Foreword

This manual describes various aspects related to the existing implementations of the XE language [3]. It also describes how to use the XE cross-compiler. XE implementations exist at the present for the Symbolics Lisp machine (the programming environment), and for the Intel 8086 processor (the cross-compiler).

Sections 1 and 2, and to a lesser extent 3 and 4, are necessary reading for an XE "cross-programmer". Section 1 is a user's manual of the cross-compiler. Section 2 describes such implementation dependent features of XE as the precision of integer arithmetics. Section 3 describes how to interface foreign routines to XE. Section 4 describes how the XE programmer can control the run time system, meaning essentially the garbage collection.

Sections 5-9 and Appendix A describe XE implementation details not visible to an XE programmer. They are mainly useful for a person maintaining or retargeting the XE compiler or otherwise interested in its implementation.

This manual is, to a large extent, based on the diploma thesis [2] where more information can be found (in Finnish) on the details of code generation and the Intel run time environment.

Acknowledgements

The programming language XE has been designed and implemented at the Helsinki University of Technology within Exbed project (Expert System Framework for Embedded Applications) funded by the Technology Development Centre, Nokia Corporation and KONE corporation.

Contents

1 Generating Intel 8086 assembler code from XE programs

XE programs are checked for syntax and semantics by the XE compiler running on a Lisp machine. If the syntax and semantics are correct, the compiler creates so called descriptor records of the program modules. The descriptors can optionally be compiled to be executed in the Lisp Machine. This is done with the *s-G* ZMACS command, or it is performed automatically by the *exec*-function. When new programs are fed to the compiler or old ones are modified, the corresponding descriptor records are changed. To generate code for an Intel-8086 machine, one must first produce EM code from the descriptor records of the program modules. EM is an intermediate language used by the Amsterdam Compiler Kit [6].

1.1 EM CODE GENERATOR

Code generation for the Intel environment is performed as a batch process; a complete program is compiled once and no incremental changes are possible. The compilation uses the descriptor records created by the XE compiler but does not modify them.

When a new version of the program for the Intel environment is required a batch compilation must be initiated. The batch compilation uses the XE compiler process, so no other XE compilation is possible during the batch compilation.

The batch compilation is initiated by the following Lisp function:

```
(icg$comp &key statement
               filename
               small-code-model-p
               stack-given-p
               stack-segment-p
               verbose
               datasize
               optimizep)
```

- *Statement* is the main program. In the Intel environment, the program starts executing from *statement* and terminates when execution of *statement* ends. This keyword argument should always be specified.
- *Filename* is the name of the file where the generated EM code will be stored. The name should include directory and file type components. This keyword argument should always be specified.
- *Small-code-model-p* is T if the program code is to be made to fit in a single memory segment (64K), and NIL otherwise. The default value is T.

- *Stack-given-p* indicates whether or not the stack needs to be allocated and initialized by the XE run-time system. Value T indicates that the stack need not be allocated by the XE run-time system, and NIL forces XE to do the allocation. The stack need not be allocated when the operating system does that already. The default value is T.
- *Stack-segment-p* indicates whether a separate stack segment is used or whether the data segment is also used for the stack. Value T indicates separate stack segment. The default value is T.
- *Datasize* is the size of the data area in bytes (decimal number). The size must be less than 64 kbytes. The default size is 20 kbytes.
- *Verbose* indicates whether or not the batch compiler produces progress output or not. The default value is NIL (no output).
- *Optimizep* is T if the program should be optimized. This is the default value. Normal users should not have any reason to want unoptimized code.

Below is an example of a call to the EM code generator function:

```
(icg$comp :statement "begin my_main(100,'a')
                            flush(:stream$primary_output())
                      end"
          :filename ">tmp>my.e"
          :small-code-model-p T
          :datasize 50000
          :stack-given-p T
          :stack-segment-p T
          :verbose T)
```

The main program consists of a call to *my_main* with the specified arguments and of flushing the standard output stream after the call to *my_main* completes. Note that the main program does not need to be a simple call to some procedure, but rather all statements of the XE language are accepted (here the statement was the *begin-end* statement). The generated EM code will reside in the file `>tmp>my.e`. The program will be allocated 50 kbytes of memory. Note that the ordering of the keyword arguments in the call need not be the same as described above.

1.2 Intel 8086 Assembly generator

The batch EM code generator produces an ascii file (extension .e) containing EM intermediate code. This file must be copied to an appropriate directory in the VAX/VMS environment where the ACK-based tools, responsible for generating Intel assembly, reside. Intel assembly generation goes through several phases:

1. Encoding of the EM from ASCII format to a more efficient format (.k), using the ACK program *em_encode*.

2. 1st Optimization phase of ACK (.t), using the ACK *opt* program.

3. 2nd Optimization phase of ACK (.m), using the ACK *opt2* program.

4. Code generation from EM to assembler (.xs), using the ACK Intel 8086 codegenerator *cg*.

5. Combining with the assembler libraries (.s).

6. Preprocessing (.c). The assembler libraries contain C-preprocessor statements. The libraries are preprocessed by the system C compiler.

7. Assembly (.o), using the ACK 8086 assembler *as*.

8. Conversion to Intel assembler format (.a86), using the *ackasm* program.

The Intel assembly generation is carried out by running the following batch command in the VAX/VMS environment:

@ack86 file

where *file* is the name of the EM ascii file without a file type extension. The resulting assembler file (.a86) contains the whole XE proram. The main program is in the procedure MAIN. This label is declared public so that the XE program can be started by calling it. Depending on the value of the *small-code-model-p*, this procedure should be called NEAR (value T) or FAR (value NIL). FAR means that the segment selector should be used in the call and NEAR means that it should not.

The foreign procedures referenced are declared as externals (EXTRN). These procedures are called from the XE program. Calls to them are always FAR and the procedures should expect to be called FAR.

The assembler file should be compiled with the Intel 8086 assembler. The produced object file contains several segments. When the value of *small-code-model-p* is T, these segments are grouped into two groups: the code group *program_cgroup* and the data group *program_dgroup*. Both of these groups should be loaded to memory before the execution of the program begins. Please note that the XE process modifies its data memory, so that restarting the process (i.e. calling MAIN again) requires the original data group to be reloaded.

2 Implementation Dependent Features of XE

This chapter describes the implementation dependent features of the XE language. Currently, there are two implementations of the XE system: the Symbolics implementation and the Intel implementation.

The implementation dependent features of the XE language are: 1) The operation of the integer arithmetic functions, including limitations of the value ranges in the integer data types, 2) The value ranges and mapping to integer values of the data type *char*, 3) The usage and semantics of the argument of the *failure* exception, 4) Interpretation of the file names, 5) Representation of the XE library data types in the *dump-undump* format and 6) Allocation errors.

Below the characteristics of these features is described both for the Symbolics and for the Intel 8086 implementation.

2.1 Symbolics Implementation

2.1.1 Characteristics of Integer Arithmetic

The integer arithmetic of the Symbolics implementation is based on two's complement modular arithmetic. This means that values exceeding the value ranges of the data types wrap around from the other end of the value range. E.g., if the value range is from -10 to 10, the expression 10+1 will yield the value -10.

Data type *byte*

The value range of the data type *byte* is from 0 to 255.

Data type *word*

The value range of the data type *word* is from 0 to 65,535.

Data type *int*

The value range of the data type *int* is from -32,768 to 32,767.

Data type *longint*

The value range of the data type *longint* is from -2,147,483,648 to 2,147,438,647.

Data type *char*

The data type *char* is based on the Symbolics standard character set, including the characters whose character code lies in the 0 to 255 range. The font and modifier bits of characters are not supported by the implementation. The operations *char$to_byte* and *char$to_int* are based on the character codes of the Symbolics standard character set and yield values in the range of 0 to 255.

2.1.2 *Failure* Exceptions

The *failure* exception takes a *string* argument. When an unhandled exception is transformed to a *failure* exception, the name of the unhandled exception as a string is given to the *failure* exception as an argument. For other implicit *failure* signals the value of the string argument is undefined, though guaranteed to be a legal XE string object.

2.1.3 Interpretation of File Names

The file names given to the routine *stream$open* are in the standard Symbolics pathname format, including the host, directory, name, type and version components.

2.1.4 *Dump-Undump* format

The data types *bool*, *char*, *byte*, *word*, *int* and *longint* are given directly to the foreign function when they appear as an argument or the return value of the foreign function (Section 4). However, they can also be *dump*ed or *undump*ed, e.g., when they appear as the element type of an *array*. All other data types are *dump*ed when they are given as an argument to a foreign function.

The *dump*ed representation of an object of type *bool* is a one byte integer with value 0 or 1. The representation of the type *char* is a one byte integer holding the character code of the character. The representation of the type *byte* is a one byte integer. The representation of the type *word* and *int* is a two byte integer in two's complement form. The representation of the type *longint* is a four byte integer in two's complement form. Note that on the Symbolics each of the above objects will physically require one 36 bit word.

The representation of the types *struct* and *record* consists of an unspecified ordering of the representations of their fields. If there is a need to know the exact ordering of the fields, the user must write his own corresponding selector generator.

The representation of the types *oneof* and *variant* is a two byte unique integer describing the current tag of the object. To get the value of type *tag* corresponding to this integer, the operation *_fieldname$index* must be applied, with the appropriate field lists. The two byte integer is followed by the representation of the current value in the object.

The representation of the types *sequence* and *array* is a two byte integer indicating the number of elements, followed by the representations of the elements in the order of increasing indices. The representation of the type *string* is a two byte integer indicating the length of the string, followed by the representations of the characters of the string in the same order as they appear in the string.

The representation of the type *null* is an unspecified one byte integer.

The representation of the types *stream* and *binary* are unspecified because they should not be of concern to an XE programmer.

Note that the representation of the *char* and *byte* values 128 to 255 and *word* values 32768 to 65535 are represented as negative integers, since the two's complement form is used. Note that when interfacing with Lisp routines the Lisp function *logand* [5] should be applied to correct the effect of the two's complement representation. It should be given the integer and a mask as arguments. The mask value for *char* and *byte* is 255 and for *word* it is 65535. Then the return value of *logand* will be the expected non-negative integer.

All *dump*ed objects are given to a foreign function as Lisp sequences, with the representation of the given data type in the elements of the sequence. Irrespective of the size of the elements (*byte*, *word*), the sequence holds one element of the representation in one of its element. Thus the sizes of integer values described earlier only tell the number of bits used in the representation, not the actual memory usage.

2.1.5 Allocation Errors

The Symbolics Lisp runtime environment allocates also the memory requested by the XE programs. If the amount of available memory becomes small, the Symbolics will inform the user of this in the form of notifications. If the memory is exhausted, the machine will crash.

2.2 Intel Implementation

2.2.1 Characteristics of Integer Arithmetic

The integer arithmetic of the Intel implementation is based on two's complement modular arithmetic. This means that values exceeding the value ranges of the data types wrap from the other end of the value range. E.g. if the value range is from -10 to 10, the expression 10+1 will yield the value -10.

Data type ***byte***

The value range of the data type *byte* is from 0 to 255.

Data type *word*

The value range of the data type *word* is from 0 to 65,535.

Data type *int*

The value range of the data type *int* is from -32,768 to 32,767.

Data type *longint*

The value range of the data type *longint* is from -2,147,483,648 to 2,147,438,647.

Data type *char*

The data type *char* is based on the IBM PC standard character set, which is the standard ASCII character set for the character codes 0..127, and some additional characters as the character codes 128 to 255. The operations *char$to_byte* and *char$to_int* are based on the character codes of the IBM PC standard character set and yield values in the range of 0 to 255.

2.2.2 *Failure* Exceptions

The *failure* exception takes a *string* argument. When an unhandled exception is transformed to a *failure* exception, the name of the unhandled exception as a string is given to the *failure* exception as an argument. For other implicit *failure* signals the value of the string argument is undefined, though guaranteed to be a legal XE string object.

2.2.3 Interpretation of File Names

The file names given to the routine *stream$open* are strings in the standard IBM PC file name format, including the disk, directory, name and extension components.

2.2.4 *Dump-Undump* format

The data types *bool*, *char*, *byte*, *word*, *int* and *longint* are given directly to the foreign function when they appear as the argument or the return value of the foreign function. However, they can also be *dump*ed or *undump*ed, e.g., when they appear as the element type of *array*. All other data types are *dump*ed when they are given as an argument to a foreign function.

The *dump*ed representation of the type *bool* is a one byte integer 0 or 1. The representation of the type *char* is a one byte integer holding the character code of the character. The representation of the type *byte* is a one byte integer. The representation of the type *word* and *int* is a two byte integer in two's complement form. The representation of the type *longint* is a four byte integer in two's complement form.

The representation of the types *struct* and *record* is the representations of their fields in some, implementation dependent, format. If there is a need to know the exact ordering of the fields, the user must write his own corresponding selector generator.

The representation of the types *oneof* and *variant* is a two byte unique integer describing the current tag of the object. To get a value of type *tag* from this integer, the operation *_fieldname$index* must be applied, with the appropriate field lists. The two byte integer is followed by the representation of the current value held by the object.

The representation of the types *sequence* and *array* is a two byte integer indicating the number of elements, followed by the representations of the elements in the order of their indices (without padding between elements). The representation of the type *string* is a two byte integer indicating the length of the string, followed by the representations of the characters of the string in the same order as they appear in the string.

The representation of the type *null* is an unspecified one byte integer. The representation of *stream* and *binary* is implementation dependent, and should not concern the XE programmer.

2.2.5 Allocation Errors

The XE runtime system stops execution of the program when the memory becomes full. Garbage collection is initiated and the program continues execution if enough memory is freed. If the garbage collection could not free enough memory, the program prints the message "Error: Out of memory." and stops.

3 Interfacing Foreign Routines with XE

Foreign routines can be called from XE by using the *foreign proc* mechanism. The foreign routine interface is implemented for the PL/M (Intel) and Lisp (Symbolics) languages. The conversions performed when passing arguments and handling return values are described in sections 2.1.4 and 2.2.4.

The calling convention of the Lisp interface uses the standard Lisp function call mechanism. When the argument is a self-contained object (types *bool*, *char*, *byte*, *word*, *int* or *longint*), it is passed directly as an argument to the Lisp function. Otherwise, a Lisp sequence is passed to the Lisp function. This sequence holds the *dump*ed representation of the object in its elements.

The calling convention of the PL/M interface uses the conventions of the PL/M 86 compiler. Segmented addresses (i.e., segment, offset pairs) are used for data pointers and routine addresses. Self contained arguments are passed directly on the stack. All arguments of type *bool* are converted to the PL/M standard boolean format: byte 0 for false and byte 255 for true. Each non self contained argument is dumped into a memory area, and a segmented pointer to this area is passed on the stack in the place of that argument. The segment part of the pointer is pushed first, followed by the offset part. Return values of types *char*, *byte* and *bool* are in the AL register of the 8086. The representation of *bool* type is a zero or nonzero byte. The values of types *int* and *word* are passed in the AX register. The values of type *longint* are passed in the AX (high) and BX (low) registers. Pointer values are returned in the AX (segment) and BX (offset) registers. The pointer values point to a memory area. This memory area is then read using *undump*.

The values of the SS, SP and DS registers are expected not to be modified by the routine called (or at least they should be restored after modification).

An example:

```
fofn1 = foreign plm proc "FOFN_1" (a,b:int,c:string) returns(int)
...
x: int := fofn1(3,4,"Foo")
```

The PL/M function *FOFN_1* takes three arguments: two integers and a pointer. The pointer points to a memory area allocated by XE. The memory area contains a word describing the number of characters in the string and the characters of the string followed by it. The return value is an integer.

4 Controlling the Run Time System

This section describes the programmer interfaces of the XE run time system and their implementation in the two current implementations of the XE system (Symbolics and Intel). The run time system of XE deals with 1) Exceptions and 2) Memory allocation.

4.1 Symbolics Run Time System

There are no user interfaces to the Symbolics run time system.

4.2 Intel Run Time System

The Intel implementation of the XE run time system offers some control to the user over the memory allocation procedures.

s_mm$gc()

This function forces a premature garbage collection. This garbage collection always succeeds. This function can be used to force a garbage collection when it is appropriate so that it does not disturb system performance or operation.

s_mm$set_gcmessages(new:bool)

Sets (argument true) or resets (argument false) the garbage collection messages. Garbage collection messages are printed when a garbage collection occurs (either through exhaustion of the memory or through forcing). The messages are printed to the stream *stream$primary_output*.

5 Primitive Data Types

The XE library is implemented with primitive data types and operations supported by the compiler. All operations of the primitive data types can be represented as CAST (Checker Abstract Syntax Tree [4]) nodes. However, because the primitive operations are normal routines of the XE language they can also appear in invocation nodes of CAST. The primitive operations can also be passed as functional arguments.

The primitive data types and routines are described in appendix A.

6 Notes on High-Level Optimization

The high-level optimization of XE is performed as a set of CAST (Checker Abstract Syntax Tree [4]) to CAST transformations. The following optimizations are performed by the optimizer when either the EM or Lisp code generators are used:

- The invocations of primitive operations are replaced by the appropriate CAST nodes. After the checking phase of the compiler, the CAST structure contains invocations of primitive routines. These invocations are changed to the direct operation nodes of CAST. This is implemented in the function *check%primitive-invocation*.
- Some library functions are directly replaced by the appropriate CAST nodes (e.g. *int$add* is replaced by the direct *add* operation in the CAST structure. This is implemented by the function *inl$inline*.
- Function invocations are inlined in expressions and statements. Also tail-recursive calls are eliminated. This is implemented by the functions *inl%inline-statement* and *inl%inline-invocation*. For instance, the program

  ```
  foo = proc(a,b:int) returns(int)
          return(a+1)
  end
  bar = proc(x,y:int) returns(int)
          return(foo(x,y))
  end
  ```

  would expand into

  ```
  bar = proc(x,y:int) returns(int)
          return(x+1)
  end
  ```

- Iterators implemented in the XE built-in library are inlined, except for the general case of *int$from_to_by*. The inlining is implemented in the routine *inl%inline-for*. An example of this inlining is given by the following two code fragments:

  ```
  for x: int in [a..b;2] do
          body
  end

  =>

  begin
          x: int := a
          temp: int := b
          while x < temp do
                  body
                  x := x + 2        % Note that this is constant
                                    % and can thus be evaluated here
          end
  ```

- Tail-recursive iterations are eliminated by the routine *inl%inline-for*. An example of a tail recursive iterator:

```
list = datatype[T:type] is elements,...
         rep = oneof{nil:null,
                     cons:record{head:T,
                                 tail:list[T]}}
         ...
         elements = iter(l:cvt) yields(T)
                      if ~is_nil(l) then
                          yield(l.cons.head)
                          for e: T in elements(:l.cons.tail) do
                              yield(e)
                          end
                      end
         end
         ...
end
```

  The iterator *elements* will be changed to:

```
         ...
         elements = iter(l:cvt) yields(T)
                      while true do
                          if ~is_nil(l) then
                              yield(l.cons.head)
                              l := l.cons.tail
                              continue
                          end
                          break
                      end
         end
         ...
```

- Iterators implemented directly by other iterators are eliminated, again by routine *inl%inline-for*. An example of an iterator implemented directly by another one:

```
myelements = iter(s:string) yields(char)
        for c: char in string$elements(s) do
                yield(c)
        end
end
```

- Peephole optimizations are performed on several CAST structures, e.g., empty *begin-end* nodes are eliminated. This is implemented by the function *inl%fold*. An example of a CAST peephole optimization:

```
if cond then
        z := 7
else
        begin
        end
```

```
end

=>

if cond then
        z := 7
end
```

- Unused *except* statements are eliminated by the routine *inl%fold-primitive.*
- *Exit* targets are propagated, i.e., if the target of an *exit* is another *exit* statement, there will be only one *exit* jump. This is implemented by the routine *inl%fold-primitive.* An example:

```
exit E1
except when E1: exit E2 end
except when E2: z := 8 end

=>

exit E2
except when E2: z := 8 end
```

7 Notes on EM-code Generation

The generation of EM code from the CAST representation is implemented as a syntax-directed process.

The EM-code generation includes the following optimizations (these optimizations are not performed in the Lisp code generator):

- Peephole optimization of the EM-code, including jump propagation and tail recursion elimination. This is implemented in the routine *icg%peephole* (which contains the top-level of the optimizer) and *icg%peephole-aux* (which contains the optimization patterns).
- Unification of similar routines to a single routine. The parametrization of XE often generates a large number of similar or equivalent routines. These are removed by this optimization, thus promoting code sharing. Unification is implemented by the function *icg%divide-to-ecs*. The optimization uses an algorithm similar to that used in reducing the number of states in automata [1].
- Removal of uncalled routines. The inline coding and other optimizations render some routines useless, since their calls have been replaced by their definition or removed alltogether. This is implemented by the function *icg%scrap-unused-routines*.

8 Notes on Intel-code Generation

The EM-language was modified to better fit the requirements of XE. Part of the modifications dealt with problems of the ACK 8086 implementation: not all operations were implemented for byte sized data (instructions *adi*, *sbi*, *adu*, *sbu*, *ngi*, *sli*, *sri*, *rol* and *ror*), and there was a too restrictive limit on the number of return values in functions (instructions *ret* and *lfr*). Some EM structures typical of code generated from XE were optimized by specialized code rules. The implementation of XE for the DX 200 digital telephone exchange [7] required addressing data in different segments. This addressing was needed in time-critical parts, e.g., during garbage collection since the DX operating system requires the stack to reside in its own segment. Thus the segmented memory accessor instructions *lbr* and *lbs* had to be implemented.

ACK was also modified to make it possible to use several code segments. The modifications included changing the procedure call and return instructions in the code generator and implementing a feature in the ACK assembler to faciliate the handling of segment, offset pairs, and assembler instructions to call, jump and return from segmented addresses.

The foreign language interface to PL/M was implemented by the new instructions *fcl* for calling PL/M functions and *flr* for taking return values of PL/M functions. A PL/M call is different from an ordinary ACK call due to different code segment and return value handling.

The tail-recursion optimization required an instruction to jump to the start of a function without creating a new stack frame. This was implemented by the instruction *jfn*. It operates by moving the stack pointer and jumping over the instructions generating a stack frame.

Here are the definitions of the new instructions:

```
lbr w                   Pops offset and segment, and reads
                        w bytes from segment:offset.
lbs w                   Pops data, offset and segment and
                        writes w bytes to segment:offset.
jfn lab                 Starts executing function lab
                        without creating a new stack frame.
fcl lab                 Calls a function by PL/M 86 call
                        conventions.
fcl n                   Organizes return values from PL/M 86
                        conventions to the ACK conventions.
                        Argument n=0 for two byte and n=1
                        for four byte return values.
```

ACK was modified in the EM definition file *etc/em_table* and in the Intel 8086 code generator file *i86/cg/table*. The ACK assembler was modified from the files *proto/as/comm0.h*, *i86/as/mach3.c*, *i86/as/mach4.c* and *i86/as/mach5.c*. The Intel code generator was split to two versions: *i86* for long function calls and *i86k2* for short function calls. The sources are the same and they differ only in that the C preprocessor definition *XELARGE* is used for *i86* and *XESMALL* is used for *i86k2*. The modifications for XE in general are compiled conditionally with the preprocessor definition *XE*.

9 Notes on the Intel Run Time System

9.1 Memory Management Implementation

Memory management is written in XE. It is implemented by the data type *s_mm*. It includes the following major routines:

init()

Initializes the *own* variables of memory management. This routine is called as the first routine in the main program.

alloc[T: type](n:word) returns(T)

Allocates a memory block of n bytes.

gc()

Performs the garbage collection.

sweepstack(fn: sweepfn)
sweepowns(fn: sweepfn)

These routines sweep the stack and *own* variables.

updateframe(fp:word,retaddr:word,fn:sweepfn)
updateframe_main(fp:word,retaddr:word,fn:sweepfn)

Sweeps through a frame and its iterator frames.

recursedo(typeptr,seg,off:word,fn:sweepfn)

Recursively traverses an object. Contains a case on the type of the object.

mark(obj:word)

Marks an object.

replace_newaddress(obj,atseg,atoff:word)

Replaces the new address of an object at a place where the object was referenced.

assign_newaddresses()

Assigns the new addresses of the referenced objects.

move_objects()

Moves the objects to their new addresses.

zero_mem()

Zeroes the memory after a garbage collection. The allocated blocks are thus initialized to zero.

9.2 Exception Handling Implementation

Exception handling is implemented differently, depending on whether we have one or several code segments. The routines are, however, similar and their description is shared below. Exception handling is implemented by the following routines (all written in XE):

s_signal
l_signal

Takes the *fp* and the signal number as an argument and returns the new settings of *cs:ip*, *sp* and *fp*.

s_find_exception_handler
l_find_exception_handler

Searches the descriptor of a routine for an exception handler. The exception handler may not be found, in which case the exception is indicated to be searched from the enclosing function call (or from the function where the iterator body belongs).

find_sp_level

Returns the appropriate stack level for the current *fp* setting. The stack level is calculated based on the maximum of the frame stack usage between the function and iterator frames in the stack.

s_get_routine_desc_addr
l_get_routine_desc_addr

Locates the address of the routine descriptor of the given routine.

9.3 Other routines

In this section the assembler routines of the Intel implementation are described.

_F__quit_iterator

Exits from an iterator. The iterator chain is updated, as well as the stack, if possible.

_F__break

Performs a *break* statement when an iterator must be broken in order to get to the target.

_F__signal

This routine is called when an XE program signals an exception. It calls the function *s_signal* and jumps according to the given return values.

_F__yield

This routine is called when an iterator performs a *yield* statement. The iterator continuation variables are set appropriately.

_F__body_end

This function is called when the body of an iteration ends and the next iteration should be started.

_F__recontinue

This function implements a direct jump to the given address. It is used in implementation of the *iterate* statement.

main

This is the main function of an XE program. It initializes registers and stack.

_f__get_fp
_f__get_data_seg
_f__get_stack_seg
_f__get_rom_seg
_f__get_types_off
_f__get_types_seg
_f__get_own_off
_f__plm_area_segment
_f__plm_area_offset
_f__get_routines_off
_f__get_code_seg

These functions return addresses and register values that ACK or XE has no facilities for referring to.

_f__get_bit
_f__set_bit

These optimized functions get and set bits in the mark bit table used by garbage collection.

References

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986.

[2] Jari Arkko: An optimized implementation of an expert system language. Diploma thesis, Helsinki University of Technology, 1989. (In Finnish).

[3] Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Esko Nuutila, Markku Tamminen: XE Reference Manual. Helsinki University of Technology, Laboratory of Information Processing Sciences, report TKO-A26, 1989.

[4] Esko Nuutila, Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Markku Tamminen: AST and DG - The XE Abstract Syntax and Descriptor Generator Tools. To appear as a report of Laboratory of Information Processing Sciences in Helsinki University of Technology in 1989.

[5] Guy L. Steele, Jr.: Common LISP: The Language. Digital Press, 1984.

[6] Andrew S. Tanenbaum, Hans van Staveren, Ed G. Keizer, Johan W. Stevenson: Description of a machine architecture for use with block structured languages. Informatica Rapport IR-81.

[7] Telenokia Oy: DX 200 General Description 1.1. Telenokia Oy report No. CAN 0826.

puppu