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 the module *Array* (Sections 2 up to 6). It also provides bitwise operations (Section 9). In the following, false stands for the bit 0 and true for the bit 1.

type t

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

val *create* : *int* → *bool* → *t*

val *init* : *int* → (*int* → *bool*) → *t*

val *set* : *t* → *int* → *bool* → *unit*

val *get* : *t* → *int* → *bool*

val *length* : *t* → *int*

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

val *max_length* : *int*

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

val *copy* : *t* → *t*

val *append* : *t* → *t* → *t*

val *concat* : *t list* → *t*

5. Sub-vectors and filling. (*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 *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* > *Bitv.length a*.

(*Bitv.fill v ofs len b*) modifies the vector v in place, storing b in elements number *ofs* to *ofs* + *len* − 1. Raise *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 $sub : t \rightarrow int \rightarrow int \rightarrow t$

val $fill : t \rightarrow int \rightarrow int \rightarrow bool \rightarrow unit$

val $blit : t \rightarrow int \rightarrow t \rightarrow int \rightarrow int \rightarrow unit$

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 $iter : (bool \rightarrow unit) \rightarrow t \rightarrow unit$
val $map : (bool \rightarrow bool) \rightarrow t \rightarrow t$

val $iteri : (int \rightarrow bool \rightarrow unit) \rightarrow t \rightarrow unit$
val $mapi : (int \rightarrow bool \rightarrow bool) \rightarrow t \rightarrow t$

val $fold_left : (\alpha \rightarrow bool \rightarrow \alpha) \rightarrow \alpha \rightarrow t \rightarrow \alpha$
val $fold_right : (bool \rightarrow \alpha \rightarrow \alpha) \rightarrow t \rightarrow \alpha \rightarrow \alpha$
val $foldi_left : (\alpha \rightarrow int \rightarrow bool \rightarrow \alpha) \rightarrow \alpha \rightarrow t \rightarrow \alpha$
val $foldi_right : (int \rightarrow bool \rightarrow \alpha \rightarrow \alpha) \rightarrow t \rightarrow \alpha \rightarrow \alpha$

7. *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 $iteri_true : (int \rightarrow unit) \rightarrow t \rightarrow unit$

8. *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 $gray_iter : (t \rightarrow unit) \rightarrow int \rightarrow unit$

9. Bitwise operations. *bwand*, *bwor* and *bwxor* implement logical and, or and exclusive or. They return fresh vectors and raise *Invalid_argument* `"Bitv.xxx"` if the two vectors do not have the same length (where `xxx` is the name of the function). *bwnot* 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 lsl and lsr). *all_zeros* and *all_ones* respectively test for a vector only containing zeros and only containing ones.

```
val bw_and : t → t → t
val bw_or : t → t → t
val bw_xor : t → t → t
val bw_not : t → t

val shiftl : t → int → t
val shiftr : t → int → t

val all_zeros : t → bool
val all_ones : t → bool
```

10. Conversions to and from strings.
With least significant bits first.

```
module L : sig
  val to_string : t → string
  val of_string : string → t
  val print : Format.formatter → t → unit
end
```

With most significant bits first.

```
module M : sig
  val to_string : t → string
  val of_string : string → t
  val print : Format.formatter → t → unit
end
```

11. Conversions to and from lists of integers. The list gives the indices of bits which are set (ie true).

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

12. 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* indicates that sign bit is kept, and *_us* that it is discarded.
type *int* (length 31/63 with sign, 30/62 without)

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*

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

val *unsafe_set* : *t* → *int* → *bool* → *unit*
val *unsafe_get* : *t* → *int* → *bool*