Just-In-Time Compiler Library

1 Introduction and rationale for libjit

Just-In-Time compilers are becoming increasingly popular for executing dynamic languages like Perl and Python and for semi-dynamic languages like Java and C#. Studies have shown that JIT techniques can get close to, and sometimes exceed, the performance of statically-compiled native code.

However, there is a problem with current JIT approaches. In almost every case, the JIT is specific to the object model, runtime support library, garbage collector, or bytecode peculiarities of a particular system. This inevitably leads to duplication of effort, where all of the good JIT work that has gone into one virtual machine cannot be reused in another.

JIT's are not only useful for implementing languages. They can also be used in other programming fields. Graphical applications can achieve greater performance if they can compile a special-purpose rendering routine on the fly, customized to the rendering task at hand, rather than using static routines. Needless to say, such applications have no need for object models, garbage collectors, or huge runtime class libraries.

Most of the work on a JIT is concerned with arithmetic, numeric type conversion, memory loads/stores, looping, performing data flow analysis, assigning registers, and generating the executable machine code. Only a very small proportion of the work is concerned with language specifics.

The goal of the `libjit` project is to provide an extensive set of routines that takes care of the bulk of the JIT process, without tying the programmer down with language specifics. Where we provide support for common object models, we do so strictly in add-on libraries, not as part of the core code.

Unlike other systems such as the JVM, .NET, and Parrot, `libjit` is not a virtual machine in its own right. It is the foundation upon which a number of different virtual machines, dynamic scripting languages, or customized rendering routines can be built.

The LLVM project (`http://www.llvm.org/`) has some similar characteristics to `libjit` in that its intermediate format is generic across front-end languages. It is written in C++ and provides a large set of compiler development and optimization components; much larger than `libjit` itself provides. According to its author, Chris Lattner, a subset of its capabilities can be used to build JIT's.

Libjit should free developers to think about the design of their front ends, and not get bogged down in the details of code execution. Meanwhile, experts in the design and implementation of JIT's can concentrate on solving code execution problems, instead of front end support issues.

This document describes how to use the library in application programs. We start with a list of features and some simple tutorials. Finally, we provide a complete reference guide for all of the API functions in `libjit`, broken down by function category.

1.1 Obtaining libjit

The latest version of `libjit` can be obtained from Southern Storm Software, Pty Ltd's Web site:

`http://www.southern-storm.com.au/libjit.html`

1.2 Further reading

While it isn't strictly necessary to know about compiler internals to use `libjit`, you can make more effective use of the library if you do. We recommend the "Dragon Book" as an excellent resource on compiler internals, particularly the sections on code generation and optimization:

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

IBM, Intel, and others have done a lot of research into JIT implementation techniques over the years. If you are interested in working on the internals of `libjit`, then you may want to make yourself familiar with the relevant literature (this is by no means a complete list):

> IBM's Jikes RVM (Research Virtual Machine),
> http://www-124.ibm.com/developerworks/oss/jikesrvm/.
>
> Intel's ORP (Open Runtime Platform),
> http://orp.sourceforge.net/.

2 Features of libjit

- The primary interface is in C, for maximal reusability. Class interfaces are available for programmers who prefer C++.
- Designed for portability to all major 32-bit and 64-bit platforms.
- Simple three-address API for library users, but opaque enough that other representations can be used inside the library in future without affecting existing users.
- Up-front or on-demand compilation of any function.
- In-built support to re-compile functions with greater optimization, automatically redirecting previous callers to the new version.
- Fallback interpreter for running code on platforms that don't have a native code generator yet. This reduces the need for programmers to write their own interpreters for such platforms.
- Arithmetic, bitwise, conversion, and comparison operators for 8-bit, 16-bit, 32-bit, or 64-bit integer types; and 32-bit, 64-bit, or longer floating point types. Includes overflow detecting arithmetic for integer types.
- Large set of mathematical and trigonometric operations (sqrt, sin, cos, min, abs, etc) for inlining floating-point library functions.
- Simplified type layout and exception handling mechanisms, upon which a variety of different object models can be built.
- Support for nested functions, able to access their parent's local variables (for implementing Pascal-style languages).

3 Tutorials in using libjit

In this chapter, we describe how to use `libjit` with a number of short tutorial exercises. Full source for these tutorials can be found in the `tutorial` directory of the `libjit` source tree.

For simplicity, we will ignore errors such as out of memory conditions, but a real program would be expected to handle such errors.

3.1 Tutorial 1 - mul_add

In the first tutorial, we will build and compile the following function (the source code can be found in `tutorial/t1.c`):

```
int mul_add(int x, int y, int z)
{
    return x * y + z;
}
```

To use the JIT, we first include the `<jit/jit.h>` file:

```
#include <jit/jit.h>
```

All of the header files are placed into the `jit` sub-directory, to separate them out from regular system headers. When `libjit` is installed, you will typically find these headers in `/usr/local/include/jit` or `/usr/include/jit`, depending upon how your system is configured. You should also link with the `-ljit` option.

Every program that uses `libjit` needs to call `jit_context_create`:

```
jit_context_t context;
...
context = jit_context_create();
```

Almost everything that is done with `libjit` is done relative to a context. In particular, a context holds all of the functions that you have built and compiled.

You can have multiple contexts at any one time, but normally you will only need one. Multiple contexts may be useful if you wish to run multiple virtual machines side by side in the same process, without them interfering with each other.

Whenever we are constructing a function, we need to lock down the context to prevent multiple threads from using the builder at a time:

```
jit_context_build_start(context);
```

The next step is to construct the function object that will represent our `mul_add` function:

```
jit_function_t function;
...
function = jit_function_create(context, signature);
```

The `signature` is a `jit_type_t` object that describes the function's parameters and return value. This tells `libjit` how to generate the proper calling conventions for the function:

```
jit_type_t params[3];
jit_type_t signature;
...
```

```
params[0] = jit_type_int;
params[1] = jit_type_int;
params[2] = jit_type_int;
signature = jit_type_create_signature
    (jit_abi_cdecl, jit_type_int, params, 3, 1);
```

This declares a function that takes three parameters of type `int` and returns a result of type `int`. We've requested that the function use the `cdecl` application binary interface (ABI), which indicates normal C calling conventions. See Chapter 6 [Types], page 20, for more information on signature types.

Now that we have a function object, we need to construct the instructions in its body. First, we obtain references to each of the function's parameter values:

```
jit_value_t x, y, z;
...
x = jit_value_get_param(function, 0);
y = jit_value_get_param(function, 1);
z = jit_value_get_param(function, 2);
```

Values are one of the two cornerstones of the `libjit` process. Values represent parameters, local variables, and intermediate temporary results. Once we have the parameters, we compute the result of `x * y + z` as follows:

```
jit_value_t temp1, temp2;
...
temp1 = jit_insn_mul(function, x, y);
temp2 = jit_insn_add(function, temp1, z);
```

This demonstrates the other cornerstone of the `libjit` process: instructions. Each of these instructions takes two values as arguments and returns a new temporary value with the result.

Students of compiler design will notice that the above statements look very suspiciously like the "three address statements" that are described in compiler textbooks. And that is indeed what they are internally within `libjit`.

If you don't know what three address statements are, then don't worry. The library hides most of the details from you. All you need to do is break your code up into simple operation steps (addition, multiplication, negation, copy, etc). Then perform the steps one at a time, using the temporary values in subsequent steps. See Chapter 8 [Instructions], page 34, for a complete list of all instructions that are supported by `libjit`.

Now that we have computed the desired result, we return it to the caller using `jit_insn_return`:

```
jit_insn_return(function, temp2);
```

We have completed the process of building the function body. Now we compile it into its executable form:

```
jit_function_compile(function);
jit_context_build_end(context);
```

As a side-effect, this will discard all of the memory associated with the values and instructions that we constructed while building the function. They are no longer required, because we now have the executable form that we require.

We also unlock the context, because it is now safe for other threads to access the function building process.

Up until this point, we haven't executed the `mul_add` function. All we have done is build and compile it, ready for execution. To execute it, we call `jit_function_apply`:

```
jit_int arg1, arg2, arg3;
void *args[3];
jit_int result;
...
arg1 = 3;
arg2 = 5;
arg3 = 2;
args[0] = &arg1;
args[1] = &arg2;
args[2] = &arg3;
jit_function_apply(function, args, &result);
printf("mul_add(3, 5, 2) = %d\n", (int)result);
```

We pass an array of pointers to `jit_function_apply`, each one pointing to the corresponding argument value. This gives us a very general purpose mechanism for calling any function that may be built and compiled using `libjit`. If all went well, the program should print the following:

```
mul_add(3, 5, 2) = 17
```

You will notice that we used `jit_int` as the type of the arguments, not `int`. The `jit_int` type is guaranteed to be 32 bits in size on all platforms, whereas `int` varies in size from platform to platform. Since we wanted our function to work the same everywhere, we used a type with a predictable size.

If you really wanted the system `int` type, you would use `jit_type_sys_int` instead of `jit_type_int` when you created the function's signature. The `jit_type_sys_int` type is guaranteed to match the local system's `int` precision.

Finally, we clean up the context and all of the memory that was used:

```
jit_context_destroy(context);
```

3.2 Tutorial 2 - gcd

In this second tutorial, we implement the subtracting Euclidean Greatest Common Divisor (GCD) algorithm over positive integers. This tutorial demonstrates how to handle conditional branching and function calls. In C, the code for the `gcd` function is as follows:

```
unsigned int gcd(unsigned int x, unsigned int y)
{
    if(x == y)
    {
        return x;
    }
    else if(x < y)
    {
        return gcd(x, y - x);
```

```
    }
    else
    {
        return gcd(x - y, y);
    }
}
```

The source code for this tutorial can be found in `tutorial/t2.c`. Many of the details are similar to the previous tutorial. We omit those details here and concentrate on how to build the function body. See Section 3.1 [Tutorial 1], page 3, for more information.

We start by checking the condition `x == y`:

```
jit_value_t x, y, temp1;
...
x = jit_value_get_param(function, 0);
y = jit_value_get_param(function, 1);
temp1 = jit_insn_eq(function, x, y);
```

This is very similar to our previous tutorial, except that we are using the `eq` operator this time. If the condition is not true, we want to skip the `return` statement. We achieve this with the `jit_insn_branch_if_not` instruction:

```
jit_label_t label1 = jit_label_undefined;
...
jit_insn_branch_if_not(function, temp1, &label1);
```

The label must be initialized to `jit_label_undefined`. It will be updated by `jit_insn_branch_if_not` to refer to a future position in the code that we haven't seen yet.

If the condition is true, then execution falls through to the next instruction where we return `x` to the caller:

```
jit_insn_return(function, x);
```

If the condition was not true, then we branched to `label1` above. We fix the location of the label using `jit_insn_label`:

```
jit_insn_label(function, &label1);
```

We use similar code to check the condition `x < y`, and branch to `label2` if it is not true:

```
jit_value_t temp2;
jit_label_t label2 = jit_label_undefined;
...
temp2 = jit_insn_lt(function, x, y);
jit_insn_branch_if_not(function, temp2, &label2);
```

At this point, we need to call the `gcd` function with the arguments `x` and `y - x`. The code for this is fairly straight-forward. The `jit_insn_call` instruction calls the function listed in its third argument. In this case, we are calling ourselves recursively:

```
jit_value_t temp_args[2];
jit_value_t temp3;
...
temp_args[0] = x;
temp_args[1] = jit_insn_sub(function, y, x);
temp3 = jit_insn_call
```

```
    (function, "gcd", function, 0, temp_args, 2, 0);
jit_insn_return(function, temp3);
```

The string `"gcd"` in the second argument is for diagnostic purposes only. It can be helpful when debugging, but the `libjit` library otherwise makes no use of it. You can set it to NULL if you wish.

In general, `libjit` does not maintain mappings from names to `jit_function_t` objects. It is assumed that the front end will take care of that, using whatever naming scheme is appropriate to its needs.

The final part of the `gcd` function is similar to the previous one:

```
jit_value_t temp4;
...
jit_insn_label(function, &label2);
temp_args[0] = jit_insn_sub(function, x, y);
temp_args[1] = y;
temp4 = jit_insn_call
    (function, "gcd", function, 0, temp_args, 2, 0);
jit_insn_return(function, temp4);
```

We can now compile the function and execute it in the usual manner.

3.3 Tutorial 3 - compiling on-demand

In the previous tutorials, we compiled everything that we needed at startup time, and then entered the execution phase. The real power of a JIT becomes apparent when you use it to compile functions only as they are called. You can thus avoid compiling functions that are never called in a given program run, saving memory and startup time.

We demonstrate how to do on-demand compilation by rewriting Tutorial 1. The source code for the modified version is in `tutorial/t3.c`.

When the `mul_add` function is created, we don't create its function body or call `jit_function_compile`. We instead provide a C function called `compile_mul_add` that performs on-demand compilation:

```
jit_function_t function;
...
function = jit_function_create(context, signature);
jit_function_set_on_demand_compiler(function, compile_mul_add);
```

We can now call this function with `jit_function_apply`, and the system will automatically call `compile_mul_add` for us if the function hasn't been built yet. The contents of `compile_mul_add` are fairly obvious:

```
int compile_mul_add(jit_function_t function)
{
    jit_value_t x, y, z;
    jit_value_t temp1, temp2;

    x = jit_value_get_param(function, 0);
    y = jit_value_get_param(function, 1);
    z = jit_value_get_param(function, 2);
```

```
    temp1 = jit_insn_mul(function, x, y);
    temp2 = jit_insn_add(function, temp1, z);

    jit_insn_return(function, temp2);
    return 1;
}
```

When the on-demand compiler returns, `libjit` will call `jit_function_compile` and then jump to the newly compiled code. Upon the second and subsequent calls to the function, `libjit` will bypass the on-demand compiler and call the compiled code directly. Note that in case of on-demand compilation `libjit` automatically locks and unlocks the corresponding context with `jit_context_build_start` and `jit_context_build_end` calls.

Sometimes you may wish to force a commonly used function to be recompiled, so that you can apply additional optimization. To do this, you must set the "recompilable" flag just after the function is first created:

```
jit_function_t function;
...
function = jit_function_create(context, signature);
jit_function_set_recompilable(function);
jit_function_set_on_demand_compiler(function, compile_mul_add);
```

Once the function is compiled (either on-demand or up-front) its intermediate representation built by `libjit` is discarded. To force the function to be recompiled you need to build it again and call `jit_function_compile` after that. As always when the function is built and compiled manually it is necessary to take care of context locking:

```
jit_context_build_start(context);
jit_function_get_on_demand_compiler(function)(function);
jit_function_compile(function);
jit_context_build_end(context);
```

After this, any existing references to the function will be redirected to the new version. However, if some thread is currently executing the previous version, then it will keep doing so until the previous version exits. Only after that will subsequent calls go to the new version.

In this tutorial, we use the same on-demand compiler when we recompile `mul_add`. In a real program, you would probably call `jit_function_set_on_demand_compiler` to set a new on-demand compiler that performs greater levels of optimization.

If you no longer intend to recompile the function, you should call `jit_function_clear_recompilable` so that `libjit` can manage the function more efficiently from then on.

The exact conditions under which a function should be recompiled are not specified by `libjit`. It may be because the function has been called several times and has reached some threshold. Or it may be because some other function that it calls has become a candidate for inlining. It is up to the front end to decide when recompilation is warranted, usually based on language-specific heuristics.

3.4 Tutorial 4 - mul_add, C++ version

While `libjit` can be easily accessed from C++ programs using the C API's, you may instead wish to use an API that better reflects the C++ programming paradigm. We demonstrate how to do this by rewriting Tutorial 3 using the `libjitplus` library.

To use the `libjitplus` library, we first include the `<jit/jit-plus.h>` file:

```
#include <jit/jit-plus.h>
```

This file incorporates all of the definitions from `<jit/jit.h>`, so you have full access to the underlying C API if you need it.

This time, instead of building the `mul_add` function with `jit_function_create` and friends, we define a class to represent it:

```
class mul_add_function : public jit_function
{
public:
    mul_add_function(jit_context& context) : jit_function(context)
    {
        create();
        set_recompilable();
    }

    virtual void build();

protected:
    virtual jit_type_t create_signature();
};
```

Where we used `jit_function_t` and `jit_context_t` before, we now use the C++ `jit_function` and `jit_context` classes.

In our constructor, we attach ourselves to the context and then call the `create()` method. This is in turn will call our overridden virtual method `create_signature()` to obtain the signature:

```
jit_type_t mul_add_function::create_signature()
{
    // Return type, followed by three parameters,
    // terminated with "end_params".
    return signature_helper
        (jit_type_int, jit_type_int, jit_type_int,
         jit_type_int, end_params);
}
```

The `signature_helper()` method is provided for your convenience, to help with building function signatures. You can create your own signature manually using `jit_type_create_signature` if you wish.

The final thing we do in the constructor is call `set_recompilable()` to mark the `mul_add` function as recompilable, just as we did in Tutorial 3.

The C++ library will create the function as compilable on-demand for us, so we don't have to do that explicitly. But we do have to override the virtual `build()` method to build the function's body on-demand:

```
void mul_add_function::build()
{
    jit_value x = get_param(0);
    jit_value y = get_param(1);
    jit_value z = get_param(2);

    insn_return(x * y + z);
}
```

This is similar to the first version that we wrote in Tutorial 1. Instructions are created with `insn_*` methods that correspond to their `jit_insn_*` counterparts in the C library.

One of the nice things about the C++ API compared to the C API is that we can use overloaded operators to manipulate `jit_value` objects. This can simplify the function build process considerably when we have lots of expressions to compile. We could have used `insn_mul` and `insn_add` instead in this example and the result would have been the same.

Now that we have our `mul_add_function` class, we can create an instance of the function and apply it as follows:

```
jit_context context;
mul_add_function mul_add(context);

jit_int arg1 = 3;
jit_int arg2 = 5;
jit_int arg3 = 2;
jit_int args[3];
args[0] = &arg1;
args[1] = &arg2;
args[2] = &arg3;

mul_add.apply(args, &result);
```

See Chapter 16 [C++ Interface], page 77, for more information on the `libjitplus` library.

3.5 Tutorial 5 - gcd, with tail calls

Astute readers would have noticed that Tutorial 2 included two instances of "tail calls". That is, calls to the same function that are immediately followed by a `return` instruction.

Libjit can optimize tail calls if you provide the `JIT_CALL_TAIL` flag to `jit_insn_call`. Previously, we used the following code to call `gcd` recursively:

```
temp3 = jit_insn_call
    (function, "gcd", function, 0, temp_args, 2, 0);
jit_insn_return(function, temp3);
```

In Tutorial 5, this is modified to the following:

```
jit_insn_call(function, "gcd", function, 0, temp_args, 2, JIT_CALL_TAIL);
```

There is no need for the `jit_insn_return`, because the call will never return to that point in the code. Behind the scenes, `libjit` will convert the call into a jump back to the head of the function.

Tail calls can only be used in certain circumstances. The source and destination of the call must have the same function signatures. None of the parameters should point to local variables in the current stack frame. And tail calls cannot be used from any source function that uses `try` or `alloca` statements.

Because it can be difficult for `libjit` to determine when these conditions have been met, it relies upon the caller to supply the `JIT_CALL_TAIL` flag when it is appropriate to use a tail call.

3.6 Dynamic Pascal - A full JIT example

This `libjit/dpas` directory contains an implementation of "Dynamic Pascal", or "dpas" as we like to call it. It is provided as an example of using `libjit` in a real working environment. We also use it to write test programs that exercise the JIT's capabilities.

Other Pascal implementations compile the source to executable form, which is then run separately. Dynamic Pascal loads the source code at runtime, dynamically JIT'ing the program as it goes. It thus has a lot in common with scripting languages like Perl and Python.

If you are writing a bytecode-based virtual machine, you would use a similar approach to Dynamic Pascal. The key difference is that you would build the JIT data structures after loading the bytecode rather than after parsing the source code.

To run a Dynamic Pascal program, use `dpas name.pas`. You may also need to pass the `-I` option to specify the location of the system library if you have used an `import` clause in your program. e.g. `dpas -I$HOME/libjit/dpas/library name.pas`.

This Pascal grammar is based on the EBNF description at the following URL:

http://www.cs.qub.ac.uk/~S.Fitzpatrick/Teaching/Pascal/EBNF.html

There are a few differences to "Standard Pascal":

1. Identifiers are case-insensitive, but case-preserving.
2. Program headings are normally `program Name (Input, Output);`. This can be abbreviated to `program Name;` as the program modifiers are ignored.
3. Some GNU Pascal operators like `xor`, `shl`, `@`, etc have been added.
4. The integer type names (`Integer`, `Cardinal`, `LongInt`, etc) follow those used in GNU Pascal also. The `Integer` type is always 32-bits in size, while `LongInt` is always 64-bits in size.
5. The types `SysInt`, `SysCard`, `SysLong`, `SysLongCard`, `SysLongestInt`, and `SysLongestCard` are guaranteed to be the same size as the underlying C system's `int`, `unsigned int`, `long`, `unsigned long`, `long long`, and `unsigned long long` types.
6. The type `Address` is logically equivalent to C's `void *`. Any pointer or array can be implicitly cast to `Address`. An explicit cast is required to cast back to a typed pointer (you cannot cast back to an array).
7. The `String` type is declared as `^Char`. Single-dimensional arrays of `Char` can be implicitly cast to any `String` destination. Strings are not bounds-checked, so be careful. Arrays are bounds-checked.

8. Pointers can be used as arrays. e.g. `p[n]` will access the n'th item of an unbounded array located at `p`. Use with care.
9. We don't support `file of` types. Data can be written to stdout using `Write` and `WriteLn`, but that is the extent of the I/O facilities.
10. The declaration `import Name1, Name2, ...;` can be used at the head of a program to declare additional files to include. e.g. `import stdio` will import the contents of `stdio.pas`. We don't support units.
11. The idiom `; ..` can be used at the end of a formal parameter list to declare that the procedure or function takes a variable number of arguments. The builtin function `va_arg(Type)` is used to extract the arguments.
12. The directive `import("Library")` can be used to declare that a function or procedure was imported from an external C library. For example, the following imports the C `puts` and `printf` functions:

    ```
    function puts (str : String) : SysInt; import ("libc")
    function printf (format : String; ..) : SysInt; import ("libc")
    ```

    Functions that are imported in this manner have case-sensitive names. i.e. using `Printf` above will fail.
13. The `throw` keyword can be used to throw an exception. The argument must be a pointer. The `try`, `catch`, and `finally` keywords are used to manage such exceptions further up the stack. e.g.

    ```
    try
        ...
    catch Name : Type
        ...
    finally
        ...
    end
    ```

    The `catch` block will be invoked with the exception pointer that was supplied to `throw`, after casting it to `Type` (which must be a pointer type). Specifying `throw` on its own without an argument will rethrow the current exception pointer, and can only be used inside a `catch` block.

    Dynamic Pascal does not actually check the type of the thrown pointer. If you have multiple kinds of exceptions, then you must store some kind of type indicator in the block that is thrown and then inspect `^Name` to see what the indicator says.
14. The `exit` keyword can be used to break out of a loop.
15. Function calls can be used as procedure calls. The return value is ignored.
16. Hexadecimal constants can be expressed as `XXH`. The first digit must be between 0 and 9, but the remaining digits can be any hex digit.
17. Ternary conditionals can be expressed as `(if e1 then e2 else e3)`. The brackets are required. This is equivalent to C's `e1 ? e2 : e3`.
18. Assigning to a function result will immediately return. i.e. it is similar to `return value;` in C. It isn't necessary to arrange for execution to flow through to the end of the function as in regular Pascal.
19. The term `sizeof(Type)` can be used to get the size of a type.

20. Procedure and function headings can appear in a record type to declare a field with a `pointer to procedure/function` type.

4 Initializing the JIT

`void jit_init` (*void*) [Function]
: This is normally the first function that you call when using `libjit`. It initializes the library and prepares for JIT operations.

  The `jit_context_create` function also calls this, so you can avoid using `jit_init` if `jit_context_create` is the first JIT function that you use.

  It is safe to initialize the JIT multiple times. Subsequent initializations are quietly ignored.

`int jit_uses_interpreter` (*void*) [Function]
: Determine if the JIT uses a fall-back interpreter to execute code rather than generating and executing native code. This can be called prior to `jit_init`.

Everything that is done with `libjit` is done relative to a context. It is possible to have more than one context at a time - each acts as an independent environment for compiling and managing code.

When you want to compile a function, you create it with `jit_function_create`, and then populate its body with calls to the value and instruction functions. See See Chapter 7 [Values], page 29, and Chapter 8 [Instructions], page 34 for more information on how to do this.

4.1 Using libjit in a multi-threaded environment

The library does not handle the creation, management, and destruction of threads itself. It is up to the front-end environment to take care of that. But the library is thread-aware, as long as you take some very simple steps.

In a multi-threaded environment, you must ensure that only one thread can build functions at any one time. Otherwise the JIT's context may become corrupted. To protect the system, you should call `jit_context_build_start` before creating the function. And then call `jit_context_build_end` once the function has been fully compiled.

You can compile multiple functions during the one build process if you wish, which is the normal case when compiling a class.

It is usually a good idea to suspend the finalization of garbage-collected objects while function building is in progress. Otherwise you may get a deadlock when the finalizer thread tries to call the builder to compile a finalization routine. Suspension of finalization is the responsibility of the caller.

4.2 Context functions

The following functions are available to create, manage, and ultimately destroy JIT contexts:

`jit_context_t jit_context_create` (*void*) [Function]
: Create a new context block for the JIT. Returns NULL if out of memory.

`void jit_context_destroy` (*jit_context_t* **`context`**) [Function]
Destroy a JIT context block and everything that is associated with it. It is very important that no threads within the program are currently running compiled code when this function is called.

`int jit_context_supports_threads` (*jit_context_t* **`context`**) [Function]
Determine if the JIT supports threads.

`void jit_context_build_start` (*jit_context_t* **`context`**) [Function]
This routine should be called before you start building a function to be JIT'ed. It acquires a lock on the context to prevent other threads from accessing the build process, since only one thread can be performing build operations at any one time.

`void jit_context_build_end` (*jit_context_t* **`context`**) [Function]
This routine should be called once you have finished building and compiling a function and are ready to resume normal execution. This routine will release the build lock, allowing other threads that are waiting on the builder to proceed.

`void jit_context_set_on_demand_driver` (*jit_context_t* **`context`**, *jit_on_demand_driver_func* **`driver`**) [Function]
Specify the C function to be called to drive on-demand compilation.

When on-demand compilation is requested the default driver provided by `libjit` takes the following actions:

1. The context is locked by calling `jit_context_build_start`.
2. If the function has already been compiled, `libjit` unlocks the context and returns immediately. This can happen because of race conditions between threads: some other thread may have beaten us to the on-demand compiler.
3. The user's on-demand compiler is called. It is responsible for building the instructions in the function's body. It should return one of the result codes `JIT_RESULT_OK`, `JIT_RESULT_COMPILE_ERROR`, or `JIT_RESULT_OUT_OF_MEMORY`.
4. If the user's on-demand function hasn't already done so, `libjit` will call `jit_function_compile` to compile the function.
5. The context is unlocked by calling `jit_context_build_end` and `libjit` jumps to the newly-compiled entry point. If an error occurs, a built-in exception of type `JIT_RESULT_COMPILE_ERROR` or `JIT_RESULT_OUT_OF_MEMORY` will be thrown.
6. The entry point of the compiled function is returned from the driver.

You may need to provide your own driver if some additional actions are required.

`int jit_context_set_meta` (*jit_context_t* **`context`**, *int* **`type`**, *void* ***`data`**, *jit_meta_free_func* **`free_data`**) [Function]
Tag a context with some metadata. Returns zero if out of memory.

Metadata may be used to store dependency graphs, branch prediction information, or any other information that is useful to optimizers or code generators. It can also be used by higher level user code to store information about the context that is specific to the virtual machine or language.

If the *type* already has some metadata associated with it, then the previous value will be freed.

`int jit_context_set_meta_numeric (`*jit_context_t* `context`, *int* `type`, *jit_nuint* `data`)` [Function]

Tag a context with numeric metadata. Returns zero if out of memory. This function is more convenient for accessing the context's special option values:

`JIT_OPTION_CACHE_LIMIT`
: A numeric option that indicates the maximum size in bytes of the function cache. If set to zero (the default), the function cache is unlimited in size.

`JIT_OPTION_CACHE_PAGE_SIZE`
: A numeric option that indicates the size in bytes of a single page in the function cache. Memory is allocated for the cache in chunks of this size. If set to zero, the cache page size is set to an internally-determined default (usually 128k). The cache page size also determines the maximum size of a single compiled function.

`JIT_OPTION_PRE_COMPILE`
: A numeric option that indicates that this context is being used for pre-compilation if it is set to a non-zero value. Code within pre-compiled contexts cannot be executed directly. Instead, they can be written out to disk in ELF format to be reloaded at some future time.

`JIT_OPTION_DONT_FOLD`
: A numeric option that disables constant folding when it is set to a non-zero value. This is useful for debugging, as it forces `libjit` to always execute constant expressions at run time, instead of at compile time.

`JIT_OPTION_POSITION_INDEPENDENT`
: A numeric option that forces generation of position-independent code (PIC) if it is set to a non-zero value. This may be mainly useful for pre-compiled contexts.

Metadata type values of 10000 or greater are reserved for internal use.

`void * jit_context_get_meta (`*jit_context_t* `context`, *int* `type`)` [Function]

Get the metadata associated with a particular tag. Returns NULL if *type* does not have any metadata associated with it.

`jit_nuint jit_context_get_meta_numeric (`*jit_context_t* `context`, *int* `type`)` [Function]

Get the metadata associated with a particular tag. Returns zero if *type* does not have any metadata associated with it. This version is more convenient for the pre-defined numeric option values.

`void jit_context_free_meta (`*jit_context_t* `context`, *int* `type`)` [Function]

Free metadata of a specific type on a context. Does nothing if the *type* does not have any metadata associated with it.

5 Building and compiling functions with the JIT

`jit_function_t jit_function_create` (*jit_context_t* **`context`**, *jit_type_t* **`signature`**) [Function]

Create a new function block and associate it with a JIT context. Returns NULL if out of memory.

A function persists for the lifetime of its containing context. It initially starts life in the "building" state, where the user constructs instructions that represents the function body. Once the build process is complete, the user calls `jit_function_compile` to convert it into its executable form.

It is recommended that you call `jit_context_build_start` before calling `jit_function_create`, and then call `jit_context_build_end` after you have called `jit_function_compile`. This will protect the JIT's internal data structures within a multi-threaded environment.

`jit_function_t jit_function_create_nested` (*jit_context_t* **`context`**, *jit_type_t* **`signature`**, *jit_function_t* **`parent`**) [Function]

Create a new function block and associate it with a JIT context. In addition, this function is nested inside the specified *parent* function and is able to access its parent's (and grandparent's) local variables.

The front end is responsible for ensuring that the nested function can never be called by anyone except its parent and sibling functions. The front end is also responsible for ensuring that the nested function is compiled before its parent.

`void jit_function_abandon` (*jit_function_t* **`func`**) [Function]

Abandon this function during the build process. This should be called when you detect a fatal error that prevents the function from being properly built. The *func* object is completely destroyed and detached from its owning context. The function is left alone if it was already compiled.

`jit_context_t jit_function_get_context` (*jit_function_t* **`func`**) [Function]

Get the context associated with a function.

`jit_type_t jit_function_get_signature` (*jit_function_t* **`func`**) [Function]

Get the signature associated with a function.

`int jit_function_set_meta` (*jit_function_t* **`func`**, *int* **`type`**, *void* ***`data`**, *jit_meta_free_func* **`free_data`**, *int* **`build_only`**) [Function]

Tag a function with some metadata. Returns zero if out of memory.

Metadata may be used to store dependency graphs, branch prediction information, or any other information that is useful to optimizers or code generators. It can also be used by higher level user code to store information about the function that is specific to the virtual machine or language.

If the *type* already has some metadata associated with it, then the previous value will be freed.

If *build_only* is non-zero, then the metadata will be freed when the function is compiled with `jit_function_compile`. Otherwise the metadata will persist until the JIT context is destroyed, or `jit_function_free_meta` is called for the specified *type*.

Metadata type values of 10000 or greater are reserved for internal use.

`void * jit_function_get_meta` (*jit_function_t* ***func***, *int* ***type***) [Function]
Get the metadata associated with a particular tag. Returns NULL if *type* does not have any metadata associated with it.

`void jit_function_free_meta` (*jit_function_t* ***func***, *int* ***type***) [Function]
Free metadata of a specific type on a function. Does nothing if the *type* does not have any metadata associated with it.

`jit_function_t jit_function_next` (*jit_context_t* ***context***, *jit_function_t* ***prev***) [Function]
Iterate over the defined functions in creation order. The *prev* argument should be NULL on the first call. Returns NULL at the end.

`jit_function_t jit_function_previous` (*jit_context_t* ***context***, *jit_function_t* ***prev***) [Function]
Iterate over the defined functions in reverse creation order.

`jit_block_t jit_function_get_entry` (*jit_function_t* ***func***) [Function]
Get the entry block for a function. This is always the first block created by `jit_function_create`.

`jit_block_t jit_function_get_current` (*jit_function_t* ***func***) [Function]
Get the current block for a function. New blocks are created by certain `jit_insn_xxx` calls.

`jit_function_t jit_function_get_nested_parent` (*jit_function_t* ***func***) [Function]
Get the nested parent for a function, or NULL if *func* does not have a nested parent.

`int jit_function_compile` (*jit_function_t* ***func***) [Function]
Compile a function to its executable form. If the function was already compiled, then do nothing. Returns zero on error.

If an error occurs, you can use `jit_function_abandon` to completely destroy the function. Once the function has been compiled successfully, it can no longer be abandoned.

Sometimes you may wish to recompile a function, to apply greater levels of optimization the second time around. You must call `jit_function_set_recompilable` before you compile the function the first time. On the second time around, build the function's instructions again, and call `jit_function_compile` a second time.

`int jit_function_compile_entry` (*jit_function_t* ***func***, *void* ***entry_point***) [Function]
Compile a function to its executable form but do not make it available for invocation yet. It may be made available later with `jit_function_setup_entry`.

`int jit_function_setup_entry` (*jit_function_t* ***func***, *void* ***entry_point***) [Function]
Make a function compiled with `jit_function_compile_entry` available for invocation and free the resources used for compilation. If *entry_point* is null then it only frees the resources.

`int jit_function_is_compiled` (*jit_function_t* ***func***) [Function]
Determine if a function has already been compiled.

`int jit_function_set_recompilable` (*jit_function_t* ***func***) [Function]
Mark this function as a candidate for recompilation. That is, it is possible that we may call `jit_function_compile` more than once, to re-optimize an existing function.

It is very important that this be called before the first time that you call `jit_function_compile`. Functions that are recompilable are invoked in a slightly different way to non-recompilable functions. If you don't set this flag, then existing invocations of the function may continue to be sent to the original compiled version, not the new version.

`void jit_function_clear_recompilable` (*jit_function_t* ***func***) [Function]
Clear the recompilable flag on this function. Normally you would use this once you have decided that the function has been optimized enough, and that you no longer intend to call `jit_function_compile` again.

Future uses of the function with `jit_insn_call` will output a direct call to the function, which is more efficient than calling its recompilable version. Pre-existing calls to the function may still use redirection stubs, and will remain so until the pre-existing functions are themselves recompiled.

`int jit_function_is_recompilable` (*jit_function_t* ***func***) [Function]
Determine if this function is recompilable.

`void * jit_function_to_closure` (*jit_function_t* ***func***) [Function]
Convert a compiled function into a closure that can called directly from C. Returns NULL if out of memory, or if closures are not supported on this platform.

If the function has not been compiled yet, then this will return a pointer to a redirector that will arrange for the function to be compiled on-demand when it is called.

Creating a closure for a nested function is not recommended as C does not have any way to call such closures directly.

`jit_function_t jit_function_from_closure` (*jit_context_t* ***context***, *void* ***closure***) [Function]
Convert a closure back into a function. Returns NULL if the closure does not correspond to a function in the specified context.

`jit_function_t jit_function_from_pc` (*jit_context_t* ***context***, *void* ***pc***, *void* ****handler***) [Function]
Get the function that contains the specified program counter location. Also return the address of the `catch` handler for the same location. Returns NULL if the program counter does not correspond to a function under the control of *context*.

`void * jit_function_to_vtable_pointer` (*jit_function_t* ***func***) [Function]
Return a pointer that is suitable for referring to this function from a vtable. Such pointers should only be used with the `jit_insn_call_vtable` instruction.

Using `jit_insn_call_vtable` is generally more efficient than `jit_insn_call_indirect` for calling virtual methods.

The vtable pointer might be the same as the closure, but this isn't guaranteed. Closures can be used with `jit_insn_call_indirect`.

`jit_function_t jit_function_from_vtable_pointer` (*jit_context_t* ***context***, *void* ***vtable_pointer***) [Function]

Convert a vtable_pointer back into a function. Returns NULL if the vtable_pointer does not correspond to a function in the specified context.

`void jit_function_set_on_demand_compiler` (*jit_function_t* ***func***, *jit_on_demand_func* ***on_demand***) [Function]

Specify the C function to be called when *func* needs to be compiled on-demand. This should be set just after the function is created, before any build or compile processes begin.

You won't need an on-demand compiler if you always build and compile your functions before you call them. But if you can call a function before it is built, then you must supply an on-demand compiler.

When on-demand compilation is requested, `libjit` takes the following actions:

1. The context is locked by calling `jit_context_build_start`.
2. If the function has already been compiled, `libjit` unlocks the context and returns immediately. This can happen because of race conditions between threads: some other thread may have beaten us to the on-demand compiler.
3. The user's on-demand compiler is called. It is responsible for building the instructions in the function's body. It should return one of the result codes `JIT_RESULT_OK`, `JIT_RESULT_COMPILE_ERROR`, or `JIT_RESULT_OUT_OF_MEMORY`.
4. If the user's on-demand function hasn't already done so, `libjit` will call `jit_function_compile` to compile the function.
5. The context is unlocked by calling `jit_context_build_end` and `libjit` jumps to the newly-compiled entry point. If an error occurs, a built-in exception of type `JIT_RESULT_COMPILE_ERROR` or `JIT_RESULT_OUT_OF_MEMORY` will be thrown.

Normally you will need some kind of context information to tell you which higher-level construct is being compiled. You can use the metadata facility to add this context information to the function just after you create it with `jit_function_create`.

`jit_on_demand_func jit_function_get_on_demand_compiler` (*jit_function_t* ***func***) [Function]

Returns function's on-demand compiler.

`int jit_function_apply` (*jit_function_t* ***func***, *void* ****args***, *void* ****return_area***) [Function]

Call the function *func* with the supplied arguments. Each element in *args* is a pointer to one of the arguments, and *return_area* points to a buffer to receive the return value. Returns zero if an exception occurred.

This is the primary means for executing a function from ordinary C code without creating a closure first with `jit_function_to_closure`. Closures may not be supported on all platforms, but function application is guaranteed to be supported everywhere.

Function applications acts as an exception blocker. If any exceptions occur during the execution of *func*, they won't travel up the stack any further than this point. This prevents ordinary C code from being accidentally presented with a situation that it cannot handle. This blocking protection is not present when a function is invoked via its closure.

`int jit_function_apply_vararg` (*jit_function_t* `func`, *jit_type_t* `signature`, *void* ***`args`, *void* *`return_area`) [Function]

Call the function *func* with the supplied arguments. There may be more arguments than are specified in the function's original signature, in which case the additional values are passed as variable arguments. This function is otherwise identical to `jit_function_apply`.

`void jit_function_set_optimization_level` (*jit_function_t* `func`, *unsigned int* `level`) [Function]

Set the optimization level for *func*. Increasing values indicate that the `libjit` dynamic compiler should expend more effort to generate better code for this function. Usually you would increase this value just before forcing *func* to recompile.

When the optimization level reaches the value returned by `jit_function_get_max_optimization_level()`, there is usually little point in continuing to recompile the function because `libjit` may not be able to do any better.

The front end is usually responsible for choosing candidates for function inlining. If it has identified more such candidates, then it may still want to recompile *func* again even once it has reached the maximum optimization level.

`unsigned int jit_function_get_optimization_level` (*jit_function_t* `func`) [Function]

Get the current optimization level for *func*.

`unsigned int jit_function_get_max_optimization_level` (*void*) [Function]

Get the maximum optimization level that is supported by `libjit`.

`jit_label_t jit_function_reserve_label` (*jit_function_t* `func`) [Function]

Allocate a new label for later use within the function *func*. Most instructions that require a label could perform label allocation themselves. A separate label allocation could be useful to fill a jump table with identical entries.

6 Manipulating system types

The functions that are defined in `<jit/jit-type.h>` allow the library user to create and manipulate objects that represent native system types. For example, `jit_type_int` represents the signed 32-bit integer type.

Each `jit_type_t` object represents a basic system type, be it a primitive, a struct, a union, a pointer, or a function signature. The library uses this information to lay out values in memory.

The following pre-defined types are available:

`jit_type_void`
: Represents the `void` type.

`jit_type_sbyte`
: Represents a signed 8-bit integer type.

`jit_type_ubyte`
: Represents an unsigned 8-bit integer type.

`jit_type_short`
: Represents a signed 16-bit integer type.

`jit_type_ushort`
: Represents an unsigned 16-bit integer type.

`jit_type_int`
: Represents a signed 32-bit integer type.

`jit_type_uint`
: Represents an unsigned 32-bit integer type.

`jit_type_nint`
: Represents a signed integer type that has the same size and alignment as a native pointer.

`jit_type_nuint`
: Represents an unsigned integer type that has the same size and alignment as a native pointer.

`jit_type_long`
: Represents a signed 64-bit integer type.

`jit_type_ulong`
: Represents an unsigned 64-bit integer type.

`jit_type_float32`
: Represents a 32-bit floating point type.

`jit_type_float64`
: Represents a 64-bit floating point type.

`jit_type_nfloat`
: Represents a floating point type that represents the greatest precision supported on the native platform.

`jit_type_void_ptr`
: Represents the system's `void *` type. This can be used wherever a native pointer type is required.

Type descriptors are reference counted. You can make a copy of a type descriptor using the `jit_type_copy` function, and free the copy with `jit_type_free`.

Some languages have special versions of the primitive numeric types (e.g. boolean types, 16-bit Unicode character types, enumerations, etc). If it is important to distinguish these special versions from the numeric types, then you should use the `jit_type_create_tagged` function below.

The following types correspond to the system types on the local platform. i.e. `jit_type_sys_int` will be the same size as `long` on the local platform, whereas `jit_type_long` is always 64 bits in size. These types should not be used to compile code that is intended to work identically on all platforms:

`jit_type_sys_bool`
: Corresponds to the system `bool` type.

`jit_type_sys_char`
: Corresponds to the system `char` type. This may be either signed or unsigned, depending upon the underlying system.

`jit_type_sys_schar`
: Corresponds to the system `signed char` type.

`jit_type_sys_uchar`
: Corresponds to the system `unsigned char` type.

`jit_type_sys_short`
: Corresponds to the system `short` type.

`jit_type_sys_ushort`
: Corresponds to the system `unsigned short` type.

`jit_type_sys_int`
: Corresponds to the system `int` type.

`jit_type_sys_uint`
: Corresponds to the system `unsigned int` type.

`jit_type_sys_long`
: Corresponds to the system `long` type.

`jit_type_sys_ulong`
: Corresponds to the system `unsigned long` type.

`jit_type_sys_longlong`
: Corresponds to the system `long long` type (`__int64` under Win32).

`jit_type_sys_ulonglong`
: Corresponds to the system `unsigned long long` type (`unsigned __int64` under Win32).

`jit_type_sys_float`
: Corresponds to the system `float` type.

`jit_type_sys_double`
: Corresponds to the system `double` type.

`jit_type_sys_long_double`
: Corresponds to the system `long double` type.

`jit_type_t jit_type_copy` (*jit_type_t* `type`) [Function]

Make a copy of the type descriptor *type* by increasing its reference count.

`void jit_type_free (`*jit_type_t* **`type`**`)` [Function]

Free a type descriptor by decreasing its reference count. This function is safe to use on pre-defined types, which are never actually freed.

`jit_type_t jit_type_create_struct (`*jit_type_t* ***`fields`**, *unsigned int* **`num_fields`**, *int* **`incref`**`)` [Function]

Create a type descriptor for a structure. Returns NULL if out of memory. If there are no fields, then the size of the structure will be zero. It is necessary to add a padding field if the language does not allow zero-sized structures. The reference counts on the field types are incremented if *incref* is non-zero.

The `libjit` library does not provide any special support for implementing structure inheritance, where one structure extends the definition of another. The effect of inheritance can be achieved by always allocating the first field of a structure to be an instance of the inherited structure. Multiple inheritance can be supported by allocating several special fields at the front of an inheriting structure.

Similarly, no special support is provided for vtables. The program is responsible for allocating an appropriate slot in a structure to contain the vtable pointer, and dereferencing it wherever necessary. The vtable will itself be a structure, containing signature types for each of the method slots.

The choice not to provide special support for inheritance and vtables in `libjit` was deliberate. The layout of objects and vtables is highly specific to the language and virtual machine being emulated, and no single scheme can hope to capture all possibilities.

`jit_type_t jit_type_create_union (`*jit_type_t* ***`fields`**, *unsigned int* **`num_fields`**, *int* **`incref`**`)` [Function]

Create a type descriptor for a union. Returns NULL if out of memory. If there are no fields, then the size of the union will be zero. It is necessary to add a padding field if the language does not allow zero-sized unions. The reference counts on the field types are incremented if *incref* is non-zero.

`jit_type_t jit_type_create_signature (`*jit_abi_t* **`abi`**, *jit_type_t* **`return_type`**, *jit_type_t* ***`params`**, *unsigned int* **`num_params`**, *int* **`incref`**`)` [Function]

Create a type descriptor for a function signature. Returns NULL if out of memory. The reference counts on the component types are incremented if *incref* is non-zero.

When used as a structure or union field, function signatures are laid out like pointers. That is, they represent a pointer to a function that has the specified parameters and return type.

The *abi* parameter specifies the Application Binary Interface (ABI) that the function uses. It may be one of the following values:

`jit_abi_cdecl`
: Use the native C ABI definitions of the underlying platform.

`jit_abi_vararg`
: Use the native C ABI definitions of the underlying platform, and allow for an optional list of variable argument parameters.

`jit_abi_stdcall`
: Use the Win32 STDCALL ABI definitions, whereby the callee pops its arguments rather than the caller. If the platform does not support this type of ABI, then `jit_abi_stdcall` will be identical to `jit_abi_cdecl`.

`jit_abi_fastcall`
: Use the Win32 FASTCALL ABI definitions, whereby the callee pops its arguments rather than the caller, and the first two word arguments are passed in ECX and EDX. If the platform does not support this type of ABI, then `jit_abi_fastcall` will be identical to `jit_abi_cdecl`.

`jit_type_t jit_type_create_pointer` (*jit_type_t* ***type***, *int* ***incref***) [Function]
: Create a type descriptor for a pointer to another type. Returns NULL if out of memory. The reference count on *type* is incremented if *incref* is non-zero.

`jit_type_t jit_type_create_tagged` (*jit_type_t* ***type***, *int* ***kind***, *void* ***data***, *jit_meta_free_func* ***free_func***, *int* ***incref***) [Function]
: Tag a type with some additional user data. Tagging is typically used by higher-level programs to embed extra information about a type that `libjit` itself does not support.

  As an example, a language might have a 16-bit Unicode character type and a 16-bit unsigned integer type that are distinct types, even though they share the same fundamental representation (`jit_ushort`). Tagging allows the program to distinguish these two types, when it is necessary to do so, without affecting `libjit`'s ability to compile the code efficiently.

  The *kind* is a small positive integer value that the program can use to distinguish multiple tag types. The *data* pointer is the actual data that you wish to store. And *free_func* is a function that is used to free *data* when the type is freed with `jit_type_free`.

  If you need to store more than one piece of information, you can tag a type multiple times. The order in which multiple tags are applied is irrelevant to `libjit`, although it may be relevant to the higher-level program.

  Tag kinds of 10000 or greater are reserved for `libjit` itself. The following special tag kinds are currently provided in the base implementation:

  `JIT_TYPETAG_NAME`
  : The *data* pointer is a `char *` string indicating a friendly name to display for the type.

  `JIT_TYPETAG_STRUCT_NAME`
  `JIT_TYPETAG_UNION_NAME`
  `JIT_TYPETAG_ENUM_NAME`
  : The *data* pointer is a `char *` string indicating a friendly name to display for a `struct`, `union`, or `enum` type. This is for languages like C that have separate naming scopes for typedef's and structures.

  `JIT_TYPETAG_CONST`
  : The underlying value is assumed to have `const` semantics. The `libjit` library doesn't enforce such semantics: it is up to the front-end to only use constant values in appopriate contexts.

`JIT_TYPETAG_VOLATILE`
: The underlying value is assumed to be volatile. The `libjit` library will automatically call `jit_value_set_volatile` when a value is constructed using this type.

`JIT_TYPETAG_REFERENCE`
: The underlying value is a pointer, but it is assumed to refer to a pass-by-reference parameter.

`JIT_TYPETAG_OUTPUT`
: This is similar to `JIT_TYPETAG_REFERENCE`, except that the underlying parameter is assumed to be output-only.

`JIT_TYPETAG_RESTRICT`
: The underlying type is marked as `restrict`. Normally ignored.

`JIT_TYPETAG_SYS_BOOL`
`JIT_TYPETAG_SYS_CHAR`
`JIT_TYPETAG_SYS_SCHAR`
`JIT_TYPETAG_SYS_UCHAR`
`JIT_TYPETAG_SYS_SHORT`
`JIT_TYPETAG_SYS_USHORT`
`JIT_TYPETAG_SYS_INT`
`JIT_TYPETAG_SYS_UINT`
`JIT_TYPETAG_SYS_LONG`
`JIT_TYPETAG_SYS_ULONG`
`JIT_TYPETAG_SYS_LONGLONG`
`JIT_TYPETAG_SYS_ULONGLONG`
`JIT_TYPETAG_SYS_FLOAT`
`JIT_TYPETAG_SYS_DOUBLE`
`JIT_TYPETAG_SYS_LONGDOUBLE`
: Used to mark types that we know for a fact correspond to the system C types of the corresponding names. This is primarily used to distinguish system types like `int` and `long` types on 32-bit platforms when it is necessary to do so. The `jit_type_sys_xxx` values are all tagged in this manner.

`int jit_type_set_names (`*jit_type_t* ***type***, *char* ****names***, *unsigned int* ***num_names***`)` [Function]

Set the field or parameter names for *type*. Returns zero if there is insufficient memory to set the names.

Normally fields are accessed via their index. Field names are a convenience for front ends that prefer to use names to indices.

`void jit_type_set_size_and_alignment (`*jit_type_t* ***type***, *jit_nint* ***size***, *jit_nint* ***alignment***`)` [Function]

Set the size and alignment information for a structure or union type. Use this for performing explicit type layout. Normally the size is computed automatically. Ignored if not a structure or union type. Setting either value to -1 will cause that value to be computed automatically.

`void jit_type_set_offset (`*jit_type_t* `type`, *unsigned int* `field_index`, *jit_nuint* `offset``)` [Function]

Set the offset of a specific structure field. Use this for performing explicit type layout. Normally the offset is computed automatically. Ignored if not a structure type, or the field index is out of range.

`int jit_type_get_kind (`*jit_type_t* `type``)` [Function]

Get a value that indicates the kind of *type*. This allows callers to quickly classify a type to determine how it should be handled further.

`JIT_TYPE_INVALID`
: The value of the *type* parameter is NULL.

`JIT_TYPE_VOID`
: The type is `jit_type_void`.

`JIT_TYPE_SBYTE`
: The type is `jit_type_sbyte`.

`JIT_TYPE_UBYTE`
: The type is `jit_type_ubyte`.

`JIT_TYPE_SHORT`
: The type is `jit_type_short`.

`JIT_TYPE_USHORT`
: The type is `jit_type_ushort`.

`JIT_TYPE_INT`
: The type is `jit_type_int`.

`JIT_TYPE_UINT`
: The type is `jit_type_uint`.

`JIT_TYPE_NINT`
: The type is `jit_type_nint`.

`JIT_TYPE_NUINT`
: The type is `jit_type_nuint`.

`JIT_TYPE_LONG`
: The type is `jit_type_long`.

`JIT_TYPE_ULONG`
: The type is `jit_type_ulong`.

`JIT_TYPE_FLOAT32`
: The type is `jit_type_float32`.

`JIT_TYPE_FLOAT64`
: The type is `jit_type_float64`.

`JIT_TYPE_NFLOAT`
: The type is `jit_type_nfloat`.

`JIT_TYPE_STRUCT`
: The type is the result of calling `jit_type_create_struct`.

`JIT_TYPE_UNION`
: The type is the result of calling `jit_type_create_union`.

`JIT_TYPE_SIGNATURE`
: The type is the result of calling `jit_type_create_signature`.

`JIT_TYPE_PTR`
: The type is the result of calling `jit_type_create_pointer`.

If this function returns `JIT_TYPE_FIRST_TAGGED` or higher, then the type is tagged and its tag kind is the return value minus `JIT_TYPE_FIRST_TAGGED`. That is, the following two expressions will be identical if *type* is tagged:

```
jit_type_get_tagged_kind(type)
jit_type_get_kind(type) - JIT_TYPE_FIRST_TAGGED
```

`jit_nuint jit_type_get_size (`*jit_type_t* *type*`)` [Function]
Get the size of a type in bytes.

`jit_nuint jit_type_get_alignment (`*jit_type_t* *type*`)` [Function]
Get the alignment of a type. An alignment value of 2 indicates that the type should be aligned on a two-byte boundary, for example.

`unsigned int jit_type_num_fields (`*jit_type_t* *type*`)` [Function]
Get the number of fields in a structure or union type.

`jit_type_t jit_type_get_field (`*jit_type_t* *type*, *unsigned int* *field_index*`)` [Function]
Get the type of a specific field within a structure or union. Returns NULL if not a structure or union, or the index is out of range.

`jit_nuint jit_type_get_offset (`*jit_type_t* *type*, *unsigned int* *field_index*`)` [Function]
Get the offset of a specific field within a structure. Returns zero if not a structure, or the index is out of range, so this is safe to use on non-structure types.

`const char * jit_type_get_name (`*jit_type_t* *type*, *unsigned int* *index*`)` [Function]
Get the name of a structure, union, or signature field/parameter. Returns NULL if not a structure, union, or signature, the index is out of range, or there is no name associated with the component.

`unsigned int jit_type_find_name (`*jit_type_t* *type*, *const char* **name*`)` [Function]
Find the field/parameter index for a particular name. Returns `JIT_INVALID_NAME` if the name was not present.

`unsigned int jit_type_num_params (`*jit_type_t* *type*`)` [Function]
Get the number of parameters in a signature type.

`jit_type_t jit_type_get_return (`*jit_type_t* *type*`)` [Function]
Get the return type from a signature type. Returns NULL if not a signature type.

`jit_type_t jit_type_get_param` (*jit_type_t* ***type***, *unsigned int* ***param_index***) [Function]

Get a specific parameter from a signature type. Returns NULL if not a signature type or the index is out of range.

`jit_abi_t jit_type_get_abi` (*jit_type_t* ***type***) [Function]

Get the ABI code from a signature type. Returns `jit_abi_cdecl` if not a signature type.

`jit_type_t jit_type_get_ref` (*jit_type_t* ***type***) [Function]

Get the type that is referred to by a pointer type. Returns NULL if not a pointer type.

`jit_type_t jit_type_get_tagged_type` (*jit_type_t* ***type***) [Function]

Get the type that underlies a tagged type. Returns NULL if not a tagged type.

`void jit_type_set_tagged_type` (*jit_type_t* ***type***, *jit_type_t* ***underlying***) [Function]

Set the type that underlies a tagged type. Ignored if *type* is not a tagged type. If *type* already has an underlying type, then the original is freed.

This function is typically used to flesh out the body of a forward-declared type. The tag is used as a placeholder until the definition can be located.

`int jit_type_get_tagged_type` (*jit_type_t* ***type***) [Function]

Get the kind of tag that is applied to a tagged type. Returns -1 if not a tagged type.

`void * jit_type_get_tagged_data` (*jit_type_t* ***type***) [Function]

Get the user data is associated with a tagged type. Returns NULL if not a tagged type.

`void jit_type_set_tagged_data` (*jit_type_t* ***type***, *void* ****data***, *jit_meta_free_func* ***free_func***) [Function]

Set the user data is associated with a tagged type. The original data, if any, is freed.

`int jit_type_is_primitive` (*jit_type_t* ***type***) [Function]

Determine if a type is primitive.

`int jit_type_is_struct` (*jit_type_t* ***type***) [Function]

Determine if a type is a structure.

`int jit_type_is_union` (*jit_type_t* ***type***) [Function]

Determine if a type is a union.

`int jit_type_is_signature` (*jit_type_t* ***type***) [Function]

Determine if a type is a function signature.

`int jit_type_is_pointer` (*jit_type_t* ***type***) [Function]

Determine if a type is a pointer.

`int jit_type_is_tagged` (*jit_type_t* ***type***) [Function]

Determine if a type is a tagged type.

`jit_nuint jit_type_best_alignment` (*void*) [Function]
Get the best alignment value for this platform.

`jit_type_t jit_type_normalize` (*jit_type_t* ***type***) [Function]
Normalize a type to its basic numeric form. e.g. "jit_type_nint" is turned into "jit_type_int" or "jit_type_long", depending upon the underlying platform. Pointers are normalized like "jit_type_nint". If the type does not have a normalized form, it is left unchanged.

Normalization is typically used prior to applying a binary numeric instruction, to make it easier to determine the common type. It will also remove tags from the specified type.

`jit_type_t jit_type_remove_tags` (*jit_type_t* ***type***) [Function]
Remove tags from a type, and return the underlying type. This is different from normalization, which will also collapses native types to their basic numeric counterparts.

`jit_type_t jit_type_promote_int` (*jit_type_t* ***type***) [Function]
If *type* is `jit_type_sbyte` or `jit_type_short`, then return `jit_type_int`. If *type* is `jit_type_ubyte` or `jit_type_ushort`, then return `jit_type_uint`. Otherwise return *type* as-is.

`int jit_type_return_via_pointer` (*jit_type_t* ***type***) [Function]
Determine if a type should be returned via a pointer if it appears as the return type in a signature.

`int jit_type_has_tag` (*jit_type_t* ***type***, *int* ***kind***) [Function]
Determine if *type* has a specific kind of tag. This will resolve multiple levels of tagging.

7 Working with temporary values in the JIT

Values form the backbone of the storage system in `libjit`. Every value in the system, be it a constant, a local variable, or a temporary result, is represented by an object of type `jit_value_t`. The JIT then allocates registers or memory locations to the values as appropriate.

We will demonstrate how to use values with a simple example of adding two local variables together and putting the result into a third local variable. First, we allocate storage for the three local variables:

```
value1 = jit_value_create(func, jit_type_int);
value2 = jit_value_create(func, jit_type_int);
value3 = jit_value_create(func, jit_type_int);
```

Here, `func` is the function that we are building. To add `value1` and `value2` and put the result into `value3`, we use the following code:

```
temp = jit_insn_add(func, value1, value2);
jit_insn_store(func, value3, temp);
```

The `jit_insn_add` function allocates a temporary value (`temp`) and places the result of the addition into it. The `jit_insn_store` function then stores the temporary result into `value3`.

You might be tempted to think that the above code is inefficient. Why do we copy the result into a temporary variable first? Why not put the result directly to `value3`?

Behind the scenes, the JIT will typically optimize `temp` away, resulting in the final code that you expect (i.e. `value3 = value1 + value2`). It is simply easier to use `libjit` if all results end up in temporary variables first, so that's what we do.

Using temporary values, it is very easy to convert stack machine bytecodes into JIT instructions. Consider the following Java Virtual Machine bytecode (representing `value4 = value1 * value2 + value3`):

```
iload 1
iload 2
imul
iload 3
iadd
istore 4
```

Let us demonstrate how this code would be translated, instruction by instruction. We assume that we have a `stack` available, which keeps track of the temporary values in the system. We also assume that `jit_value_t` objects representing the local variables are already stored in an array called `locals`.

First, we load local variable 1 onto the stack:

```
stack[size++] = jit_insn_load(func, locals[1]);
```

We repeat this for local variable 2:

```
stack[size++] = jit_insn_load(func, locals[2]);
```

Now we pop these two values and push their multiplication:

```
stack[size - 2] = jit_insn_mul(func, stack[size - 2], stack[size - 1]);
--size;
```

Next, we need to push the value of local variable 3 and add it to the product that we just computed:

```
stack[size++] = jit_insn_load(func, locals[3]);
stack[size - 2] = jit_insn_add(func, stack[size - 2], stack[size - 1]);
--size;
```

Finally, we store the result into local variable 4:

```
jit_insn_store(func, locals[4], stack[--size]);
```

Collecting up all of the above code, we get the following:

```
stack[size++] = jit_insn_load(func, locals[1]);
stack[size++] = jit_insn_load(func, locals[2]);
stack[size - 2] = jit_insn_mul(func, stack[size - 2], stack[size - 1]);
--size;
stack[size++] = jit_insn_load(func, locals[3]);
stack[size - 2] = jit_insn_add(func, stack[size - 2], stack[size - 1]);
--size;
jit_insn_store(func, locals[4], stack[--size]);
```

The JIT will optimize away most of these temporary results, leaving the final machine code that you expect.

If the virtual machine was register-based, then a slightly different translation strategy would be used. Consider the following code, which computes `reg4 = reg1 * reg2 + reg3`, with the intermediate result stored temporarily in `reg5`:

```
mul reg5, reg1, reg2
add reg4, reg5, reg3
```

You would start by allocating value objects for all of the registers in your system (with `jit_value_create`):

```
reg[1] = jit_value_create(func, jit_type_int);
reg[2] = jit_value_create(func, jit_type_int);
reg[3] = jit_value_create(func, jit_type_int);
reg[4] = jit_value_create(func, jit_type_int);
reg[5] = jit_value_create(func, jit_type_int);
```

Then, the virtual register machine code is translated as follows:

```
temp1 = jit_insn_mul(func, reg[1], reg[2]);
jit_insn_store(reg[5], temp1);
temp2 = jit_insn_add(func, reg[5], reg[3]);
jit_insn_store(reg[4], temp2);
```

Each virtual register machine instruction turns into two `libjit` function calls. The JIT will normally optimize away the temporary results. If the value in `reg5` is not used further down the code, then the JIT may also be able to optimize `reg5` away.

The rest of this section describes the functions that are available to create and manipulate values.

`jit_value_t jit_value_create` (*jit_function_t* ***func***, *jit_type_t* ***type***) [Function]

Create a new value in the context of a function's current block. The value initially starts off as a block-specific temporary. It will be converted into a function-wide local variable if it is ever referenced from a different block. Returns NULL if out of memory.

Note: It isn't possible to refer to global variables directly using values. If you need to access a global variable, then load its address into a temporary and use `jit_insn_load_relative` or `jit_insn_store_relative` to manipulate it. It simplifies the JIT if it can assume that all values are local.

`jit_value_t jit_value_create_nint_constant` (*jit_function_t* ***func***, *jit_type_t* ***type***, *jit_nint* ***const_value***) [Function]

Create a new native integer constant in the specified function. Returns NULL if out of memory.

The *type* parameter indicates the actual type of the constant, if it happens to be something other than `jit_type_nint`. For example, the following will create an unsigned byte constant:

```
value = jit_value_create_nint_constant(context, jit_type_ubyte, 128);
```

This function can be used to create constants of type `jit_type_sbyte`, `jit_type_ubyte`, `jit_type_short`, `jit_type_ushort`, `jit_type_int`, `jit_type_uint`, `jit_type_nint`, `jit_type_nuint`, and all pointer types.

`jit_value_t jit_value_create_long_constant (`*jit_function_t* **`func`**, *jit_type_t* **`type`**, *jit_long* **`const_value`**`)` [Function]
Create a new 64-bit integer constant in the specified function. This can also be used to create constants of type `jit_type_ulong`. Returns NULL if out of memory.

`jit_value_t jit_value_create_float32_constant (`*jit_function_t* **`func`**, *jit_type_t* **`type`**, *jit_float32* **`const_value`**`)` [Function]
Create a new 32-bit floating-point constant in the specified function. Returns NULL if out of memory.

`jit_value_t jit_value_create_float64_constant (`*jit_function_t* **`func`**, *jit_type_t* **`type`**, *jit_float64* **`const_value`**`)` [Function]
Create a new 64-bit floating-point constant in the specified function. Returns NULL if out of memory.

`jit_value_t jit_value_create_nfloat_constant (`*jit_function_t* **`func`**, *jit_type_t* **`type`**, *jit_nfloat* **`const_value`**`)` [Function]
Create a new native floating-point constant in the specified function. Returns NULL if out of memory.

`jit_value_t jit_value_create_constant (`*jit_function_t* **`func`**, *const jit_constant* ***`const_value`**`)` [Function]
Create a new constant from a generic constant structure in the specified function. Returns NULL if out of memory or if the type in *const_value* is not suitable for a constant.

`jit_value_t jit_value_get_param (`*jit_function_t* **`func`**, *unsigned int* **`param`**`)` [Function]
Get the value that corresponds to a specified function parameter. Returns NULL if out of memory.

`jit_value_t jit_value_get_struct_pointer (`*jit_function_t* **`func`**`)` [Function]
Get the value that contains the structure return pointer for a function. If the function does not have a structure return pointer (i.e. structures are returned in registers), then this returns NULL.

`int jit_value_is_temporary (`*jit_value_t* **`value`**`)` [Function]
Determine if a value is temporary. i.e. its scope extends over a single block within its function.

`int jit_value_is_local (`*jit_value_t* **`value`**`)` [Function]
Determine if a value is local. i.e. its scope extends over multiple blocks within its function.

`int jit_value_is_constant (`*jit_value_t* **`value`**`)` [Function]
Determine if a value is a constant.

`int jit_value_is_parameter (`*jit_value_t* **`value`**`)` [Function]
Determine if a value is a function parameter.

`void jit_value_ref (`*jit_function_t* `func`, *jit_value_t* `value``)` [Function]
Create a reference to the specified *value* from the current block in *func*. This will convert a temporary value into a local value if *value* is being referenced from a different block than its original.

It is not necessary that *func* be the same function as the one where the value was originally created. It may be a nested function, referring to a local variable in its parent function.

`void jit_value_set_volatile (`*jit_value_t* `value``)` [Function]
Set a flag on a value to indicate that it is volatile. The contents of the value must always be reloaded from memory, never from a cached register copy.

`int jit_value_is_volatile (`*jit_value_t* `value``)` [Function]
Determine if a value is volatile.

`void jit_value_set_addressable (`*jit_value_t* `value``)` [Function]
Set a flag on a value to indicate that it is addressable. This should be used when you want to take the address of a value (e.g. `&variable` in C). The value is guaranteed to not be stored in a register across a function call. If you refer to a value from a nested function (`jit_value_ref`), then the value will be automatically marked as addressable.

`int jit_value_is_addressable (`*jit_value_t* `value``)` [Function]
Determine if a value is addressable.

`jit_type_t jit_value_get_type (`*jit_value_t* `value``)` [Function]
Get the type that is associated with a value.

`jit_function_t jit_value_get_function (`*jit_value_t* `value``)` [Function]
Get the function which owns a particular *value*.

`jit_block_t jit_value_get_block (`*jit_value_t* `value``)` [Function]
Get the block which owns a particular *value*.

`jit_context_t jit_value_get_context (`*jit_value_t* `value``)` [Function]
Get the context which owns a particular *value*.

`jit_constant_t jit_value_get_constant (`*jit_value_t* `value``)` [Function]
Get the constant value within a particular *value*. The returned structure's `type` field will be `jit_type_void` if `value` is not a constant.

`jit_nint jit_value_get_nint_constant (`*jit_value_t* `value``)` [Function]
Get the constant value within a particular *value*, assuming that its type is compatible with `jit_type_nint`.

`jit_nint jit_value_get_long_constant (`*jit_value_t* `value``)` [Function]
Get the constant value within a particular *value*, assuming that its type is compatible with `jit_type_long`.

`jit_float32 jit_value_get_float32_constant (`*jit_value_t* `value``)` [Function]
Get the constant value within a particular *value*, assuming that its type is compatible with `jit_type_float32`.

`jit_float64 jit_value_get_float64_constant (jit_value_t value)` [Function]
Get the constant value within a particular *value*, assuming that its type is compatible with `jit_type_float64`.

`jit_nfloat jit_value_get_nfloat_constant (jit_value_t value)` [Function]
Get the constant value within a particular *value*, assuming that its type is compatible with `jit_type_nfloat`.

`int jit_value_is_true (jit_value_t value)` [Function]
Determine if *value* is constant and non-zero.

`int jit_constant_convert (jit_constant_t *result, const jit_constant_t *value, jit_type_t type, int overflow_check)` [Function]
Convert a the constant *value* into a new *type*, and return its value in *result*. Returns zero if the conversion is not possible, usually due to overflow.

8 Working with instructions in the JIT

`int jit_insn_get_opcode (jit_insn_t insn)` [Function]
Get the opcode that is associated with an instruction.

`jit_value_t jit_insn_get_dest (jit_insn_t insn)` [Function]
Get the destination value that is associated with an instruction. Returns NULL if the instruction does not have a destination.

`jit_value_t jit_insn_get_value1 (jit_insn_t insn)` [Function]
Get the first argument value that is associated with an instruction. Returns NULL if the instruction does not have a first argument value.

`jit_value_t jit_insn_get_value2 (jit_insn_t insn)` [Function]
Get the second argument value that is associated with an instruction. Returns NULL if the instruction does not have a second argument value.

`jit_label_t jit_insn_get_label (jit_insn_t insn)` [Function]
Get the label for a branch target from an instruction. Returns NULL if the instruction does not have a branch target.

`jit_function_t jit_insn_get_function (jit_insn_t insn)` [Function]
Get the function for a call instruction. Returns NULL if the instruction does not refer to a called function.

`void * jit_insn_get_native (jit_insn_t insn)` [Function]
Get the function pointer for a native call instruction. Returns NULL if the instruction does not refer to a native function call.

`const char * jit_insn_get_name (jit_insn_t insn)` [Function]
Get the diagnostic name for a function call. Returns NULL if the instruction does not have a diagnostic name.

`jit_type_t jit_insn_get_signature` (*jit_insn_t* ***insn***) [Function]
Get the signature for a function call instruction. Returns NULL if the instruction is not a function call.

`int jit_insn_dest_is_value` (*jit_insn_t* ***insn***) [Function]
Returns a non-zero value if the destination for *insn* is actually a source value. This can happen with instructions such as `jit_insn_store_relative` where the instruction needs three source operands, and the real destination is a side-effect on one of the sources.

`void jit_insn_label` (*jit_function_t* ***func***, *jit_label_t* ****label***) [Function]
Start a new block within the function *func* and give it the specified *label*. Returns zero if out of memory.

If the contents of *label* are `jit_label_undefined`, then this function will allocate a new label for this block. Otherwise it will reuse the specified label from a previous branch instruction.

`int jit_insn_new_block` (*jit_function_t* ***func***) [Function]
Start a new basic block, without giving it an explicit label.

`jit_value_t jit_insn_load` (*jit_function_t* ***func***, *jit_value_t* ***value***) [Function]
Load the contents of *value* into a new temporary, essentially duplicating the value. Constants are not duplicated.

`jit_value_t jit_insn_dup` (*jit_function_t* ***func***, *jit_value_t* ***value***) [Function]
This is the same as `jit_insn_load`, but the name may better reflect how it is used in some front ends.

`jit_value_t jit_insn_load_small` (*jit_function_t* ***func***, *jit_value_t* ***value***) [Function]
If *value* is of type `sbyte`, `byte`, `short`, `ushort`, a structure, or a union, then make a copy of it and return the temporary copy. Otherwise return *value* as-is.

This is useful where you want to use *value* directly without duplicating it first. However, certain types usually cannot be operated on directly without first copying them elsewhere. This function will do that whenever necessary.

`void jit_insn_store` (*jit_function_t* ***func***, *jit_value_t* ***dest***, *jit_value_t* ***value***) [Function]
Store the contents of *value* at the location referred to by *dest*. The *dest* should be a `jit_value_t` representing a local variable or temporary. Use `jit_insn_store_relative` to store to a location referred to by a pointer.

`jit_value_t jit_insn_load_relative` (*jit_function_t* ***func***, *jit_value_t* ***value***, *jit_nint* ***offset***, *jit_type_t* ***type***) [Function]
Load a value of the specified *type* from the effective address `(value + offset)`, where *value* is a pointer.

`int jit_insn_store_relative` (*jit_function_t* ***func***, *jit_value_t* ***dest***, *jit_nint* ***offset***, *jit_value_t* ***value***) [Function]
Store *value* at the effective address `(dest + offset)`, where *dest* is a pointer.

`jit_value_t jit_insn_add_relative` (*jit_function_t* `func`, *jit_value_t* `value`, *jit_nint* `offset`) [Function]

Add the constant *offset* to the specified pointer *value*. This is functionally identical to calling `jit_insn_add`, but the JIT can optimize the code better if it knows that the addition is being used to perform a relative adjustment on a pointer. In particular, multiple relative adjustments on the same pointer can be collapsed into a single adjustment.

`jit_value_t jit_insn_load_elem` (*jit_function_t* `func`, *jit_value_t* `base_addr`, *jit_value_t* `index`, *jit_type_t* `elem_type`) [Function]

Load an element of type *elem_type* from position *index* within the array starting at *base_addr*. The effective address of the array element is `base_addr + index * sizeof(elem_type)`.

`jit_value_t jit_insn_load_elem_address` (*jit_function_t* `func`, *jit_value_t* `base_addr`, *jit_value_t* `index`, *jit_type_t* `elem_type`) [Function]

Load the effective address of an element of type *elem_type* at position *index* within the array starting at *base_addr*. Essentially, this computes the expression `base_addr + index * sizeof(elem_type)`, but may be more efficient than performing the steps with `jit_insn_mul` and `jit_insn_add`.

`int jit_insn_store_elem` (*jit_function_t* `func`, *jit_value_t* `base_addr`, *jit_value_t* `index`, *jit_value_t* `value`) [Function]

Store *value* at position *index* of the array starting at *base_addr*. The effective address of the storage location is `base_addr + index * sizeof(jit_value_get_type(value))`.

`int jit_insn_check_null` (*jit_function_t* `func`, *jit_value_t* `value`) [Function]

Check *value* to see if it is NULL. If it is, then throw the built-in `JIT_RESULT_NULL_REFERENCE` exception.

`jit_value_t jit_insn_add` (*jit_function_t* `func`, *jit_value_t* `value1`, *jit_value_t* `value2`) [Function]

Add two values together and return the result in a new temporary value.

`jit_value_t jit_insn_add_ovf` (*jit_function_t* `func`, *jit_value_t* `value1`, *jit_value_t* `value2`) [Function]

Add two values together and return the result in a new temporary value. Throw an exception if overflow occurs.

`jit_value_t jit_insn_sub` (*jit_function_t* `func`, *jit_value_t* `value1`, *jit_value_t* `value2`) [Function]

Subtract two values and return the result in a new temporary value.

`jit_value_t jit_insn_sub_ovf` (*jit_function_t* `func`, *jit_value_t* `value1`, *jit_value_t* `value2`) [Function]

Subtract two values and return the result in a new temporary value. Throw an exception if overflow occurs.

`jit_value_t jit_insn_mul` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Multiply two values and return the result in a new temporary value.

`jit_value_t jit_insn_mul_ovf` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Multiply two values and return the result in a new temporary value. Throw an exception if overflow occurs.

`jit_value_t jit_insn_div` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Divide two values and return the quotient in a new temporary value. Throws an exception on division by zero or arithmetic error (an arithmetic error is one where the minimum possible signed integer value is divided by -1).

`jit_value_t jit_insn_rem` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Divide two values and return the remainder in a new temporary value. Throws an exception on division by zero or arithmetic error (an arithmetic error is one where the minimum possible signed integer value is divided by -1).

`jit_value_t jit_insn_rem_ieee` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Divide two values and return the remainder in a new temporary value. Throws an exception on division by zero or arithmetic error (an arithmetic error is one where the minimum possible signed integer value is divided by -1). This function is identical to `jit_insn_rem`, except that it uses IEEE rules for computing the remainder of floating-point values.

`jit_value_t jit_insn_neg` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**) [Function]

Negate a value and return the result in a new temporary value.

`jit_value_t jit_insn_and` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Bitwise AND two values and return the result in a new temporary value.

`jit_value_t jit_insn_or` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Bitwise OR two values and return the result in a new temporary value.

`jit_value_t jit_insn_xor` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Bitwise XOR two values and return the result in a new temporary value.

`jit_value_t jit_insn_not` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**) [Function]

Bitwise NOT a value and return the result in a new temporary value.

`jit_value_t jit_insn_shl` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Perform a bitwise left shift on two values and return the result in a new temporary value.

`jit_value_t jit_insn_shr` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Perform a bitwise right shift on two values and return the result in a new temporary value. This performs a signed shift on signed operators, and an unsigned shift on unsigned operands.

`jit_value_t jit_insn_ushr` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Perform a bitwise right shift on two values and return the result in a new temporary value. This performs an unsigned shift on both signed and unsigned operands.

`jit_value_t jit_insn_sshr` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Perform a bitwise right shift on two values and return the result in a new temporary value. This performs an signed shift on both signed and unsigned operands.

`jit_value_t jit_insn_eq` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Compare two values for equality and return the result in a new temporary value.

`jit_value_t jit_insn_ne` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Compare two values for inequality and return the result in a new temporary value.

`jit_value_t jit_insn_lt` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Compare two values for less than and return the result in a new temporary value.

`jit_value_t jit_insn_le` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Compare two values for less than or equal and return the result in a new temporary value.

`jit_value_t jit_insn_gt` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Compare two values for greater than and return the result in a new temporary value.

`jit_value_t jit_insn_ge` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Compare two values for greater than or equal and return the result in a new temporary value.

`jit_value_t jit_insn_cmpl` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Compare two values, and return a -1, 0, or 1 result. If either value is "not a number", then -1 is returned.

`jit_value_t jit_insn_cmpg` (*jit_function_t* **`func`**, *jit_value_t* **`value1`**, *jit_value_t* **`value2`**) [Function]

Compare two values, and return a -1, 0, or 1 result. If either value is "not a number", then 1 is returned.

`jit_value_t jit_insn_to_bool` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]

Convert a value into a boolean 0 or 1 result of type `jit_type_int`.

`jit_value_t jit_insn_to_not_bool` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]

Convert a value into a boolean 1 or 0 result of type `jit_type_int` (i.e. the inverse of `jit_insn_to_bool`).

`jit_value_t jit_insn_acos` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_asin` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_atan` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_atan2` (*jit_function_t* **func**, *jit_value_t* **value1**, *jit_value_t* **value2**) [Function]
`jit_value_t jit_insn_ceil` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_cos` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_cosh` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_exp` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_floor` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_log` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_log10` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_pow` (*jit_function_t* **func**, *jit_value_t* **value1**, *jit_value_t* **value2**) [Function]
`jit_value_t jit_insn_rint` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_round` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_sin` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_sinh` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_sqrt` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_tan` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_tanh` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]

Apply a mathematical function to floating-point arguments.

`jit_value_t jit_insn_is_nan` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_is_finite` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_is_inf` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]

Test a floating point value for not a number, finite, or infinity.

`jit_value_t jit_insn_abs` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
`jit_value_t jit_insn_min` (*jit_function_t* **func**, *jit_value_t* **value1**, *jit_value_t* **value2**) [Function]
`jit_value_t jit_insn_max` (*jit_function_t* **func**, *jit_value_t* **value1**, *jit_value_t* **value2**) [Function]

`jit_value_t jit_insn_sign` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
Calculate the absolute value, minimum, maximum, or sign of the specified values.

`int jit_insn_branch` (*jit_function_t* **func**, *jit_label_t* ***label**) [Function]
Terminate the current block by branching unconditionally to a specific label. Returns zero if out of memory.

`int jit_insn_branch_if` (*jit_function_t* **func**, *jit_value_t* **value**, *jit_label_t* ***label**) [Function]
Terminate the current block by branching to a specific label if the specified value is non-zero. Returns zero if out of memory.

If *value* refers to a conditional expression that was created by `jit_insn_eq`, `jit_insn_ne`, etc, then the conditional expression will be replaced by an appropriate conditional branch instruction.

`int jit_insn_branch_if_not` (*jit_function_t* **func**, *jit_value_t* **value**, *jit_label_t* ***label**) [Function]
Terminate the current block by branching to a specific label if the specified value is zero. Returns zero if out of memory.

If *value* refers to a conditional expression that was created by `jit_insn_eq`, `jit_insn_ne`, etc, then the conditional expression will be replaced by an appropriate conditional branch instruction.

`int jit_insn_jump_table` (*jit_function_t* **func**, *jit_value_t* **value**, *jit_label_t* ***labels**, *unsigned int* **num_labels**) [Function]
Branch to a label from the *labels* table. The *value* is the index of the label. It is allowed to have identical labels in the table. If an entry in the table has `jit_label_undefined` value then it is replaced with a newly allocated label.

`jit_value_t jit_insn_address_of` (*jit_function_t* **func**, *jit_value_t* **value1**) [Function]
Get the address of a value into a new temporary.

`jit_value_t jit_insn_address_of_label` (*jit_function_t* **func**, *jit_label_t* ***label**) [Function]
Get the address of *label* into a new temporary. This is typically used for exception handling, to track where in a function an exception was actually thrown.

`jit_value_t jit_insn_convert` (*jit_function_t* **func**, *jit_value_t* **value**, *jit_type_t* **type**, *int* **overflow_check**) [Function]
Convert the contents of a value into a new type, with optional overflow checking.

`jit_value_t jit_insn_call` (*jit_function_t* **func**, *const char* ***name**, *jit_function_t* **jit_func**, *jit_type_t* **signature**, *jit_value_t* ***args**, *unsigned int* **num_args**, *int* **flags**) [Function]
Call the function *jit_func*, which may or may not be translated yet. The *name* is for diagnostic purposes only, and can be NULL.

If *signature* is NULL, then the actual signature of *jit_func* is used in its place. This is the usual case. However, if the function takes a variable number of arguments, then you may need to construct an explicit signature for the non-fixed argument values.

The *flags* parameter specifies additional information about the type of call to perform:

`JIT_CALL_NOTHROW`
: The function never throws exceptions.

`JIT_CALL_NORETURN`
: The function will never return directly to its caller. It may however return to the caller indirectly by throwing an exception that the caller catches.

`JIT_CALL_TAIL`
: Apply tail call optimizations, as the result of this function call will be immediately returned from the containing function. Tail calls are only appropriate when the signature of the called function matches the callee, and none of the parameters point to local variables.

If *jit_func* has already been compiled, then `jit_insn_call` may be able to intuit some of the above flags for itself. Otherwise it is up to the caller to determine when the flags may be appropriate.

`jit_value_t jit_insn_call_indirect` (*jit_function_t* **`func`**, *jit_value_t* **`value`**, *jit_type_t* **`signature`**, *jit_value_t* ***`args`**, *unsigned int* **`num_args`**, *int* **`flags`**) [Function]

Call a function via an indirect pointer.

`jit_value_t jit_insn_call_indirect_vtable` (*jit_function_t* **`func`**, *jit_value_t* **`value`**, *jit_type_t* **`signature`**, *jit_value_t* ***`args`**, *unsigned int* **`num_args`**, *int* **`flags`**) [Function]

Call a function via an indirect pointer. This version differs from `jit_insn_call_indirect` in that we assume that *value* contains a pointer that resulted from calling `jit_function_to_vtable_pointer`. Indirect vtable pointer calls may be more efficient on some platforms than regular indirect calls.

`jit_value_t jit_insn_call_native` (*jit_function_t* **`func`**, *const char* ***`name`**, *void* ***`native_func`**, *jit_type_t* **`signature`**, *jit_value_t* ***`args`**, *unsigned int* **`num_args`**, *int* **`exception_return`**, *int* **`flags`**) [Function]

Output an instruction that calls an external native function. The *name* is for diagnostic purposes only, and can be NULL.

`jit_value_t jit_insn_call_intrinsic` (*jit_function_t* **`func`**, *const char* ***`name`**, *void* ***`intrinsic_func`**, *const jit_intrinsic_descr_t* ***`descriptor`**, *jit_value_t* **`arg1`**, *jit_value_t* **`arg2`**) [Function]

Output an instruction that calls an intrinsic function. The descriptor contains the following fields:

`return_type`
: The type of value that is returned from the intrinsic.

`ptr_result_type`
: This should be NULL for an ordinary intrinsic, or the result type if the intrinsic reports exceptions.

`arg1_type`
: The type of the first argument.

`arg2_type`
: The type of the second argument, or NULL for a unary intrinsic.

If all of the arguments are constant, then `jit_insn_call_intrinsic` will call the intrinsic directly to calculate the constant result. If the constant computation will result in an exception, then code is output to cause the exception at runtime.

The *name* is for diagnostic purposes only, and can be NULL.

`int jit_insn_incoming_reg (`*jit_function_t* **`func`**`, `*jit_value_t* **`value`**`, `*int* **`reg`**`)` [Function]

Output an instruction that notes that the contents of *value* can be found in the register *reg* at this point in the code.

You normally wouldn't call this yourself - it is used internally by the CPU back ends to set up the function's entry frame and the values of registers on return from a subroutine call.

`int jit_insn_incoming_frame_posn (`*jit_function_t* **`func`**`, `*jit_value_t* **`value`**`, `*jit_nint* **`frame_offset`**`)` [Function]

Output an instruction that notes that the contents of *value* can be found in the stack frame at *frame_offset* at this point in the code.

You normally wouldn't call this yourself - it is used internally by the CPU back ends to set up the function's entry frame.

`int jit_insn_outgoing_reg (`*jit_function_t* **`func`**`, `*jit_value_t* **`value`**`, `*int* **`reg`**`)` [Function]

Output an instruction that copies the contents of *value* into the register *reg* at this point in the code. This is typically used just before making an outgoing subroutine call.

You normally wouldn't call this yourself - it is used internally by the CPU back ends to set up the registers for a subroutine call.

`int jit_insn_outgoing_frame_posn (`*jit_function_t* **`func`**`, `*jit_value_t* **`value`**`, `*jit_nint* **`frame_offset`**`)` [Function]

Output an instruction that notes that the contents of *value* should be stored in the stack frame at *frame_offset* at this point in the code.

You normally wouldn't call this yourself - it is used internally by the CPU back ends to set up an outgoing frame for tail calls.

`int jit_insn_return_reg (`*jit_function_t* **`func`**`, `*jit_value_t* **`value`**`, `*int* **`reg`**`)` [Function]

Output an instruction that notes that the contents of *value* can be found in the register *reg* at this point in the code. This is similar to `jit_insn_incoming_reg`, except that it refers to return values, not parameter values.

You normally wouldn't call this yourself - it is used internally by the CPU back ends to handle returns from subroutine calls.

`int jit_insn_setup_for_nested (`*jit_function_t* `func`, *int* `nested_level`, *int* `reg``)` [Function]

Output an instruction to set up for a nested function call. The *nested_level* value will be -1 to call a child, zero to call a sibling of *func*, 1 to call a sibling of the parent, 2 to call a sibling of the grandparent, etc. If *reg* is not -1, then it indicates the register to receive the parent frame information. If *reg* is -1, then the frame information will be pushed on the stack.

You normally wouldn't call this yourself - it is used internally by the CPU back ends to set up the parameters for a nested subroutine call.

`int jit_insn_flush_struct (`*jit_function_t* `func`, *jit_value_t* `value``)` [Function]

Flush a small structure return value out of registers and back into the local variable frame. You normally wouldn't call this yourself - it is used internally by the CPU back ends to handle structure returns from functions.

`jit_value_t jit_insn_import (`*jit_function_t* `func`, *jit_value_t* `value``)` [Function]

Import *value* from an outer nested scope into *func*. Returns the effective address of the value for local access via a pointer. Returns NULL if out of memory or the value is not accessible via a parent, grandparent, or other ancestor of *func*.

`int jit_insn_push (`*jit_function_t* `func`, *jit_value_t* `value``)` [Function]

Push a value onto the function call stack, in preparation for a call. You normally wouldn't call this yourself - it is used internally by the CPU back ends to set up the stack for a subroutine call.

`int jit_insn_push_ptr (`*jit_function_t* `func`, *jit_value_t* `value`, *jit_type_t* `type``)` [Function]

Push `*value` onto the function call stack, in preparation for a call. This is normally used for returning `struct` and `union` values where you have the effective address of the structure, rather than the structure's contents, in *value*.

You normally wouldn't call this yourself - it is used internally by the CPU back ends to set up the stack for a subroutine call.

`int jit_insn_set_param (`*jit_function_t* `func`, *jit_value_t* `value`, *jit_nint* `offset``)` [Function]

Set the parameter slot at *offset* in the outgoing parameter area to *value*. This may be used instead of `jit_insn_push` if it is more efficient to store directly to the stack than to push. The outgoing parameter area is allocated within the frame when the function is first entered.

You normally wouldn't call this yourself - it is used internally by the CPU back ends to set up the stack for a subroutine call.

`int jit_insn_set_param_ptr (`*jit_function_t* `func`, *jit_value_t* `value`, *jit_type_t* `type`, *jit_nint* `offset``)` [Function]

Same as `jit_insn_set_param_ptr`, except that the parameter is at `*value`.

`int jit_insn_push_return_area_ptr (`*jit_function_t* `func``)` [Function]
Push the interpreter's return area pointer onto the stack. You normally wouldn't call this yourself - it is used internally by the CPU back ends to set up the stack for a subroutine call.

`int jit_insn_pop_stack (`*jit_function_t* `func`, *jit_nint* `num_items``)` [Function]
Pop *num_items* items from the function call stack. You normally wouldn't call this yourself - it is used by CPU back ends to clean up the stack after calling a subroutine. The size of an item is specific to the back end (it could be bytes, words, or some other measurement).

`int jit_insn_defer_pop_stack (`*jit_function_t* `func`, *jit_nint* `num_items``)` [Function]
This is similar to `jit_insn_pop_stack`, except that it tries to defer the pop as long as possible. Multiple subroutine calls may result in parameters collecting up on the stack, and only being popped at the next branch or label instruction. You normally wouldn't call this yourself - it is used by CPU back ends.

`int jit_insn_flush_defer_pop (`*jit_function_t* `func`, *jit_nint* `num_items``)` [Function]
Flush any deferred items that were scheduled for popping by `jit_insn_defer_pop_stack` if there are *num_items* or more items scheduled. You normally wouldn't call this yourself - it is used by CPU back ends to clean up the stack just prior to a subroutine call when too many items have collected up. Calling `jit_insn_flush_defer_pop(func, 0)` will flush all deferred items.

`int jit_insn_return (`*jit_function_t* `func`, *jit_value_t* `value``)` [Function]
Output an instruction to return *value* as the function's result. If *value* is NULL, then the function is assumed to return `void`. If the function returns a structure, this will copy the value into the memory at the structure return address.

`int jit_insn_return_ptr (`*jit_function_t* `func`, *jit_value_t* `value`, *jit_type_t* `type``)` [Function]
Output an instruction to return `*value` as the function's result. This is normally used for returning `struct` and `union` values where you have the effective address of the structure, rather than the structure's contents, in *value*.

`int jit_insn_default_return (`*jit_function_t* `func``)` [Function]
Add an instruction to return a default value if control reaches this point. This is typically used at the end of a function to ensure that all paths return to the caller. Returns zero if out of memory, 1 if a default return was added, and 2 if a default return was not needed.

Note: if this returns 1, but the function signature does not return `void`, then it indicates that a higher-level language error has occurred and the function should be abandoned.

`int jit_insn_throw (`*jit_function_t* `func`, *jit_value_t* `value``)` [Function]
Throw a pointer *value* as an exception object. This can also be used to "rethrow" an object from a catch handler that is not interested in handling the exception.

`jit_value_t jit_insn_get_call_stack` (*jit_function_t* ***func***) [Function]
Get an object that represents the current position in the code, and all of the functions that are currently on the call stack. This is equivalent to calling `jit_exception_get_stack_trace`, and is normally used just prior to `jit_insn_throw` to record the location of the exception that is being thrown.

`jit_value_t jit_insn_thrown_exception` (*jit_function_t* ***func***) [Function]
Get the value that holds the most recent thrown exception. This is typically used in `catch` clauses.

`int jit_insn_uses_catcher` (*jit_function_t* ***func***) [Function]
Notify the function building process that *func* contains some form of `catch` clause for catching exceptions. This must be called before any instruction that is covered by a `try`, ideally at the start of the function output process.

`jit_value_t jit_insn_start_catcher` (*jit_function_t* ***func***) [Function]
Start the catcher block for *func*. There should be exactly one catcher block for any function that involves a `try`. All exceptions that are thrown within the function will cause control to jump to this point. Returns a value that holds the exception that was thrown.

`int jit_insn_branch_if_pc_not_in_range` (*jit_function_t* ***func***, [Function]
jit_label_t ***start_label***, *jit_label_t* ***end_label***, *jit_label_t* ****label***)
Branch to *label* if the program counter where an exception occurred does not fall between *start_label* and *end_label*.

`int jit_insn_rethrow_unhandled` (*jit_function_t* ***func***) [Function]
Rethrow the current exception because it cannot be handled by any of the `catch` blocks in the current function.

Note: this is intended for use within catcher blocks. It should not be used to rethrow exceptions in response to programmer requests (e.g. `throw;` in C#). The `jit_insn_throw` function should be used for that purpose.

`int jit_insn_start_finally` (*jit_function_t* ***func***, *jit_label_t* [Function]
****finally_label***)
Start a `finally` clause.

`int jit_insn_return_from_finally` (*jit_function_t* ***func***) [Function]
Return from the `finally` clause to where it was called from. This is usually the last instruction in a `finally` clause.

`int jit_insn_call_finally` (*jit_function_t* ***func***, *jit_label_t* [Function]
****finally_label***)
Call a `finally` clause.

`jit_value_t jit_insn_start_filter` (*jit_function_t* ***func***, *jit_label_t* [Function]
****label***, *jit_type_t* ***type***)
Define the start of a filter. Filters are embedded subroutines within functions that are used to filter exceptions in `catch` blocks.

A filter subroutine takes a single argument (usually a pointer) and returns a single result (usually a boolean). The filter has complete access to the local variables of the function, and can use any of them in the filtering process.

This function returns a temporary value of the specified *type*, indicating the parameter that is supplied to the filter.

`int jit_insn_return_from_filter (`*jit_function_t* **`func`**, *jit_value_t* **`value`**`)` [Function]

Return from a filter subroutine with the specified `value` as its result.

`jit_value_t jit_insn_call_filter (`*jit_function_t* **`func`**, *jit_label_t* ***`label`**, *jit_value_t* **`value`**, *jit_type_t* **`type`**`)` [Function]

Call the filter subroutine at *label*, passing it *value* as its argument. This function returns a value of the specified *type*, indicating the filter's result.

`int jit_insn_memcpy (`*jit_function_t* **`func`**, *jit_value_t* **`dest`**, *jit_value_t* **`src`**, *jit_value_t* **`size`**`)` [Function]

Copy the *size* bytes of memory at *src* to *dest*. It is assumed that the source and destination do not overlap.

`int jit_insn_memmove (`*jit_function_t* **`func`**, *jit_value_t* **`dest`**, *jit_value_t* **`src`**, *jit_value_t* **`size`**`)` [Function]

Copy the *size* bytes of memory at *src* to *dest*. This is save to use if the source and destination overlap.

`int jit_insn_memset (`*jit_function_t* **`func`**, *jit_value_t* **`dest`**, *jit_value_t* **`value`**, *jit_value_t* **`size`**`)` [Function]

Set the *size* bytes at *dest* to *value*.

`jit_value_t jit_insn_alloca (`*jit_function_t* **`func`**, *jit_value_t* **`size`**`)` [Function]

Allocate *size* bytes of memory from the stack.

`int jit_insn_move_blocks_to_end (`*jit_function_t* **`func`**, *jit_label_t* **`from_label`**, *jit_label_t* **`to_label`**`)` [Function]

Move all of the blocks between *from_label* (inclusive) and *to_label* (exclusive) to the end of the current function. This is typically used to move the expression in a `while` loop to the end of the body, where it can be executed more efficiently.

`int jit_insn_move_blocks_to_start (`*jit_function_t* **`func`**, *jit_label_t* **`from_label`**, *jit_label_t* **`to_label`**`)` [Function]

Move all of the blocks between *from_label* (inclusive) and *to_label* (exclusive) to the start of the current function. This is typically used to move initialization code to the head of the function.

`int jit_insn_mark_offset (`*jit_function_t* **`func`**, *jit_int* **`offset`**`)` [Function]

Mark the current position in *func* as corresponding to the specified bytecode *offset*. This value will be returned by `jit_stack_trace_get_offset`, and is useful for associating code positions with source line numbers.

`void jit_insn_iter_init (`*jit_insn_iter_t* ***`iter`**, *jit_block_t* **`block`**`)` [Function]

Initialize an iterator to point to the first instruction in *block*.

`void jit_insn_iter_init_last (`*jit_insn_iter_t* `*iter`, *jit_block_t* `block``)` [Function]
Initialize an iterator to point to the last instruction in *block*.

`jit_insn_t jit_insn_iter_next (`*jit_insn_iter_t* `*iter``)` [Function]
Get the next instruction in an iterator's block. Returns NULL when there are no further instructions in the block.

`jit_insn_t jit_insn_iter_previous (`*jit_insn_iter_t* `*iter``)` [Function]
Get the previous instruction in an iterator's block. Returns NULL when there are no further instructions in the block.

9 Working with basic blocks in the JIT

`jit_function_t jit_block_get_function (`*jit_block_t* `block``)` [Function]
Get the function that a particular *block* belongs to.

`jit_context_t jit_block_get_context (`*jit_block_t* `block``)` [Function]
Get the context that a particular *block* belongs to.

`jit_label_t jit_block_get_label (`*jit_block_t* `block``)` [Function]
Get the label associated with a block.

`jit_block_t jit_block_next (`*jit_function_t* `func`, *jit_block_t* `previous``)` [Function]
Iterate over the blocks in a function, in order of their creation. The *previous* argument should be NULL on the first call. This function will return NULL if there are no further blocks to iterate.

`jit_block_t jit_block_previous (`*jit_function_t* `func`, *jit_block_t* `previous``)` [Function]
Iterate over the blocks in a function, in reverse order of their creation. The *previous* argument should be NULL on the first call. This function will return NULL if there are no further blocks to iterate.

`jit_block_t jit_block_from_label (`*jit_function_t* `func`, *jit_label_t* `label``)` [Function]
Get the block that corresponds to a particular *label*. Returns NULL if there is no block associated with the label.

`int jit_block_set_meta (`*jit_block_t* `block`, *int* `type`, *void* `*data`, *jit_meta_free_func* `free_data``)` [Function]
Tag a block with some metadata. Returns zero if out of memory. If the *type* already has some metadata associated with it, then the previous value will be freed. Metadata may be used to store dependency graphs, branch prediction information, or any other information that is useful to optimizers or code generators.

Metadata type values of 10000 or greater are reserved for internal use.

`void * jit_block_get_meta` (*jit_block_t* **block**, *int* **type**) [Function]
Get the metadata associated with a particular tag. Returns NULL if *type* does not have any metadata associated with it.

`void jit_block_free_meta` (*jit_block_t* **block**, *int* **type**) [Function]
Free metadata of a specific type on a block. Does nothing if the *type* does not have any metadata associated with it.

`int jit_block_is_reachable` (*jit_block_t* **block**) [Function]
Determine if a block is reachable from some other point in its function. Unreachable blocks can be discarded in their entirety. If the JIT is uncertain as to whether a block is reachable, or it does not wish to perform expensive flow analysis to find out, then it will err on the side of caution and assume that it is reachable.

`int jit_block_ends_in_dead` (*jit_block_t* **block**) [Function]
Determine if a block ends in a "dead" marker. That is, control will not fall out through the end of the block.

`int jit_block_current_is_dead` (*jit_function_t* **func**) [Function]
Determine if the current point in the function is dead. That is, there are no existing branches or fall-throughs to this point. This differs slightly from `jit_block_ends_in_dead` in that this can skip past zero-length blocks that may not appear to be dead to find the dead block at the head of a chain of empty blocks.

10 Intrinsic functions available to libjit users

Intrinsics are functions that are provided to ease code generation on platforms that may not be able to perform all operations natively.

For example, on a CPU without a floating-point unit, the back end code generator will output a call to an intrinsic function when a floating-point operation is performed. CPU's with a floating-point unit would use a native instruction instead.

Some platforms may already have appropriate intrinsics (e.g. the ARM floating-point emulation routines). The back end code generator may choose to use either the system-supplied intrinsics or the ones supplied by this library. We supply all of them in our library just in case a particular platform lacks an appropriate intrinsic.

Some intrinsics have no equivalent in existing system libraries; particularly those that deal with overflow checking.

Functions that perform overflow checking or which divide integer operands return a built-in exception code to indicate the type of exception to be thrown (the caller is responsible for throwing the actual exception). See Chapter 11 [Exceptions], page 59, for a list of built-in exception codes.

The following functions are defined in `<jit/jit-intrinsic.h>`:

`jit_int jit_int_add` (*jit_int* **value1**, *jit_int* **value2**) [Function]
`jit_int jit_int_sub` (*jit_int* **value1**, *jit_int* **value2**) [Function]
`jit_int jit_int_mul` (*jit_int* **value1**, *jit_int* **value2**) [Function]
`jit_int jit_int_neg` (*jit_int* **value1**) [Function]

`jit_int jit_int_and` (*jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_or` (*jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_xor` (*jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_not` (*jit_int* `value1`) [Function]
`jit_int jit_int_not` (*jit_int* `value1`) [Function]
`jit_int jit_int_shl` (*jit_int* `value1`, *jit_uint* `value2`) [Function]
`jit_int jit_int_shr` (*jit_int* `value1`, *jit_uint* `value2`) [Function]
Perform an arithmetic operation on signed 32-bit integers.

`jit_int jit_int_add_ovf` (*jit_int* *`result`, *jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_sub_ovf` (*jit_int* *`result`, *jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_mul_ovf` (*jit_int* *`result`, *jit_int* `value1`, *jit_int* `value2`) [Function]
Perform an arithmetic operation on two signed 32-bit integers, with overflow checking. Returns `JIT_RESULT_OK` or `JIT_RESULT_OVERFLOW`.

`jit_int jit_int_div_ovf` (*jit_int* *`result`, *jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_rem_ovf` (*jit_int* *`result`, *jit_int* `value1`, *jit_int* `value2`) [Function]
Perform a division or remainder operation on two signed 32-bit integers. Returns `JIT_RESULT_OK`, `JIT_RESULT_DIVISION_BY_ZERO`, or `JIT_RESULT_ARITHMETIC`.

`jit_int jit_int_eq` (*jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_ne` (*jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_lt` (*jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_le` (*jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_gt` (*jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_ge` (*jit_int* `value1`, *jit_int* `value2`) [Function]
Compare two signed 32-bit integers, returning 0 or 1 based on their relationship.

`jit_int jit_int_cmp` (*jit_int* `value1`, *jit_int* `value2`) [Function]
Compare two signed 32-bit integers and return -1, 0, or 1 based on their relationship.

`jit_int jit_int_abs` (*jit_int* `value1`) [Function]
`jit_int jit_int_min` (*jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_max` (*jit_int* `value1`, *jit_int* `value2`) [Function]
`jit_int jit_int_sign` (*jit_int* `value1`) [Function]
Calculate the absolute value, minimum, maximum, or sign for signed 32-bit integer values.

`jit_uint jit_uint_add` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_uint jit_uint_sub` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_uint jit_uint_mul` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_uint jit_uint_neg` (*jit_uint* `value1`) [Function]
`jit_uint jit_uint_and` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_uint jit_uint_or` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]

`jit_uint jit_uint_xor` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_uint jit_uint_not` (*jit_uint* `value1`) [Function]
`jit_uint jit_uint_not` (*jit_uint* `value1`) [Function]
`jit_uint jit_uint_shl` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_uint jit_uint_shr` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
Perform an arithmetic operation on unsigned 32-bit integers.

`jit_int jit_uint_add_ovf` (*jit_uint* *`result`, *jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_int jit_uint_sub_ovf` (*jit_uint* *`result`, *jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_int jit_uint_mul_ovf` (*jit_uint* *`result`, *jit_uint* `value1`, *jit_uint* `value2`) [Function]
Perform an arithmetic operation on two unsigned 32-bit integers, with overflow checking. Returns `JIT_RESULT_OK` or `JIT_RESULT_OVERFLOW`.

`jit_int jit_uint_div_ovf` (*jit_uint* *`result`, *jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_int jit_uint_rem_ovf` (*jit_uint* *`result`, *jit_uint* `value1`, *jit_uint* `value2`) [Function]
Perform a division or remainder operation on two unsigned 32-bit integers. Returns `JIT_RESULT_OK` or `JIT_RESULT_DIVISION_BY_ZERO` (`JIT_RESULT_ARITHMETIC` is not possible with unsigned integers).

`jit_int jit_uint_eq` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_int jit_uint_ne` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_int jit_uint_lt` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_int jit_uint_le` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_int jit_uint_gt` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_int jit_uint_ge` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
Compare two unsigned 32-bit integers, returning 0 or 1 based on their relationship.

`jit_int jit_uint_cmp` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
Compare two unsigned 32-bit integers and return -1, 0, or 1 based on their relationship.

`jit_uint jit_uint_min` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
`jit_uint jit_uint_max` (*jit_uint* `value1`, *jit_uint* `value2`) [Function]
Calculate the minimum or maximum for unsigned 32-bit integer values.

`jit_long jit_long_add` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_long jit_long_sub` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_long jit_long_mul` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_long jit_long_neg` (*jit_long* `value1`) [Function]
`jit_long jit_long_and` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_long jit_long_or` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_long jit_long_xor` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_long jit_long_not` (*jit_long* `value1`) [Function]
`jit_long jit_long_not` (*jit_long* `value1`) [Function]

`jit_long jit_long_shl` (*jit_long* `value1`, *jit_uint* `value2`) [Function]
`jit_long jit_long_shr` (*jit_long* `value1`, *jit_uint* `value2`) [Function]
Perform an arithmetic operation on signed 64-bit integers.

`jit_int jit_long_add_ovf` (*jit_long* `*result`, *jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_int jit_long_sub_ovf` (*jit_long* `*result`, *jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_int jit_long_mul_ovf` (*jit_long* `*result`, *jit_long* `value1`, *jit_long* `value2`) [Function]
Perform an arithmetic operation on two signed 64-bit integers, with overflow checking. Returns `JIT_RESULT_OK` or `JIT_RESULT_OVERFLOW`.

`jit_int jit_long_div_ovf` (*jit_long* `*result`, *jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_int jit_long_rem_ovf` (*jit_long* `*result`, *jit_long* `value1`, *jit_long* `value2`) [Function]
Perform a division or remainder operation on two signed 64-bit integers. Returns `JIT_RESULT_OK`, `JIT_RESULT_DIVISION_BY_ZERO`, or `JIT_RESULT_ARITHMETIC`.

`jit_int jit_long_eq` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_int jit_long_ne` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_int jit_long_lt` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_int jit_long_le` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_int jit_long_gt` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_int jit_long_ge` (*jit_long* `value1`, *jit_long* `value2`) [Function]
Compare two signed 64-bit integers, returning 0 or 1 based on their relationship.

`jit_int jit_long_cmp` (*jit_long* `value1`, *jit_long* `value2`) [Function]
Compare two signed 64-bit integers and return -1, 0, or 1 based on their relationship.

`jit_long jit_long_abs` (*jit_long* `value1`) [Function]
`jit_long jit_long_min` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_long jit_long_max` (*jit_long* `value1`, *jit_long* `value2`) [Function]
`jit_int jit_long_sign` (*jit_long* `value1`) [Function]
Calculate the absolute value, minimum, maximum, or sign for signed 64-bit integer values.

`jit_ulong jit_ulong_add` (*jit_ulong* `value1`, *jit_ulong* `value2`) [Function]
`jit_ulong jit_ulong_sub` (*jit_ulong* `value1`, *jit_ulong* `value2`) [Function]
`jit_ulong jit_ulong_mul` (*jit_ulong* `value1`, *jit_ulong* `value2`) [Function]
`jit_ulong jit_ulong_neg` (*jit_ulong* `value1`) [Function]
`jit_ulong jit_ulong_and` (*jit_ulong* `value1`, *jit_ulong* `value2`) [Function]
`jit_ulong jit_ulong_or` (*jit_ulong* `value1`, *jit_ulong* `value2`) [Function]
`jit_ulong jit_ulong_xor` (*jit_ulong* `value1`, *jit_ulong* `value2`) [Function]
`jit_ulong jit_ulong_not` (*jit_ulong* `value1`) [Function]
`jit_ulong jit_ulong_not` (*jit_ulong* `value1`) [Function]
`jit_ulong jit_ulong_shl` (*jit_ulong* `value1`, *jit_uint* `value2`) [Function]
`jit_ulong jit_ulong_shr` (*jit_ulong* `value1`, *jit_uint* `value2`) [Function]
Perform an arithmetic operation on unsigned 64-bit integers.

`jit_int jit_ulong_add_ovf` (*jit_ulong* ***result***, *jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]
`jit_int jit_ulong_sub_ovf` (*jit_ulong* ***result***, *jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]
`jit_int jit_ulong_mul_ovf` (*jit_ulong* ***result***, *jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]

Perform an arithmetic operation on two unsigned 64-bit integers, with overflow checking. Returns `JIT_RESULT_OK` or `JIT_RESULT_OVERFLOW`.

`jit_int jit_ulong_div_ovf` (*jit_ulong* ***result***, *jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]
`jit_int jit_ulong_rem_ovf` (*jit_ulong* ***result***, *jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]

Perform a division or remainder operation on two unsigned 64-bit integers. Returns `JIT_RESULT_OK` or `JIT_RESULT_DIVISION_BY_ZERO` (`JIT_RESULT_ARITHMETIC` is not possible with unsigned integers).

`jit_int jit_ulong_eq` (*jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]
`jit_int jit_ulong_ne` (*jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]
`jit_int jit_ulong_lt` (*jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]
`jit_int jit_ulong_le` (*jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]
`jit_int jit_ulong_gt` (*jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]
`jit_int jit_ulong_ge` (*jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]

Compare two unsigned 64-bit integers, returning 0 or 1 based on their relationship.

`jit_int jit_ulong_cmp` (*jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]

Compare two unsigned 64-bit integers and return -1, 0, or 1 based on their relationship.

`jit_ulong jit_ulong_min` (*jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]
`jit_ulong jit_ulong_max` (*jit_ulong* ***value1***, *jit_ulong* ***value2***) [Function]

Calculate the minimum or maximum for unsigned 64-bit integer values.

`jit_float32 jit_float32_add` (*jit_float32* ***value1***, *jit_float32* ***value2***) [Function]
`jit_float32 jit_float32_sub` (*jit_float32* ***value1***, *jit_float32* ***value2***) [Function]
`jit_float32 jit_float32_mul` (*jit_float32* ***value1***, *jit_float32* ***value2***) [Function]
`jit_float32 jit_float32_div` (*jit_float32* ***value1***, *jit_float32* ***value2***) [Function]
`jit_float32 jit_float32_rem` (*jit_float32* ***value1***, *jit_float32* ***value2***) [Function]
`jit_float32 jit_float32_ieee_rem` (*jit_float32* ***value1***, *jit_float32* ***value2***) [Function]
`jit_float32 jit_float32_neg` (*jit_float32* ***value1***) [Function]

Perform an arithmetic operation on 32-bit floating-point values.

`jit_int jit_float32_eq` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
`jit_int jit_float32_ne` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
`jit_int jit_float32_lt` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
`jit_int jit_float32_le` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
`jit_int jit_float32_gt` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
`jit_int jit_float32_ge` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
Compare two 32-bit floating-point values, returning 0 or 1 based on their relationship.

`jit_int jit_float32_cmpl` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
Compare two 32-bit floating-point values and return -1, 0, or 1 based on their relationship. If either value is "not a number", then -1 is returned.

`jit_int jit_float32_cmpg` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
Compare two 32-bit floating-point values and return -1, 0, or 1 based on their relationship. If either value is "not a number", then 1 is returned.

`jit_float32 jit_float32_abs` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_min` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
`jit_float32 jit_float32_max` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
`jit_int jit_float32_sign` (*jit_float32* `value1`) [Function]
Calculate the absolute value, minimum, maximum, or sign for 32-bit floating point values.

`jit_float32 jit_float32_acos` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_asin` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_atan` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_atan2` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
`jit_float32 jit_float32_ceil` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_cos` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_cosh` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_exp` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_floor` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_log` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_log10` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_pow` (*jit_float32* `value1`, *jit_float32* `value2`) [Function]
`jit_float32 jit_float32_sin` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_sinh` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_sqrt` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_tan` (*jit_float32* `value1`) [Function]
`jit_float32 jit_float32_tanh` (*jit_float32* `value1`) [Function]
Apply a mathematical function to one or two 32-bit floating-point values.

`jit_float32 jit_float32_rint` (*jit_float32* `value1`) [Function]
Round *value1* to the nearest integer. Half-way cases are rounded to an even number.

`jit_float32 jit_float32_round` (*jit_float32* `value1`) [Function]
Round *value1* to the nearest integer. Half-way cases are rounded away from zero.

`jit_int jit_float32_is_finite` (*jit_float32* `value`) [Function]
Determine if a 32-bit floating point value is finite, returning non-zero if it is, or zero if it is not. If the value is "not a number", this function returns zero.

`jit_int jit_float32_is_nan` (*jit_float32* `value`) [Function]
Determine if a 32-bit floating point value is "not a number", returning non-zero if it is, or zero if it is not.

`jit_int jit_float32_is_inf` (*jit_float32* `value`) [Function]
Determine if a 32-bit floating point value is infinite or not. Returns -1 for negative infinity, 1 for positive infinity, and 0 for everything else.

Note: this function is preferable to the system `isinf` intrinsic because some systems have a broken `isinf` function that returns 1 for both positive and negative infinity.

`jit_float64 jit_float64_add` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
`jit_float64 jit_float64_sub` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
`jit_float64 jit_float64_mul` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
`jit_float64 jit_float64_div` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
`jit_float64 jit_float64_rem` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
`jit_float64 jit_float64_ieee_rem` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
`jit_float64 jit_float64_neg` (*jit_float64* `value1`) [Function]
Perform an arithmetic operation on 64-bit floating-point values.

`jit_int jit_float64_eq` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
`jit_int jit_float64_ne` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
`jit_int jit_float64_lt` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
`jit_int jit_float64_le` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
`jit_int jit_float64_gt` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
`jit_int jit_float64_ge` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
Compare two 64-bit floating-point values, returning 0 or 1 based on their relationship.

`jit_int jit_float64_cmpl` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
Compare two 64-bit floating-point values and return -1, 0, or 1 based on their relationship. If either value is "not a number", then -1 is returned.

`jit_int jit_float64_cmpg` (*jit_float64* `value1`, *jit_float64* `value2`) [Function]
Compare two 64-bit floating-point values and return -1, 0, or 1 based on their relationship. If either value is "not a number", then 1 is returned.

`jit_float64 jit_float64_abs (`*jit_float64* `value1``)` [Function]
`jit_float64 jit_float64_min (`*jit_float64* `value1`, *jit_float64* `value2``)` [Function]
`jit_float64 jit_float64_max (`*jit_float64* `value1`, *jit_float64* `value2``)` [Function]
`jit_int jit_float64_sign (`*jit_float64* `value1`)` [Function]
Calculate the absolute value, minimum, maximum, or sign for 64-bit floating point values.

`jit_float64 jit_float64_acos (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_asin (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_atan (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_atan2 (`*jit_float64* `value1`, *jit_float64* `value2`)` [Function]
`jit_float64 jit_float64_ceil (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_cos (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_cosh (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_exp (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_floor (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_log (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_log10 (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_pow (`*jit_float64* `value1`, *jit_float64* `value2`)` [Function]
`jit_float64 jit_float64_sin (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_sinh (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_sqrt (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_tan (`*jit_float64* `value1`)` [Function]
`jit_float64 jit_float64_tanh (`*jit_float64* `value1`)` [Function]
Apply a mathematical function to one or two 64-bit floating-point values.

`jit_float64 jit_float64_rint (`*jit_float64* `value1`)` [Function]
Round *value1* to the nearest integer. Half-way cases are rounded to an even number.

`jit_float64 jit_float64_round (`*jit_float64* `value1`)` [Function]
Round *value1* to the nearest integer. Half-way cases are rounded away from zero.

`jit_int jit_float64_is_finite (`*jit_float64* `value`)` [Function]
Determine if a 64-bit floating point value is finite, returning non-zero if it is, or zero if it is not. If the value is "not a number", this function returns zero.

`jit_int jit_float64_is_nan (`*jit_float64* `value`)` [Function]
Determine if a 64-bit floating point value is "not a number", returning non-zero if it is, or zero if it is not.

`jit_int jit_float64_is_inf (`*jit_float64* `value`)` [Function]
Determine if a 64-bit floating point value is infinite or not. Returns -1 for negative infinity, 1 for positive infinity, and 0 for everything else.

Note: this function is preferable to the system `isinf` intrinsic because some systems have a broken `isinf` function that returns 1 for both positive and negative infinity.

`jit_nfloat jit_nfloat_add (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
`jit_nfloat jit_nfloat_sub (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
`jit_nfloat jit_nfloat_mul (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
`jit_nfloat jit_nfloat_div (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
`jit_nfloat jit_nfloat_rem (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
`jit_nfloat jit_nfloat_ieee_rem (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
`jit_nfloat jit_nfloat_neg (`*jit_nfloat* `value1``)` [Function]
Perform an arithmetic operation on native floating-point values.

`jit_int jit_nfloat_eq (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
`jit_int jit_nfloat_ne (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
`jit_int jit_nfloat_lt (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
`jit_int jit_nfloat_le (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
`jit_int jit_nfloat_gt (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
`jit_int jit_nfloat_ge (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
Compare two native floating-point values, returning 0 or 1 based on their relationship.

`jit_int jit_nfloat_cmpl (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
Compare two native floating-point values and return -1, 0, or 1 based on their relationship. If either value is "not a number", then -1 is returned.

`jit_int jit_nfloat_cmpg (`*jit_nfloat* `value1`, *jit_nfloat* `value2``)` [Function]
Compare two native floating-point values and return -1, 0, or 1 based on their relationship. If either value is "not a number", then 1 is returned.

`jit_nfloat jit_nfloat_abs (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_min (`*jit_nfloat* `value1`, *jit_nfloat* `value2```)` [Function]
`jit_nfloat jit_nfloat_max (`*jit_nfloat* `value1`, *jit_nfloat* `value2```)` [Function]
`jit_int jit_nfloat_sign (`*jit_nfloat* `value1```)` [Function]
Calculate the absolute value, minimum, maximum, or sign for native floating point values.

`jit_nfloat jit_nfloat_acos (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_asin (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_atan (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_atan2 (`*jit_nfloat* `value1`, *jit_nfloat* `value2```)` [Function]
`jit_nfloat jit_nfloat_ceil (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_cos (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_cosh (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_exp (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_floor (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_log (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_log10 (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_pow (`*jit_nfloat* `value1`, *jit_nfloat* `value2```)` [Function]
`jit_nfloat jit_nfloat_sin (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_sinh (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_sqrt (`*jit_nfloat* `value1```)` [Function]
`jit_nfloat jit_nfloat_tan (`*jit_nfloat* `value1```)` [Function]

`jit_nfloat jit_nfloat_tanh` (*jit_nfloat* `value1`) [Function]
Apply a mathematical function to one or two native floating-point values.

`jit_nfloat jit_nfloat_rint` (*jit_nfloat* `value1`) [Function]
Round *value1* to the nearest integer. Half-way cases are rounded to an even number.

`jit_nfloat jit_nfloat_round` (*jit_nfloat* `value1`) [Function]
Round *value1* to the nearest integer. Half-way cases are rounded away from zero.

`jit_int jit_nfloat_is_finite` (*jit_nfloat* `value`) [Function]
Determine if a native floating point value is finite, returning non-zero if it is, or zero if it is not. If the value is "not a number", this function returns zero.

`jit_int jit_nfloat_is_nan` (*jit_nfloat* `value`) [Function]
Determine if a native floating point value is "not a number", returning non-zero if it is, or zero if it is not.

`jit_int jit_nfloat_is_inf` (*jit_nfloat* `value`) [Function]
Determine if a native floating point value is infinite or not. Returns -1 for negative infinity, 1 for positive infinity, and 0 for everything else.

Note: this function is preferable to the system `isinf` intrinsic because some systems have a broken `isinf` function that returns 1 for both positive and negative infinity.

`jit_int jit_int_to_sbyte` (*jit_int* `value`) [Function]
`jit_int jit_int_to_ubyte` (*jit_int* `value`) [Function]
`jit_int jit_int_to_short` (*jit_int* `value`) [Function]
`jit_int jit_int_to_ushort` (*jit_int* `value`) [Function]
`jit_int jit_int_to_int` (*jit_int* `value`) [Function]
`jit_uint jit_int_to_uint` (*jit_int* `value`) [Function]
`jit_long jit_int_to_long` (*jit_int* `value`) [Function]
`jit_ulong jit_int_to_ulong` (*jit_int* `value`) [Function]
`jit_int jit_uint_to_int` (*jit_uint* `value`) [Function]
`jit_uint jit_uint_to_uint` (*jit_uint* `value`) [Function]
`jit_long jit_uint_to_long` (*jit_uint* `value`) [Function]
`jit_ulong jit_uint_to_ulong` (*jit_uint* `value`) [Function]
`jit_int jit_long_to_int` (*jit_long* `value`) [Function]
`jit_uint jit_long_to_uint` (*jit_long* `value`) [Function]
`jit_long jit_long_to_long` (*jit_long* `value`) [Function]
`jit_ulong jit_long_to_ulong` (*jit_long* `value`) [Function]
`jit_int jit_ulong_to_int` (*jit_ulong* `value`) [Function]
`jit_uint jit_ulong_to_uint` (*jit_ulong* `value`) [Function]
`jit_long jit_ulong_to_long` (*jit_ulong* `value`) [Function]
`jit_ulong jit_ulong_to_ulong` (*jit_ulong* `value`) [Function]
Convert between integer types.

`jit_int jit_int_to_sbyte_ovf` (*jit_int* *`result`, *jit_int* `value`) [Function]
`jit_int jit_int_to_ubyte_ovf` (*jit_int* *`result`, *jit_int* `value`) [Function]
`jit_int jit_int_to_short_ovf` (*jit_int* *`result`, *jit_int* `value`) [Function]
`jit_int jit_int_to_ushort_ovf` (*jit_int* *`result`, *jit_int* `value`) [Function]

`jit_int jit_int_to_int_ovf (jit_int *result, jit_int value)` [Function]
`jit_int jit_int_to_uint_ovf (jit_uint *result, jit_int value)` [Function]
`jit_int jit_int_to_long_ovf (jit_long *result, jit_int value)` [Function]
`jit_int jit_int_to_ulong_ovf (jit_ulong *result, jit_int value)` [Function]
`jit_int jit_uint_to_int_ovf (jit_int *result, jit_uint value)` [Function]
`jit_int jit_uint_to_uint_ovf (jit_uint *result, jit_uint value)` [Function]
`jit_int jit_uint_to_long_ovf (jit_long *result, jit_uint value)` [Function]
`jit_int jit_uint_to_ulong_ovf (jit_ulong *result, jit_uint value)` [Function]
`jit_int jit_long_to_int_ovf (jit_int *result, jit_long value)` [Function]
`jit_int jit_long_to_uint_ovf (jit_uint *result, jit_long value)` [Function]
`jit_int jit_long_to_long_ovf (jit_long *result, jit_long value)` [Function]
`jit_int jit_long_to_ulong_ovf (jit_ulong *result, jit_long value)` [Function]
`jit_int jit_ulong_to_int_ovf (jit_int *result, jit_ulong value)` [Function]
`jit_int jit_ulong_to_uint_ovf (jit_uint *result, jit_ulong value)` [Function]
`jit_int jit_ulong_to_long_ovf (jit_long *result, jit_ulong value)` [Function]
`jit_int jit_ulong_to_ulong_ovf (jit_ulong *result, jit_ulong value)` [Function]

Convert between integer types with overflow detection.

`jit_int jit_nfloat_to_int (jit_nfloat value)` [Function]
`jit_uint jit_nfloat_to_uint (jit_nfloat value)` [Function]
`jit_long jit_nfloat_to_long (jit_nfloat value)` [Function]
`jit_ulong jit_nfloat_to_ulong (jit_nfloat value)` [Function]

Convert a native floating-point value into an integer.

`jit_int jit_nfloat_to_int_ovf (jit_int *result, jit_nfloat value)` [Function]
`jit_uint jit_nfloat_to_uint_ovf (jit_uint *result, jit_nfloat value)` [Function]
`jit_long jit_nfloat_to_long_ovf (jit_long *result, jit_nfloat value)` [Function]
`jit_ulong jit_nfloat_to_ulong_ovf (jit_ulong *result, jit_nfloat value)` [Function]

Convert a native floating-point value into an integer, with overflow detection. Returns `JIT_RESULT_OK` if the conversion was successful or `JIT_RESULT_OVERFLOW` if an overflow occurred.

`jit_nfloat jit_int_to_nfloat (jit_int value)` [Function]
`jit_nfloat jit_uint_to_nfloat (jit_uint value)` [Function]
`jit_nfloat jit_long_to_nfloat (jit_long value)` [Function]
`jit_nfloat jit_ulong_to_nfloat (jit_ulong value)` [Function]

Convert an integer into native floating-point value.

`jit_nfloat jit_float32_to_nfloat (jit_float32 value)` [Function]
`jit_nfloat jit_float64_to_nfloat (jit_float64 value)` [Function]
`jit_float32 jit_nfloat_to_float32 (jit_nfloat value)` [Function]
`jit_float64 jit_nfloat_to_float64 (jit_nfloat value)` [Function]

Convert between floating-point types.

11 Handling exceptions

`void * jit_exception_get_last` (*void*) [Function]
Get the last exception object that occurred on this thread, or NULL if there is no exception object on this thread. As far as `libjit` is concerned, an exception is just a pointer. The precise meaning of the data at the pointer is determined by the front end.

`void * jit_exception_get_last_and_clear` (*void*) [Function]
Get the last exception object that occurred on this thread and also clear the exception state to NULL. This combines the effect of both `jit_exception_get_last` and `jit_exception_clear_last`.

`void jit_exception_set_last` (*void *object*) [Function]
Set the last exception object that occurred on this thread, so that it can be retrieved by a later call to `jit_exception_get_last`. This is normally used by `jit_function_apply` to save the exception object before returning to regular code.

`void jit_exception_clear_last` (*void*) [Function]
Clear the last exception object that occurred on this thread. This is equivalent to calling `jit_exception_set_last` with a parameter of NULL.

`void jit_exception_throw` (*void *object*) [Function]
Throw an exception object within the current thread. As far as `libjit` is concerned, the exception object is just a pointer. The precise meaning of the data at the pointer is determined by the front end.

Note: as an exception object works its way back up the stack, it may be temporarily stored in memory that is not normally visible to a garbage collector. The front-end is responsible for taking steps to "pin" the object so that it is uncollectable until explicitly copied back into a location that is visible to the collector once more.

`void jit_exception_builtin` (*int exception_type*) [Function]
This function is called to report a builtin exception. The JIT will automatically embed calls to this function wherever a builtin exception needs to be reported.

When a builtin exception occurs, the current thread's exception handler is called to construct an appropriate object, which is then thrown.

If there is no exception handler set, or the handler returns NULL, then `libjit` will print an error message to stderr and cause the program to exit with a status of 1. You normally don't want this behavior and you should override it if possible.

The following builtin exception types are currently supported:

`JIT_RESULT_OK`
: The operation was performed successfully (value is 1).

`JIT_RESULT_OVERFLOW`
: The operation resulted in an overflow exception (value is 0).

`JIT_RESULT_ARITHMETIC`
: The operation resulted in an arithmetic exception. i.e. an attempt was made to divide the minimum integer value by -1 (value is -1).

`JIT_RESULT_DIVISION_BY_ZERO`
: The operation resulted in a division by zero exception (value is -2).

`JIT_RESULT_COMPILE_ERROR`
: An error occurred when attempting to dynamically compile a function (value is -3).

`JIT_RESULT_OUT_OF_MEMORY`
: The system ran out of memory while performing an operation (value is -4).

`JIT_RESULT_NULL_REFERENCE`
: An attempt was made to dereference a NULL pointer (value is -5).

`JIT_RESULT_NULL_FUNCTION`
: An attempt was made to call a function with a NULL function pointer (value is -6).

`JIT_RESULT_CALLED_NESTED`
: An attempt was made to call a nested function from a non-nested context (value is -7).

`jit_exception_func jit_exception_set_handler` [Function]
(*jit_exception_func* **`handler`**)

Set the builtin exception handler for the current thread. Returns the previous exception handler.

`jit_exception_func jit_exception_get_handler` (*void*) [Function]

Get the builtin exception handler for the current thread.

`jit_stack_trace_t jit_exception_get_stack_trace` (*void*) [Function]

Create an object that represents the current call stack. This is normally used to indicate the location of an exception. Returns NULL if a stack trace is not available, or there is insufficient memory to create it.

`unsigned int jit_stack_trace_get_size` (*jit_stack_trace_t* **`trace`**) [Function]

Get the size of a stack trace.

`jit_function_t jit_stack_trace_get_function` (*jit_context_t* **`context`**, *jit_stack_trace_t* **`trace`**, *unsigned int* **`posn`**) [Function]

Get the function that is at position *posn* within a stack trace. Position 0 is the function that created the stack trace. If this returns NULL, then it indicates that there is a native callout at *posn* within the stack trace.

`void * jit_stack_trace_get_pc` (*jit_stack_trace_t* **`trace`**, *unsigned int* **`posn`**) [Function]

Get the program counter that corresponds to position *posn* within a stack trace. This is the point within the function where execution had reached at the time of the trace.

`unsigned int jit_stack_trace_get_offset` (*jit_stack_trace_t* **`trace`**, *unsigned int* **`posn`**) [Function]

Get the bytecode offset that is recorded for position *posn* within a stack trace. This will be `JIT_NO_OFFSET` if there is no bytecode offset associated with *posn*.

`void jit_stack_trace_free (`*jit_stack_trace_t* `trace``)` [Function]
Free the memory associated with a stack trace.

12 Hooking a breakpoint debugger into libjit

The `libjit` library provides support routines for breakpoint-based single-step debugging. It isn't a full debugger, but provides the infrastructure necessary to support one.

The front end virtual machine is responsible for inserting "potential breakpoints" into the code when functions are built and compiled. This is performed using `jit_insn_mark_breakpoint`:

`int jit_insn_mark_breakpoint (`*jit_function_t* `func`, *jit_nint* `data1`, *jit_nint* `data2``)` [Function]
Mark the current position in *func* as corresponding to a breakpoint location. When a break occurs, the debugging routines are passed *func*, *data1*, and *data2* as arguments. By convention, *data1* is the type of breakpoint (source line, function entry, function exit, etc).

There are two ways for a front end to receive notification about breakpoints. The bulk of this chapter describes the `jit_debugger_t` interface, which handles most of the ugly details. In addition, a low-level "debug hook mechanism" is provided for front ends that wish more control over the process. The debug hook mechanism is described below, under the `jit_debugger_set_hook` function.

This debugger implementation requires a threading system to work successfully. At least two threads are required, in addition to those of the program being debugged:

1. Event thread which calls `jit_debugger_wait_event` to receive notifications of breakpoints and other interesting events.
2. User interface thread which calls functions like `jit_debugger_run`, `jit_debugger_step`, etc, to control the debug process.

These two threads should be set to "unbreakable" with a call to `jit_debugger_set_breakable`. This prevents them from accidentally stopping at a breakpoint, which would cause a system deadlock. Other housekeeping threads, such as a finalization thread, should also be set to "unbreakable" for the same reason.

Events have the following members:

`type` The type of event (see the next table for details).

`thread` The thread that the event occurred on.

`function` The function that the breakpoint occurred within.

`data1`
`data2` The data values at the breakpoint. These values are inserted into the function's code with `jit_insn_mark_breakpoint`.

`id` The identifier for the breakpoint.

`trace` The stack trace corresponding to the location where the breakpoint occurred. This value is automatically freed upon the next call to `jit_debugger_wait_event`. If you wish to preserve the value, then you must call `jit_stack_trace_copy`.

The following event types are currently supported:

`JIT_DEBUGGER_TYPE_QUIT`
A thread called `jit_debugger_quit`, indicating that it wanted the event thread to terminate.

`JIT_DEBUGGER_TYPE_HARD_BREAKPOINT`
A thread stopped at a hard breakpoint. That is, a breakpoint defined by a call to `jit_debugger_add_breakpoint`.

`JIT_DEBUGGER_TYPE_SOFT_BREAKPOINT`
A thread stopped at a breakpoint that wasn't explicitly defined by a call to `jit_debugger_add_breakpoint`. This typicaly results from a call to a "step" function like `jit_debugger_step`, where execution stopped at the next line but there isn't an explicit breakpoint on that line.

`JIT_DEBUGGER_TYPE_USER_BREAKPOINT`
A thread stopped because of a call to `jit_debugger_break`.

`JIT_DEBUGGER_TYPE_ATTACH_THREAD`
A thread called `jit_debugger_attach_self`. The `data1` field of the event is set to the value of `stop_immediately` for the call.

`JIT_DEBUGGER_TYPE_DETACH_THREAD`
A thread called `jit_debugger_detach_self`.

`int jit_insn_mark_breakpoint_variable` (*jit_function_t* ***func***, *jit_value_t* ***data1***, *jit_value_t* ***data2***) [Function]
This function is similar to `jit_insn_mark_breakpoint` except that values in *data1* and *data2* can be computed at runtime. You can use this function for example to get address of local variable.

`int jit_debugging_possible` (*void*) [Function]
Determine if debugging is possible. i.e. that threading is available and compatible with the debugger's requirements.

`jit_debugger_t jit_debugger_create` (*jit_context_t* ***context***) [Function]
Create a new debugger instance and attach it to a JIT *context*. If the context already has a debugger associated with it, then this function will return the previous debugger.

`void jit_debugger_destroy` (*jit_debugger_t* ***dbg***) [Function]
Destroy a debugger instance.

`jit_context_t jit_debugger_get_context` (*jit_debugger_t* ***dbg***) [Function]
Get the JIT context that is associated with a debugger instance.

`jit_debugger_t jit_debugger_from_context (`*jit_context_t* `context``)` [Function]

Get the debugger that is currently associated with a JIT *context*, or NULL if there is no debugger associated with the context.

`jit_debugger_thread_id_t jit_debugger_get_self (`*jit_debugger_t* `dbg``)` [Function]

Get the thread identifier associated with the current thread. The return values are normally values like 1, 2, 3, etc, allowing the user interface to report messages like "thread 3 has stopped at a breakpoint".

`jit_debugger_thread_id_t jit_debugger_get_thread` [Function]
`(`*jit_debugger_t* `dbg`, *const void* *`native_thread`)

Get the thread identifier for a specific native thread. The *native_thread* pointer is assumed to point at a block of memory containing a native thread handle. This would be a `pthread_t` on Pthreads platforms or a `HANDLE` on Win32 platforms. If the native thread has not been seen previously, then a new thread identifier is allocated.

`int jit_debugger_get_native_thread (`*jit_debugger_t* `dbg`, [Function]
jit_debugger_thread_id_t `thread`, *void* *`native_thread`)

Get the native thread handle associated with a debugger thread identifier. Returns non-zero if OK, or zero if the debugger thread identifier is not yet associated with a native thread handle.

`void jit_debugger_set_breakable (`*jit_debugger_t* `dbg`, *const void* [Function]
*`native_thread`, *int* `flag`)

Set a flag that indicates if a native thread can stop at breakpoints. If set to 1 (the default), breakpoints will be active on the thread. If set to 0, breakpoints will be ignored on the thread. Typically this is used to mark threads associated with the debugger's user interface, or the virtual machine's finalization thread, so that they aren't accidentally suspended by the debugger (which might cause a deadlock).

`void jit_debugger_attach_self (`*jit_debugger_t* `dbg`, *int* [Function]
`stop_immediately`)

Attach the current thread to a debugger. If *stop_immediately* is non-zero, then the current thread immediately suspends, waiting for the user to start it with `jit_debugger_run`. This function is typically called in a thread's startup code just before any "real work" is performed.

`void jit_debugger_detach_self (`*jit_debugger_t* `dbg`) [Function]

Detach the current thread from the debugger. This is typically called just before the thread exits.

`int jit_debugger_wait_event (`*jit_debugger_t* `dbg`, [Function]
jit_debugger_event_t *`event`, *jit_nint* `timeout`)

Wait for the next debugger event to arrive. Debugger events typically indicate breakpoints that have occurred. The *timeout* is in milliseconds, or -1 for an infinite timeout period. Returns non-zero if an event has arrived, or zero on timeout.

`jit_debugger_breakpoint_id_t jit_debugger_add_breakpoint` [Function]
(*jit_debugger_t* ***dbg***, *jit_debugger_breakpoint_info_t* ***info***)

Add a hard breakpoint to a debugger instance. The *info* structure defines the conditions under which the breakpoint should fire. The fields of *info* are as follows:

`flags` Flags that indicate which of the following fields should be matched. If a flag is not present, then all possible values of the field will match. Valid flags are `JIT_DEBUGGER_FLAG_THREAD`, `JIT_DEBUGGER_FLAG_FUNCTION`, `JIT_DEBUGGER_FLAG_DATA1`, and `JIT_DEBUGGER_FLAG_DATA2`.

`thread` The thread to match against, if `JIT_DEBUGGER_FLAG_THREAD` is set.

`function` The function to match against, if `JIT_DEBUGGER_FLAG_FUNCTION` is set.

`data1` The `data1` value to match against, if `JIT_DEBUGGER_FLAG_DATA1` is set.

`data2` The `data2` value to match against, if `JIT_DEBUGGER_FLAG_DATA2` is set.

The following special values for `data1` are recommended for marking breakpoint locations with `jit_insn_mark_breakpoint`:

`JIT_DEBUGGER_DATA1_LINE`
Breakpoint location that corresponds to a source line. This is used to determine where to continue to upon a "step".

`JIT_DEBUGGER_DATA1_ENTER`
Breakpoint location that corresponds to the start of a function.

`JIT_DEBUGGER_DATA1_LEAVE`
Breakpoint location that corresponds to the end of a function, just prior to a `return` statement. This is used to determine where to continue to upon a "finish".

`JIT_DEBUGGER_DATA1_THROW`
Breakpoint location that corresponds to an exception throw.

`void jit_debugger_remove_breakpoint` (*jit_debugger_t* ***dbg***, *jit_debugger_breakpoint_id_t* ***id***) [Function]

Remove a previously defined breakpoint from a debugger instance.

`void jit_debugger_remove_all_breakpoints` (*jit_debugger_t* ***dbg***) [Function]

Remove all breakpoints from a debugger instance.

`int jit_debugger_is_alive` (*jit_debugger_t* ***dbg***, *jit_debugger_thread_id_t* ***thread***) [Function]

Determine if a particular thread is still alive.

`int jit_debugger_is_running` (*jit_debugger_t* ***dbg***, *jit_debugger_thread_id_t* ***thread***) [Function]

Determine if a particular thread is currently running (non-zero) or stopped (zero).

`void jit_debugger_run` (*jit_debugger_t* ***dbg***, *jit_debugger_thread_id_t* ***thread***) [Function]

Start the specified thread running, or continue from the last breakpoint.

This function, and the others that follow, sends a request to the specified thread and then returns to the caller immediately.

`void jit_debugger_step` (*jit_debugger_t* ***dbg***, *jit_debugger_thread_id_t* ***thread***) [Function]

Step over a single line of code. If the line performs a method call, then this will step into the call. The request will be ignored if the thread is currently running.

`void jit_debugger_next` (*jit_debugger_t* ***dbg***, *jit_debugger_thread_id_t* ***thread***) [Function]

Step over a single line of code but do not step into method calls. The request will be ignored if the thread is currently running.

`void jit_debugger_finish` (*jit_debugger_t* ***dbg***, *jit_debugger_thread_id_t* ***thread***) [Function]

Keep running until the end of the current function. The request will be ignored if the thread is currently running.

`void jit_debugger_break` (*jit_debugger_t* ***dbg***) [Function]

Force an explicit user breakpoint at the current location within the current thread. Control returns to the caller when the debugger calls one of the above "run" or "step" functions in another thread.

`void jit_debugger_quit` (*jit_debugger_t* ***dbg***) [Function]

Sends a request to the thread that called `jit_debugger_wait_event` indicating that the debugger should quit.

`jit_debugger_hook_func jit_debugger_set_hook` (*jit_context_t* ***context***, *jit_debugger_hook_func* ***hook***) [Function]

Set a debugger hook on a JIT context. Returns the previous hook.

Debug hooks are a very low-level breakpoint mechanism. Upon reaching each breakpoint in a function, a user-supplied hook function is called. It is up to the hook function to decide whether to stop execution or to ignore the breakpoint. The hook function has the following prototype:

```
void hook(jit_function_t func, jit_nint data1, jit_nint data2);
```

The `func` argument indicates the function that the breakpoint occurred within. The `data1` and `data2` arguments are those supplied to `jit_insn_mark_breakpoint`. The debugger can use these values to indicate information about the breakpoint's type and location.

Hook functions can be used for other purposes besides breakpoint debugging. For example, a program could be instrumented with hooks that tally up the number of times that each function is called, or which profile the amount of time spent in each function.

By convention, `data1` values less than 10000 are intended for use by user-defined hook functions. Values of 10000 and greater are reserved for the full-blown debugger system described earlier.

13 Manipulating ELF binaries

The `libjit` library contains routines that permit pre-compiling JIT'ed functions into an on-disk representation. This representation can be loaded at some future time, to avoid the overhead of compiling the functions at runtime.

We use the ELF format for this purpose, which is a common binary format used by modern operating systems and compilers.

It isn't necessary for your operating system to be based on ELF natively. We use our own routines to read and write ELF binaries. We chose ELF because it has all of the features that we require, and reusing an existing format was better than inventing a completely new one.

13.1 Reading ELF binaries

`int jit_readelf_open (`*jit_readelf_t* `*readelf`, *const char* `*filename`, *int* `force``)` [Function]

Open the specified *filename* and load the ELF binary that is contained within it. Returns one of the following result codes:

`JIT_READELF_OK`
: The ELF binary was opened successfully.

`JIT_READELF_CANNOT_OPEN`
: Could not open the file at the filesystem level (reason in `errno`).

`JIT_READELF_NOT_ELF`
: The file was opened, but it is not an ELF binary.

`JIT_READELF_WRONG_ARCH`
: The file is an ELF binary, but it does not pertain to the architecture of this machine.

`JIT_READELF_BAD_FORMAT`
: The file is an ELF binary, but the format is corrupted in some fashion.

`JIT_READELF_MEMORY`
: There is insufficient memory to open the ELF binary.

The following flags may be supplied to alter the manner in which the ELF binary is loaded:

`JIT_READELF_FLAG_FORCE`
: Force `jit_readelf_open` to open the ELF binary, even if the architecture does not match this machine. Useful for debugging.

`JIT_READELF_FLAG_DEBUG`
: Print additional debug information to stdout.

`void jit_readelf_close (`*jit_readelf_t* `readelf``)` [Function]

Close an ELF reader, reclaiming all of the memory that was used.

`const char * jit_readelf_get_name` (*jit_readelf_t* **`readelf`**) [Function]

Get the library name that is embedded inside an ELF binary. ELF binaries can refer to each other using this name.

`void *jit_readelf_get_symbol` (*jit_readelf_t* **`readelf`**, *const char* ***`name`**) [Function]

Look up the symbol called *name* in the ELF binary represented by *readelf*. Returns NULL if the symbol is not present.

External references from this ELF binary to others are not resolved until the ELF binary is loaded into a JIT context using `jit_readelf_add_to_context` and `jit_readelf_resolve_all`. You should not call functions within this ELF binary until after you have fully resolved it.

`void * jit_readelf_get_section` (*jit_readelf_t* **`readelf`**, *const char* ***`name`**, *jit_nuint* ***`size`**) [Function]

Get the address and size of a particular section from an ELF binary. Returns NULL if the section is not present in the ELF binary.

The virtual machine may have stored auxillary information in the section when the binary was first generated. This function allows the virtual machine to retrieve its auxillary information.

Examples of such information may be version numbers, timestamps, checksums, and other identifying information for the bytecode that was previously compiled by the virtual machine. The virtual machine can use this to determine if the ELF binary is up to date and relevant to its needs.

It is recommended that virtual machines prefix their special sections with a unique string (e.g. `.foovm`) to prevent clashes with system-defined section names. The prefix `.libjit` is reserved for use by `libjit` itself.

`void * jit_readelf_get_section_by_type` (*jit_readelf_t* **`readelf`**, *jit_int* **`type`**, *jit_nuint* ***`size`**) [Function]

Get a particular section using its raw ELF section type (i.e. one of the `SHT_*` constants in `jit-elf-defs.h`). This is mostly for internal use, but some virtual machines may find it useful for debugging purposes.

`void * jit_readelf_map_vaddr` (*jit_readelf_t* **`readelf`**, *jit_nuint* **`vaddr`**) [Function]

Map a virtual address to an actual address in a loaded ELF binary. Returns NULL if *vaddr* could not be mapped.

`unsigned int jit_readelf_num_needed` (*jit_readelf_t* **`readelf`**) [Function]

Get the number of dependent libraries that are needed by this ELF binary. The virtual machine will normally need to arrange to load these libraries with `jit_readelf_open` as well, so that all of the necessary symbols can be resolved.

`const char * jit_readelf_get_needed` (*jit_readelf_t* **`readelf`**, *unsigned int* **`index`**) [Function]

Get the name of the dependent library at position *index* within the needed libraries list of this ELF binary. Returns NULL if the *index* is invalid.

`void jit_readelf_add_to_context (`*jit_readelf_t* **`readelf`**`,` *jit_context_t* **`context`**`)` [Function]

Add this ELF binary to a JIT context, so that its contents can be used when executing JIT-managed code. The binary will be closed automatically if the context is destroyed and `jit_readelf_close` has not been called explicitly yet.

The functions in the ELF binary cannot be used until you also call `jit_readelf_resolve_all` to resolve cross-library symbol references. The reason why adding and resolution are separate steps is to allow for resolving circular dependencies between ELF binaries.

`int jit_readelf_resolve_all (`*jit_context_t* **`context`**`,` *int* **`print_failures`**`)` [Function]

Resolve all of the cross-library symbol references in ELF binaries that have been added to *context* but which were not resolved in the previous call to this function. If *print_failures* is non-zero, then diagnostic messages will be written to stdout for any symbol resolutions that fail.

Returns zero on failure, or non-zero if all symbols were successfully resolved. If there are no ELF binaries awaiting resolution, then this function will return a non-zero result.

`int jit_readelf_register_symbol (`*jit_context_t* **`context`**`,` *const char* ***`name`**`,` *void* ***`value`**`,` *int* **`after`**`)` [Function]

Register *value* with *name* on the specified *context*. Whenever symbols are resolved with `jit_readelf_resolve_all`, and the symbol *name* is encountered, *value* will be substituted. Returns zero if out of memory or there is something wrong with the parameters.

If *after* is non-zero, then *name* will be resolved after all other ELF libraries; otherwise it will be resolved before the ELF libraries.

This function is used to register intrinsic symbols that are specific to the front end virtual machine. References to intrinsics within `libjit` itself are resolved automatically.

14 Miscellaneous utility routines

The `libjit` library provides a number of utility routines that it itself uses internally, but which may also be useful to front ends.

14.1 Memory allocation

The `libjit` library provides an interface to the traditional system `malloc` routines. All heap allocation in `libjit` goes through these functions. If you need to perform some other kind of memory allocation, you can replace these functions with your own versions.

`void * jit_malloc (`*unsigned int* **`size`**`)` [Function]

Allocate *size* bytes of memory from the heap.

`type * jit_new (type)` [Function]
Allocate `sizeof(type)` bytes of memory from the heap and cast the return pointer to `type *`. This is a macro that wraps up the underlying `jit_malloc` function and is less error-prone when allocating structures.

`void * jit_calloc (unsigned int num, unsigned int size)` [Function]
Allocate `num * size` bytes of memory from the heap and clear them to zero.

`type * jit_cnew (type)` [Function]
Allocate `sizeof(type)` bytes of memory from the heap and cast the return pointer to `type *`. The memory is cleared to zero.

`void * jit_realloc (void *ptr, unsigned int size)` [Function]
Re-allocate the memory at *ptr* to be *size* bytes in size. The memory block at *ptr* must have been allocated by a previous call to `jit_malloc`, `jit_calloc`, or `jit_realloc`.

`void jit_free (void *ptr)` [Function]
Free the memory at *ptr*. It is safe to pass a NULL pointer.

`void * jit_malloc_exec (unsigned int size)` [Function]
Allocate a block of memory that is read/write/executable. Such blocks are used to store JIT'ed code, function closures, and other trampolines. The size should be a multiple of `jit_exec_page_size()`.

This will usually be identical to `jit_malloc`. However, some systems may need special handling to create executable code segments, so this function must be used instead.

You must never mix regular and executable segment allocation. That is, do not use `jit_free` to free the result of `jit_malloc_exec`.

`void jit_free_exec (void *ptr, unsigned int size)` [Function]
Free a block of memory that was previously allocated by `jit_malloc_exec`. The *size* must be identical to the original allocated size, as some systems need to know this information to be able to free the block.

`void jit_flush_exec (void *ptr, unsigned int size)` [Function]
Flush the contents of the block at *ptr* from the CPU's data and instruction caches. This must be used after the code is written to an executable code segment, but before the code is executed, to prepare it for execution.

`unsigned int jit_exec_page_size (void)` [Function]
Get the page allocation size for the system. This is the preferred unit when making calls to `jit_malloc_exec`. It is not required that you supply a multiple of this size when allocating, but it can lead to better performance on some systems.

14.2 Memory set, copy, compare, etc

The following functions are provided to set, copy, compare, and search memory blocks.

`void * jit_memset (void *dest, int ch, unsigned int len)` [Function]
Set the *len* bytes at *dest* to the value *ch*. Returns *dest*.

`void * jit_memcpy` (*void *dest, const void *src, unsigned int len*) [Function]
Copy the *len* bytes at *src* to *dest*. Returns *dest*. The behavior is undefined if the blocks overlap (use *jit_memmove* instead for that case).

`void * jit_memmove` (*void *dest, const void *src, unsigned int len*) [Function]
Copy the *len* bytes at *src* to *dest* and handle overlapping blocks correctly. Returns *dest*.

`int jit_memcmp` (*const void *s1, const void *s2, unsigned int len*) [Function]
Compare *len* bytes at *s1* and *s2*, returning a negative, zero, or positive result depending upon their relationship. It is system-specific as to whether this function uses signed or unsigned byte comparisons.

`void * jit_memchr` (*void *str, int ch, unsigned int len*) [Function]
Search the *len* bytes at *str* for the first instance of the value *ch*. Returns the location of *ch* if it was found, or NULL if it was not found.

14.3 String operations

The following functions are provided to manipulate NULL-terminated strings. It is highly recommended that you use these functions in preference to system functions, because the corresponding system functions are extremely non-portable.

`unsigned int jit_strlen` (*const char *str*) [Function]
Returns the length of *str*.

`char * jit_strcpy` (*char *dest, const char *src*) [Function]
Copy the string at *src* to *dest*. Returns *dest*.

`char * jit_strcat` (*char *dest, const char *src*) [Function]
Copy the string at *src* to the end of the string at *dest*. Returns *dest*.

`char * jit_strncpy` (*char *dest, const char *src, unsigned int len*) [Function]
Copy at most *len* characters from the string at *src* to *dest*. Returns *dest*.

`char * jit_strdup` (*const char *str*) [Function]
Allocate a block of memory using `jit_malloc` and copy *str* into it. Returns NULL if *str* is NULL or there is insufficient memory to perform the `jit_malloc` operation.

`char * jit_strndup` (*const char *str, unsigned int len*) [Function]
Allocate a block of memory using `jit_malloc` and copy at most *len* characters of *str* into it. The copied string is then NULL-terminated. Returns NULL if *str* is NULL or there is insufficient memory to perform the `jit_malloc` operation.

`int jit_strcmp` (*const char *str1, const char *str2*) [Function]
Compare the two strings *str1* and *str2*, returning a negative, zero, or positive value depending upon their relationship.

`int jit_strncmp` (*const char *str1, const char *str2, unsigned int len*) [Function]
Compare the two strings *str1* and *str2*, returning a negative, zero, or positive value depending upon their relationship. At most *len* characters are compared.

`int jit_stricmp (const char *str1, const char *str2)` [Function]

Compare the two strings *str1* and *str2*, returning a negative, zero, or positive value depending upon their relationship. Instances of the English letters A to Z are converted into their lower case counterparts before comparison.

Note: this function is guaranteed to use English case comparison rules, no matter what the current locale is set to, making it suitable for comparing token tags and simple programming language identifiers.

Locale-sensitive string comparison is complicated and usually specific to the front end language or its supporting runtime library. We deliberately chose not to handle this in `libjit`.

`int jit_strnicmp (const char *str1, const char *str2, unsigned int len)` [Function]

Compare the two strings *str1* and *str2*, returning a negative, zero, or positive value depending upon their relationship. At most *len* characters are compared. Instances of the English letters A to Z are converted into their lower case counterparts before comparison.

`char * jit_strchr (const char *str, int ch)` [Function]

Search *str* for the first occurrence of *ch*. Returns the address where *ch* was found, or NULL if not found.

`char * jit_strrchr (const char *str, int ch)` [Function]

Search *str* for the first occurrence of *ch*, starting at the end of the string. Returns the address where *ch* was found, or NULL if not found.

14.4 Metadata handling

Many of the structures in the `libjit` library can have user-supplied metadata associated with them. Metadata may be used to store dependency graphs, branch prediction information, or any other information that is useful to optimizers or code generators.

Metadata can also be used by higher level user code to store information about the structures that is specific to the user's virtual machine or language.

The library structures have special-purpose metadata routines associated with them (e.g. `jit_function_set_meta`, `jit_block_get_meta`). However, sometimes you may wish to create your own metadata lists and attach them to your own structures. The functions below enable you to do this:

`int jit_meta_set (jit_meta_t *list, int type, void *data, jit_meta_free_func free_data, jit_function_t pool_owner)` [Function]

Set a metadata value on a list. If the *type* is already present in the list, then its previous value will be freed. The *free_func* is called when the metadata value is freed with `jit_meta_free` or `jit_meta_destroy`. Returns zero if out of memory.

If *pool_owner* is not NULL, then the metadata value will persist until the specified function is finished building. Normally you would set this to NULL.

Metadata type values of 10000 or greater are reserved for internal use. They should never be used by external user code.

`void * jit_meta_get (`*jit_meta_t* `list`, *int* `type``)` [Function]
Get the value associated with *type* in the specified *list*. Returns NULL if *type* is not present.

`void jit_meta_free (`*jit_meta_t* `*list`, *int* `type``)` [Function]
Free the metadata value in the *list* that has the specified *type*. Does nothing if the *type* is not present.

`void jit_meta_destroy (`*jit_meta_t* `*list`)` [Function]
Destroy all of the metadata values in the specified *list*.

14.5 Function application and closures

Sometimes all you have for a function is a pointer to it and a dynamic description of its arguments. Calling such a function can be extremely difficult in standard C. The routines in this section, particularly `jit_apply`, provide a convenient interface for doing this.

At other times, you may wish to wrap up one of your own dynamic functions in such a way that it appears to be a regular C function. This is performed with `jit_closure_create`.

`void jit_apply (`*jit_type_t signature, void* `*func`, *void* `**args`, *unsigned int* `num_fixed_args`, *void* `*return_value``)` [Function]
Call a function that has a particular function signature. If the signature has more than *num_fixed_args* arguments, then it is assumed to be a vararg call, with the additional arguments passed in the vararg argument area on the stack. The *signature* must specify the type of all arguments, including those in the vararg argument area.

`void jit_apply_raw (`*jit_type_t* `signature`, *void* `*func`, *void* `*args`, *void* `*return_value``)` [Function]
Call a function, passing a set of raw arguments. This can only be used if `jit_raw_supported` returns non-zero for the signature. The *args* value is assumed to be an array of `jit_nint` values that correspond to each of the arguments. Raw function calls are slightly faster than their non-raw counterparts, but can only be used in certain circumstances.

`int jit_raw_supported (`*jit_type_t* `signature`)` [Function]
Determine if `jit_apply_raw` can be used to call functions with a particular signature. Returns zero if not.

`void * jit_closure_create (`*jit_context_t* `context`, *jit_type_t* `signature`, *jit_closure_func* `func`, *void* `*user_data`)` [Function]
Create a closure from a function signature, a closure handling function, and a user data value. Returns NULL if out of memory, or if closures are not supported. The *func* argument should have the following prototype:

```
void func (jit_type_t signature, void *result,
void **args, void *user_data);
```

If the closure signature includes variable arguments, then `args` will contain pointers to the fixed arguments, followed by a `jit_closure_va_list_t` value for accessing the remainder of the arguments.

The memory for the closure will be reclaimed when the *context* is destroyed.

`int jit_closures_supported` (*void*) [Function]
Determine if this platform has support for closures.

`jit_nint jit_closure_va_get_nint` (*jit_closure_va_list_t* `va`) [Function]
`jit_nuint jit_closure_va_get_nuint` (*jit_closure_va_list_t* `va`) [Function]
`jit_long jit_closure_va_get_long` (*jit_closure_va_list_t* `va`) [Function]
`jit_ulong jit_closure_va_get_ulong` (*jit_closure_va_list_t* `va`) [Function]
`jit_float32 jit_closure_va_get_float32` (*jit_closure_va_list_t* `va`) [Function]
`jit_float64 jit_closure_va_get_float64` (*jit_closure_va_list_t* `va`) [Function]
`jit_nfloat jit_closure_va_get_nfloat` (*jit_closure_va_list_t* `va`) [Function]
`void * jit_closure_va_get_ptr` (*jit_closure_va_list_t* `va`) [Function]
Get the next value of a specific type from a closure's variable arguments.

`void jit_closure_va_get_struct` (*jit_closure_va_list_t* `va`, *void* *`buf`, *jit_type_t* `type`) [Function]
Get a structure or union value of a specific *type* from a closure's variable arguments, and copy it into *buf*.

14.6 Stack walking

The functions in `<jit/jit-walk.h>` allow the caller to walk up the native execution stack, inspecting frames and return addresses.

`void * jit_get_frame_address` (*unsigned int* `n`) [Function]
Get the frame address for the call frame *n* levels up the stack. Setting *n* to zero will retrieve the frame address for the current function. Returns NULL if it isn't possible to retrieve the address of the specified frame.

`void * jit_get_current_frame` (*void*) [Function]
Get the frame address for the current function. This may be more efficient on some platforms than using `jit_get_frame_address(0)`. Returns NULL if it isn't possible to retrieve the address of the current frame.

`void * jit_get_next_frame_address` (*void* *`frame`) [Function]
Get the address of the next frame up the stack from *frame*. Returns NULL if it isn't possible to retrieve the address of the next frame up the stack.

`void * jit_get_return_address` (*void* *`frame`) [Function]
Get the return address from a specified frame. The address represents the place where execution returns to when the specified frame exits. Returns NULL if it isn't possible to retrieve the return address of the specified frame.

`void * jit_get_current_return` (*void*) [Function]
Get the return address for the current function. This may be more efficient on some platforms than using `jit_get_return_address(0)`. Returns NULL if it isn't possible to retrieve the return address of the current frame.

`int jit_frame_contains_crawl_mark (void *frame, jit_crawl_mark_t *mark)` [Function]

Determine if the stack frame that resides just above *frame* contains a local variable whose address is *mark*. The *mark* parameter should be the address of a local variable that is declared with `jit_declare_crawl_mark(name)`.

Crawl marks are used internally by libjit to determine where control passes between JIT'ed and ordinary code during an exception throw. They can also be used to mark frames that have special security conditions associated with them.

14.7 Dynamic libraries

The following routines are supplied to help load and inspect dynamic libraries. They should be used in place of the traditional `dlopen`, `dlclose`, and `dlsym` functions, which are not portable across operating systems.

You must include `<jit/jit-dynamic.h>` to use these routines, and then link with `-ljitdynamic -ljit`.

`jit_dynlib_handle_t jit_dynlib_open (const char *name)` [Function]

Opens the dynamic library called *name*, returning a handle for it.

`void jit_dynlib_close (jit_dynlib_handle_t handle)` [Function]

Close a dynamic library.

`void * jit_dynlib_get_symbol (jit_dynlib_handle_t handle, const char *symbol)` [Function]

Retrieve the symbol *symbol* from the specified dynamic library. Returns NULL if the symbol could not be found. This will try both non-prefixed and underscore-prefixed forms of *symbol* on platforms where it makes sense to do so, so there is no need for the caller to perform prefixing.

`void jit_dynlib_set_debug (int flag)` [Function]

Enable or disable additional debug messages to stderr. Debugging is disabled by default. Normally the dynamic library routines will silently report errors via NULL return values, leaving reporting up to the caller. However, it can be useful to turn on additional diagnostics when tracking down problems with dynamic loading.

`const char * jit_dynlib_get_suffix (void)` [Function]

Get the preferred dynamic library suffix for this platform. Usually something like `so`, `dll`, or `dylib`.

Sometimes you want to retrieve a C++ method from a dynamic library using `jit_dynlib_get_symbol`. Unfortunately, C++ name mangling rules differ from one system to another, making this process very error-prone.

The functions that follow try to help. They aren't necessarily fool-proof, but they should work in the most common cases. The only alternative is to wrap your C++ library with C functions, so that the names are predictable.

The basic idea is that you supply a description of the C++ method that you wish to access, and these functions return a number of candidate forms that you can try with `jit_dynlib_get_symbol`. If one form fails, you move on and try the next form, until either symbol lookup succeeds or until all forms have been exhausted.

The following code demonstrates how to resolve a global function:

```
jit_dynlib_handle_t handle;
jit_type_t signature;
int form = 0;
void *address = 0;
char *mangled;

while((mangled = jit_mangle_global_function
("foo", signature, form)) != 0)
{
address = jit_dynlib_get_symbol(handle, mangled);
if(address != 0)
{
break;
}
jit_free(mangled);
++form;
}

if(address)
{
printf("%s = 0x%lxn", mangled, (long)address);
}
else
{
printf("could not resolve foon");
}
```

This mechanism typically cannot be used to obtain the entry points for `inline` methods. You will need to make other arrangements to simulate the behaviour of inline methods, or recompile your dynamic C++ library in a mode that explicitly exports inlines.

C++ method names are very picky about types. On 32-bit systems, `int` and `long` are the same size, but they are mangled to different characters. To ensure that the correct function is picked, you should use `jit_type_sys_int`, `jit_type_sys_long`, etc instead of the platform independent types. If you do use a platform independent type like `jit_type_int`, this library will try to guess which system type you mean, but the guess will most likely be wrong.

`char * jit_mangle_global_function` (*const char *`name`, jit_type_t `signature`, int `form`*) [Function]

Mangle the name of a global C++ function using the specified *form*. Returns NULL if out of memory, or if the form is not supported.

`char * jit_mangle_member_function` (*const char *`class_name`, const char *`name`, jit_type_t `signature`, int `form`, int `flags`*) [Function]

Mangle the name of a C++ member function using the specified *form*. Returns NULL if out of memory, or if the form is not supported. The following flags may be specified to modify the mangling rules:

`JIT_MANGLE_PUBLIC`
: The method has `public` access within its containing class.

`JIT_MANGLE_PROTECTED`
: The method has `protected` access within its containing class.

`JIT_MANGLE_PRIVATE`
: The method has `private` access within its containing class.

`JIT_MANGLE_STATIC`
: The method is `static`.

`JIT_MANGLE_VIRTUAL`
: The method is a virtual instance method. If neither `JIT_MANGLE_STATIC` nor `JIT_MANGLE_VIRTUAL` are supplied, then the method is assumed to be a non-virtual instance method.

`JIT_MANGLE_CONST`
: The method is an instance method with the `const` qualifier.

`JIT_MANGLE_EXPLICIT_THIS`
: The *signature* includes an extra pointer parameter at the start that indicates the type of the `this` pointer. This parameter won't be included in the final mangled name.

`JIT_MANGLE_IS_CTOR`
: The method is a constructor. The *name* parameter will be ignored.

`JIT_MANGLE_IS_DTOR`
: The method is a destructor. The *name* parameter will be ignored.

`JIT_MANGLE_BASE`
: Fetch the "base" constructor or destructor entry point, rather than the "complete" entry point.

The *class_name* may include namespace and nested parent qualifiers by separating them with `::` or `..` Class names that involve template parameters are not supported yet.

15 Diagnostic routines

`void jit_dump_type (`*FILE *`**`stream`*`, jit_type_t `**`type`*`)` [Function]
Dump the name of a type to a stdio stream.

`void jit_dump_value (`*FILE *`**`stream`*`, jit_function_t `**`func`*`, jit_value_t `**`value`*`, const char *`**`prefix`*`)` [Function]
Dump the name of a value to a stdio stream. If *prefix* is not NULL, then it indicates a type prefix to add to the value name. If *prefix* is NULL, then this function intuits the type prefix.

`void jit_dump_insn (`*FILE *`**`stream`*`, jit_function_t `**`func`*`, jit_value_t `**`value`*`)` [Function]
Dump the contents of an instruction to a stdio stream.

`void jit_dump_function` (*FILE *stream*, *jit_function_t func*, *const char *name*) [Function]

Dump the three-address instructions within a function to a stream. The *name* is attached to the output as a friendly label, but has no other significance.

If the function has not been compiled yet, then this will dump the three address instructions from the build process. Otherwise it will disassemble and dump the compiled native code.

16 Using libjit from C++

This chapter describes the classes and methods that are available in the `libjitplus` library. To use this library, you must include the header `<jit/jit-plus.h>` and link with the `-ljitplus` and `-ljit` options.

17 Contexts in C++

The `jit_context` class provides a C++ counterpart to the C `jit_context_t` type. See Chapter 4 [Initialization], page 13, for more information on creating and managing contexts.

`jit_context` () [Constructor on `jit_context`]

Construct a new JIT context. This is equivalent to calling `jit_context_create` in the C API. The raw C context is destroyed when the `jit_context` object is destructed.

`jit_context` (*jit_context_t context*) [Constructor on `jit_context`]

Construct a new JIT context by wrapping up an existing raw C context. This is useful for importing a context from third party C code into a program that prefers to use C++.

When you use this form of construction, `jit_context_destroy` will not be called on the context when the `jit_context` object is destructed. You will need to arrange for that manually.

`~jit_context` () [Destructor on `jit_context`]

Destruct a JIT context.

`void build_start` () [Method on `jit_context`]

Start an explicit build process. Not needed if you will be using on-demand compilation.

`void build_end` () [Method on `jit_context`]

End an explicit build process.

`jit_context_t raw` () *const* [Method on `jit_context`]

Get the raw C context pointer that underlies this object.

18 Values in C++

The `jit_value` class provides a C++ counterpart to the `jit_value_t` type. Values normally result by calling methods on the `jit_function` class during the function building process. See Chapter 7 [Values], page 29, for more information on creating and managing values.

`jit_value ()` [Constructor on `jit_value`]
Construct an empty value.

`jit_value (`*jit_value_t* `value)` [Constructor on `jit_value`]
Construct a value by wrapping up a raw C `jit_value_t` object.

`jit_value (`*const jit_value*`& value)` [Constructor on `jit_value`]
Create a copy of *value*.

`~jit_value ()` [Destructor on `jit_value`]
Destroy the C++ value wrapper, but leave the underlying raw C value alone.

`jit_value& operator= (`*const jit_value*`& value)` [Operator on `jit_value`]
Copy `jit_value` objects.

`jit_value_t raw ()` *const* [Method on `jit_value`]
Get the raw C `jit_value_t` value that underlies this object.

`int is_valid ()` *const* [Method on `jit_value`]
Determine if this `jit_value` object contains a valid raw C `jit_value_t` value.

`int is_temporary ()` *const* [Method on `jit_value`]
`int is_local ()` *const* [Method on `jit_value`]
`int is_constant ()` *const* [Method on `jit_value`]
Determine if this `jit_value` is temporary, local, or constant.

`void set_volatile ()` [Method on `jit_value`]
`int is_volatile ()` *const* [Method on `jit_value`]
Set or check the "volatile" state on this value.

`void set_addressable ()` [Method on `jit_value`]
`int is_addressable ()` *const* [Method on `jit_value`]
Set or check the "addressable" state on this value.

`jit_type_t type ()` *const* [Method on `jit_value`]
Get the type of this value.

`jit_function_t function ()` *const* [Method on `jit_value`]
`jit_block_t block ()` *const* [Method on `jit_value`]
`jit_context_t context ()` *const* [Method on `jit_value`]
Get the owning function, block, or context for this value.

`jit_constant_t constant` () *const* [Method on `jit_value`]
`jit_nint nint_constant` () *const* [Method on `jit_value`]
`jit_long long_constant` () *const* [Method on `jit_value`]
`jit_float32 float32_constant` () *const* [Method on `jit_value`]
`jit_float64 float64_constant` () *const* [Method on `jit_value`]
`jit_nfloat nfloat_constant` () *const* [Method on `jit_value`]

Extract the constant stored in this value.

`jit_value operator+` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator-` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator*` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator/` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator%` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator-` (*const jit_value*& `value1`) [Operator on `jit_value`]
`jit_value operator&` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator|` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator^` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator~` (*const jit_value*& `value1`) [Operator on `jit_value`]
`jit_value operator<<` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator>>` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator==` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator!=` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator<` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator<=` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator>` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]
`jit_value operator>=` (*const jit_value*& `value1`, *const jit_value*& `value2`) [Operator on `jit_value`]

Generate an arithmetic, bitwise, or comparison instruction based on one or two `jit_value` objects. These operators are shortcuts for calling `insn_add`, `insn_sub`, etc on the `jit_function` object.

19 Functions in C++

The `jit_function` class provides a C++ counterpart to the C `jit_function_t` type. See Chapter 5 [Functions], page 16, for more information on creating and managing functions.

The `jit_function` class also provides a large number of methods for creating the instructions within a function body. See Chapter 8 [Instructions], page 34, for more information on creating and managing instructions.

`jit_function` (*jit_context*`&` ***context***, *jit_type_t* ***signature***) [Constructor on `jit_function`]

Constructs a new function handler with the specified *signature* in the given *context*. It then calls `create(`***signature***`)` to create the actual function.

`jit_function` (*jit_context*`&` ***context***) [Constructor on `jit_function`]

Constructs a new function handler in the specified *context*. The actual function is not created until you call `create()`.

`jit_function` (*jit_function_t* ***func***) [Constructor on `jit_function`]

Constructs a new function handler and wraps it around the specified raw C `jit_function_t` object. This can be useful for layering the C++ on-demand building facility on top of an existing C function.

`~jit_function` () [Destructor on `jit_function`]

Destroy this function handler. The raw function will persist until the context is destroyed.

`jit_function_t raw` () *const* [Method on `jit_function`]

Get the raw C `jit_function_t` value that underlies this object.

`int is_valid` () *const* [Method on `jit_function`]

Determine if the raw C `jit_function_t` value that underlies this object is valid.

`static jit_function * from_raw` (*jit_function_t* ***func***) [Method on `jit_function`]

Find the C++ `jit_function` object that is associated with a raw C `jit_function_t` pointer. Returns NULL if there is no such object.

`jit_type_t signature` () *const* [Method on `jit_function`]

Get the signature type for this function.

`void create` (*jit_type_t* ***signature***) [Method on `jit_function`]

Create this function if it doesn't already exist.

`void create` () [Method on `jit_function`]

Create this function if it doesn't already exist. This version will call the virtual `create_signature()` method to obtain the signature from the subclass.

`int compile` () [Method on `jit_function`]

Compile this function explicity. You normally don't need to use this method because the function will be compiled on-demand. If you do choose to build the function manually, then the correct sequence of operations is as follows:

1. Invoke the `build_start` method to lock down the function builder.
2. Build the function by calling the value-related and instruction-related methods within `jit_function`.
3. Compile the function with the `compile` method.
4. Unlock the function builder by invoking `build_end`.

`int is_compiled` () *const* [Method on `jit_function`]
Determine if this function has already been compiled.

`void set_optimization_level` (*unsigned int* ***level***) [Method on `jit_function`]
`unsigned int optimization_level` () *const* [Method on `jit_function`]
Set or get the optimization level for this function.

`static unsigned int max_optimization_level` () [Method on `jit_function`]
Get the maximum optimization level for `libjit`.

`void * closure` () *const* [Method on `jit_function`]
`void * vtable_pointer` () *const* [Method on `jit_function`]
Get the closure or vtable pointer form of this function.

`int apply` (*void*** ***args***, *void** ***result***) [Method on `jit_function`]
`int apply` (*jit_type_t* ***signature***, *void*** ***args***, *void** ***return_area***) [Method on `jit_function`]
Call this function, applying the specified arguments.

`static jit_type_t signature_helper` (*jit_type_t* ***return_type***, *...*) [Method on `jit_function`]
You can call this method from `create_signature()` to help build the correct signature for your function. The first parameter is the return type, following by zero or more types for the parameters. The parameter list is terminated with the special value `jit_function::end_params`.

A maximum of 32 parameter types can be supplied, and the signature ABI is always set to `jit_abi_cdecl`.

`void build` () [Method on `jit_function`]
This method is called when the function has to be build on-demand, or in response to an explicit `recompile` request. You build the function by calling the value-related and instruction-related methods within `jit_function` that are described below.

The default implementation of `build` will fail, so you must override it if you didn't build the function manually and call `compile`.

`jit_type_t create_signature` () [Method on `jit_function`]
This method is called by `create()` to create the function's signature. The default implementation creates a signature that returns `void` and has no parameters.

`void fail` () [Method on `jit_function`]
This method can be called by `build` to fail the on-demand compilation process. It throws an exception to unwind the build.

`void out_of_memory ()` [Method on `jit_function`]
This method can be called by `build` to indicate that the on-demand compilation process ran out of memory. It throws an exception to unwind the build.

`void build_start ()` [Method on `jit_function`]
Start an explicit build process. Not needed if you will be using on-demand compilation.

`void build_end ()` [Method on `jit_function`]
End an explicit build process.

`jit_value new_value (`*jit_type_t* `type`*`)`* [Method on `jit_function`]
Create a new temporary value. This is the C++ counterpart to `jit_value_create`.

`jit_value new_constant (`*jit_sbyte* `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*jit_ubyte* `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*jit_short* `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*jit_ushort* `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*jit_int* `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*jit_uint* `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*jit_long* `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*jit_ulong* `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*jit_float32* `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*jit_float64* `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*jit_nfloat* `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*void** `value`*, jit_type_t* `type`*`)`* [Method on `jit_function`]
`jit_value new_constant (`*const jit_constant_t&* `value`*`)`* [Method on `jit_function`]
Create constant values of various kinds. See Chapter 7 [Values], page 29, for more information on creating and managing constants.

`jit_value get_param (`*unsigned int* `param`*`)`* [Method on `jit_function`]
Get the value that corresponds to parameter *param*.

`jit_value get_struct_pointer ()` [Method on `jit_function`]
Get the value that corresponds to the structure pointer parameter, if this function has one. Returns an empty value if it does not.

`jit_label new_label ()` [Method on `jit_function`]
Create a new label. This is the C++ counterpart to `jit_function_reserve_label`.

`void insn_label (`*jit_label&* `label)` [Method on `jit_function`]
`void insn_new_block ()` [Method on `jit_function`]
`jit_value insn_load (`*const jit_value&* `value)` [Method on `jit_function`]
`jit_value insn_dup (`*const jit_value&* `value)` [Method on `jit_function`]
`jit_value insn_load_small (`*const jit_value&* `value)` [Method on `jit_function`]
`void store (`*const jit_value&* `dest`, *const jit_value&* `value)` [Method on `jit_function`]
`jit_value insn_load_relative (`*const jit_value&* `value`, *jit_nint* `offset`, *jit_type_t* `type)` [Method on `jit_function`]
`void insn_store_relative (`*const jit_value&* `dest`, *jit_nint* `offset`, *const jit_value&* `value)` [Method on `jit_function`]
`jit_value insn_add_relative (`*const jit_value&* `value`, *jit_nint* `offset)` [Method on `jit_function`]
`jit_value insn_load_elem (`*const jit_value&* `base_addr`, *const jit_value&* `index`, *jit_type_t* `elem_type)` [Method on `jit_function`]
`jit_value insn_load_elem_address (`*const jit_value&* `base_addr`, *const jit_value&* `index`, *jit_type_t* `elem_type)` [Method on `jit_function`]
`void insn_store_elem (`*const jit_value&* `base_addr`, *const jit_value&* `index`, *const jit_value&* `value)` [Method on `jit_function`]
`void insn_check_null (`*const jit_value&* `value)` [Method on `jit_function`]
`jit_value insn_add (`*const jit_value&* `value1`, *const jit_value&* `value2)` [Method on `jit_function`]
`jit_value insn_add_ovf (`*const jit_value&* `value1`, *const jit_value&* `value2)` [Method on `jit_function`]
`jit_value insn_sub (`*const jit_value&* `value1`, *const jit_value&* `value2)` [Method on `jit_function`]
`jit_value insn_sub_ovf (`*const jit_value&* `value1`, *const jit_value&* `value2)` [Method on `jit_function`]
`jit_value insn_mul (`*const jit_value&* `value1`, *const jit_value&* `value2)` [Method on `jit_function`]
`jit_value insn_mul_ovf (`*const jit_value&* `value1`, *const jit_value&* `value2)` [Method on `jit_function`]
`jit_value insn_div (`*const jit_value&* `value1`, *const jit_value&* `value2)` [Method on `jit_function`]
`jit_value insn_rem (`*const jit_value&* `value1`, *const jit_value&* `value2)` [Method on `jit_function`]
`jit_value insn_rem_ieee (`*const jit_value&* `value1`, *const jit_value&* `value2)` [Method on `jit_function`]
`jit_value insn_neg (`*const jit_value&* `value1)` [Method on `jit_function`]
`jit_value insn_and (`*const jit_value&* `value1`, *const jit_value&* `value2)` [Method on `jit_function`]
`jit_value insn_or (`*const jit_value&* `value1`, *const jit_value&* `value2)` [Method on `jit_function`]

`jit_value insn_xor` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_not` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_shl` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_shr` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_ushr` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_sshr` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_eq` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_ne` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_lt` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_le` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_gt` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_ge` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_cmpl` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_cmpg` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_to_bool` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_to_not_bool` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_acos` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_asin` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_atan` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_atan2` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_ceil` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_cos` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_cosh` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_exp` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_floor` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_log` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_log10` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_pow` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_rint` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_round` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_sin` (*const jit_value&* `value1`) [Method on `jit_function`]

`jit_value insn_sinh` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_sqrt` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_tan` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_tanh` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_is_nan` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_is_finite` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_is_inf` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_abs` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_min` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_max` (*const jit_value&* `value1`, *const jit_value&* `value2`) [Method on `jit_function`]
`jit_value insn_sign` (*const jit_value&* `value1`) [Method on `jit_function`]
`void insn_branch` (*jit_label&* `label`) [Method on `jit_function`]
`void insn_branch_if` (*const jit_value&* `value`, *jit_label&* `label`) [Method on `jit_function`]
`void insn_branch_if_not` (*const jit_value& value*, *jit_label&* `label`) [Method on `jit_function`]
`jit_value insn_address_of` (*const jit_value&* `value1`) [Method on `jit_function`]
`jit_value insn_address_of_label` (*jit_label&* `label`) [Method on `jit_function`]
`jit_value insn_convert` (*const jit_value&* `value`, *jit_type_t* `type`, *int* `overflow_check`) [Method on `jit_function`]
`jit_value insn_call` (*const char** `name`, *jit_function_t* `jit_func`, *jit_type_t* `signature`, *jit_value_t** `args`, *unsigned int* `num_args`, *int* `flags`) [Method on `jit_function`]
`jit_value insn_call_indirect` (*const jit_value&* `value`, *jit_type_t* `signature`, *jit_value_t** `args`, *unsigned int* `num_args`, *int* `flags`) [Method on `jit_function`]
`jit_value insn_call_indirect_vtable` (*const jit_value&* `value`, *jit_type_t* `signature`, *jit_value_t * * `args`, *unsigned int* `num_args`, *int* `flags`) [Method on `jit_function`]
`jit_value insn_call_native` (*const char** `name`, *void** `native_func`, *jit_type_t* `signature`, *jit_value_t** `args`, *unsigned int* `num_args`, *int* `flags`) [Method on `jit_function`]
`jit_value insn_call_intrinsic` (*const char** `name`, *void** `intrinsic_func`, *const jit_intrinsic_descr_t&* `descriptor`, *const jit_value&* `arg1`, *const jit_value&* `arg2`) [Method on `jit_function`]
`void insn_incoming_reg` (*const jit_value&* `value`, *int* `reg`) [Method on `jit_function`]
`void insn_incoming_frame_posn` (*const jit_value&* `value`, *jit_nint* `posn`) [Method on `jit_function`]
`void insn_outgoing_reg` (*const jit_value&* `value`, *int* `reg`) [Method on `jit_function`]
`void insn_outgoing_frame_posn` (*const jit_value&* `value`, *jit_nint* `posn`) [Method on `jit_function`]

`void insn_return_reg (const jit_value& value, int reg)` [Method on `jit_function`]
`void insn_setup_for_nested (int nested_level, int reg)` [Method on `jit_function`]
`void insn_flush_struct (const jit_value& value)` [Method on `jit_function`]
`jit_value insn_import (jit_value value)` [Method on `jit_function`]
`void insn_push (const jit_value& value)` [Method on `jit_function`]
`void insn_push_ptr (const jit_value& value, jit_type_t type)` [Method on `jit_function`]
`void insn_set_param (const jit_value& value, jit_nint offset)` [Method on `jit_function`]
`void insn_set_param_ptr (const jit_value& value, jit_type_t type, jit_nint offset)` [Method on `jit_function`]
`void insn_push_return_area_ptr ()` [Method on `jit_function`]
`void insn_return (const jit_value& value)` [Method on `jit_function`]
`void insn_return ()` [Method on `jit_function`]
`void insn_return_ptr (const jit_value& value, jit_type_t type)` [Method on `jit_function`]
`void insn_default_return ()` [Method on `jit_function`]
`void insn_throw (const jit_value& value)` [Method on `jit_function`]
`jit_value insn_get_call_stack ()` [Method on `jit_function`]
`jit_value insn_thrown_exception ()` [Method on `jit_function`]
`void insn_uses_catcher ()` [Method on `jit_function`]
`jit_value insn_start_catcher ()` [Method on `jit_function`]
`void insn_branch_if_pc_not_in_range (const jit_label& start_label, const jit_label& end_label, jit_label& label)` [Method on `jit_function`]
`void insn_rethrow_unhandled ()` [Method on `jit_function`]
`void insn_start_finally (jit_label& label)` [Method on `jit_function`]
`void insn_return_from_finally ()` [Method on `jit_function`]
`void insn_call_finally (jit_label& label)` [Method on `jit_function`]
`jit_value insn_start_filter (jit_label& label, jit_type_t type)` [Method on `jit_function`]
`void insn_return_from_filter (const jit_value& value)` [Method on `jit_function`]
`jit_value insn_call_filter (jit_label& label, const jit_value& value, jit_type_t type)` [Method on `jit_function`]
`void insn_memcpy (const jit_value& dest, const jit_value& src, const jit_value& size)` [Method on `jit_function`]
`void insn_memmove (const jit_value& dest, const jit_value& src, const jit_value& size)` [Method on `jit_function`]
`void jit_insn_memset (const jit_value& dest, const jit_value& value, const jit_value& size)` [Method on `jit_function`]
`jit_value jit_insn_alloca (const jit_value& size)` [Method on `jit_function`]
`void insn_move_blocks_to_end (const jit_label& from_label, const jit_label& to_label)` [Method on `jit_function`]
`void insn_move_blocks_to_start (const jit_label& from_label, const jit_label& to_label)` [Method on `jit_function`]

`void insn_mark_offset (`*jit_int* *offset*`)` [Method on `jit_function`]
`void insn_mark_breakpoint (`*jit_nint* *data1*, *jit_nint* *data2*`)` [Method on `jit_function`]

Create instructions of various kinds. See Chapter 8 [Instructions], page 34, for more information on the individual instructions and their arguments.

20 Porting libjit to new architectures

This chapter describes what needs to be done to port `libjit` to a new CPU architecture. It is assumed that the reader is familiar with compiler implementation techniques and the particulars of their target CPU's instruction set.

We will use `ARCH` to represent the name of the architecture in the sections that follow. It is usually the name of the CPU in lower case (e.g. `x86`, `arm`, `ppc`, etc). By convention, all back end functions should be prefixed with `_jit`, because they are not part of the public API.

20.1 Porting the function apply facility

The first step in porting `libjit` to a new architecture is to port the `jit_apply` facility. This provides support for calling arbitrary C functions from your application or from JIT'ed code. If you are familiar with `libffi` or `ffcall`, then `jit_apply` provides a similar facility.

Even if you don't intend to write a native code generator, you will probably still need to port `jit_apply` to each new architecture.

The `libjit` library makes use of gcc's `__builtin_apply` facility to do most of the hard work of function application. This gcc facility takes three arguments: a pointer to the function to invoke, a structure containing register arguments, and a size value that indicates the number of bytes to push onto the stack for the call.

Unfortunately, the register argument structure is very system dependent. There is no standard format for it, but it usually looks something like this:

`stack_args`
: Pointer to an array of argument values to push onto the stack.

`struct_ptr`
: Pointer to the buffer to receive a `struct` return value. The `struct_ptr` field is only present if the architecture passes `struct` pointers in a special register.

`word_reg[0..N]`
: Values for the word registers. Platforms that pass values in registers will populate these fields. Not present if the architecture does not use word registers for function calls.

`float_reg[0..N]`
: Values for the floating-point registers. Not present if the architecture does not use floating-point registers for function calls.

It is possible to automatically detect the particulars of this structure by making test function calls and inspecting where the arguments end up in the structure. The `gen-apply`

program in `libjit/tools` takes care of this. It outputs the `jit-apply-rules.h` file, which tells `jit_apply` how to operate.

The `gen-apply` program will normally "just work", but it is possible that some architectures will be stranger than usual. You will need to modify `gen-apply` to detect this additional strangeness, and perhaps also modify `libjit/jit/jit-apply.c`.

If you aren't using gcc to compile `libjit`, then things may not be quite this easy. You may have to write some inline assembly code to emulate `__builtin_apply`. See the file `jit-apply-x86.h` for an example of how to do this. Be sure to add an `#include` line to `jit-apply-func.h` once you do this.

The other half of `jit_apply` is closure and redirector support. Closures are used to wrap up interpreted functions so that they can be called as regular C functions. Redirectors are used to help compile a JIT'ed function on-demand, and then redirect control to it.

Unfortunately, you will have to write some assembly code to support closures and redirectors. The builtin gcc facilities are not complete enough to handle the task. See `jit-apply-x86.c` and `jit-apply-arm.c` for some examples from existing architectures. You may be able to get some ideas from the `libffi` and `ffcall` libraries as to what you need to do on your architecture.

20.2 Creating the instruction generation macros

You will need a large number of macros and support functions to generate the raw instructions for your chosen CPU. These macros are fairly generic and are not necessarily specific to `libjit`. There may already be a suitable set of macros for your CPU in some other Free Software project.

Typically, the macros are placed into a file called `jit-gen-ARCH.h` in the `libjit/jit` directory. If some of the macros are complicated, you can place helper functions into the file `jit-gen-ARCH.c`. Remember to add both `jit-gen-ARCH.h` and `jit-gen-ARCH.c` to `Makefile.am` in `libjit/jit`.

Existing examples that you can look at for ideas are `jit-gen-x86.h` and `jit-gen-arm.h`. The macros in these existing files assume that instructions can be output to a buffer in a linear fashion, and that each instruction is relatively independent of the next.

This independence principle may not be true of all CPU's. For example, the `ia64` packs up to three instructions into a single "bundle" for parallel execution. We recommend that the macros should appear to use linear output, but call helper functions to pack bundles after the fact. This will make it easier to write the architecture definition rules. A similar approach could be used for performing instruction scheduling on platforms that require it.

20.3 Writing the architecture definition rules

The architecture definition rules for a CPU are placed into the files `jit-rules-ARCH.h` and `jit-rules-ARCH.c`. You should add both of these files to `Makefile.am` in `libjit/jit`.

You will also need to edit `jit-rules.h` in two places. First, place detection logic at the top of the file to detect your platform and define `JIT_BACKEND_ARCH` to 1. Further down the file, you should add the following two lines to the include file logic:

```
#elif defined(JIT_BACKEND_ARCH)
#include "jit-rules-ARCH.h"
```

20.3.1 Defining the registers

Every rule header file needs to define the macro `JIT_REG_INFO` to an array of values that represents the properties of the CPU's registers. The `_jit_reg_info` array is populated with these values. `JIT_NUM_REGS` defines the number of elements in the array. Each element in the array has the following members:

`name` The name of the register. This is used for debugging purposes.

`cpu_reg` The raw CPU register number. Registers in `libjit` are referred to by their pseudo register numbers, corresponding to their index within `JIT_REG_INFO`. However, these pseudo register numbers may not necessarily correspond to the register numbers used by the actual CPU. This field provides a mapping.

`other_reg`
The second pseudo register in a 64-bit register pair, or -1 if the current register cannot be used as the first pseudo register in a 64-bit register pair. This field only has meaning on 32-bit platforms, and should always be set to -1 on 64-bit platforms.

`flags` Flag bits that describe the pseudo register's properties.

The following flags may be present:

`JIT_REG_WORD`
This register can hold an integer word value.

`JIT_REG_LONG`
This register can hold a 64-bit long value without needing a second register. Normally only used on 64-bit platforms.

`JIT_REG_FLOAT32`
This register can hold a 32-bit floating-point value.

`JIT_REG_FLOAT64`
This register can hold a 64-bit floating-point value.

`JIT_REG_NFLOAT`
This register can hold a native floating-point value.

`JIT_REG_FRAME`
This register holds the frame pointer. You will almost always supply `JIT_REG_FIXED` for this register.

`JIT_REG_STACK_PTR`
This register holds the stack pointer. You will almost always supply `JIT_REG_FIXED` for this register.

`JIT_REG_FIXED`
This register has a fixed meaning and cannot be used for general allocation.

`JIT_REG_CALL_USED`
This register will be destroyed by a function call.

`JIT_REG_IN_STACK`
This register is in a stack-like arrangement.

`JIT_REG_GLOBAL`
: This register is a candidate for global register allocation.

A CPU may have some registers arranged into a stack. In this case operations can typically only occur at the top of the stack, and may automatically pop values as a side-effect of the operation. An example of such architecture is x87 floating point unit. Such CPU requires three additional macros.

`JIT_REG_STACK`
: If defined, this indicates the presence of the register stack.

`JIT_REG_STACK_START`
: The index of the first register in the `JIT_REG_INFO` array that is used in a stack-like arrangement.

`JIT_REG_STACK_END`
: The index of the last register in the `JIT_REG_INFO` array that is used in a stack-like arrangement.

The entries in the `JIT_REG_INFO` array from `JIT_REG_STACK_START` up to `JIT_REG_STACK_END` must also have the `JIT_REG_IN_STACK` flag set.

20.3.2 Other architecture macros

The rule file may also have definitions of the following macros:

`JIT_NUM_GLOBAL_REGS`
: The number of registers that are used for global register allocation. Set to zero if global register allocation should not be used.

`JIT_ALWAYS_REG_REG`
: Define this to 1 if arithmetic operations must always be performed on registers. Define this to 0 if register/memory and memory/register operations are possible.

`JIT_PROLOG_SIZE`
: If defined, this indicates the maximum size of the function prolog.

`JIT_FUNCTION_ALIGNMENT`
: This value indicates the alignment required for the start of a function. e.g. define this to 32 if functions should be aligned on a 32-byte boundary.

`JIT_ALIGN_OVERRIDES`
: Define this to 1 if the platform allows reads and writes on any byte boundary. Define to 0 if only properly-aligned memory accesses are allowed. Normally only defined to 1 under x86.

`jit_extra_gen_state`
`jit_extra_gen_init`
`jit_extra_gen_cleanup`
: The `jit_extra_gen_state` macro can be supplied to add extra fields to the `struct jit_gencode` type in `jit-rules.h`, for extra CPU-specific code generation state information.

  The `jit_extra_gen_init` macro initializes this extra information, and the `jit_extra_gen_cleanup` macro cleans it up when code generation is complete.

20.3.3 Architecture-dependent functions

`void _jit_init_backend` (*void*) [Function]
Initialize the backend. This is normally used to configure registers that may not appear on all CPU's in a given family. For example, only some ARM cores have floating-point registers.

`void _jit_gen_get_elf_info` (*jit_elf_info_t* ***info**) [Function]
Get the ELF machine and ABI type information for this platform. The `machine` field should be set to one of the `EM_*` values in `jit-elf-defs.h`. The `abi` field should be set to one of the `ELFOSABI_*` values in `jit-elf-defs.h` (`ELFOSABI_SYSV` will normally suffice if you are unsure). The `abi_version` field should be set to the ABI version, which is usually zero.

`int _jit_create_entry_insns` (*jit_function_t* **func**) [Function]
Create instructions in the entry block to initialize the registers and frame offsets that contain the parameters. Returns zero if out of memory.

This function is called when a builder is initialized. It should scan the signature and decide which register or frame position contains each of the parameters and then call either `jit_insn_incoming_reg` or `jit_insn_incoming_frame_posn` to notify `libjit` of the location.

`int _jit_create_call_setup_insns` (*jit_function_t* **func**, *jit_type_t* **signature**, *jit_value_t* ***args**, *unsigned int* **num_args**, *int* **is_nested**, *int* **nested_level**, *jit_value_t* ***struct_return**, *int* **flags**) [Function]
Create instructions within *func* necessary to set up for a function call to a function with the specified *signature*. Use `jit_insn_push` to push values onto the system stack, or `jit_insn_outgoing_reg` to copy values into call registers.

If *is_nested* is non-zero, then it indicates that we are calling a nested function within the current function's nested relationship tree. The *nested_level* value will be -1 to call a child, zero to call a sibling of *func*, 1 to call a sibling of the parent, 2 to call a sibling of the grandparent, etc. The `jit_insn_setup_for_nested` instruction should be used to create the nested function setup code.

If the function returns a structure by pointer, then *struct_return* must be set to a new local variable that will contain the returned structure. Otherwise it should be set to NULL.

`int _jit_setup_indirect_pointer` (*jit_function_t* **func**, *jit_value_t* **value**) [Function]
Place the indirect function pointer *value* into a suitable register or stack location for a subsequent indirect call.

`int _jit_create_call_return_insns` (*jit_function_t* **func**, *jit_type_t* **signature**, *jit_value_t* ***args**, *unsigned int* **num_args**, *jit_value_t* **return_value**, *int* **is_nested**) [Function]
Create instructions within *func* to clean up after a function call and to place the function's result into *return_value*. This should use `jit_insn_pop_stack` to pop

values off the system stack and `jit_insn_return_reg` to tell `libjit` which register contains the return value. In the case of a `void` function, *return_value* will be NULL.

Note: the argument values are passed again because it may not be possible to determine how many bytes to pop from the stack from the *signature* alone; especially if the called function is vararg.

`int _jit_opcode_is_supported (int opcode)` [Function]

Not all CPU's support all arithmetic, conversion, bitwise, or comparison operators natively. For example, most ARM platforms need to call out to helper functions to perform floating-point.

If this function returns zero, then `jit-insn.c` will output a call to an intrinsic function that is equivalent to the desired opcode. This is how you tell `libjit` that you cannot handle the opcode natively.

This function can also help you develop your back end incrementally. Initially, you can report that only integer operations are supported, and then once you have them working you can move on to the floating point operations.

`void * _jit_gen_prolog (jit_gencode_t gen, jit_function_t func, void *buf)` [Function]

Generate the prolog for a function into a previously-prepared buffer area of `JIT_PROLOG_SIZE` bytes in size. Returns the start of the prolog, which may be different than *buf*.

This function is called at the end of the code generation process, not the beginning. At this point, it is known which callee save registers must be preserved, allowing the back end to output the most compact prolog possible.

`void _jit_gen_epilog (jit_gencode_t gen, jit_function_t func)` [Function]

Generate a function epilog, restoring the registers that were saved on entry to the function, and then returning.

Only one epilog is generated per function. Functions with multiple `jit_insn_return` instructions will all jump to the common epilog. This is needed because the code generator may not know which callee save registers need to be restored by the epilog until the full function has been processed.

`void * _jit_gen_redirector (jit_gencode_t gen, jit_function_t func)` [Function]

Generate code for a redirector, which makes an indirect jump to the contents of `func->entry_point`. Redirectors are used on recompilable functions in place of the regular entry point. This allows `libjit` to redirect existing calls to the new version after recompilation.

`void _jit_gen_spill_reg (jit_gencode_t gen, int reg, int other_reg, jit_value_t value)` [Function]

Generate instructions to spill a pseudo register to the local variable frame. If *other_reg* is not -1, then it indicates the second register in a 64-bit register pair.

This function will typically call `_jit_gen_fix_value` to fix the value's frame position, and will then generate the appropriate spill instructions.

`void _jit_gen_free_reg (`*jit_gencode_t* **`gen`**, *int* **`reg`**, *int* **`other_reg`**, *int* **`value_used`**`)` [Function]

Generate instructions to free a register without spilling its value. This is called when a register's contents become invalid, or its value is no longer required. If *value_used* is set to a non-zero value, then it indicates that the register's value was just used. Otherwise, there is a value in the register but it was never used.

On most platforms, this function won't need to do anything to free the register. But some do need to take explicit action. For example, x86 needs an explicit instruction to remove a floating-point value from the FPU's stack if its value has not been used yet.

`void _jit_gen_load_value (`*jit_gencode_t* **`gen`**, *int* **`reg`**, *int* **`other_reg`**, *jit_value_t* **`value`**`)` [Function]

Generate instructions to load a value into a register. The value will either be a constant or a slot in the frame. You should fix frame slots with `_jit_gen_fix_value`.

`void _jit_gen_spill_global (`*jit_gencode_t* **`gen`**, *int* **`reg`**, *jit_value_t* **`value`**`)` [Function]

Spill the contents of *value* from its corresponding global register. This is used in rare cases when a machine instruction requires its operand to be in the specific register that happens to be global. In such cases the register is spilled just before the instruction and loaded back immediately after it.

`void _jit_gen_load_global (`*jit_gencode_t* **`gen`**, *int* **`reg`**, *jit_value_t* **`value`**`)` [Function]

Load the contents of *value* into its corresponding global register. This is used at the head of a function to pull parameters out of stack slots into their global register copies.

`void _jit_gen_exch_top (`*jit_gencode_t* **`gen`**, *int* **`reg`**`)` [Function]

Generate instructions to exchange the contents of the top stack register with a stack register specified by the *reg* argument.

It needs to be implemented only by backends that support stack registers.

`void _jit_gen_move_top (`*jit_gencode_t* **`gen`**, *int* **`reg`**`)` [Function]

Generate instructions to copy the contents of the top stack register into a stack register specified by the `reg` argument and pop the top register after this. If `reg` is equal to the top register then the top register is just popped without copying it.

It needs to be implemented only by backends that support stack registers.

`void _jit_gen_spill_top (`*jit_gencode_t* **`gen`**, *int* **`reg`**, *jit_value_t* **`value`**, *int* **`pop`**`)` [Function]

Generate instructions to spill the top stack register to the local variable frame. The *pop* argument indicates if the top register is popped from the stack.

It needs to be implemented only by backends that support stack registers.

`void _jit_gen_fix_value (`*jit_value_t* **`value`**`)` [Function]

Fix the position of a value within the local variable frame. If it doesn't already have a position, then assign one for it.

`void _jit_gen_insn (`*jit_gencode_t* **`gen`**, *jit_function_t* **`func`**, *jit_block_t* **`block`**, *jit_insn_t* **`insn`**`)` [Function]
Generate native code for the specified *insn*. This function should call the appropriate register allocation routines, output the instruction, and then arrange for the result to be placed in an appropriate register or memory destination.

`void _jit_gen_start_block (`*jit_gencode_t* **`gen`**, *jit_block_t* **`block`**`)` [Function]
Called to notify the back end that the start of a basic block has been reached.

`void _jit_gen_end_block (`*jit_gencode_t* **`gen`**`)` [Function]
Called to notify the back end that the end of a basic block has been reached.

`int _jit_gen_is_global_candidate (`*jit_type_t* **`type`**`)` [Function]
Determine if *type* is a candidate for allocation within global registers.

20.4 Allocating registers in the back end

The `libjit` library provides a number of functions for performing register allocation within basic blocks so that you mostly don't have to worry about it:

`void _jit_regs_lookup (`*char *name*`)` [Function]
Get the pseudo register by its name.

`int _jit_regs_needs_long_pair (`*jit_type_t type*`)` [Function]
Determine if a type requires a long register pair.

`int _jit_regs_get_cpu (`*jit_gencode_t gen, int reg, int *other_reg*`)` [Function]
Get the CPU register that corresponds to a pseudo register. "other_reg" will be set to the other register in a pair, or -1 if the register is not part of a pair.

`void _jit_regs_alloc_global (`*jit_gencode_t gen, jit_function_t func*`)` [Function]
Perform global register allocation on the values in `func`. This is called during function compilation just after variable liveness has been computed.

`void _jit_regs_init_for_block (`*jit_gencode_t gen*`)` [Function]
Initialize the register allocation state for a new block.

`void _jit_regs_spill_all (`*jit_gencode_t gen*`)` [Function]
Spill all of the temporary registers to memory locations. Normally used at the end of a block, but may also be used in situations where a value must be in a certain register and it is too hard to swap things around to put it there.

`void _jit_regs_set_incoming (`*jit_gencode_t gen, int reg, jit_value_t value*`)` [Function]
Set pseudo register `reg` to record that it currently holds the contents of `value`. The register must not contain any other live value at this point.

`void _jit_regs_set_outgoing (`*jit_gencode_t gen, int reg, jit_value_t value*`)` [Function]
Load the contents of `value` into pseudo register `reg`, spilling out the current contents. This is used to set up outgoing parameters for a function call.

`void _jit_regs_force_out` (*jit_gencode_t gen, jit_value_t value, int is_dest*) [Function]

If `value` is currently in a register, then force its value out into the stack frame. The `is_dest` flag indicates that the value will be a destination, so we don't care about the original value.

`int _jit_regs_load_value` (*jit_gencode_t gen, jit_value_t value, int destroy, int used_again*) [Function]

Load a value into any register that is suitable and return that register. If the value needs a long pair, then this will return the first register in the pair. Returns -1 if the value will not fit into any register.

If `destroy` is non-zero, then we are about to destroy the register, so the system must make sure that such destruction will not side-effect `value` or any of the other values currently in that register.

If `used_again` is non-zero, then it indicates that the value is used again further down the block.

Index of concepts and facilities

I

J

L

M

N

O

P

R

S

T

U

V

W

Table of Contents