HiPE Technical Reference
Version 0.99.0

Erik Johansson

February 21, 2003

Abstract

This manual is still in a beta state, please report any errors to the author (happi@csd.uu.se). I would also like to know if there is something that is missing or unclear in the presentation. The best source of information beside this manual is the source files, see the appendix for a description of the files.

Contents

1 Introduction **1**

1.1 What's new in this document? 1

1.2 The History of HiPE 2

2 The HiPE system **3**

2.1 Starting a HiPE system 3

2.2 Using the HiPE native code compiler 3

2.3 Overview of the compilation process 4

2.4 Handling of exceptions 5

2.5 Handling of closures 5

2.5.1 Closure structures 6

2.5.2 Creation of a closure 7

2.5.3 Calls to a closure 10

2.5.4 Closures and hot code loading 10

2.5.5 Closures in a distributed environment 10

2.6 Primitive operators in HiPE 10

3 New BIFs (in module hipe_bifs) **13**

3.1 Compiler and linker support 13

3.1.1 address_to_fun(Address) 13

3.1.2 alloc_code(SizeInBytes) 13

3.1.3 alloc_constant(SizeInBytes) 13

3.1.4 array(Size, InitVal) 13

3.1.5 array_length(Array) 13

3.1.6 array_sub(Array, Offset) 13

3.1.7 array_update(Array, Offset, Value) 13

3.1.8 atom_to_word(Atom) 13

3.1.9 bif_address(Mod, Fun, Arity) 13

3.1.10 catch_index_to_word(CatchIndex) 14

3.1.11 catch_table_nil() 14

3.1.12 catch_table_insert(Table, Catch) 14

3.1.13 catch_table_remove(?,?,?) 14

3.1.14 copy_term(Term, Address, Size) 14

3.1.15 emu_stub(Mod, Fun, Arity) 14
3.1.16 fun_to_address(MFA) 14
3.1.17 get_funinfo(MFA) 14
3.1.18 make_fe(NativeAddress, Module, BEAMAddress) 14
3.1.19 primop_address(Op) 14
3.1.20 read_u8(Address) 14
3.1.21 read_s32(Address) 14
3.1.22 read_u32(Address) 14
3.1.23 ref(Value) 14
3.1.24 ref_get(Ref) 14
3.1.25 ref_set(Ref, NewValue) 14
3.1.26 set_funinfo(Key) 15
3.1.27 set_native_address(MFA, Address, IsClosure) 15
3.1.28 term_size(Term) 15
3.1.29 write_u8(Address, Value) 15
3.1.30 write_s32(Address, Value) 15
3.1.31 write_u32(Address, Value) 15
3.2 Measurements 15
3.2.1 call_count_on(MFA) 15
3.2.2 call_count_off(MFA) 15
3.2.3 call_count_get(MFA) 15
3.2.4 call_count_clear(MFA) 15
3.3 Miscellaneous add-ons 15
3.3.1 Debugging 15

4 Intermediate code in HiPE **17**
4.1 Initializing 17
4.2 Control Flow Graphs 17
4.3 Basic Blocks 18

5 Icode **20**
5.1 Control flow instructions 22
5.1.1 label(Label) 22
5.1.2 goto(Label) 22
5.1.3 type(test, argument, true_label, false_label, prediction) 22
5.1.4 'if'(condition, args, true_label, false_label, prediction) 23

5.2 Operations . . . 23
5.2.1 mov(dst, src) . . . 23
5.3 Function application . . . 24
5.3.1 call([dst], fun, [arg], type, continuation, fail) . . . 24
5.3.2 enter(fun, [arg], type) . . . 24
5.3.3 return([var]) . . . 24
5.4 Error handling . . . 24
5.4.1 pushcatch(Label) . . . 25
5.4.2 restore_catch(Dst, Label) . . . 25
5.4.3 remove_catch(type, id) . . . 25
5.4.4 fail(Reason, Type) . . . 25
5.5 Comments . . . 26
5.5.1 comment(text) . . . 26
5.6 Arguments . . . 26
5.6.1 Constants . . . 26
5.6.2 Variables . . . 26

6 RTL- register transfer language 27

6.1 Variables . . . 28
6.2 Immediates . . . 28
6.3 RTL-Instructions . . . 28
6.4 Control flow instructions . . . 29
6.4.1 label(Name, Info) . . . 29
6.4.2 branch(Src1, RelOp, Src2, TrueLabel, FalseLabel, Prediction, Info) . . . 29
6.4.3 alu_branch(Dst, Src1, Op, Src2, Cond, TrueLabel, FalseLabel, Prediction, Info) . . . 29
6.4.4 goto(Label, Info) . . . 30
6.5 Operations . . . 31
6.5.1 move(Dst, Src, Info) . . . 31
6.5.2 alu(Dst, Src1, Op, Src2, Info) . . . 31
6.5.3 *(fp(Dst, Src1, Op, Src2, Info))* . . . 32
6.5.4 load(Dst, Src, Off, Info) . . . 32
6.5.5 load_atom(Dst, Atom, Info) . . . 32
6.5.6 load_address(Dst, Addr, Type, Info) . . . 32
6.5.7 store(Dst, Off, Src, Info) . . . 32
6.6 Comments . . . 33
6.6.1 comment(Text, Info) . . . 33

6.7 Function application 33
6.7.1 call(Dst, Fun, Args, Type, Info) 33
6.7.2 enter(Fun, Args, Type, Info) 33
6.7.3 return(Vars, Info) 33
6.7.4 gctest(Words, Info) 33

7 SPARC 34

7.1 Control flow instructions 34
7.1.1 label(Name, Info) 34
7.1.2 branch(Cond, TrueLabel, FalseLabel, Prediction, Annul, Info) 34
7.1.3 branch_on_reg(Reg, RegCond, TrueLabel, FalseLabel, Prediction, Annul, Info) 34
7.1.4 goto(Label, Info) 35
7.2 Operations 35
7.2.1 alu_cc(Dest, Src1, Op, Src2, Info) 35
7.2.2 alu(Dest, Src1, Op, Src2, Info) 35
7.2.3 load_atom(Dest, Atom, Info) 35
7.2.4 load(Dest, Type, Source, Off, Info) 36
7.2.5 move(Dest, Source, Info) 36
7.2.6 cmove_cc(Dest, Source, Cond, Info) 36
7.2.7 cmov_on_reg(Dest,Source,Reg,RegCond,Info) 36
7.2.8 store(Target, Off, Type, Source, Info) 36
7.2.9 sethi(Dest,Const,Info) 37
7.3 Function application 37
7.3.1 jmp_link( ) 37
7.3.2 jmp( ) 37
7.3.3 call_link( ) 37
7.4 Stack and memory 37
7.4.1 align( ) 37
7.5 Comments and NOP 38
7.5.1 comment(Comment,Info) 38
7.5.2 nop(Info) 38
7.6 Auxillary functions 38

Index 40

1 Introduction

The HiPE system is intended as an experimental platform for research in language implementation and compiler techniques as applied to ERLANG. One goal with this system is that is should be a complete ERLANG implementation. This can also be stated as the two subgoals that all ERLANG programs should execute in HiPE and that the system and compiler should be bug-free.

We in the HiPE development group have tried hard to achieve these goals and we think that we have reach the first subgoal, and we are constantly working on the second subgoal.

Another goal with the system is that it should be modular and easy to use for experimental research. How we have succeeded in this area we let you judge for yourself. Your comments and suggestions are welcome.

This document is a step in the direction of making the system easy to use. We hope that you, with the help of this document, will be able to add your own optimizations to any stage of the compiler and to understand and be able to modify the runtime system.

This document is not about ERLANG, if you want to learn ERLANG we recommend the documentation that can be found at:
`http://www.erlang.org/`

This document is not about how to install and run a HiPE system, for that kind of information and information about how to use the HiPE compiler please refer to the "HiPE User's Manual". (Although we briefly touch these subjects in Section 2.

→ Section 2

This document is about the internal structure of the HiPE runtime system and the HiPE compiler, it is intended for those of you that want to add your own optimizations to the compiler.

1.1 What's new in this document?

Version 0.99: Major rewrite bringing the manual from HiPE version 0.97 to HiPE version 1.0.2.

Version 0.9: Major rewrite bringing the manual from HiPE version 0.91 to HiPE version 0.97.

Version 0.8: RTL almost finished. Introduction added.

Version 0.7: RTL updated and RTL(1)-RTL(2) instructions are put in separate sections. All common RTL instruction are described. Most RTL(2) instructions are described.

Version 0.6: Description of catches in Icode. Correct description of apply in Icode.

1.2 The History of HiPE

The first HiPE compiler (code name Jerico) was written entirely in C and based on the ERLANG runtime system version 4.3 provided by Ericsson/OTP. The Jerico compiler compiled JAM byte-code to SPARC machine code. The last stable version of this HiPE system is version 0.28. Since it is based on a commercial version of ERLANG it is not distributed freely.

The HiPE compiler was then completely rewritten in ERLANG itself, and the runtime system was ported to version 4.5.3 of the runtime system provided by Ericsson/OTP. The last stable version of this system is HiPE version 0.90. This system is also a JAM to SPARC compiler and not freely distributed.

The runtime system was then ported to the JAM version of Open Source ERLANG version 4.7.3, and the compiler have been updated to, for example, use low bit tags. This system is now released under the OSE (Open Source ERLANG) licens as HiPE version 0.92.0.

Since Ericsson dropped support for JAM in their ERLANG system we had to follow suite and port HiPE to OTP-R7 and include BEAM support. This system is available as a stable but incomplete extension as HiPE 0.96.5.

To be able to make the HiPE system complete some changes in the runtime system where needed and Ericsson agreed to implement these in OTP-R8. HiPE versions 0.97.0 to 0.99.x were based on a prerelease (OTP-P8) of the upcomming OTP release.

HiPE 1.0.0 was released together with OTP-R8 in the fall of 2001.

The current version of HiPE (1.0.2) is based on a prerelease of OTP-R9.

2 The HiPE system

The HiPE ERLANG system is made up of four parts: a runtime system, an emulator, a compiler for the emulator, and a native code compiler. The current emulator is BEAM written by Ericsson. The compiler for the emulator is hence the BEAM compiler provided by Ericsson. The native code compiler is called HiPE and it can compile BEAM files to SPARC or IA32 native code.

2.1 Starting a HiPE system

You start a HiPE system just as you start an ordinary ERLANG system by typing `erl` (assuming you have built the system with `--enable_hipe`). At startup a HiPE system should say something like:

```
Erlang (BEAM) emulator version 2002.01.09 [source] [hipe]

Eshell  V2002.01.09 (abort with ^G)
1>
```

And the call `erlang:info(system_version).` should return something like:
`"Erlang (BEAM) emulator version 2002.01.09 [source] [hipe]\n"`
The HiPE version number, given by `hipe:version()` should be at least 1.0.0.

2.2 Using the HiPE native code compiler

The HiPE compiler is defined in the ERLANG module hipe. For a more thorough description of how to use the compiler see the HiPE User's Manual. The following functions are defined:

c `c{Mod,Fun,Arity}` - Compiles the function `Mod:Fun/Arity` with the default options.

c `c(Module)` - Compiles the entire module Module with the default options.

c `c({Mod,Fun,Arity},Options)` - Compiles `Mod:Fun/Arity` with the options `Options`. (For options see below.)

c `c(Module,Options)` - Compiles the entire module Module with the options `Options`. (For options see below.)

version `version()` - Returns the version of HiPE.

The argument `Options` is a list of options (for example `[o2, pp_sparc]`). Some options are:

- `safe` - Only realy trusted optimizations.
- `o1` - Some optimizations.
- `o2` - Several optimizations.
- `o3` - Hard (experimental) optimizations.

- `pp_all` - Pretty Print All.

For a full and up to date list of options call `hipe:help_options()` in the shell.

You can also write `{no,Option}` to turn the `Option` off. In that case, it will look to the system as if you haven't specified the option (and it won't be used as a default option either)

! → NOTE: the options are processed in the order they appear; an early option will 'shadow' a later one.
(E.g., `has_option([cse,{no,cse}]) = true`)

2.3 Overview of the compilation process

The compiler has four intermediate representations; an internal representation of BEAM code, a high level intermediate code called Icode, a general register transfer language called RTL (with two flavors RTL(1) and RTL(2)), and a machine-specific assembly language, either SPARC or x86; see Fig. 2.1.

ICode, RTL, and SPARC are all represented as control flow graphs of basic blocks. The BEAM code is translated to symbolic form by a straightforward process. Internal atom numbers are converted to real atoms, and branches to local functions are translated to call instructions with symbolic function names.

Icode is based on a register-oriented virtual machine for Erlang. Arguments and temporaries are located in an infinite number of registers, and all values are proper Erlang terms. The call stack is implicit, and calls preserve registers. Bookkeeping operations, such as heap overflow checks, context switching, and time-slice decrements, are implicit.

The Icode is then optimized with copy and constant propagation, and constant folding. This is done in one pass over all extended basic blocks. Dead code removal is then performed to remove assignments to dead temporaries.

Unreachable code is removed by the translation to RTL, since only reachable basic blocks are inserted in the RTL control flow graph. Operations on Erlang values are expanded to make data tagging and untagging explicit.

The same optimisations that were performed on Icode are then applied to the RTL code. Heap overflow tests, call stack management, and the saving and restoring of registers around calls are made explicit, and the standard optimisations are applied again. In order to limit the number of heap overflow tests, they are propagated backwards as far as possible, and adjacent tests are merged.

The RTL code then is translated to abstract SPARC code, and registers are assigned using a simple graph-colouring register allocator. Finally, symbolic references to atoms and functions are replaced by their values in the running system, memory is allocated for the code, and the code is linked into the system.

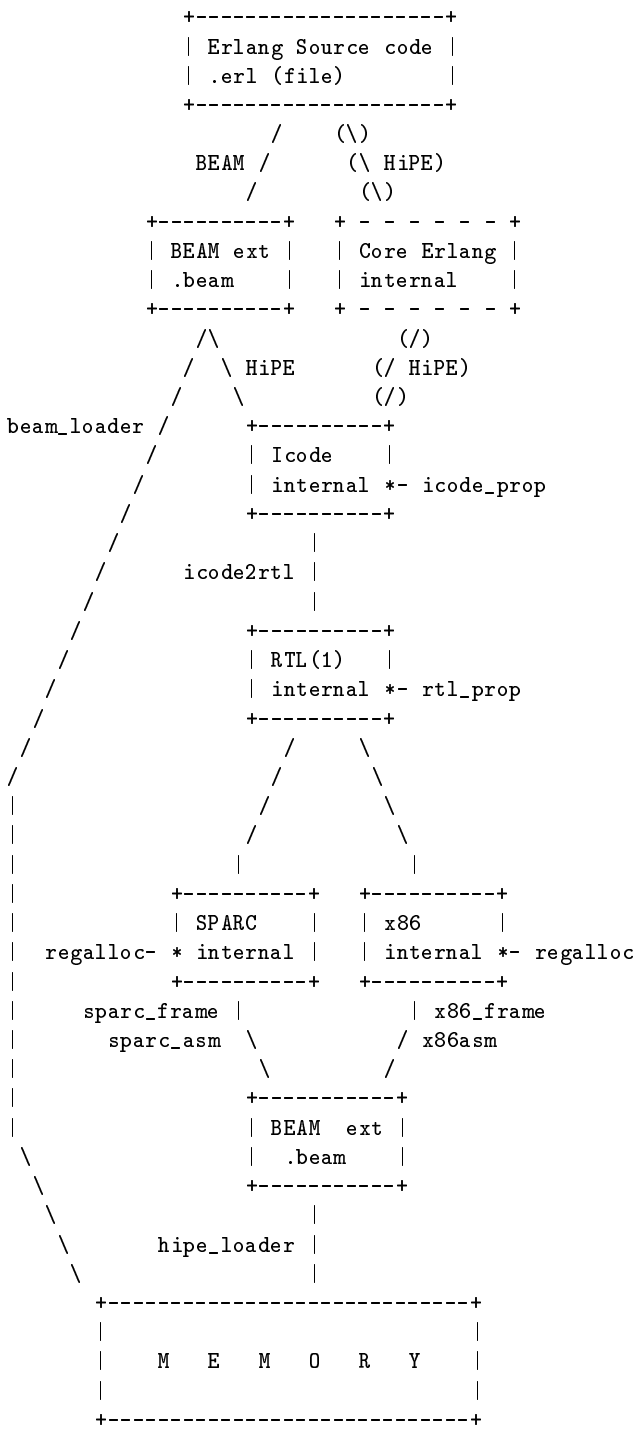

Figure 2.1: Intermediate representations in HiPE.

2.4 Handling of exceptions

2.5 Handling of closures

This section describes how *closures* are handled in HiPE. Some knowledge about the internals of the BEAM system is needed to fully understand the whole description, but the general principle is quite simple. There are three parts to a closure: 1) the code of the

closure, 2) a fun-entry with a pointer to the code created at load time, and 3) a closure created at runtime. The closure contains the free variable and a pointer to the code (direct for native code and indirect via the fun-entry for emulated code). When a closure is called the closure itself is passed as the last argument giving the code access to the values in these variables. The design is complicate since the BEAM code expects to get the free variables as arguments, and since there has to be support for hot code loading.

The following paragraphs describes the implementation in detail.

2.5.1 Closure structures

There are two structures that keeps the information about a closure, the erl_fun_entry (contains pointers to code) and the erl_fun_thing (contains free variables, the actual closure).

```
typedef struct erl_fun_entry {
    HashBucket bucket;
#if 0
    /* XXX Not yet */
    byte mod_id[16]; /* MD5 for module. */
    int index; /* New style index. */
#endif
    int old_uniq; /* Unique number (old_style) */
    int old_index; /* Old style index */
    Eterm* address; /* Pointer to code for fun */
#ifdef HIPE
    Eterm* native_address; /* Native code for the fun. */
#endif
    Eterm module; /* Tagged atom for module. */
    int refc; /* Reference count: One for code
                                    + one for each
   fun object in each process. */
} ErlFunEntry;

/*
 * This structure represents a 'fun' (lambda).
 * It is stored on process heaps.
 * It has variable size depending on the size
 * of the environment.
 */

typedef struct erl_fun_thing {
    Eterm thing_word; /* Subtag FUN_SUBTAG. */
    struct erl_fun_thing* next; /* Next fun in mso list. */
    Eterm creator; /* Pid of creator process
                                     (contains node). */
    ErlFunEntry* fe; /* Pointer to fun entry. */
#ifdef HIPE
    Eterm* native_address; /* Native code for the fun. */
    Uint real_arity;            /* Arity of the native fun. */
#endif
    Uint num_free; /* Number of free variables
                                    (in env). */
    Eterm env[1]; /* Environment
                                    (free variables). */
```

```
} ErlFunThing;
```

2.5.2 Creation of a closure

Assume a module M with a function F/A containing a fun of arity A' with FV free variables.

BEAM handles creation of the closure as follows:

Compile time A new local function F' (actually named F/A-fun-I- , where I is the index of the fun within the function) with the code for the fun is created. This function assumes that the free variables of the fun are passed in as arguments to the function. The instruction make_fun(L, Magic, FreeVariables), where L is the address of F', is inserted into the BEAM code for F.

Load time A function entry (ErlFunEntry fe) for the fun F' is created. The fun name in the make_fun instruction is patched with a pointer to the function entry (`&fe`). A pointer to a native code stub of the right arity is inserted in `fe->native_address`. To get this address the arity of the fun is needed, this is calculated as:
`arity = ((int)(fe->address[-1])) - num_free;`
For this num_free is needed, but num_free and `fe->address` are not known at the same time in the beam loader. HACK: num_free is stored in the Lambda structure of the BEAM loader when fe is created. ...

Runtime When the BEAM executes the instruction make_fun a closure (ErlFunThing) is created as follows: Space is allocated on the heap for the closure

```
unsigned needed = ERL_FUN_SIZE + num_free;
ErlFunThing* funp =
  (ErlFunThing *) HAlloc(p, needed);
funp->thing_word = HEADER_FUN;
```

The reference counter of the function entry is increased

```
fe->refc++;
```

The closure is linked in to the list of closures in the current process.

```
funp->next = p->off_heap.funs;
p->off_heap.funs = funp;
```

The fields in the closure are initialized

```
funp->num_free = num_free;
funp->creator = p->id;
funp->fe = fe;
```

The free variables are copied from the BEAM register file (reg) to the closure

```
Eterm* hp = funp->env;
int i;
for (i = 0; i < num_free; i++) {
  *hp++ = reg[i];
}
```

Then the closure is tagged

```
make_fun(funp);
```

HiPE handles creation of the closure as follows:

Compile time The BEAM instruction
`make_fun(F', Magic, FreeVariables)`
is patched to

```
Fun :=
 patched_make_fun(M, F', A+FV, UNIQUE, FV,
                  MODINDEX, [FreeVars])
```

this is then translated to code that creates the closure.

Given: `(FV = length(FreeVars))` in translation to rtl, make fun is translated to:

```
%% set funp to point to the closure
funp := heaptop
%% copy free vars to the closure
x0 := funp + ?EFT_ENV
[x0+0] = FreeVar1
  ...
[x0+FV*4] = FreeVarFV

%% Get the address to the fun-entry of F'
x1 := load_address(F', 'closure')
%% Store fe in funp->fe
[funp+?EFT_FE] := x1
%% Cache the native address and arity in
%%  funp->native_address
%%  funp->real_arity
x2 := [x1+?EFE_NATIVE_ADDRESS]
[funp+?EFT_NATIVE_ADDRESS] := x2
x3 := ARITY-FV
[funp+?EFT_REAL_ARITY] := x3

%% Increase the reference count in fe->refc
x4 := [x1+?EFE_REFC]
x5 := x4 + 1
[x1+?EFE_REFC] := x5

%% Store number of free vars in funp->num_free
[funp+?EFT_NUM_FREE] := FV
%% Store the pid in funp->creator
x6 := [p+?P_ID]
[funp+?EFT_CREATOR] := x6
[funp+?EFT_THING] := ?HEADER_FUN

%% Link funp to list in p->off_heap.funs
x7 := [p+?P_OFF_HEAP_FUNS]
[funp+?EFT_NEXT] := x7
[p+?P_OFF_HEAP_FUNS] := funp

%% Tag funp and allocate space on the heap
fun := tag_fun(funp)
```

```
heaptop :=
  heaptop + ((?ERL_FUN_SIZE + FV) *4)
```

The BEAM code for the closure is patched with a prelude:

```
F'(Arg_1, ..., Arg_A, FreeV_1, ..., FreeV_FV, Closure) ->
 L1: Instr1
     ...
```

Is patched to:

```
F'(Arg_1, ..., ArgA, Closure) ->
 L0: FreeV_FV := hipe:closure_element(FV, Closure),
     ...
     FreeV_1 := hipe:closure_element(1, Closure),
     Goto L1
 L1: Instr1
    ...
```

Load time A function entry (`fe`) is created for the closure. A pointer to an emulator stub is inserted into `fe->address`. (If there existed BEAM code for the closure, then that code is used as the stub.) The BEAM code for the sub is patched so that calls are redirected to hipe-mode-switch-call-closure.

For SPARC six stubs are needed (`nbif_ccallemu0` to `nbif_ccallemu5`), they look as:

```
nbif_ccallemu0:
ba .ccall
st ARG0,[P+P_CLOSURE]
nbif_ccallemu1:
ba .args0
st ARG1,[P+P_CLOSURE]
...
nbif_ccallemu5:
ld TEMP, [NSP-4]
ba .args4
st TEMP,[P+P_CLOSURE]
.global nbif_ccallemu4

args4: st ARG4,[P+P_ARG4]
args3: st ARG3,[P+P_ARG3]
args2: st ARG2,[P+P_ARG2]
args1: st ARG1,[P+P_ARG1]
args0: st ARG0,[P+P_ARG0]
ccall: st RA,[P+P_NRA] ! native return address
ba .flush_exit
mov HIPE_MODE_SWITCH_RES_CLOSURE,TEMP
```

The last stub works as long as the calling convention puts the last argument on the top of the stack (true for SPARC). For other calling conventions one can get the closure in a special stub created for each arity as needed.

The instruction

```
x1 := load_address(F', 'closure')
```

is patched to

```
x1 := load(&fe)
```

Runtime At runtime a test is performed to ensure that enough room is available on the heap, then the code to create the closure that was described above is executed.

2.5.3 Calls to a closure

BEAM calls a closure The address to the closure is found at `funp->fe->address`. The free variables are copied from `funp->env` to `reg[arity...arity+num_free-1]`. The last argument `reg[arity+num_free]` is initialized to the closure (funp). If the number of arguments is equal to the arity of the closure execution continues at the given address.

If the closure is compiled to native code then the address points to a stub. The stub code traps to hipe-mode-switch which puts actuals and closure in native regs (and on the stack if needed) then it makes a normal native call to `funp->native_address`.

Native call to a closure `C(A1, ..., AN)` First the type of C is cheked to ensure it is a closure, otherwise `{'EXIT', badfun}` is thrown. Then the arity of C is found by

```
funp = UNTAG(C)
arity = funp->real_arity
```

Then a check ensures that `arity == N (== length(Args))` (`{'EXIT', badarity}` is thrown otherwise).

The call passes actuals and closure as normal parameters and jumps to `funp->native_address`. If the function is not compiled to native code `funp->native_address` points to a stub (as described above).

2.5.4 Closures and hot code loading

TO BE WRITTEN

2.5.5 Closures in a distributed environment

TO BE WRITTEN

2.6 Primitive operators in HiPE

Where the primop can be one of the operator described in Table 2.1.

→ Table 2.1

Some primops can be created with a shortcut:

mk_gctest `mk_gctest() - -` mk_primop([],gc_test,[])

mk_redtest `mk_redtest() - -` mk_primop([],redtest,[])

Op	Dest	Arg	Description
Arithmetic			
'+'	[Dst]	[Arg1,Arg2]	Dst:=Arg1+Arg2
'-'	[Dst]	[Arg1,Arg2]	Dst:=Arg1-Arg2
'*'	[Dst]	[Arg1,Arg2]	Dst:=Arg1*Arg2
'/'	[Dst]	[Arg1,Arg2]	Dst:=Arg1/Arg2 fp.
'div'	[Dst]	[Arg1,Arg2]	Dst:=Arg1/Arg2 int.
'rem'	[Dst]	[Arg1,Arg2]	Dst :=reminder(Arg1/Arg2)
Bitwise operations			
'band'	[Dst]	[Arg1,Arg2]	
'bor'	[Dst]	[Arg1,Arg2]	
'bxor'	[Dst]	[Arg1,Arg2]	
'bsl'	[Dst]	[Arg1,Arg2]	
'bsr'	[Dst]	[Arg1,Arg2]	
'bnot'	[Dst]	[Arg]	
Lists and tuples[a]			
cons	[Dst]	[Car,Cdr]	Cons the car and cdr.
unsafe_hd	[Dst]	[Arg]	Get the head of the cons Arg
unsafe_tl	[Dst]	[Arg]	Get the tail of the cons Arg
mktuple	[Dst]	Elements	Make a tuple of the Elements.
{unsafe_element,N}	[Dst]	[Tuple]	Nth element of Tuple
Other			
self	[Dst]	[]	Get the PID
gc_test	[]	[]	Do a GC test.
redtest	[]	[]	Do a reduction test.
send	[Dst]	[To, Mess]	To ! Mess, Dst := Mess
get_msg	[Dst]	[]	Look at next message.
next_msg	[]	[]	Advance message ptr.
join	[]	[]	Remove selected mess.
set_timeout	[]	Time	Set timeout.
BIFs			
<bif_NAME_ARITY>	[Dst,F]	Args	F is used to test for success.

Table 2.1: Primitive operators.

[a] The *unsafe* operations do no type test nor any range check.

The following operators and arguments can be used as guardops:

Op	Dest	Arg	Description
Arithmetic			
'+'	[Dst]	[Arg1,Arg2]	Dst:=Arg1+Arg2
'-'	[Dst]	[Arg1,Arg2]	Dst:=Arg1-Arg2
'*'	[Dst]	[Arg1,Arg2]	Dst:=Arg1*Arg2
'/'	[Dst]	[Arg1,Arg2]	Dst:=Arg1/Arg2 fp.
'div'	[Dst]	[Arg1,Arg2]	Dst:=Arg1/Arg2 int.
'rem'	[Dst]	[Arg1,Arg2]	Dst :=reminder(Arg1/Arg2)
Bitwise operations			
'band'	[Dst]	[Arg1,Arg2]	
'bor'	[Dst]	[Arg1,Arg2]	
'bxor'	[Dst]	[Arg1,Arg2]	
'bsl'	[Dst]	[Arg1,Arg2]	
'bsr'	[Dst]	[Arg1,Arg2]	
'bnot'	[Dst]	[Arg]	
Other			
self	[Dst]	[]	Get the PID
{unsafe_element, N}	[Dst]	[Arg]	Nth element of Arg
unsafe_hd	[Dst]	[Arg]	Get the head of the cons Arg
unsafe_tl	[Dst]	[Arg]	Get the tail of the cons Arg

The word unsafe in the last three operators means that no type checking or range checking is done. Be sure to only use these when Arg can be guaranteed to have the right type.

3 New BIFs (in module hipe_bifs)

3.1 Compiler and linker support

3.1.1 address_to_fun(Address)

Returns the name of the ERLANG function who's JAM code starts at address `Adress`.

3.1.2 alloc_code(SizeInBytes)

Allocates a block of memory of size `SizeInBytes` bytes at a memory address that fits in an ERLANG fixnum. The address is returned as a fixnum.

3.1.3 alloc_constant(SizeInBytes)

The function `alloc_constant/1` is like `alloc_code/1`, except it returns memory suitable for storing constant Erlang values.

These constants must not be forwarded by the gc. Therefore, the gc needs to be able to distinguish between collectible objects and constants. Unfortunately, an Erlang process' collectible objects are scattered around in two heaps and a list of message buffers, so testing "is X a collectible object?" can be expensive.

Instead, constants are placed in a single contiguous area, which allows for an inexpensive "is X a constant?" test.

3.1.4 array(Size, InitVal)

Creates a mutable array of `Size` immediates (small integers or atoms) and fills it with `InitVal`.

3.1.5 array_length(Array)

Returns the size of the array.

3.1.6 array_sub(Array, Offset)

Returns the element at position `Offset` in the array.

3.1.7 array_update(Array, Offset, Value)

Sets the element at position `Offset` in the array to `Value`.

3.1.8 atom_to_word(Atom)

Returns the atom number of `Atom`.

3.1.9 bif_address(Mod, Fun, Arity)

Returns the native address of a built-in-function (of `false` if the function is not implemented in native code (or C)).

3.1.10 catch_index_to_word(CatchIndex)

3.1.11 catch_table_nil()

3.1.12 catch_table_insert(Table, Catch)

3.1.13 catch_table_remove(?,?,?)

3.1.14 copy_term(Term, Address, Size)

Copies the ERLANG term `Term` of size `Size` to the address `Address`.

3.1.15 emu_stub(Mod, Fun, Arity)

Creates a BEAM-code stub for the function `Mod:Fun/Arity` and returns the address to this stub.

3.1.16 fun_to_address(MFA)

`MFA` is a tuple: `{Module, Function, Arity}`. Returns the address to the BEAM-code of the function `MFA`.

3.1.17 get_funinfo(MFA)

Returns native code information about the function `MFA`.

3.1.18 make_fe(NativeAddress, Module, BEAMAddress)

Creates a ErlFunEntry for the function with code at address NativeAddress. (See Section 2.5.)

3.1.19 primop_address(Op)

Returns the address to the primop `Op`.

3.1.20 read_u8(Address)

Returns the (unsigned) byte at `Address`.

3.1.21 read_s32(Address)

Returns the (signed) 32-bit word at `Address`.

3.1.22 read_u32(Address)

Returns the (unsigned) 32-bit word at `Address`.

3.1.23 ref(Value)

Creates a mutable reference to the immediate value `Value`.

3.1.24 ref_get(Ref)

Returns the value of the reference `Ref`.

3.1.25 ref_set(Ref, NewValue)

Mutates the reference `Ref`, by setting te value to `NewValue`.

3.1.26 set_funinfo(Key)

`Key` should be of the form {`MFA, Value`}. Sets the native information about the fun `MFA` to `Value`.

3.1.27 set_native_address(MFA, Address, IsClosure)

Sets the address for native code for the function `MFA` to `Address`, if `IsClosure` is the atom `true` then the BEAM-code is patched to call a closure otherwise the beam code is patched to call an ordinary function.

3.1.28 term_size(Term)

Calculates the size of an ERLANG term.

3.1.29 write_u8(Address, Value)

Writes the (unsigned) byte `Value` to `Address`.

3.1.30 write_s32(Address, Value)

Writes the (unsigned) 32-bit word `Value` to `Address`.

3.1.31 write_u32(Address, Value)

Writes the (unsigned) 32-bit word `Value` to `Address`.

3.2 Measurements

3.2.1 call_count_on(MFA)

Turns on emulator profiling (counting of calls to) for the function `MFA`.

3.2.2 call_count_off(MFA)

Turns off emulator profiling (counting of calls to) for the function `MFA`.

3.2.3 call_count_get(MFA)

Returns the number of times the emulated function `MFA` has been called since call count was turned on (or cleared).

3.2.4 call_count_clear(MFA)

Clears the call counter for the function `MFA`.

3.3 Miscellaneous add-ons

3.3.1 Debugging

show_estack `show_estack(PID)` - Prints the emulator stack for the process `PID` to stdout.

show_heap `show_heap(PID)` - Prints the heap of the process `PID` to stdout.

show_nstack `show_nstack (PID)` - Prints the native stack for the process `PID` to stdout.

show_p_struct `show_p_struct(PID)` - Prints the process-struct of the process `PID` to stdout.

show_term `show_term(Term)` - Prints the hexadecimal value of the representation of the term `Term`, the term itself is also printed.

4 Intermediate code in HiPE

There are three levels of intermediate code in HiPE: Icode - imperative code with high level instructions and primitive operations inherited from ERLANG, RTL - a generic RISC three address code, and SPARC - specific assembler code for SPARC (level 9), or X86 - specific assembler code for IA32.

All three comes in two different layouts: *CFG - control flow graph*, and *linear*. In CFG the code is divided into *Basic Blocks* with predecessors and successors. The linear code is just a list of instructions with some meta information added to it.

4.1 Initializing

To get a dense numbering of labels and still get unique labels, we use our own gensym. To make this work you need to initialize the gensym server. If you for example want to be able to work with an "old" rtl-cfg you need to do the following:

```
{LLow, LHigh} = rtl_cfg:label_range(CFG),
gensym:set_label(LHigh),
{VLow, VHigh} = rtl_cfg:var_range(CFG),
gensym:set_var(VHigh),
Start = rtl_cfg:start_label(CFG)
```

4.2 Control Flow Graphs

There are functions for constructing and manipulating the control flow graph of a function. A unique label identifies each Basic Block in the CFG, all labels in the CFG is within the label range of the CFG. All variables (temporaries) are within the variable range of the CFG.

There is a module for each one of the codes:
`icode_cfg.erl`, `rtl_cfg.erl`, and `sparc_cfg.erl`. Each module contains the following functions:

- `init(Code)`- Makes a CFG out of `Code` (See the chapters on Icode, RTL and SPARC for descriptions on the format of `Code`).
- `bb(CFG, Label)`- Returns the basic block associated with `Label`.
- `bb_update(CFG, Label, NewBB)`- Makes `NewBB` the basic block associated with `Label`.
- `succ_map(CFG)`- Returns a mapping from labels to successors.
- `succ(Map, Label)`- Returns a list of successors of the basic block with the associated with `Label`.
- `pred_map(CFG)`- Returns a mapping from labels to predecessors.

- `pred(Map, Label)`- Returns the predecessors of the basic block associated with `Label`.
- `fallthrough(CFG, Label)`- Returns fall-through successor of the basic block associated with `Label` (or the atom `none`).
- `cond(CFG, Label)`- Returns conditional successor (or the atom `none`).
- `start(CFG)`- Returns the label of the entry basic block.
- `fail_entrypoints(CFG)`- Returns a list of labels that are fail entry points. Execution of a catch can start at all of these points.
- `params(CFG)`- Returns the list of parameter names.
- `labels(CFG)`- Returns a list of labels of all basic blocks in CFG.
- `postorder(CFG)`- Returns a list of labels in postorder.
- `reverse_postorder(CFG)`- Returns a list of labels in reverse postorder.
- `linearize(CFG)`- Converts CFG to linear code.
- `var_range(CFG)`- Returns the variable range as a tuple: `{Min, Max}`.
- `label_range(CFG)`- Returns the label range as a tuple: `{Min, Max}`.

The following functions are also available...
... a description will be added soon.

```
add_fail_entrypoint/2,
is_entry/2,
info/1,
info_update/2,
start_label/1,
start_label_update/2,
preorder/1,
reverse_preorder/1,
bb_add/3,
bb_remove/2,
redirect/4,
var_range_update/2,
label_range_update/2,
info/1,
info_add/2,
info_update/2,
pp/1
```

4.3 Basic Blocks

The functions for manipulating basic blocks can be found in the module `bb`.

- `mk_bb(Code)`- Creates a basic block from the list of instructions `Code`.
- `mk_bb(Code, Annot)`- Creates a basic block from the list of instructions `Code` and annotates it with `Annot`.

- `code(BB)`- Returns the code in the basic block.
- `annot(BB)`- Returns the annotations for the basic block.
- `code_update(BB, NewCode)`- Replaces the code in `BB` with `NewCode`.
- `last(BB)`- Returns the last instruction in the BB.
- `butlast(BB)`- Returns the code except for the last instruction.

5 Icode

The module `icode` contains all functions necessary to work with Icode. Linear Icode consists of:

- The name of a function (as a MFA `{M,F,A}`).
- A list of parameters (as Icode variables).
- A list of Icode instructions.
- A variable range.
- A label range.
- And optional additional information.

The following functions can be used to handle linear Icode:

mk_icode `mk_icode(Fun, Params, CodeList, VarRange, LabelRange)` - Creates linear Icode from the given components.

icode_fun `icode_fun(Icode)` - Returns the function as a MFA.

icode_params `icode_params(Icode)` - Returns the list of parameters.

icode_code `icode_code(Icode)` - Returns the list of instructions.

icode_var_range `icode_var_range(Icode)` - Returns the "var range".

icode_label_range `icode_label_range(Icode)` - Returns the "label range".

icode_info `icode_info(Icode)` - Returns additional information about the code.

icode_info_add `icode_info_add(Icode, Info)` - Add some additional information to the code.

icode_info_update `icode_info_update(Icode,Info)` - Replace the additional information about the code.

pp `pp(Icode)` - Pretty-print the code.

pp_exit `pp_exit(Icode)` - Pretty-print the code, exit with the exception `{'EXIT',{pp,Instr}}` if anything goes wrong.

remove_empty_bbs `remove_empty_bbsT` - rim the code.

is_leaf `is_leaf(Icode)` - True if the function is a leaf function.

All instructions has an information field and these functions works on all Icode instructions:

info_add `info_add(Instr,Info)` - Adds extra information to an instruction.

info_update `info_update(Instr,NewInfo)` - Replaces the information for the instruction.

info `info(Instr)` - Returns the information.

The following instructions also work on all Icode instructions:

type `type(Instr)` - Return the type of the Icode instruction as an atom.

is_branch `is_branch(Instr)` - True if the instruction is a branch.

is_uncond `is_uncond(Instr)` - True if the instruction is an unconditional branch.

successors `successors(Instr)` - Returns a list of successors to a branch instruction.

is_pure `is_pure(Instr)` - True only if the instruction can not fail. (Can still be false for some instructions that can not fail.)

uses `uses(Instr)` - Returns a list of variables that the instruction uses (reads from).

defines `defines(Instr)` - Returns a list of variables that the instruction defines (writes to).

pp_instr `pp_instr(Instr)` - Pretty print the instruction.

subst `subst(Substs,Instr)` - `Substs` is a list of the form `[{NewVar,OldVar},...]`. Replaces all occurrences of `OldVar` in the instructions with `NewVar`.

subst_uses `subst_uses(Substs,Instr)` - Subst as above. Replaces all uses of `OldVar` in the instructions with `NewVar`.

subst_defines `subst_defines(Substs,Instr)` - Subst as above. Replaces all definitions of `OldVar` in the instructions with `NewVar`.

redirect_jmp `redirect_jmp(Instr,ToOld,ToNew)` - Makes `ToNew` the destination of any destinations that previously was `ToOld`.

In the following sections all Icode instruction are described. They all come with their own constructor functions, selector functions, and recognition functions. These are on the forms `mk_INSTR(Parts)`, `INSTR_PART(Instr)`, and `is_INSTR(Instr)` respectively.

The *icode instructions* are:

Instruction	Arguments
'if'	cond, args, true_label, false_label
switch_val	arg, fail_label, length, [cases]
switch_tuple_arity	arg, fail_label, length, [cases]
type	typ_expr, arg, true_label, false_label
goto	label
label	name
mov	dst, src
call	[dsts], fun, [args], type, continuation, fail, in_guard, code_change
enter	fun, [arg], type, code_change
return	[var]
pushcatch	label
restore_catch	dst, label
remove_catch	label
fail	reason, type
comment	text

5.1 Control flow instructions

5.1.1 label(Label)

Names a point in the code.

mk_label `mk_label(Label)` - Creates a label with name Label.

mk_new_label `mk_new_label()` - Creates a new unique label. (If you have initialized gensym correctly.)

label_name `label_name(Instr)` - Returns the name of the label.

is_label `is_label(Instr)` - True if `Instr` is a label.

5.1.2 goto(Label)

Unconditional branch to a label.

mk_goto `mk_goto(Label)` - Creats a goto instruction.

goto_label `goto_label(Instr)` - Returns the label name.

is_goto `is_goto(Instr)` - True if the instruction is a goto.

5.1.3 type(test, argument, true_label, false_label, prediction)

Where test is one of the tests described below. If the type expression applied to argument is true then go to the true label else go to the false label. The prediction is a number between 0.0 and 1.0 indicating the probability that the test will succeed.

mk_type `mk_type(Arg, TypeExpr, TrueLbl, FalseLbl)` - Create a type test with prediction 0.5.

mk_type `mk_type(Arg, TypeExpr, TrueLbl, FalseLbl, Prediction)` - Create a type test with the given prediction.

type_var `type_var(Instr)` - Return the argument.

type_type `type_type(Instr)` - Return the type expression.

type_true_label `type_true_label(Instr)` - Return the true label.

type_false_label `type_false_label(Instr)` - Return the false label.

type_pred `type_pred(Instr)` - Return the prediction.

is_type `is_type(Instr)` - True if the instruction is a type test instruction.

The following type tests are available:

Type Expr	Meaning
list	nil or cons
nil	[]
cons	cons
tuple	tuple of any arity
{tuple, N}	tuple of arity N
atom	Any atom
{atom, Atom}	the atom Atom
constant	any constant
number	any number
integer	any integer
{integer, N}	the integer N
fixnum	any fixnum
bignum	any bignum
float	any float
pid	is pid
port	is port
reference	is reference
binary	is binary
function	is function

5.1.4 'if'(condition, args, true_label, false_label, prediction)

Where condition is one of the operators described below. If the condition applied to the arguments is true then go to the true label else go to the false label. The optional prediction is a number between 0 and 1 predicting the chance of success.

mk_if `mk_if(Op, Args, TrueLbl, FalseLbl)` - Constructor, the prediction defaults to 0.5.

mk_if `mk_if(Op, Args, TrueLbl, FalseLbl, P)` - Constructor.

if_op `if_op(Instr)` -

if_true_label `if_true_label(Instr)` -

if_false_label `if_false_label(Instr)` -

if_args `if_args(Instr)` -

if_pred `if_pred(Instr)` -

is_if `is_if(Instr)` -

The following *binary operators* can be used:
`>, <, >=, =<, ==, /= , =:=, =/=`

5.2 Operations

5.2.1 mov(dst, src)

Copy the source to the destination. A constant can only show up on the right hand side (as the source).

mk_mov `mk_mov(Dst, Src)` -

mk_movs `mk_movs(DstList, SrcList)` -

mov_dst `mov_dst(Instr)` -

mov_src `mov_src(Instr)` -

mov_src_update `mov_src_update(Instr,NewSrc)` -

is_mov `is_mov(Instr)` -

5.3 Function application

Here 'fun' is a tuple: Module, Function, Arity.

5.3.1 call([dst], fun, [arg], type, continuation, fail)

Call the function fun with the arguments and store the results in the destinations (multiple return values are partly supported). The call is also annotated with information about whether it is a local or remote call (in the type field).

After the call execution continues at `continuation`. If the call fails and `fail` in not nil then the exception will be caught at `fail`.

mk_call `mk_call(Dst, Mod, Fun, Args, Type, Continuation, Fail)` -

call_dst `call_dst(Instr)` -

call_args `call_args(Instr)` -

call_fun `call_fun(Instr)` -

call_type `call_type(Instr)` -

is_call `is_call(Instr)` -

5.3.2 enter(fun, [arg], type)

A tail recursive call. (No return values.)

mk_enter `mk_enter(Mod, Fun, Args, Type)` -

enter_fun `enter_fun(Instr)` -

enter_args `enter_args(Instr)` -

enter_type `enter_type(Instr)` -

is_enter `is_enter(Instr)` -

5.3.3 return([var])

Return from the function possibly with multiple return values.

mk_return `mk_return(Vars)` -

is_return `is_return(Instr)` -

return_vars `return_vars(Instr)` -

5.4 Error handling

A catch sequence in Icode is built as follows:

```
  pushcatch('catchlblx')
  'CODE TO CATCH... '
  remove_catch('catchlblx')
  goto 'catchend'
'catchlblx':
  restore_catch(Vi, 'catchlblx')
  goto 'catchend'
'catchend':
```

5.4.1 pushcatch(Label)

Make a catch frame on the stack. The `Label` is a label where execution should continue if an exception is catch.

mk_pushcatch `mk_pushcatch(Label)` -

pushcatch_label `pushcatch_label(Instr)` -

is_pushcatch `is_pushcatch(Instr)` -

5.4.2 restore_catch(Dst, Label)

Restores the saved variables in the list `var` from a catch frame and moves the thrown (or exit) value to the variable `dst`. The `id` identifies the catch to restore, it should be the same as the id for the corresponding `push_catch`.

mk_restore_catch `mk_restore_catch(Dst, Id)` -

is_restore_catch `is_restore_catch(Instr)` -

restore_catch_dst `restore_catch_dst(Instr)` -

restore_catch_id `restore_catch_id(Instr)` | -

5.4.3 remove_catch(type, id)

Removes the catch-frame `id` where `size` variables are stored. If the `type` is `not_taken` then it handles the case where there where no exit or throw. The `type taken` handles the case where there where an exit or throw.

mk_remove_catch `mk_remove_catch(Type, Id)` -

is_remove_catch `is_remove_catch(Instr)` -

remove_catch_type `remove_catch_type(Instr)` -

remove_catch_id `remove_catch_id(Instr)` -

5.4.4 fail(Reason, Type)

Fail with reason `Reason`, `Type` is either `exit` or `throw`.

mk_fail `mk_fail(Reason, Type)` -

is_fail `is_fail(Instr)` -

fail_reason `fail_reason(Instr)` -

fail_type `fail_type(Instr)` -

5.5 Comments

5.5.1 comment(text)

A comment

mk_comment `mk_comment(Text)` -

is_comment `is_comment(Instr)` -

comment_text `comment_text(Instr)` -

5.6 Arguments

5.6.1 Constants

mk_const `mk_const(Const)` -

is_const `is_const(Instr)` -

const_value `const_value(Instr)` -

5.6.2 Variables

mk_var `mk_var(Id)` -

mk_new_var `mk_new_var()` -

var_name `var_name(Instr)` -

is_var `is_var(Instr)` -

6 RTL- register transfer language

The module `rtl` contains all functions necessary to work with RTL. Linear RTL consists of:

- The name of a function (as a MFA `{M,F,A}`).
- A list of parameters (as RTL variables).
- A list of RTL instructions.
- A variable range.
- A label range.
- And optional additonal information.

The following functions can be used to handle linear RTL:

mk_rtl : `mk_rtl(Fun, Args, Code, VarRange, LabelRange)` -

rtl_fun : `rtl_fun(RTLcode)` -

rtl_params : `rtl_params(RTL)` -

rtl_code : `rtl_code(RTL)` -

rtl_code_update : `rtl_code_update(RTL,Code)` -

rtl_var_range : `rtl_var_range(RTL)` -

rtl_var_range_update : `rtl_var_range_update(RTL,NewVarRange)` -

rtl_label_range : `rtl_label_range(RTL)` -

rtl_label_range_update : `rtl_label_range_update(RTL,NewLabelRange)` -

rtl_info : `rtl_info(RTL)` -

rtl_info_update : `rtl_info_update(RTL,Info)` -

remove_empty_bbs : `remove_empty_bbs(RTL)` -

pp : `pp(RTL)` -

pp : `pp(Dev, RTL)` -

All instructions has an information field and these functions works on all RTL instructions:

`info(Instr)` :

`info_add(Instr, Info)` :

`info_update(Instr, NewInfo)` :

The following instructions also work on all RTL instructions:

`type(Instr)` :

`uses(Instr)` :

`defines(Instr)` :

`redirect_jmp(Instr, ToOld, ToNew)` :

```
is_pure(Instr) :
```

6.1 Variables

There are three kinds of *variables* in RTL. Variables containing tagged data that are traced by the GC. Registers that are ignored by the GC. Floating point registers.

mk_var(Name)
mk_new_var()
is_var(Val)
var_name(Val)

mk_reg(Name)
mk_new_reg()
is_reg(Val)
reg_name(Val)

mk_fpreg(Name)
is_fpreg(Val)
fpreg_name(Val)

6.2 Immediates

mk_imm(Imm)
is_imm(Val)
imm_value(Val)

6.3 RTL-Instructions

The *RTL instructions* are:

Instruction	Arguments
label	Name
branch	Src1, RelOp, Src2, TrueLabel, FalseLabel, Prediction
alu_branch	Dst, Src1, Op, Src2, Cond, TrueLabel, FalseLabel, Prediction
goto	Label
move	Dst, Src
alu	Dst, Src1, Op, Src2
fp	Dst, Src1, Op, Src2
load	Dst, Src, Off
load_atom	Dst, Atom
load_address	Dst, Addr, Type
store	Dst, Off, Src
comment	Text
call	Dst, Fun, Args, Continuation, Fail, Type
enter	Fun, Args, Continuation, Type
return	Vars
gctest	Words

6.4 Control flow instructions

6.4.1 label(Name, Info)

Gives this point in the code the name `Name` so that branches can jump to this point.

`mk_label(Name)` : Make a new label named `Name`

`mk_new_label()` : If gensym is initialized correctly this instruction generates a new label that is unique for the function being compiled.

`label_name(Instr)` : Returns the name of a label.

`is_label(Instr)` : True if the instruction is a label.

6.4.2 branch(Src1, RelOp, Src2, TrueLabel, FalseLabel, Prediction, Info)

Conditional branch. Compares the arguments `Src1` and `Src2` with regards to the relation operation `RelOp` (See Table 6.1. The arguments can be variables, immediates or registers. If the comparison succeeds the execution continues at `TrueLabel` otherwise the execution continues at `FalseLabel`.

→ Table 6.1

The `Prediction` can be used to indicate the predicted direction of the branch. The number is a float between 0.0 and 1.0 indicating the likelihood of the destination being the `TrueLabel`.

`mk_branch(Src1, RelOp, Src2, TrueLabel, FalseLabel)` : Makes a conditional branch with prediction 0.5.

`mk_branch(Src1, RelOp, Src2, TrueLabel, FalseLabel, Prediction)` : Makes a conditional branch with the given prediction.

`branch_src1(Instr)` : Returns `Src1`.

`branch_src2(Instr)` : Returns `Src2`.

`branch_cond(Instr)` : Returns `RelOp`.

`branch_true_label(Instr)` : Returns `TrueLabel`.

`branch_false_label(Instr)` : Returns `FalseLabel`.

`branch_pred(Instr)` : Returns `Prediction`.

`is_branch(Instr)` : Returns `true` if the instruction is a brach instruction, and `false` otherwise.

`branch_true_label_update(Instr,NewLabel)` : Replaces the destination of a successful comparison (`TrueLabel`) with `NewLabel`.

`branch_false_label_update(Instr,NewLabel)` : Replaces the destination of an unsuccessful comparison (`FalseLabel`) with `NewLabel`.

6.4.3 alu_branch(Dst, Src1, Op, Src2, Cond, TrueLabel, FalseLabel, Prediction, Info)

The arithmetic-logic branch calculates `Dst := Src1 Op Src2` to generate a condition code. Then depending on the condition `Cond` the execution continues at `TrueLabel` or `FalseLabel`. The operation (`Op`) can be one of the operations described in Table 6.2 . The condition can be one of de relational operator described in Table 6.1.The destination `Dst` have to be a register or a variable, the

→ Table 6.2

→ Table 6.1

RelOp	Src1 Type	Operation	Src2 Type
eq	sw	equal	sw
ne	sw	not equal	sw
gt	sw	greater	sw
gtu	uw	unsigned greater	uw
ge	sw	greater or equal	sw
geu	uw	unsigned greater or equal	uw
lt	sw	less	sw
ltu	uw	unsigned less	uw
le	sw	less or equal	sw
leu	sw	unsigned less or equal	sw
overflow	sw	overflow	sw
not_overflow	sw	not overflow	sw

Table 6.1: Valid relational operators in conditional branches.

sources can also be immediates. The prediction is a number between 0.0 and 1.0 indicating the probability that the condition holds.

`mk_alub(Dst, Src1, Op, Src2, Cond, True, False)` : Generate a arithmetic-logic branch with prediction 0.5.

`mk_alub(Dst, Src1, Op, Src2, Cond, True, False, P)` : Generate a arithmetic-logic branch with the prediction `P`.

`alub_dst(Instr)` : Return the register `Dst`.

`alub_src1(Instr)` : Return `Src1`.

`alub_op(Instr)` : Return the operation `Op`.

`alub_src2(Instr)` : Return `Src2`.

`alub_cond(Instr)` : Return the condition `Cond`.

`alub_true_label(Instr)` : Return `TrueLabel`.

`alub_true_label_update(Instr,NewLabel)` : Set the new success destination (`TrueLabel`) to `NewLabel`.

`alub_false_label(Instr)` : Return `FalseLabel`.

`alub_false_label_update(Instr,NewLabel)` : Set the new destination for an unsuccessful comparison (`FalseLabel`) to `NewLabel`.

`alub_pred(Instr)` : Return the prediction.

`alub_info(Instr)` : Return additional information.

`is_alub(Instr)` : Returns `true` if the instruction is an arithmetic-logic branch and `false` otherwise.

6.4.4 goto(Label, Info)

Continue execution at the label `Label`.

`mk_goto(Label)` : Create a goto instruction.

`goto_label(Instr)` : Return the destination label.

`is_goto(Instr)` : `true` if the instruction is a goto, `false` otherwise.

RelOp	Src1 Type	Operation	Src2 Type
add	sw	+	sw
sub	sw	-	sw
'or'	uw	or	uw
'and'	uw	and	uw
'xor'	uw	xor	uw
'xornot'	uw	xnor	uw
andnot	uw	and not	uw
sll	uw	shift left logical	uw
sllx	udw (64b)	shift left logical	udw (64b)
srl	uw	shift right logical	uw
srlx	udw (64b)	shift right logical	udw (64b)
sra	sw	shift right arithmetic	uw
srax	sdw (64b)	shift right arithmetic	udw (64b)

Table 6.2: Valid operators in alu operations and alu branches.

`goto_label_update(Instr, Label)` : Update the destination of a goto instruction.

6.5 Operations

6.5.1 move(Dst, Src, Info)

Move the immediate value, or the contents of the register/variable `Src` to the register/variable `Dst`.

`mk_move(Dst, Src)` : Create a move instruction.

`move_dst(Instr)` : Return the move destination `Dst`.

`move_src(Instr)` : Return the move source `Src`.

`move_src_update(Instr,NewSrc)` : Updates the source of the move.

`is_move(Instr)` : True if the instruction is a move instruction and false otherwise.

6.5.2 alu(Dst, Src1, Op, Src2, Info)

Calculates `Dst := Src1 Op Src2`. Where `Dst` is a register/variable and `Src1` and `Src2` are immediates, registers or variables. The operation `Op` can be one of the operations given in Table 6.2.

→ Table 6.2

`mk_alu(Dst, Src1, Op, Src2)` : Create an arithmetic-logic operation.

`alu_dst(Instr)` : Return `Dst`.

`alu_src1(Instr)` : Return `Src1`.

`alu_src1_update(Instr, NewSrc)` : Replaces `Src1` with `NewSrc`.

`alu_src2(Instr)` : Return `Srs2`.

`alu_src2_update(Instr, NewSrc)` : Replaces `Src2` with `NewSrc`.

`alu_op(Instr)` : Return `Op`.

`is_alu_op(Instr)` : `true` if the instruction is an arithmetic-logic operation.

6.5.3 *(fp(Dst, Src1, Op, Src2, Info))*

! → *This instruction is not fully supported yet...*

6.5.4 load(Dst, Src, Off, Info)

Loads an unsigned word from the address `Src+Off` into `Dst` (`Dst := [Src+Off]`).

`mk_load(Dst,Src,Info)` : Create a load instruction.

`load_dst(Instr)` : Return `Dst`.

`load_src(Instr)` : Return `Src`.

`load_offset(Instr)` : Return `Off`.

6.5.5 load_atom(Dst, Atom, Info)

Moves the atom index for the atom `Atom` into `Dst`. This is used to keep a symbolic representation of atoms.

`mk_load_atom(Dst, Atom)` : Creates a load-atom instruction.

`load_atom_dst(Instr)` : Return `Dst`.

`load_atom_atom(Instr)` : Return `Atom`.

6.5.6 load_address(Dst, Addr, Type, Info)

Loads the address of `Addr` into `Dst`. There are three types of addresses that can be loaded: addresses of labels (`Type = label`), addresses of functions (`Type = function`), or addresses of constants (`Type = constant`). If the address is a label then that label has to exist somewhere in the code that is linked together with the instruction. If the address is a function it can either be an atom that denotes the name and arity of a bif or it can be a MFA (`{Module, Function, Arity}`). If the type is a constant then the address should be the name of a constant that is stored in the constant table.

`mk_load_address(Dst, Addr, Type)` : Create a load-address instruction.

`load_address_dst(Instr)` : Return `Dst`.

`load_address_dst_update(Instr, NewDst)` : Updates the destination to `NewDst`.

`load_address_address(Instr)` : Return `Address`.

`load_address_type(Instr)` : Return `Type`.

`load_address_type_update(Instr, NewType)` : Updates the type to `NewType`.

6.5.7 store(Dst, Off, Src, Info)

Writes the immediate or the register/variable `Src` to the memory address `Dst+Off`, where `Dst` is a register or a variable and `Off` is an immediate or a register/variable. (`[Dst+Off] := Src`)

`mk_store(Dst, Off, Src)` : Create a new store instruction.

`store_dst(Instr)` : Return `Dst`.

`store_src(Instr)` : Return `Src`.

`store_offset(Instr)` : Return `Off`.

6.6 Comments

6.6.1 comment(Text, Info)

`mk_comment(Text)` : Creates a comment.

`comment_text(Instr)` : Returns the text in the comment.

`is_comment(Instr)` : True if the instruction is a comment.

6.7 Function application

6.7.1 call(Dst, Fun, Args, Type, Info)

Calls the function `Fun` (=`{Mod, Fun, Arity}`) with the list of arguments `Args`. The result of the call is stored in the registers in the list of destinations `Dst`. The call type is either `local` or `remote`.

`mk_call(Dst, Fun, Args, Type)` : Creates a call instruction.

`call_fun(Instr)` : Returns the MFA.

`call_args(Instr)` : Returns the list of arguments.

`call_dst(Instr)` : Returns the list of destinations.

`call_type(Instr)` : Returns the type of the call.

`call_args_update(Instr, NewArgs)` : Replaces the list of arguments with `NewArgs`.

6.7.2 enter(Fun, Args, Type, Info)

The `enter` instruction does a tail-call to the `Fun` (=`{Mod, Fun, Arity}`) with the arguments in the list `Arg`. The type is either `local` or `remote`. There is no return value destination since this call will not return.

`mk_enter(Fun, Args, Type)` : Creates an enter instruction.

`enter_fun(Instr)` : Returns the MFA.

`enter_args(Instr)` : Returns the list of arguments.

`enter_type(Instr)` : Returns the type of the call.

`enter_args_update(Instr, NewArgs)` : Replaces the list of arguments with `NewArgs`.

6.7.3 return(Vars, Info)

Returns the list of variables `Vars` to the calling function.

`mk_return(Vars)` : Create a return instruction.

`return_vars(Instr)` : Return the list of return variables.

6.7.4 gctest(Words, Info)

Ensures that there is space for `Words` words on the heap.

`mk_gctest(Words)` : Creates a gctest instruction.

`gctest_words(Instr)` : Returns the number of words.

`gctest_info(Instr)` :

`gctest_info_update(Instr, NewInfo)` :

7 SPARC

mk_sparc(Fun, Code, VarRange, LabelRange)
sparc_fun(Sparc)
sparc_code(Sparc)
sparc_code_update(Sparc, Code)
sparc_var_range(Sparc)
sparc_var_range_update(Sparc, NewVarRange)
sparc_label_range(Sparc)
sparc_label_range_update(Sparc, NewLabelRange)
sparc_size(Sparc)

7.1 Control flow instructions

7.1.1 label(Name, Info)

label_create(Name, Info)
label_create_new()
label_name(Instr)
label_name_update(Instr, NewName)

7.1.2 branch(Cond, TrueLabel, FalseLabel, Prediction, Annul, Info)

b_create(Cond, TrueLabel, FalseLabel, Prediction, Annul, Info)
b_annul(Instr)
b_annul_update(Instr, NewAnnul)
b_cond(Instr)
b_cond_update(Instr, Cond)
b_label(Instr)
b_label_update(Instr, Label)
b_true_label(Instr)
b_true_label_update(Instr, NewLabel)
b_false_label(Instr)
b_false_label_update(Instr, NewLabel)
b_pred(Instr)
b_pred_update(Instr, NewPred)
b_taken(Instr)

7.1.3 branch_on_reg(Reg, RegCond, TrueLabel, FalseLabel, Prediction, Annul, Info)

br_create(Reg,RegCond,TrueLabel,FalseLabel,Prediction,Annul,Info)
br_annul(Instr)
br_annul_update(Instr, NewAnnul)
br_label(Instr)
br_label_update(Instr, NewLabel)
br_true_label(Instr)
br_true_label_update(Instr, NewLabel)
br_false_label(Instr)

br_false_label_update(Instr, NewLabel)
br_pred(Instr)
br_pred_update(Instr, NewPrediction)
br_taken(Instr)
br_reg(Instr)
br_reg_update(Instr, NewReg)
br_regcond(Instr)
br_regcond_update(Instr, NewCond)

7.1.4 goto(Label, Info)

goto_create(Label, Info)
goto_label(Instr)
goto_label_update(Instr, Label)

7.2 Operations

7.2.1 alu_cc(Dest, Src1, Op, Src2, Info)

alu_cc_create(Dest, Src1, Op, Src2, Info)
alu_cc_dest(Instr)
alu_cc_dest_update(Instr, Dest)
alu_cc_operator(Instr)
alu_cc_operator_update(Instr, NewOp)
alu_cc_src1(Instr)
alu_cc_src1_update(Instr, NewSrc1)
alu_cc_src2(Instr)
alu_cc_src2_update(Instr, NewSrc2)

7.2.2 alu(Dest, Src1, Op, Src2, Info)

alu_create(Dest, Src1, Op, Src2, Info)
alu_dest(Instr)
alu_dest_update(Instr, NewDest)
alu_operator(Instr)
alu_operator_update(Instr, NewOp)
alu_src1(Instr)
alu_src1_update(Instr, NewSrc1)
alu_src2(Instr)
alu_src2_update(Instr, NewSrc2)

7.2.3 load_atom(Dest, Atom, Info)

load_atom_create(Dest, Atom, Info)
load_atom_dest(Instr)
load_atom_dest_update(Instr, NewDest)
load_atom_atom(Instr)
load_address(Dst, Addr, Type, Info)
load_address_create(Dst, Addr, Type, Info)
load_address_dest(Instr)
load_address_address(Instr)
load_address_type(Instr)

load_address_type_update(Instr, NewType)

7.2.4 load(Dest, Type, Source, Off, Info)

load_create(Dest, Type, Source, Off, Info)
load_dest(Instr)
load_dest_update(Instr, Dest)
load_off(Instr)
load_off_update(Instr, NewOff)
load_src(Instr)
load_src_update(Instr, NewScr)
load_type(Instr)
load_type_update(Instr, NewType)

7.2.5 move(Dest, Source, Info)

move_create(Dest, Source, Info)
move_dest(Instr)
move_dest_update(Instr, NewDest)
move_src(Instr)
move_src_update(Instr, NewSrc)

7.2.6 cmove_cc(Dest, Source, Cond, Info)

cmov_cc_cond(Instr)
cmov_cc_cond_update(Instr, NewCond)
cmov_cc_create(Dest, Source, Cond, Info)
cmov_cc_dest(Instr)
cmov_cc_dest_update(Instr, NewDest)
cmov_cc_src(Instr)
cmov_cc_src_update(Instr, NewSrc)

7.2.7 cmov_on_reg(Dest,Source,Reg,RegCond,Info)

cmov_r_create(Dest,Source,Reg,RegCond,Info)
cmov_r_dest(Instr)
cmov_r_dest_update(Instr, NewDest)
cmov_r_reg(Instr)
cmov_r_reg_update(Instr, NewReg)
cmov_r_regcond(Instr)
cmov_r_regcond_update(Instr, NewRegCond)
cmov_r_src(Instr)
cmov_r_src_update(Instr, Src)

7.2.8 store(Target, Off, Type, Source, Info)

store_create(Target, Off, Type, Source, Info)
store_dest(Instr)
store_dest_update(Instr, NewDest)
store_off(Instr)
store_off_update(Instr, NewOff)
store_src(Instr)

store_src_update(Instr, NewSrc)
store_type(Instr)
store_type_update(Instr, NewType)

7.2.9 sethi(Dest,Const,Info)

sethi_const(Instr)
sethi_const_update(Instr, NewConst)
sethi_create(Dest,Const,Info)
sethi_dest(Instr)
sethi_dest_update(Instr, NewDest)

7.3 Function application

7.3.1 jmp_link( )

jmp_link_create/5,
jmp_link_off(Instr)
jmp_link_off_update/2,
jmp_link_target(Instr)
jmp_link_target_update/2,
jmp_link_args(Instr)
jmp_link_args_update/2,
jmp_link_link(Instr)
jmp_link_link_update/2,

7.3.2 jmp( )

jmp_create/4,
jmp_off(Instr)
jmp_off_update/2,
jmp_target(Instr)
jmp_target_update/2,
jmp_args(Instr)
jmp_args_update/2,

7.3.3 call_link( )

call_link_create/4,
call_link_target(Instr)
call_link_target_update/2,
call_link_link(Instr)
call_link_link_update/2,
call_link_args(Instr)
call_link_args_update/2,

7.4 Stack and memory

7.4.1 align( )

align_alignment/1,
align_alignment_update/2,

align_create/2,

7.5 Comments and NOP

7.5.1 comment(Comment,Info)

comment_create/2,
comment_text(Instr)
comment_text_update/2,

7.5.2 nop(Info)

nop_create/1,

7.6 Auxillary functions

ext_cc/1,
fp_cc/1,
fp_op/1,
imm16/1,
imm19/1,
info(Instr)
info_update/2,
is_align/1,
is_alu(Instr)
is_alu_cc(Instr)
is_any_alu(Instr)
is_any_branch(Instr)
is_any_cmov(Instr)
is_any_memop(Instr)
is_b(Instr)
is_br(Instr)
is_call_link(Instr)
is_cmov_cc(Instr)
is_cmov_r(Instr)
is_comment(Instr)
is_jmp(Instr)
is_jmp_link(Instr)
is_label(Instr)
is_load(Instr)
is_move(Instr)
is_nop(Instr)
is_sethi(Instr)
is_store(Instr)
redirect_jmp/3,
cc_negate/1,
uses/1,
all_uses/1,
imm_uses/1,
defines/1,
def_use/1,
subst/2,
subst_uses/2,

subst_defines/2,
verify/2,
pp/1,
pp/2,
pp_instr/1
mk_reg/1,
mk_new_reg/0,
is_reg/1,
reg_nr/1,
mk_fpreg/1,
is_fpreg/1,
fpreg_nr/1,
mk_imm/1,
is_imm/1,
imm_value/1

Index

address_to_fun, *see* hipe_bifs
alloc_code, *see* hipe_bifs
alloc_constant, *see* hipe_bifs
annot, *see* icode_bb, *see* rtl_bb, *see* sparc_bb
array, *see* hipe_bifs
array_length, *see* hipe_bifs
array_sub, *see* hipe_bifs
atom_to_word, *see* hipe_bifs

Basic Blocks, 17
bb, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
bb_update, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
bif_address, *see* hipe_bifs
binary operators, 23
butlast, *see* icode_bb, *see* rtl_bb, *see* sparc_bb

c, *see* hipe
call_args, *see* icode
call_count_clear, *see* hipe_bifs
call_count_get, *see* hipe_bifs
call_count_off, *see* hipe_bifs
call_count_on, *see* hipe_bifs
call_dst, *see* icode
call_fun, *see* icode
call_type, *see* icode
catch_index_to_word, *see* hipe_bifs
catch_table_insert, *see* hipe_bifs
catch_table_nil, *see* hipe_bifs
catch_table_remove, *see* hipe_bifs
CFG, 17
closure, 5
code, *see* icode_bb, *see* rtl_bb, *see* sparc_bb
code_update, *see* icode_bb, *see* rtl_bb, *see* sparc_bb
comment_text, *see* icode
cond, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
const_value, *see* icode
control flow graph, 17
copy_term, *see* hipe_bifs

defines, *see* icode

emu_stub, *see* hipe_bifs
enter_args, *see* icode
enter_fun, *see* icode
enter_type, *see* icode

fail_entrypoints, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
fail_reason, *see* icode
fail_type, *see* icode
fallthrough, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
fun_to_address, *see* hipe

get_funinfo, *see* hipe_bifs
goto_label, *see* icode

hipe
  c/1, 3
  c/2, 3
  fun_to_address/1, 14
  version/0, 3
hipe_bifs
  address_to_fun/1, 13
  alloc_code/1, 13
  alloc_constant/1, 13
  array/2, 13
  array_length/1, 13
  array_sub/2, 13
  atom_to_word/1, 13
  bif_address/1, 13
  call_count_clear/1, 15
  call_count_get/1, 15
  call_count_off/1, 15
  call_count_on/1, 15
  catch_index_to_word/1, 13
  catch_table_insert/2, 14
  catch_table_nil/0, 14
  catch_table_remove/3, 14
  copy_term/3, 14
  emu_stub/3, 14
  get_funinfo/1, 14
  make_fe/3, 14
  primop_address/1, 14
  read_s32/1, 14
  read_u32/1, 14
  read_u8/1, 14
  ref/1, 14
  ref_get/1, 14
  ref_set/2, 14
  set_funinfo/1, 14
  set_native_address/3, 15
  show_estack/1, 15
  show_heap/1, 15
  show_nstack /1, 15
  show_p_struct/1, 16
  show_term/1, 16
  term_size/1, 15
  write_s32/2, 15
  write_u32/2, 15
  write_u8/2, 15

icode

call_args/1, 24
call_dst/1, 24
call_fun/1, 24
call_type/1, 24
comment_text/1, 26
const_value/1, 26
defines/1, 21
enter_args/1, 24
enter_fun/1, 24
enter_type/1, 24
fail_reason/1, 25
fail_type/1, 25
goto_label/1, 22
icode_code/1, 20
icode_fun/1, 20
icode_info/1, 20
icode_info_add/2, 20
icode_info_update/2, 20
icode_label_range/1, 20
icode_params/1, 20
icode_var_range/1, 20
if_args/1, 23
if_false_label/1, 23
if_op/1, 23
if_pred/1, 23
if_true_label/1, 23
info/1, 20
info_add/2, 20
info_update/2, 20
is_branch/1, 21
is_call/1, 24
is_comment/1, 26
is_const/1, 26
is_enter/1, 24
is_fail/1, 25
is_goto/1, 22
is_if/1, 23
is_label/1, 22
is_leaf/1, 20
is_mov/1, 24
is_pure/1, 21
is_pushcatch/1, 25
is_remove_catch/1, 25
is_restore_catch/1, 25
is_return/1, 24
is_type/1, 22
is_uncond/1, 21
is_var/1, 26
label_name/1, 22
mk_call/7, 24
mk_comment/1, 26
mk_const/1, 26
mk_enter/4, 24
mk_fail/2, 25
mk_gctest/0, 10
mk_goto/1, 22
mk_icode/5, 20
mk_if/4, 23
mk_if/5, 23
mk_label/1, 22
mk_mov/2, 23
mk_movs/2, 23
mk_new_label/0, 22
mk_new_var/0, 26
mk_pushcatch/1, 25
mk_redtest/0, 10
mk_remove_catch/2, 25
mk_restore_catch/2, 25
mk_return/1, 24
mk_type/4, 22
mk_type/5, 22
mk_var/1, 26
mov_dst/1, 23
mov_src/1, 23
mov_src_update/2, 24
pp/1, 20
pp_exit/1, 20
pp_instr/1, 21
pushcatch_label/1, 25
redirect_jmp/3, 21
remove_catch_id/1, 25
remove_catch_type/1, 25
remove_empty_bbs/(Icode), 20
restore_catch_dst/1, 25
restore_catch_id/1, 25
return_vars/1, 24
subst/2, 21
subst_defines/2, 21
subst_uses/2, 21
successors/1, 21
type/1, 20
type_false_label/1, 22
type_pred/1, 22
type_true_label/1, 22
type_type/1, 22
type_var/1, 22
uses/1, 21
var_name/1, 26
icode instructions, 21
icode_bb
annot/1, 19
butlast/1, 19
code/1, 18
code_update/2, 19
last/1, 19
mk_bb/1, 18
mk_bb/2, 18
icode_cfg
bb/2, 17
bb_update/3, 17
cond/2, 18
fail_entrypoints/1, 18

fallthrough/2, 18
init/1, 17
label_range/1, 18
labels/1, 18
linearize/1, 18
params/1, 18
postorder/1, 18
pred/2, 17
pred_map/1, 17
reverse_postorder/1, 18
start/1, 18
succ/2, 17
succ_map/1, 17
var_range/1, 18
icode_code, *see* icode
icode_fun, *see* icode
icode_info, *see* icode
icode_info_add, *see* icode
icode_info_update, *see* icode
icode_label_range, *see* icode
icode_params, *see* icode
icode_var_range, *see* icode
if_args, *see* icode
if_false_label, *see* icode
if_op, *see* icode
if_pred, *see* icode
if_true_label, *see* icode
info, *see* icode
info_add, *see* icode
info_update, *see* icode
init, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
is_branch, *see* icode
is_call, *see* icode
is_comment, *see* icode
is_const, *see* icode
is_enter, *see* icode
is_fail, *see* icode
is_goto, *see* icode
is_if, *see* icode
is_label, *see* icode
is_leaf, *see* icode
is_mov, *see* icode
is_pure, *see* icode
is_pushcatch, *see* icode
is_remove_catch, *see* icode
is_restore_catch, *see* icode
is_return, *see* icode
is_type, *see* icode
is_uncond, *see* icode
is_var, *see* icode

label_name, *see* icode
label_range, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
labels, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
last, *see* icode_bb, *see* rtl_bb, *see* sparc_bb
linear, 17
linearize, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg

make_fe, *see* hipe_bifs
mk_bb, *see* icode_bb, *see* rtl_bb, *see* sparc_bb
mk_call, *see* icode
mk_comment, *see* icode
mk_const, *see* icode
mk_enter, *see* icode
mk_fail, *see* icode
mk_gctest, *see* icode
mk_goto, *see* icode
mk_icode, *see* icode
mk_if, *see* icode
mk_label, *see* icode
mk_mov, *see* icode
mk_movs, *see* icode
mk_new_label, *see* icode
mk_new_var, *see* icode
mk_pushcatch, *see* icode
mk_redtest, *see* icode
mk_remove_catch, *see* icode
mk_restore_catch, *see* icode
mk_return, *see* icode
mk_rtl, *see* rtl
mk_type, *see* icode
mk_var, *see* icode
mov_dst, *see* icode
mov_src, *see* icode
mov_src_update, *see* icode

params, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
postorder, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
pp, *see* icode, *see* rtl
pp_exit, *see* icode
pp_instr, *see* icode
pred, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
pred_map, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
primop_address, *see* hipe_bifs
pushcatch_label, *see* icode

read_s32, *see* hipe_bifs
read_u32, *see* hipe_bifs
read_u8, *see* hipe_bifs
redirect_jmp, *see* icode
ref, *see* hipe_bifs
ref_get, *see* hipe_bifs
ref_set, *see* hipe_bifs
remove_catch_id, *see* icode
remove_catch_type, *see* icode
remove_empty_bbs, *see* icode, *see* rtl
restore_catch_dst, *see* icode
restore_catch_id, *see* icode
return_vars, *see* icode
reverse_postorder, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
rtl

mk_rtl/5, 27
pp/1, 27
pp/2, 27
remove_empty_bbs/1, 27
rtl_code/1, 27
rtl_code_update/2, 27
rtl_fun/1, 27
rtl_info/1, 27
rtl_info_update/2, 27
rtl_label_range/1, 27
rtl_label_range_update/2, 27
rtl_params/1, 27
rtl_var_range/1, 27
rtl_var_range_update/2, 27
RTL instructions, 28
rtl_bb
annot/1, 19
butlast/1, 19
code/1, 18
code_update/2, 19
last/1, 19
mk_bb/1, 18
mk_bb/2, 18
rtl_cfg
bb/2, 17
bb_update/3, 17
cond/2, 18
fail_entrypoints/1, 18
fallthrough/2, 18
init/1, 17
label_range/1, 18
labels/1, 18
linearize/1, 18
params/1, 18
postorder/1, 18
pred/2, 17
pred_map/1, 17
reverse_postorder/1, 18
start/1, 18
succ/2, 17
succ_map/1, 17
var_range/1, 18
rtl_code, *see* rtl
rtl_code_update, *see* rtl
rtl_fun, *see* rtl
rtl_info, *see* rtl
rtl_info_update, *see* rtl
rtl_label_range, *see* rtl
rtl_label_range_update, *see* rtl
rtl_params, *see* rtl
rtl_var_range, *see* rtl
rtl_var_range_update, *see* rtl

set_funinfo, *see* hipe_bifs
set_native_address, *see* hipe_bifs
show_estack, *see* hipe_bifs
show_heap, *see* hipe_bifs
show_nstack , *see* hipe_bifs
show_p_struct, *see* hipe_bifs
show_term, *see* hipe_bifs
sparc_bb
annot/1, 19
butlast/1, 19
code/1, 18
code_update/2, 19
last/1, 19
mk_bb/1, 18
mk_bb/2, 18
sparc_cfg
bb/2, 17
bb_update/3, 17
cond/2, 18
fail_entrypoints/1, 18
fallthrough/2, 18
init/1, 17
label_range/1, 18
labels/1, 18
linearize/1, 18
params/1, 18
postorder/1, 18
pred/2, 17
pred_map/1, 17
reverse_postorder/1, 18
start/1, 18
succ/2, 17
succ_map/1, 17
var_range/1, 18
start, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
subst, *see* icode
subst_defines, *see* icode
subst_uses, *see* icode
succ, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
succ_map, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
successors, *see* icode

term_size, *see* hipe_bifs
type, *see* icode
type_false_label, *see* icode
type_pred, *see* icode
type_true_label, *see* icode
type_type, *see* icode
type_var, *see* icode

uses, *see* icode

var_name, *see* icode
var_range, *see* icode_cfg, *see* rtl_cfg, *see* sparc_cfg
version, *see* hipe

write_s32, *see* hipe_bifs
write_u32, *see* hipe_bifs
write_u8, *see* hipe_bifs