Bitwise primitives proposal for Plutus

Koz Ross

February 25, 2022

1 Motivation

Bitwise operations are one of the most fundamental building blocks of algorithms and data structures. They can be used for a wide variety of applications, ranging from representing and manipulating sets of integers efficiently, to implementations of cryptographic primitives, to fast searches. Their wide availability, law-abiding behaviour and efficiency are the key reasons why they are widely used, and widely depended on.

At present, Plutus lacks meaningful support for bitwise operations, which significantly limits what can be usefully done on-chain. While it is possible to mimic some of these capabilities with what currently exists, and it is always possible to introduce new primitives for any task, this is extremely unsustainable, and often leads to significant inefficiencies and duplication of effort.

We describe a list of bitwise operations, as well as their intended semantics, designed to address this problem.

2 Example applications

We provide a range of applications that could be useful or beneficial on-chain, but are difficult or impossible to implement without some, or all, of the primitives we propose.

2.1 Succinct data structures

Due to the on-chain size limit, many data structures become impractical or impossible, as they require too much space either for their elements, or their overheads, to allow them to fit alongside the operations we want to perform on them. Succinct data structures could serve as a solution to this, as they represent data in an amount of space much closer to the entropy limit and ensure only constant overheads. There are several examples of these, and all rely on bitwise operations for their implementations.

For example, consider wanting to store a set of `BuiltinInteger`s on-chain. Given current on-chain primitives, the most viable option involves some variant on a `BuiltinList` of `BuiltinInteger`s; however, this is unviable in practice unless the set is small. To see why, suppose that we have an upper limit of k on the `BuiltinInteger`s we want to store; this is realistic in practically all cases. To store n `BuiltinInteger`s under the above scheme requires

$$n \cdot \left(\left\lceil \frac{\log_2(k)}{64} \right\rceil \cdot 64 + c \right)$$

bits, where c denotes the constant overhead for each cons cell of the `BuiltinList` holding the data. If the set being represented is dense (meaning that the number of entries is a sizeable fraction of k), this cost becomes intolerable quickly, especially when taking into account the need to also store the operations manipulating such a structure on-chain with the script where the set is being used.

If we instead represented the same set as a bitmap based on `BuiltinByteString`, the amount of space required would instead be

$$\left\lceil \frac{k}{8} \right\rceil \cdot 8 + \left\lceil \frac{\log_2(k)}{64} \right\rceil \cdot 64$$

bits. This is significantly better unless n is small. Furthermore, this representation would likely be more efficient in terms of time in practice, as instead of having to crawl through a cons-like structure, we can implement set operations on a memory-contiguous byte string:

- The cardinality of the set can be computed as a population count. This can have terrifyingly efficient implementations: the Muła-Kurz-Lemire algorithm (the current state of the art) can process four kilobytes per loop iteration, which amounts to over four thousand potential stored integers.
- Insertion or removal is a bit set or bit clear respectively.
- Finding the smallest element is a find-first-one.
- Testing for membership is a check to see if the bit is set.
- Set intersection is bitwise and.
- Set union is bitwise inclusive or.
- Set symmetric difference is bitwise exclusive or.

A potential implementation could use a range of techniques to make these operations extremely efficient, by relying on SWAR (SIMD-within-a-register) techniques if portability is desired, and SIMD instructions for maximum speed. This would allow both potentially large integer sets to be represented on-chain without breaking the size limit, and nodes to efficiently compute with such, reducing the usage of resources by the chain. Lastly, in practice, if compression techniques are used (which also rely on bitwise operations!), the number of required bits can be reduced considerably in most cases without compromising performance: the current state-of-the-art (Roaring Bitmaps) can be used as an example of the possible gains.

In order to make such techniques viable, bitwise primitives are mandatory. Furthermore, succinct data structures are not limited to sets of integers, but *all* require bitwise operations to be implementable.

2.2 On-chain vectors

For linear structures on-chain, we are currently limited to `BuiltinList` and `BuiltinMap`, which don't allow constant-time indexing. This is a significant restriction, especially when many data structures and algorithms rely on the broad availability of a constant-time-indexable linear structure, such as a C array or Haskell `Vector`. While we could introduce a primitive of this sort, this is a significant undertaking, and would require both implementing and costing a possibly large API.

While for variable-length data, we don't have any alternatives if constant-time indexing is a goal, for fixed-length (or limited-length at least) data, there is a possibility, based on a similar approach taken by the `finitary` library. Essentially, given finitary data, we can transform any item into a numerical index, which is then stored by embedding into a byte array. As the indexes are of a fixed maximum size, this can be done efficiently, but only if there is a way of converting indices into bitstrings, and vice versa. Such a construction would allow using a (wrapper around) `BuiltinByteString` as a constant-time indexable structure of any finitary type. This is not much of a restriction in practice, as on-chain, fixed-width or size-bounded types are preferable due to the on-chain size limit.

Currently, all the pieces to make this work already exist: the only missing piece is the ability to convert indices (which would have to be `BuiltinInteger`s) into bit strings (which would have to be `BuiltinByteString`s) and back again. With this capability, it would be possible to use these techniques to implement something like an array or vector without new primitive data types.

2.3 Binary representations and encodings

On-chain, space is at a premium. One way that space can be saved is with binary representations, which can potentially represent something much closer to the entropy limit, especially if the structure or value being represented has significant redundant structure. While some possibilities for a more efficient 'packing' already exist in the form of `BuiltinData`, it is rather idiosyncratic to the needs of Plutus, and its decoding is potentially quite costly.

Bitwise primitives would allow more compact binary encodings to be defined, where complex structures or values are represented using fixed-size `BuiltinByteString`s. The encoders and decoders for these could also be implemented more efficiently than currently possible, as there exist numerous bitwise techniques for this.

3 Goals

To ensure a focused and meaningful proposal, we specify our goals below.

3.1 Useful primitives

The primitives provided should enable implementations of algorithms and data structures that are currently impossible or impractical. Furthermore, the primitives provided should have a high power-to-weight ratio: having them should enable as much as possible to be implemented.

3.2 Maintaining as many algebraic laws as possible

Bitwise operations, via Boolean algebras, have a long and storied history of algebraic laws, dating back to important results by the like of de Morgan, Post and many others. These algebraic laws are useful for a range of reasons: they guide implementations, enable easier testing (especially property testing) and in some cases much more efficient implementations. To some extent, they also formalize our intuition about how these operations 'should work'. Thus, maintaining as many of these laws in our implementation, and being clear about them, is important.

3.3 Allowing efficient, portable implementations

Providing primitives alone is not enough: they should also be efficient. This is not least of all because many would associate 'primitive operation' with a notion of being 'close to the machine', and therefore fast. Thus, it is on us to ensure that the implementations of the primitives we provide have to be implementable in an efficient way, across a range of hardware.

3.4 Clear indication of failure

While totality is desirable, in some cases, there isn't a sensible answer for us to give. A good example is a division-by-zero: if we are asked to do such a thing, the only choice we have is to reject it. However, we need to make it as easy as possible for someone to realize why their program is failing, by emitting a sensible message which can later be inspected.

4 Non-goals

We also specify some specific non-goals of this proposal.

4.1 No metaphor-mixing between numbers and bits

A widespread legacy of C is the mixing of treatment of numbers and blobs of bits: specifically, the allowing of logical operations on representations of numbers. This applies to Haskell as much as any other language: according to the Haskell Report, it is in fact *required* that any type implementing `Bits` implement `Num` first. While GHC Haskell only mandates `Eq`, it still defines `Bits` instances for types clearly meant to represent numbers. This is a bad choice, as it creates complex situations and partiality in several cases, for arguably no real gain other than C-like bit twiddling code.

Even if two types share a representation, their type distinctness is meant to be a semantic or abstraction boundary: just because a number is represented as a blob of bits does not necessarily mean that arbitrary bit manipulations are sensible. However, by defining such a capability, we create several semantic problems:

- Some operations end up needing multiple definitions to take this into account. A good example are shifts: instead of simply having left or right shifts, we now have to distinguish *arithmetic* versus *logical* shifts, simply to take into account that a shift can be used on something which is meant to be a number, which could be signed. This creates unnecessary complexity and duplication of operations.
- As Plutus `BuiltinInteger`s are of arbitrary precision, certain bitwise operations are not well-defined on them. A good example is bitwise complement: the bitwise complement of 0 cannot be defined sensibly, and in fact, is partial in its `Bits` instance.
- Certain bitwise operations on `BuiltinInteger` would have quite undesirable semantic changes in order to be implementable. A good example are bitwise rotations: we should be able to 'decompose' a rotation left or right by n into two rotations (by m_1 and m_2 such that $m_1 + m_2 = n$) without changing the outcome. However, because trailing zeroes are not tracked by the implementation, this can fail depending on the choice of decomposition, which seems needlessly annoying for no good reason.

- Certain bitwise operations on `BuiltinInteger` would require additional arguments and padding to define them sensibly. Consider bitwise logical AND: in order to perform this sensibly on `BuiltinInteger`s we would need to specify what 'length' we assume they have, and some policy of 'padding' when the length requested is longer than one, or both, arguments. This feels unnecessary, and it isn't even clear exactly how we should do this: for example, how would negative numbers be padded?

These complexities, and many more besides, are poor choices, owing more to the legacy of C than any real useful functionality. Furthermore, they feel like a casual and senseless undermining of type safety and its guarantees for very small and questionable gains. Therefore, defining bitwise operations on `BuiltinInteger` is not something we wish to support.

There are legitimate cases where a conversion from `BuiltinInteger` to `BuiltinByteString` is desirable; this conversion should be provided, and be both explicit and specified in a way that is independent of the machine or the implementation of `BuiltinInteger`, as well as total and round-tripping. Arguably, it is also desirable to provide built-in support for `BuiltinByteString` literals specified in a way convenient to their treatment as blobs of bytes (for example, hexadecimal or binary notation), but this is outside the scope of this proposal.

5 Proposed operations

We propose several classes of operations. Firstly, we propose two operations for inter-conversion between `BuiltinByteString` and `BuiltinInteger`, whose semantics are specified in Subsection 6.2:

`integerToByteString :: BuiltinInteger -> BuiltinByteString`: Convert a number to a bitwise representation.

`byteStringToInteger :: BuiltinByteString -> BuiltinInteger`: Reinterpret a bitwise representation as a number.

We also propose several logical operations on `BuiltinByteString`s, whose semantics are specified in Subsection 6.3:

`andByteString :: BuiltinByteString -> BuiltinByteString -> BuiltinByteString`: Perform a bitwise logical AND on arguments of the same length, producing a result of the same length, erroring otherwise.

`iorByteString :: BuiltinByteString -> BuiltinByteString -> BuiltinByteString`: Perform a bitwise logical IOR on arguments of the same length, producing a result of the same length, erroring otherwise.

`xorByteString :: BuiltinByteString -> BuiltinByteString -> BuiltinByteString`: Perform a bitwise logical XOR on arguments of the same length, producing a result of the same length, erroring otherwise.

`complementByteString :: BuiltinByteString -> BuiltinByteString`: Complement all the bits in the argument, producing a result of the same length.

Lastly, we define the following additional operations, whose semantics are specified in Subsection 6.4:

`shiftByteString :: BuiltinByteString -> BuiltinInteger -> BuiltinByteString`: Performs a bitwise shift of the first argument by the absolute value of the second argument, with padding, the direction being indicated by the sign of the second argument.

`rotateByteString :: BuiltinByteString -> BuiltinInteger -> BuiltinByteString`: Performs a bitwise rotation of the first argument by the absolute value of the second argument, the direction being indicated by the sign of the second argument.

`popCountByteString :: BuiltinByteString -> BuiltinInteger`: Returns the number of 1 bits in the argument.

`testBitByteString :: BuiltinByteString -> BuiltinInteger -> BuiltinBool`: If the position given by the second argument is not in bounds for the first argument, error; otherwise, if the bit given by that position is 1, return `True`, and `False` otherwise.

`writeBitByteString :: BuiltinByteString -> BuiltinInteger -> BuiltinBool -> BuiltinByteString`: If the position given by the second argument is not in bound for the first argument, error; otherwise, set the bit given by that position to 1 if the third argument is `True`, and 0 otherwise.

`findFirstSetByteString :: BuiltinByteString -> BuiltinInteger`: Return the lowest index such that `testBitByteString` with the first argument and that index would be `True`. If no such index exists, return -1 instead.

5.1 Why these operations?

As stated in Subsection 4.1, there needs to be a well-defined interface between the 'world' of `BuiltinInteger` and `BuiltinByteString`. To provide this, we require `integerToByteString` and `byteStringToInteger`, which is designed to roundtrip (that is, describe an isomorphism). Furthermore, by spelling out a precise description of the conversions in Subsection 6.2, we make this predictable and portable.

Our choice of logical AND, IOR, XOR and complement as the primary logical operations is driven by a mixture of prior art, utility and convenience. These are the typical bitwise logical operations provided in hardware, and in most programming languages; for example, in the x86 instruction set, the following bitwise operations have existed since the 8086:

`AND`: Bitwise AND.

`OR`: Bitwise IOR.

`NOT`: Bitwise complement.

`XOR`: Bitwise XOR.

Likewise, on the ARM instruction set, the following bitwise operations have existed since ARM2:

`AND`: Bitwise AND.

`ORR`: Bitwise IOR.

`EOR`: Bitwise XOR.

`ORN`: Bitwise IOR with complement of the second argument.

`BIC`: Bitwise AND with complement of the second argument.

Going 'up a level', the C and Forth programming languages (according to C89 and ANS Forth respectively) define bitwise AND (denoted `&` and `AND` respectively), bitwise IOR (denoted `|` and `OR` respectively), bitwise XOR (denoted `^` and `XOR` respectively) and bitwise complement (denoted `~` and `NOT` respectively) as the primitive bitwise operations. This is followed by basically all languages 'higher-up' than C and Forth: Haskell's `Bits` type class defines these same four as `.&.`, `.|.`, `xor` and `complement`.

This ubiquity in choices leads to most algorithm descriptions that rely on bitwise operations to assume that these four are primitive, and thus, constant-time and cost. While we could reduce this number (and, in fact, due to Post, we know that there exist two *sole* sufficient operators), this would be both inconvenient and inefficient. As an example, consider implementing XOR using AND, IOR and complement: this would translate x XOR y into

$$(\text{COMPLEMENT } x \text{ AND } y) \text{ IOR } (x \text{ AND COMPLEMENT } y)$$

This is both needlessly complex and also inefficient, as it requires copying the arguments twice, only to throw away both copies.

Like our 'baseline' bitwise operations above, shifts and rotations are widely used, and considered as primitive. For example, x86 platforms have had the following available since the 8086:

`RCL`: Rotate left.

`RCR`: Rotate right.

`SHL`: Shift left.

`SHR`: Shift right.

Likewise, ARM platforms have had the following available since ARM2:

`ROR`: Rotate right.

`LSL`: Shift left.

`LSR`: Shift right.

While C and Forth both have shifts (denoted with `<<` and `>>` in C, and `LSHIFT` and `RSHIFT` in Forth), they don't have rotations; however, many higher-level languages do: Haskell's `Bits` type class has `rotate`, which enables both left and right rotations.

While `popCountByteString` could in theory be simulated using `testBitByteString` and a fold, this is quite inefficient: the best way to simulate this operation would involve using something similar to the Harley-Seal algorithm, which requires a large lookup table, making it impractical on-chain. Furthermore, population counting is important for several classes of succinct data structure (particularly rank-select dictionaries and bitmaps), and is in fact provided as part of the `SSE4.2` x86 instruction set as a primitive `POPCNT`.

In order to usefully manipulate individual bits, both `testBitByteString` and `writeBitByteString` are needed. They can also be used as part of specifying, and verifying, that other bitwise operations, both primitive and non-primitive, are behaving correctly. They are also particularly essential for binary encodings.

`findFirstSetByteString` is an essential primitive for several succinct data structures: both Roaring Bitmaps and rank-select dictionaries rely on it being efficient for much of their usefulness. Furthermore, this operation is provided in hardware by several instruction sets: on x86, there exist (at least) `BSF`, `BSR`, `LZCNT` and `TZCNT`, which allow finding both the first *and* last set bits, while on ARM, there exists `CLZ`, which can be used to simulate finding the first set bit. The instruction also exists in higher-level languages: for example, GHC's `FiniteBits` type class has `countTrailingZeros` and `countLeadingZeros`. The main reason we propose taking 'finding the first set bit' as primitive, rather than 'counting leading zeroes' or 'counting trailing zeroes' is that finding the first set bit is required specifically for several succinct data structures.

5.2 Costing

All of the primitives we describe are linear in one of their arguments. For a more precise description, see Table 1.

Primitive	Linear in
`integerToByteString`	Argument (only one)
`byteStringToInteger`	Argument (only one)
`andByteString`	One argument (same length for both)
`iorByteString`	One argument (same length for both)
`xorByteString`	One argument (same length for both)
`complementByteString`	Argument (only one)
`shiftByteString`	`BuiltinByteString` argument
`rotateByteString`	`BuiltinByteString` argument
`popCountByteString`	Argument (only one)
`testBitByteString`	`BuiltinByteString` argument
`writeBitByteString`	`BuiltinByteString` argument
`findFirstSetByteString`	Argument (only one)

Table 1: Primitives and which argument they are linear in

6 Semantics

6.1 Preliminaries

We define $\mathbb{N}^+ = \{x \in \mathbb{N} \mid x \neq 0\}$. We assume that `BuiltinInteger` is a faithful representation of $\mathbb{Z}$. A *bit sequence* $s = s_n s_{n-1} \dots s_0$ is a sequence such that for all $i \in \{0, 1, \dots, n\}$, $s_i \in \{0, 1\}$. A bit sequence $s = s_n s_{n-1} \dots s_0$ is a

byte sequence if $n = 8k - 1$ for some $k \in \mathbb{N}$. We denote the *empty bit sequence* (and, indeed, byte sequence as well) by $\emptyset$.

We intend that `BuiltinByteString`s represent byte sequences, with the sequence of bits being exactly as the description above. For example, given the byte sequence `0110111100001100`, the `BuiltinByteString` corresponding to it would be `"o\f"`.

Let $i \in \mathbb{N}^+$. We define the sequence $\texttt{binary}(i) = (d_0, m_0), (d_1, m_1), \ldots$ as

1. $m_0 = i \mod 2$, $d_0 = \frac{i}{2}$ if i is even, and $\frac{i-1}{2}$ if it is odd.
2. $m_j = d_{j-1} \mod 2$, $d_j = \frac{d_{j-1}}{2}$ if d_j is even, and $\frac{d_{j-1}-1}{2}$ if it is odd.

6.2 Representation of `BuiltinInteger` as `BuiltinByteString` and conversions

We describe the translation of `BuiltinInteger` into `BuiltinByteString` which is implemented as the `integerToByteString` primitive. Informally, we represent `BuiltinInteger`s with the least significant bit at bit position 0, using a twos-complement representation. More precisely, let $i \in \mathbb{N}^+$. We represent i as the bit sequence $s = s_n s_{n-1} \ldots s_0$, such that:

1. $\sum_{j \in \{0,1,\ldots,n\}} s_j \cdot 2^j = i$; and
2. $s_n = 0$.
3. Let $\texttt{binary}(j) = (d_0, m_0), (d_1, m_1), \ldots$. For any $j \in \{0, 1, \ldots, n-1\}$, $s_j = m_j$; and
4. $n + 1 = 8k$ for the smallest $k \in \mathbb{N}^+$ consistent with the previous requirements.

For 0, we represent it as the sequence `00000000` (one zero byte). We represent any $i \in \{x \in \mathbb{Z} \mid x < 0\}$ as the twos-complement of the representation of its additive inverse. We observe that any such sequence is by definition a byte sequence.

To interpret a byte sequence $s = s_n s_{n-1} \ldots s_0$ as a `BuiltinInteger`, we use the following process:

1. If s is `00000000`, then the result is 0.
2. Otherwise, if $s_n = 1$, let s' be the twos-complement of s. Then the result is the additive inverse of the result of interpreting s'.
3. Otherwise, the result is $\sum_{i \in \{0,1,\ldots,n\}} s_i \cdot 2^i$.

The above interpretation is implemented as the `byteStringToInteger` primitive. We observe that `byteStringToInteger` and `integerToByteString` form an isomorphism. More specifically:

```
byteStringToInteger . integerToByteString =
integerToByteString . byteStringToInteger =
id
```

6.3 Bitwise logical operations on `BuiltinByteString`

Throughout, let $s = s_n s_{n-1} \ldots s_0$ and $t = t_m t_{m-1} \ldots t_0$ be two byte sequences. Whenever we specify a *mismatched length error* result, its error message must contain at least the following information:

- The name of the failed operation;
- The reason (mismatched lengths); and
- The lengths of the arguments.

We describe the semantics of `andByteString`. For inputs s and t, if $n \neq m$, the result is a mismatched length error. Otherwise, the result is the byte sequence $u = u_n u_{n-1} \ldots, u_0$ such that for all $i \in \{0, 1, \ldots, n\}$ we have

$$u_i = \begin{cases} 1 & s_i = t_i = 1 \\ 0 & \text{otherwise} \end{cases}$$

For `iorByteString`, for inputs s and t, if $n \neq m$, the result is a mismatched length error. Otherwise, the result is the byte sequence $u = u_n u_{n-1} \dots u_0$ such that for all $i \in \{0, 1, \dots, n\}$ we have

$$u_i = \begin{cases} 1 & s_i = 1 \\ 1 & t_i = 1 \\ 0 & \text{otherwise} \end{cases}$$

For `xorByteString`, for inputs s and t, if $n \neq m$, the result is a mismatched length error. Otherwise, the result is the byte sequence $u = u_n u_{n-1} \dots u_0$ such that for all $i \in \{0, 1, \dots, n\}$ we have

$$u_i = \begin{cases} 0 & s_i = t_i \\ 1 & \text{otherwise} \end{cases}$$

We observe that, for length-matched arguments, each of `andByteString`, `iorByteString` and `xorByteString` describes a commutative and associative operation. Furthermore, for any given length k, each of these operations have an identity element: for `iorByteString`, this is the bit sequence of length k where each element is 0, and for `andByteString` and `xorByteString`, this is the bit sequence of length k where each element is 1. Lastly, for any length k, the bit sequence of length k where each element is 0 is an absorbing element for `andByteString`, and the bit sequence of length k where each element is 1 is an absorbing element for `iorByteString`.

We now describe the semantics of `complementByteString`. For input s, the result is the byte sequence $u = u_n u_{n-1} \dots u_0$ such that for all $i \in \{0, 1, \dots, n\}$ we have

$$u_i = \begin{cases} 1 & s_i = 0 \\ 0 & \text{otherwise} \end{cases}$$

We observe that `complementByteString` is self-inverting. We also note the following equivalences hold assuming `b` and `b'` have the same length; these are the DeMorgan laws:

```
complementByteString (andByteString b b') =
iorByteString (complementByteString b) (complementByteString b')

complementByteString (iorByteString b b') =
andByteString (complementByteString b) (complementByteString b')
```

6.4 Mixed operations

Throughout this section, let $s = s_n s_{n-1} \dots s_0$ and $t = t_m t_{m-1} \dots t_0$ be byte sequences, and let $i \in \mathbb{Z}$.

We describe the semantics of `shiftByteString`. Informally, these are logical shifts, with negative shifts 'moving' away from bit index 0, and positive shifts 'moving' towards bit index 0. More precisely, given the argument s and i, the result of `shiftByteString` is the byte sequence $u_n u_{n-1} \dots u_0$, such that for all $j \in \{0, 1, \dots, n\}$, we have

$$u_j = \begin{cases} s_{j+i} & j - i \in \{0, 1, \dots, n\} \\ 0 & \text{otherwise} \end{cases}$$

We observe that for k, ℓ with the same sign and any `bs`, we have

```
shiftByteString (shiftBytestring bs k) l = shiftByteString bs (k + l)
```

We now describe `rotateByteString`, assuming the same inputs as the description of `shiftByteString` above. Informally, the 'direction' of the rotations matches that of `shiftByteString` above. More precisely, the result of `rotateByteString` on the given inputs is the byte sequence $u_n u_{n-1} \dots u_0$ such that for all $j \in \{0, 1, \dots, n\}$, we have $u_j = s_{j+i \mod (n+1)}$. We observe that for any k, ℓ, and any `bs`, we have

```
rotateByteString (rotateByteString bs k) l = rotateByteString bs (k + l)
```

We also note that

```
rotateByteString bs 0 = shiftByteString bs 0 = bs
```

For `popCountByteString` with argument s, the result is

$$\sum_{j \in \{0,1,\ldots,n\}} s_j$$

Informally, this is just the total count of 1 bits. We observe that for any `bs` and `bs'`, we have

```
popCountByteString bs + popCountByteString bs' =
popCountByteString (appendByteString bs bs')
```

We now describe the semantics of `testBitByteString` and `writeBitByteString`. Throughout, whenever we specify an *out-of-bounds error* result, its error message must contain at least the following information:

- The name of the failed operation;
- The reason (out of bounds access);
- What index was accessed out-of-bounds; and
- The valid range of indexes.

For `testBitByteString` with arguments s and i, if $0 \leq i \leq n$, then the result is `True` if $s_i = 1$, and `False` if $s_i = 0$; otherwise, the result is an out-of-bounds error. Let `b :: BuiltinBool`; for `writeBitByteString` with arguments s, i and `b`, if $0 \leq i \leq n$, then the result is the byte sequence $u_n u_{n-1} \ldots u_0$ such that for all $j \in \{0, 1, \ldots, n\}$, we have

$$u_j = \begin{cases} 1 & i = j \text{ and } \texttt{b} = \texttt{True} \\ 0 & i = j \text{ and } \texttt{b} = \texttt{False} \\ s_j & \text{otherwise} \end{cases}$$

If $i < 0$ or $i > n$, the result is an out-of-bounds error.

Lastly, we describe the semantics of `findFirstSetByteString`. Given the argument s, if for any $j \in \{0, 1, \ldots, n\}$, $s_j = 0$, the result is -1; otherwise, the result is k such that all of the following hold:

- $k \in \{0, 1, \ldots, n\}$;
- $s_k = 1$; and
- For all $0 \leq k' < k$, $s_{k'} = 0$.