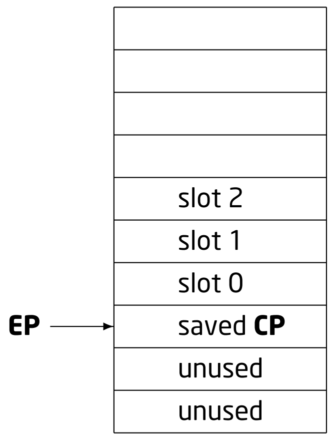
BEAM instructions semantics

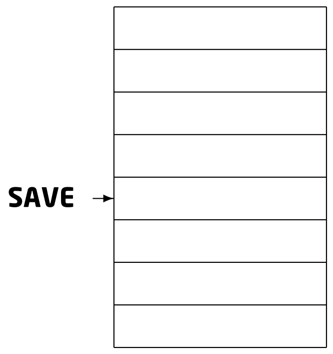
February 16, 2012

Abstract machine elements

1. Registers. R0 is a special fast register.
2. **IP** (instruction pointer) contains the reference to the instruction being executed.
3. **CP** (continuation pointer) contains a return address of the current function call.
4. The stack:



5. **EP** (stack pointer) points to the last value pushed on stack.
6. **HTOP** (heap top) points to the first empty word of the heap.
7. The message queue:



8. **SAVE** pointer marks the message being currently matched.
9. **tmp_arg1** and **tmp_arg2** variables that temporarily hold values for the imminent arithmetic or logical operation.

Instructions semantics

`allocate t t`

Allocates Arg0 slots on the stack. Arg1 is Live. See Fig. 1.

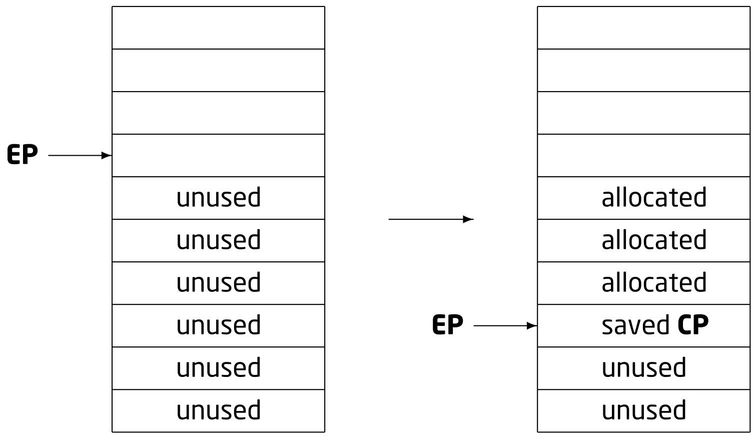


Figure 1 Allocating 3 slots on the stack.

`allocate_heap t I t`

Allocates Arg0 slots on the stack. See `allocate t t`. Ensures that heap contains at least Arg1 words after **HTOP**. Arg2 is Live.

`deallocate I`

Restores **CP** to the value references by **EP**. Adds Arg0+1 to **EP** to remove the saved **CP** and Arg0 slots from the stack.

`init y`

Sets the Arg0 slot to nil. The op was called "kill" earlier.

`init2 y y`

Sets slots Arg0 and Arg1 to nil.

`init3 y y y`

Sets slots Arg0, Arg1, and Arg2 to nil.

`i_trim I`

Removes Arg0 slots from the stack. Th saved **CP** is moved to the new top of the stack.

`test_heap I t`

Ensures that the heap has at least Arg0 words available. Arg1 is Live.

`allocate_zero t t`

Same as `allocate t t` but sets the slots to nil.

`allocate_heap_zero t I t`

Same as `allocate_heap t I t` but sets the slots to nil.

`i_select_val r f I`
`i_select_val x f I`
`i_select_val y f I`

Searches an array of `{val,addr}` pairs for the value of Arg0. The length of the array is Arg2. If the value is found the execution continues at the corresponding `addr`. Otherwise, jumps to Arg1. Uses binary search and *bitwise* comparisons.

`i_select_val2 r f c f c f`
`i_select_val2 x f c f c f`
`i_select_val2 y f c f c f`

Same as `i_select_val rxy f I` but uses an array made of `{Arg2,Arg3}` and `{Arg4,Arg5}`.

`i_jump_on_val rxy f I I`

Implements a "jump table". Calculates the index by adding Arg0 to the minimum value Arg2. Jumps to Arg1 if the index falls outside of the table boundaries. Arg3 is the length of the jump table. Picks the label from the table using the calculated index and jumps there.

`i_jump_on_val_zero rxy f I`

Same as `i_jump_on_val rxy f I I` but the minimum value is assumed to be zero.

`i_select_tuple_arity r f I`
`i_select_tuple_arity x f I`
`i_select_tuple_arity y f I`

Checks that Arg0 is tuple and then searches the array for `{arity,addr}` for the right arity. See `i_select_val rxy f I`.

`i_select_tuple_arity2 r f A f A f`
`i_select_tuple_arity2 x f A f A f`
`i_select_tuple_arity2 y f A f A f`

Same as `i_select_tuple_arity rxy f I` but uses the array made up of two pairs: `{Arg2,Arg3}` and `{Arg4,Arg5}`.

`i_func_info I a a I`

Raises `function_clause` exception. Arg1, Arg2, Arg3 are MFA. Arg0 is a mystery coming out of nowhere, probably, needed for NIFs.

`return`

IP is set to **CP**.

`get_list rxy rxy rxy`

Gets the head of the list from Arg0 and puts it to Arg1, the tail – to Arg2.

`catch y f`

Creates a new "catch" context. Saves value of the label from Arg1 to the Arg0 stack slot.

`catch_end y`

Pops a "catch" context. Erases the label saved in the Arg0 slot. Noval in R0 indicates that something is caught. If R1 contains atom `throw` then R0 is set to R2. If R1 contains atom `error` than a stack trace is added to R2. R0 is set to `{exit,R2}`.

`try_end y`

Pops a "catch" context. Erases the label saved in the Arg0 slot. Noval in R0 indicate that something is caught. If so, R0 is set to R1, R1 – to R2, R2 – to R3.

`try_case_end s`

Raises a `try_clause` exception with the value read from Arg0.

`set_tuple_element s d P`

Sets the Arg2'th element of the tuple Arg1 to to the value of Arg0. The index in Arg2 is 1-based.

`i_get_tuple_element rxy P rxy`

Gets the value of the the Arg1'th element of the tuple Arg0 and puts it to Arg2. Arg1 is 1-based.

`is_number f rxy`

Jumps to Arg0 if Arg1 is not integer and is not float.

`jump f`

Continues execution at the location Arg0.

case_end rxy

Raises the `case_clause` exception with the value of Arg0.

badmatch rxy

Raises the `badmatch` exception with the value of Arg0.

if_end

Raises the `if_clause` exception.

raise s s

Raises the exception. The instruction is garbled by backward compatibility. Arg0 is a stack trace and Arg1 is the value accompanying the exception. The reason of the raised exception is dug up from the stack trace.

badarg j

If Arg0 is not 0 then jumps to the label Arg0. Otherwise, raises the `badarg` exception.

system_limit j

Similar to `badarg j` but raises `{system_limit}` exception.

move_jump f ncxy

Move the value from Arg1 to R0 and then jumps to the label Arg0.

move_x1 c

Sets R1 to the value of Arg0.

move_x2 c

Sets R2 to the value of Arg0.

```
move2 x y x y
move2 y x y x
move2 x x x x
```

Combine two move operations.

```
move rxync rxy
```

Copies Arg0 to Arg1.

```
recv_mark f
```

Saves the last pointer of the message queue and the label of the soon expected `loop_rec f r`.

```
i_recv_set f
```

Sets the **SAVE** pointer to the saved last pointer if the label is right. This is somehow needed for in-order processing of monitoring messages.

```
remove_message
```

Removes the (matched) message from the message queue.

```
timeout
```

Timeout has occured. Resets the **SAVE** pointer to the first message in the message queue.

```
timeout_locked
```

Same as `timeout` but with SMP precautions.

```
i_loop_rec f r
```

Picks the next message from the message queue and places it in R0. If there are no messages then jumps to Arg0 which points to `wait` or `wait_timeout` instruction.

```
loop_rec_end f
```

Advances the **SAVE** pointer to the next message as the current message did not match. Jumps to Arg0 that points to `loop_rec f r` instruction.

`wait f`

Prepares to wait indefinitely for a message to arrive. When the message arrives transfers control to the `loop_rec` instruction at the label Arg0.

`wait_locked f`

Same as `wait f`.

`wait_unlocked f`

Same as `wait f` but messes with SMP locks.

`i_wait_timeout f I`
`i_wait_timeout f s`

Sets the timer to the appropriate time as indicated by Arg1. Afterwards acts as `wait f`.

`i_wait_timeout_locked f I`
`i_wait_timeout_locked f s`

Same as `i_wait_timeout f I` but with SMP trickery.

`i_wait_error`

Raises `timeout_value` exception.

`i_wait_error_locked`

Same as `i_wait_error` but with due respect to SMP.

`send`

Sends the message from R1 to the process R0.

`i_is_eq_exact_immed f rxy c`

Checks if Arg1 equals Arg2 using *bitwise* comparison. Jumps to Arg0 it does not.

`i_is_ne_exact_immed f rxy c`

Checks if Arg1 equals Arg2 using *bitwise* comparison. Jumps to Arg0 it does.

`i_is_eq_exact_literal f rxy c`

Checks if Arg1 equals Arg2 using the term comparison. Jumps to Arg0 it does not.

`i_is_ne_exact_literal f rxy c`

Checks if Arg1 equals Arg2 using the term comparison. Jumps to Arg0 it does.

`i_is_eq_exact f`

Checks if **tmp_arg1** "exactly" equals **tmp_arg2**. Jumps to the label Arg0 if it does not.

`i_is_ne_exact f`

Checks if **tmp_arg1** "exactly" equals **tmp_arg2**. Jumps to the label Arg0 it does.

`i_is_lt f`

Checks if **tmp_arg1** is less than **tmp_arg2**. Jumps to the label Arg0 if it is not.

`i_is_ge f`

Checks if **tmp_arg1** is greater than or equal to **tmp_arg2**. Jumps to the label Arg0 if it is not.

`i_is_eq f`

Checks if **tmp_arg1** is equal to **tmp_arg2**. Jumps to the label Arg0 if it is not.

`i_is_ne f`

Checks if **tmp_arg1** is equal to **tmp_arg2**. Jumps to the label Arg0 if it is.

`i_put_tuple rxy I`

Creates a tuple on the heap and places it in Arg0. Arg1 is the arity of the tuple. The elements of the tuple follow the instruction. Some of the elements may refer to registers or stack slots.

`put_list s s d`

Create a cons cell on the heap and places in onto Arg2. Arg0 is the head of the list saved to **HTOP**[0] and Arg1 is the tail saved to **HTOP**[1].

`i_fetch s s`

Fetches values from Arg0 and Arg1 and places them in **tmp_arg1** and **tmp_arg2** respectively.

`move_return xcn r`

Combines `move xcn r` and `return`.

`move_deallocate_return xycn r Q`

Combines `move xycn r`, `deallocate Q`, and `return`.

`deallocate_return Q`

Combines `deallocate Q` and `return`.

`test_heap_1_put_list I y`

Test that the heap has a room for Arg0 words and than conses value of Arg1 with R0.

`is_tuple_of_arity f rxy A`

Jumps to the label Arg0 if Arg1 is not a tuple or its arity is not Arg2. If Arg1 is tuple its value is saved to **tmp_arg1**.

`is_tuple f rxy`

Jumps to the label Arg0 if Arg1 is not a tuple.

`test_arity f rxy A`

Jumps to the label Arg0 is the arity of the tuple Arg1 is not Arg2. The value of Arg1 is saved to **tmp_arg1**.

`extract_next_element xy`

The pointer in **tmp_arg1** moved to the next element of the tuple. The element is read from the location pointed to by **tmp_arg1** to Arg0.

`extract_next_element2 xy`

The pointer in **tmp_arg1** moved to the next element of the tuple. Copies 2 terms from the array pointed to by **tmp_arg1** to the registers or slots pointed to by Arg0. For instance, if Arg0 is R3 then tuple elements are loaded to R3 and R4.

`extract_next_element3 xy`

Same as `extract_next_element2 xy` but copies three elements.

`is_integer_allocate f rx I I`

Combines `is_integer f rx` and `allocate I I`.

`is_integer f rxy`

Jumps to the label Arg0 if Arg1 is not integer.

`is_list f rxy`

Jumps to the label Arg0 if Arg1 is not a list.

`is_nonempty_list_allocate f rx I t`

Combines `is_nonempty_list f rx` and `allocate I t`.

`is_nonempty_list_test_heap f r I t`

Combines `is_nonempty_list f r` and `test_heap I t`.

`is_nonempty_list f rxy`

Jumps to the label Arg0 if Arg1 is not an nonempty list.

`is_atom f rxy`

Jumps to the label Arg0 if Arg1 is not an atom.

`is_float f rxy`

Jumps to the label Arg0 if Arg1 is not a float.

`is_nil f rxy`

Jumps to the label Arg0 if Arg1 is not nil.

`is_bitstring f rxy`

Jumps to the label Arg0 if Arg1 is not a bitstring (or a binary).

`is_reference f rxy`

Jumps to the label Arg0 if Arg1 is not a reference.

`is_pid f rxy`

Jumps to the label Arg0 if Arg1 is not a pid.

`is_port f rxy`

Jumps to the label Arg0 if Arg1 is not a port.

`is_boolean f rxy`

Jumps to the label Arg0 if Arg1 is not true or false.

`is_function2 f s s`

Jumps to the label Arg0 if Arg1 is not fun or have an arity different from Arg1.

`allocate_init t I y`

Combines `allocate t I` and `init y`.

`i_apply`

Applies the function identified by R0 (module), R1 (function), and R2 (arguments). Sets **CP** to the point after the instruction.

`i_apply_last P`

Applies the function identified by R0 (module), R1 (function), and R2 (arguments). Afterwards, restores **CP** from stack and deallocates Arg0 slots from stack.

`i_apply_only`

Applies the function identified by R0 (module), R1 (function), and R2 (arguments).

`apply I`

Finds the address from the export entry for the function (possibly BIF) identified by the module R(N), the function R($N + 1$), and the arity N. The arity is taken from Arg0. Sets **CP** to the next instruction and jumps to the found address.

`apply_last I P`

Similar to `apply I` but restores **CP** from stack and drops Arg1 slots from it before jumping to the found address.

`i_apply_fun`

Applies the fun R0 to the argument list in R1. Set **CP** to the location right after the instruction.

`i_apply_fun_last P`

Applies the fun R0 to the argument list in R1. Restores **CP** from stack and deallocates Arg0 slots from stack.

`i_apply_fun_only`

Applies the fun R0 to the argument list in R1.

`i_hibernate`

Puts the process into a hibernated state dropping its stack and minimizing its heap. When a message arrives or if the message queue is not empty the process is waken up and starts executing a function identified by R0 (module), R1 (function), and R2 (arguments).

`call_bif0 e`

Calls the BIF from Arg0 without arguments and returns result in R0.

`call_bif1 e`

Calls the BIF from Arg0 with R0 as the argument and returns result in R0.

`call_bif2 e`

Calls the BIF from Arg0 with arguments in R0 and R1 and returns result in R0.

`call_bif3 e`

Calls the BIF from Arg0 with arguments in R0, R1, and R2 and returns result in R0.

`i_get s d`

Reads the process dictionary key Arg0 and places the result to Arg1.

`self rxy`

Put the identifier of the current process to Arg0.

`node rxy`

Put the node identifier of the virtual machine to Arg0.

`i_fast_element rxy j I d`

Gets the Arg2'th element of the tuple Arg0 and places it to Arg3. Arg1 is effectively disregarded.

`i_element rxy j s d`

Gets the Arg2'th element of the tuple Arg0 and places it to Arg3. Arg1 is effectively disregarded.

`bif1 f b s d`

Calls the BIF from Arg1 with the argument Arg2 and places the result to Arg3. Upon error jumps to Arg0 as appropriate for the guard BIF.

`bif1_body b s d`

Calls the BIF from Arg0 with the argument Argr1 and places the result to Arg3. This is the case of a guard BIF in the function body. It fails like an ordinary BIF.

`i_bif2 f b d`

Calls the guard BIF from Arg1 with the two arguments in **tmp_arg1** and **tmp_arg2** and places the result to Arg2. Upon error jumps to Arg0 as appropriate for the guard BIF.

`i_bif2_body b d`

Calls the guard BIF from Arg0 with the two arguments in **tmp_arg1** and **tmp_arg2** and places the result to Arg1. This is the case of a guard BIF in the function body. It fails like an ordinary BIF.

`i_move_call c r f`

Combines `move c r` and `call f`.

`i_move_call_last f P c r`

Combines `move c r` and `call_last f P`.

`i_move_call_only f c r`

Combines `move c r` and `call_only f`.

`move_call xy r f`

Same as `i_move_call xy r f`.

`move_call_last xy r f Q`

Same as `i_move_last f Q xy r`.

`move_call_only x r f`

Same as `i_move_only xy r f`.

`i_call f`

Sets **CP** to the location after the instruction and then jumps to the label Arg0.

`i_call_last f P`

Restores **CP** from the stack, drops Arg1 slots from stack, and jumps to the label Arg0.

`i_call_only f`

Jumps to Arg0.

`i_call_ext e`

Sets **CP** to the point after the instruction and then jumps to the address of the export entry Arg0.

`i_call_ext_last e P`

Restores **CP** from the stack, drops Arg1 slots from the stack, and then jumps to the address of the export entry Arg0.

`i_call_ext_only e`

Jumps to the address of the export entry Arg0.

`i_move_call_ext c r e`

Combines `move c r` with `i_call_ext e`.

`i_move_call_ext_last e P c r`

Combines `move c r` with `i_call_last e`.

`i_move_call_ext_only e c r`

Combines `move c r` with `i_call_only e`.

i_call_fun I

Calls the fun with the arity Arg0. The fun is found in the arity'th registers. Thus the fun call arguments are in registers starting with R0 and the fun itself immediately follows. The result is returned in R0. Sets **CP** to the point after the instruction and jumps to the fun's entry.

i_call_fun_last I P

Same as `i_call_fun I` but restores **CP** from stack and drops Arg1 slots from it before proceeding to the fun's entry.

i_make_fun I t

Creates a fun and puts it to R0. Arg0 contains a reference to the entry of the lamdba table of the module. Arg1 is the number of free variables. The values of free variables are taken from registers starting with R0. Only these registers are concidered live. It means that Arg1 is Live also.

is_function f rxy

Jumps to the label Arg0 if Arg1 is not a fun.

i_bs_start_match2 rxy f I I d

Checks that Arg0 is the matching context with enough slots for saved offsets. The number of slots needed is given by Arg3. If there not enough slots of if Arg0 is a regular binary then recreates the matching context and saves it to Arg4. If something does not work jumps to Arg1. Arg2 is Live.

i_bs_save2 rx I

Saves the offset from the matching context Arg0 to the saved offsets array at the index Arg1.

i_bs_restore2 rx I

Does the opposite of `i_bs_save2 rx I`. Restores the offset of the matching context Arg0 from the saved offsets array at index Arg1.

i_bs_match_string rx f I I

Compares Arg2 bits from the matching context Arg2 with the data referenced by the raw data pointer Arg3. Jumps to the label Arg1 if there is no match.

`i_bs_get_integer_small_imm rx I f I d`

Gets a (small) integer of size (in bits) Arg1 from the matching context Arg0 and puts it to Arg4. Arg3 are flags. Jumps to the label Arg2 on failure.

`i_bs_get_integer_imm rx I I f I d`

Gets an integer of size (in bits) Arg1 from the matching context Arg0 and puts it to Arg5. Arg4 are flags. Arg2 is Live. Jumps to the label Arg3 on failure.

`i_bs_get_integer f I I d`

Gets an integer of size **tmp_arg2** (in units) from the matching context **tmp_arg1** and saves it to Arg3. Arg1 is Live. Arg2 are flags and unit. Jumps to Arg0 on failure.

`i_bs_get_integer_8 rx f d`

Gets an 8-bit integer from the matching context Arg0 and saves it to Arg2. Jumps to Arg1 if there are not enough bits left.

`i_bs_get_integer_16 rx f d`

Gets an 16-bit integer from the matching context Arg0 and saves it to Arg2. Jumps to Arg1 if there are not enough bits left.

`i_bs_get_integer_32 rx f I d`

Gets an 32-bit integer from the matching context Arg0 and saves it to Arg3. Jumps to Arg1 if there are not enough bits left. Arg2 is Live as the integer may not be small.

`i_bs_get_binary_imm2 f rx I I I d`

Gets a binary from the matching context Arg1 of size (in bits) Arg3. Arg2 is Live. Arg4 are flags. The binary is put to Arg5. Jumps yo Arg0 on failure.

`i_bs_get_binary2 f rx I s I d`

Gets a binary from the matching context Arg1 of size (in units) Arg3. Arg2 is Live. Arg4 are flags and unit. Puts result to Arg5. Jumps to Arg0 on failure.

`i_bs_get_binary_all2 f rx I I d`

Gets the rest of the matching context Arg1 as a binary and puts it to Arg4. The binary size should be divisable by unit Arg3. Arg2 is Live. Jumps to Arg0 upon failure.

`i_bs_get_binary_all_reuse rx f I`

Replaces the matching context Arg0 with the rest of its reminder represented as a binary. Arg2 is the unit. The size of the binary should be divisible by unit. Jumps to the label Arg1 on failure.

`i_bs_get_float2 f rx I s I d`

Gets the float value form the matching context Arg1. Arg3 is the size in units. Arg4 are unit and flags. Puts the value to Arg5. Arg2 is Live. Jumps to Arg0 on failure.

`i_bs_skip_bits2_imm2 f rx I`

Move the offset of the matching context Arg1 by Arg2 bits. Jumps to Arg0 on failure.

`i_bs_skip_bits2 f rx rxy I`

Skips2 Arg2 units of the matching context Arg1. Arg3 is the unit size. Jumps to Arg0 on failure.

`i_bs_skip_bits_all2 f rx I`

Skips all remaining bits of the matching context Arg1. The number of bits skipped should be divisible by unit Arg2. Jumps to the label Arg0 on failure.

`bs_test_zero_tail2 f rx`

Jumps to the label in Arg0 if the matching context Arg1 still have unmatched bits.

`bs_test_tail_imm2 f rx I`

Checks that the matching context Arg1 has exactly Arg2 unmatched bits. Jumps to the label Arg0 if it is not so.

`bs_test_unit f rx I`

Checks that the size of the remainder of the matching context Arg1 is divisible by unit Arg2. Jumps to the label Arg0 if it is not so.

`bs_test_unit8 f rx`

Checks that the size of the remainder of the matching context Arg1 is divisible by 8. Jumps to the label Arg1 if it is not.

`bs_context_to_binary rxy`

Converts the matching context to a (sub)binary using almost the same code as `i_bs_get_binary_all_reuse rx f I`.

`i_bs_get_utf8 rx f d`

Extracts a UTF-8 encoded character from the matching context Arg0 and places it to Arg2. Jumps to Arg1 upon failure.

`i_bs_get_utf16 rx f I d`

Extracts a UTF-16 encoded character from the matching context Arg0 using flags Arg2 and places it to Arg3. Jumps to Arg1 upon failure.

`i_bs_validate_unicode_retract j`

Verifies that a matched out integer **tmp_arg1** falls into the valid range for Unicode characters. If not the matching context **tmp_arg2** is rectracted by 32 bits. Arg0 looks like a failure label but it is never used.

`i_bs_init_fail rxy j I d`

The same as `i_bs_init I I d` but may fail if the size in Arg0 is invalid. Arg1 is not used.

`i_bs_init_fail_heap I j I d`

Follows the `i_fetch d d` that loads binary data size into **tmp_arg1**. Allocates space for a shared binary data of **tmp_arg1** size. Creates a ProcBin referencing the data. Makes sure that heap has enough space for a subbinary and Arg0 more words. Arg2 is Live. Puts the ProcBin to Arg3.

`i_bs_init I I d`

Allocates space for a shared binary data of Arg0 size. Creates a ProcBin referencing the data. Makes sure that the heap has enough space for a subbinary. Arg1 is Live. Puts the ProcBin into Arg2.

`i_bs_init_heap I I I d`

Allocates space for a shared binary data of Arg0 size. Create a ProcBin referencing the data. Makes sure that the heap has enough space for a subbinary and Arg1 more words. Arg2 is Live. Puts the ProcBin into Arg3.

`i_bs_init_heap_bin I I d`

Allocates a heap binary of size Arg0 and makes sure that the heap has enough space for a subbinary. Arg1 is Live. The heap binary is put to Arg2.

`i_bs_init_heap_bin_heap I I I d`

Allocates a heap binary of size Arg0 and makes sure that the heap has enough space for a subbinary and Arg1 more words. Arg2 is Live. The heap binary is put to Arg3.

`i_bs_init_bits_fail rxy j I d`

Same as `i_bs_init_bits_heap I I I d` but the number of extra words allocated on the heap is zero and may fail if the number of bits in Arg0 is not valid. Arg1 is not used.

`i_bs_init_bits_fail_heap I j I d`

Same as `i_bs_init_bits_heap I I I d` but the number of bits is fetched to **tmp_arg1** by previous instruction and may fail if the number of bits in **tmp_arg1** is not valid. Arg1 is not used.

`i_bs_init_bits I I d`

Same as `i_bs_init_bits_heap I I I d` but the number of extra words allocated on heap is zero.

`i_bs_init_bits_heap I I I d`

Allocates a heap binary of size (in bits) Arg0. If the size is not divisible by 8 than allocates a subbinary referencing the exact number of bits of the heap binary. Makes sure that heap has Arg1 extra words available. Arg2 is Live. Puts the result (the heap binary or the subbinary) to Arg3.

`i_bs_add j I d`

Calculates the total of the number of bits in **tmp_arg1** and the number of units in **tmp_arg2**. Arg1 is the unit. Stores the result to Arg2.

`i_bs_init_writable`

Allocates a shared data space of size R0. Creates a ProcBin on the heap referencing the data. Create a subbinary referencing the ProcBin. Save the subbinary to R0.

`i_bs_append j I I I d`

Appends **tmp_arg1** bits to a binary **tmp_arg2**. If the binary is a writable subbinary referencing a ProcBin with enough empty space then a new writable subbinary is created and the old one is made non-writable. In other cases, creates a new shared data, a new ProcBin, and a new subbinary. For all heap allocation, a space for more Arg1 words are requested. Arg2 is Live. Arg3 is unit. Saves the resultant subbinary to Arg4.

`i_bs_private_append j I d`

Same as `i_bs_append j I I I d` but assumes that the binary writable. Reallocates the shared data if there is not enough space. This should be safe. Changes the subbinary **tmp_arg2** in place. No heap allocations. Arg1 is the unit. Save the result to Arg2.

`i_new_bs_put_integer j s I s`

Assumes that previous instructions created a binary creation context. Packs an integer value Arg3 into Arg1 units of the context. Arg2 are the unit and flags. The failure label Arg0 is unused.

`i_new_bs_put_integer_imm j I I s`

Assumes that previous instructions created a binary creation context. Packs an integer value Arg3 into Arg1 bits of the context. Arg2 are flags. The failure label Arg0 is unused.

`i_bs_utf8_size s d`

```
/*
  * Calculate the number of bytes needed to encode the source
 * operarand to UTF-8. If the source operand is invalid (e.g. wrong
 * type or range) we return a nonsense integer result (2 or 4). We
 * can get away with that because we KNOW that bs_put_utf16 will do
  * full error checking.
  */
```

`i_bs_utf16_size s d`

```
/*
  * Calculate the number of bytes needed to encode the source
 * operarand to UTF-16. If the source operand is invalid (e.g. wrong
 * type or range) we return a nonsense integer result (2 or 4). We
 * can get away with that because we KNOW that bs_put_utf16 will do
  * full error checking
  */
```

`i_bs_put_utf8 j s`

Assumes that previous instructions created a binary creation context. Packs a UTF-8 character Arg1 into the context. The failure label Arg0 is unused.

`i_bs_put_utf16 j I s`

Assumes that previous instructions created a binary creation context. Packs a UTF-16 character Arg2 into the context. Arg1 are flags. The failure label Arg0 is unused.

`i_bs_validate_unicode j s`

Validates a single unicode character Arg1. Raises `badarg` on error. Arg0 is not used. NB the existence of the instruction may be dependent on the implementation detail of binary construction see comment in beam_emu.c, line 3963.

`i_new_bs_put_float j s I s`

Assumes that previous instructions created a binary creation context. Packs a float of Arg1 units size into the context. Arg2 are the unit and flags. The failure label Arg0 is not used.

`i_new_bs_put_float_imm j I I s`

Assumes that previous instructions created a binary creation context. Packs a float Arg3 of Arg1 bits size into the context. Arg2 are flags. The failure label Arg0 is not used.

`i_new_bs_put_binary j s I s`

Assumes that previous instructions created a binary creation context. Packs a binary Arg3 of Arg1 units size into the context. Arg2 are flags and the unit. The failure label Arg0 is not used.

`i_new_bs_put_binary_imm j I s`

Assumes that previous instructions created a binary creation context. Packs a binary Arg2 of Arg1 bits size into the context. The failure label Arg0 is not used.

`i_new_bs_put_binary_all j s I`

Assumes that previous instructions created a binary creation context. Packs a whole binary Arg1 into the context. Arg2 is the unit. The failure label Arg0 is not used.

`bs_put_string I I`

Assumes that previous instructions created a binary creation context. Packs a string of characters Arg0 bytes long referenced by the pointer Arg1 into the context.

`fmove qdl ld`

Copies Arg0 to Arg1.

`fconv d l`

Converts Arg0 to float and places the result to Arg1.

`i_fadd l l l`

$Arg2 = Arg0 + Arg1$.

`i_fsub l l l`

$Arg2 = Arg0 - Arg1$.

`i_fmul l l l`

$Arg2 = Arg0 \times Arg1$.

`i_fdiv l l l`

$Arg2 = Arg0/Arg1$.

`i_fnegate l l l`

$Arg1 = -Arg0$.

`i_fcheckerror`

Checks if the FR0 contains Nan or -Inf. Raises `badarith` if this is the case.

`fclearerror`

Maybe clears FP error flag.

`i_increment rxy I I d`

$Arg3 = Arg0 + Arg1$. Arg2 is Live.

`i_plus j I d`

$Arg2 =$ **tmp_arg1** $+$ **tmp_arg2**. Arg1 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_minus j I d`

$Arg2 =$ **tmp_arg1** $-$ **tmp_arg2**. Arg1 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_times j I d`

$Arg2 =$ **tmp_arg1** $\times$ **tmp_arg2**. Mixed. Arg1 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_m_div j I d`

$Arg2 =$ **tmp_arg1**/**tmp_arg2**. Mixed. Arg1 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_int_div j I d`

$Arg2 =$ **tmp_arg1**/**tmp_arg2**. Division is integer. Arg1 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_rem j I d`

$Arg2 =$ **tmp_arg1**%**tmp_arg2**. Division is integer. Arg1 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_bsl j I d`

$Arg2 =$ **tmp_arg1** $<< / >>$ **tmp_arg2**. The direction of the shift depends on the sign of **tmp_arg2**. Arg1 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_bsr j I d`

$Arg2 =$ **tmp_arg1** $>> / <<$ **tmp_arg2**. The direction of the shift depends on the sign of **tmp_arg2**. Arg1 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_band j I d`

$Arg2 =$ **tmp_arg1**&**tmp_arg2**. Arg1 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_bor j I d`

$Arg2 =$ **tmp_arg1**|**tmp_arg2**. Arg1 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_bxor j I d`

$Arg2 =$ **tmp_arg1** $\oplus$ **tmp_arg2**. Arg1 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_int_bnot j s I d`

$Arg3 = \neg Arg1$. Arg2 is Live. Jumps to Arg0 on error if Arg0 is not zero.

`i_gc_bif1 j I s I d`

Executes a guard BIF at the address Arg1 with a single parameter Arg2. Arg3 is Live. Stores the result in Arg4. Jumps to Arg0 on error if it is not zero. Otherwise, raises an exception.

`i_gc_bif2 j I I d`

Executes a guard BIF at the address Arg1 with two parameters – **tmp_arg1** and **tmp_arg2**. Arg2 is Live. Stores the result in Arg3. Jumps to Arg0 on error if it is not zero. Otherwise, raises an exception.

`i_gc_bif3 j I s I d`

Executes a guard BIF at the address Arg1 with three parameters – Arg2, **tmp_arg1**, and **tmp_arg2**. Arg3 is Live. Stores the result in Arg4. Jumps to Arg0 on error if it is not zero. Otherwise, raises an exception.

Message passing

```
/*
 * Skeleton for receive statement:
 *
 *            recv_mark L1                     Optional
 *            call make_ref/monitor            Optional
 *            ...
 *            recv_set L1                      Optional
 *      L1:         <-------------------+
 *                  <-----------+       |
 *                              |       |
 *            loop_rec L2 ------+---+   |
 *            ...               |   |   |
 *            remove_message    |   |   |
 *            jump L3           |   |   |
 *            ...               |   |   |
 *            loop_rec_end L1 --+   |   |
 *      L2:         <---------------+   |
 *            wait L1  -----------------+      or wait_timeout
 *            timeout
 *
 *      L3:    Code after receive...
 *
 *
 */
```

Meta instructions

All meta instructions are omitted by the loader. Their presence is recognized by the interpreter as an error.

`int_code_end`

Maybe marks the end of the code.

`label L`

The label L is set to the current code location.

`line I`

Maybe augments debugging output when backtracking.

Superficial instructions

The use of the following instructions supported by BEAM is avoided in Ling:

- `too_old_compiler`
- `i_trace_breakpoint`
- `i_debug_breakpoint`
- `i_count_breakpoint`
- `i_time_breakpoint`
- `i_return_time_trace`
- `i_return_to_trace`
- `i_yield`
- `normal_exit`
- `continue_exit`
- `apply_bif`
- `call_nif`
- `call_error_handler`
- `call_traced_function`
- `return_trace`
- `unsupported_guard_bif`
- `on_load`