progress, Internet-Draft, draft-bellis-unheaded-protocol-foundation-00, March 2026, <https://github.com/unheaded/unheaded>.

[WOTAN] Bellis, S., "Wotan: The Unheaded Message Bus Protocol", Work in Progress, Internet-Draft, draft-bellis-unheaded-wotan-memory-00, March 2026, <https://github.com/unheaded/wotan>.

12.2. Informative References

[RFC9000] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[SOPHIA] Bellis, S., "Sophia: The Unheaded Dictionary and Knowledge Graph Service", Work in Progress, Internet-Draft, draft-bellis-unheaded-sophia-dictionary-00, March 2026, <https://github.com/unheaded/unheaded>.

[EBPF-IO] eBPF Foundation, "eBPF: Introduction and Overview", <https://ebpf.io/>.

Author’s Address

Stevie Bellis
Unheaded
United States of America
Email: stevie@bellis.tech