1. **Module Bitv.** This module implements bit vectors, as an abstract datatype \( t \). Since bit vectors are particular cases of arrays, this module provides the same operations as module Array (Sections 2 up to 6). It also provides bitwise operations (Section 10) and conversions to/from integer types.

In the following, \texttt{false} stands for bit 0 and \texttt{true} for bit 1.

type \( t \)

2. **Creation, access and assignment.** \( (\text{Bitv.create } n \ b) \) creates a new bit vector of length \( n \), initialized with \( b \). \( (\text{Bitv.init } n \ f) \) returns a fresh vector of length \( n \), with bit number \( i \) initialized to the result of \( (f \ i) \). \( (\text{Bitv.set } v \ n \ b) \) sets the \( n \)th bit of \( v \) to the value \( b \). \( (\text{Bitv.get } v \ n) \) returns the \( n \)th bit of \( v \). \texttt{Bitv.length} returns the length (number of elements) of the given vector.

val create : int \( \rightarrow \) bool \( \rightarrow \) \( t \)
val init : int \( \rightarrow \) (int \( \rightarrow \) bool) \( \rightarrow \) \( t \)
val set : \( t \) \( \rightarrow \) int \( \rightarrow \) bool \( \rightarrow \) unit
val get : \( t \) \( \rightarrow \) int \( \rightarrow \) bool
val length : \( t \) \( \rightarrow \) int

3. **max_length** is the maximum length of a bit vector (System dependent).

val max_length : int

4. **Copies and concatenations.** \( (\text{Bitv.copy } v) \) returns a copy of \( v \), that is, a fresh vector containing the same elements as \( v \). \( (\text{Bitv.append } v1 \ v2) \) returns a fresh vector containing the concatenation of the vectors \( v1 \) and \( v2 \). \texttt{Bitv.concat} is similar to \texttt{Bitv.append}, but concatenates a list of vectors.

val copy : \( t \) \( \rightarrow \) \( t \)
val append : \( t \) \( \rightarrow \) \( t \) \( \rightarrow \) \( t \)
val concat : \( t \) \( \rightarrow \) \( t \) \( \rightarrow \) \( t \)

5. **Sub-vectors and filling.** \( (\text{Bitv.sub } v \ start \ len) \) returns a fresh vector of length \( len \), containing the bits number \( start \) to \( start + len - 1 \) of vector \( v \). Raise \texttt{Invalid_argument "Bitv.sub"} if \( start \) and \( len \) do not designate a valid subvector of \( v \); that is, if \( start < 0 \), or \( len < 0 \), or \( start + len > \text{Bitv.length} a \).

\( (\text{Bitv.fill } v \ ofs \ len \ b) \) modifies the vector \( v \) in place, storing \( b \) in elements number \( ofs \) to \( ofs + len - 1 \). Raise \texttt{Invalid_argument "Bitv.fill"} if \( ofs \) and \( len \) do not designate a valid subvector of \( v \).
(Bitv.blit v1 o1 v2 o2 len) copies len elements from vector v1, starting at element number o1, to vector v2, starting at element number o2. It does not work correctly if v1 and v2 are the same vector with the source and destination chunks overlapping. Raise Invalid_argument "Bitv.blit" if o1 and len do not designate a valid subvector of v1, or if o2 and len do not designate a valid subvector of v2.

val sub : t → int → int → t
val fill : t → int → int → bool → unit
val blit : t → int → t → int → int → unit

6. Iterators. (Bitv.iter f v) applies function f in turn to all the elements of v. Given a function f, (Bitv.map f v) applies f to all the elements of v, and builds a vector with the results returned by f. Bitv.iteri and Bitv.mapi are similar to Bitv.iter and Bitv.map respectively, but the function is applied to the index of the element as first argument, and the element itself as second argument.

(Bitv.fold_left f x v) computes f (... (f (f x (get v 0)) (get v 1)) ...) (get v (n - 1)), where n is the length of the vector v.
(Bitv.fold_right f a x) computes f (get v 0) (f (get v 1) (... (f (get v (n - 1)) x) ...)), where n is the length of the vector v.

val iter : (bool → unit) → t → unit
val map : (bool → bool) → t → t
val iteri : (int → bool → unit) → t → unit
val mapi : (int → bool → bool) → t → t
val fold_left : (α → bool → α) → α → t → α
val fold_right : (bool → α → α) → t → α → α
val foldi_left : (α → int → bool → α) → α → t → α
val foldi_right : (int → bool → α → α) → t → α → α

7. Population count, i.e., number of 1 bits
val pop : t → int

8. iteri_true f v applies function f in turn to all indexes of the elements of v which are set (i.e. true); indexes are visited from least significant to most significant.

val iteri_true : (int → unit) → t → unit

9. gray_iter f n iterates function f on all bit vectors of length n, once each, using a Gray code. The order in which bit vectors are processed is unspecified.

val gray_iter : (t → unit) → int → unit
10. **Bitwise operations.** \(bw\text{and}, bw\text{or} \) and \(bw\text{xor}\) implement logical and, or and exclusive or. They return fresh vectors and raise \texttt{Invalid_argument} "\texttt{Bitv.xxx}" if the two vectors do not have the same length (where \texttt{xxx} is the name of the function). \(bw\text{not}\) implements the logical negation. It returns a fresh vector. \(shiftl\) and \(shiftr\) implement shifts. They return fresh vectors. \(shiftl\) moves bits from least to most significant, and \(shiftr\) from most to least significant (think \texttt{lsl} and \texttt{lsr}). \texttt{all_zeros} and \texttt{all_ones} respectively test for a vector only containing zeros and only containing ones.

\[
\begin{align*}
\text{val } bw\_and &: \ t \to t \to t \\
\text{val } bw\_or &: \ t \to t \to t \\
\text{val } bw\_xor &: \ t \to t \to t \\
\text{val } bw\_not &: \ t \to t \\
\text{val } shiftl &: \ t \to \text{int} \to t \\
\text{val } shiftr &: \ t \to \text{int} \to t \\
\text{val } all\_zeros &: \ t \to \text{bool} \\
\text{val } all\_ones &: \ t \to \text{bool}
\end{align*}
\]

11. **Conversions to and from strings.**
With least significant bits first.

\[
\begin{align*}
\text{module } L &: \ \text{sig} \\
\text{val } \text{to\_string} &: \ t \to \text{string} \\
\text{val } \text{of\_string} &: \ \text{string} \to t \\
\text{val } \text{print} &: \ \text{Format.formatter} \to t \to \text{unit}
\end{align*}
\]

With most significant bits first.

\[
\begin{align*}
\text{module } M &: \ \text{sig} \\
\text{val } \text{to\_string} &: \ t \to \text{string} \\
\text{val } \text{of\_string} &: \ \text{string} \to t \\
\text{val } \text{print} &: \ \text{Format.formatter} \to t \to \text{unit}
\end{align*}
\]

12. **Input/output in a machine-independent format.** The following functions export/import a bit vector to/from a channel, in a way that is compact, independent of the machine architecture, and independent of the OCaml version. For a bit vector of length \(n\), the number of bytes of this external representation is \(4+\text{ceil}(n/8)\) on a 32-bit machine and \(8+\text{ceil}(n/8)\) on a 64-bit machine.

\[
\begin{align*}
\text{val } \text{output\_bin} &: \ \text{out\_channel} \to t \to \text{unit} \\
\text{val } \text{input\_bin} &: \ \text{in\_channel} \to t
\end{align*}
\]
13. Conversions to and from lists of integers. The list gives the indices of bits which are set (ie true).

```plaintext
val to_list : t → int list
val of_list : int list → t
val of_list_with_length : int list → int → t
```

14. Interpretation of bit vectors as integers. Least significant bit comes first (ie is at index 0 in the bit vector). `to_xxx` functions truncate when the bit vector is too wide, and raise `Invalid_argument` when it is too short. Suffix `_s` means that sign bit is kept, and `_us` that it is discarded.

Type `int` (length 31/63 with sign, 30/62 without)

```plaintext
val of_int_s : int → t
val to_int_s : t → int
val of_int_us : int → t
val to_int_us : t → int
(* type Int32.t (length 32 with sign, 31 without) *)
val of_int32_s : Int32.t → t
val to_int32_s : t → Int32.t
val of_int32_us : Int32.t → t
val to_int32_us : t → Int32.t
(* type Int64.t (length 64 with sign, 63 without) *)
val of_int64_s : Int64.t → t
val to_int64_s : t → Int64.t
val of_int64_us : Int64.t → t
val to_int64_us : t → Int64.t
(* type Nativeint.t (length 32/64 with sign, 31/63 without) *)
val of_nativeint_s : Nativeint.t → t
val to_nativeint_s : t → Nativeint.t
val of_nativeint_us : Nativeint.t → t
val to_nativeint_us : t → Nativeint.t
```

15. Only if you know what you are doing...

```plaintext
val unsafe_set : t → int → bool → unit
val unsafe_get : t → int → bool
```