

MyJIT: Documentation

version 0.9.0.x

Petr Krajča, 2015

Dept. Computer Science
Palacký University Olomouc

Contents

1	About MyJIT	1
2	Instruction Set	2
2.1	Registers	2
2.2	Notation	3
2.3	Instructions	3
2.3.1	Transfer Operations	3
2.3.2	Binary Arithmetic Operations	3
2.3.3	Unary Arithmetic Operations	5
2.3.4	Load Operations	5
2.3.5	Store Operations	6
2.3.6	Compare Instructions	6
2.3.7	Conversions	7
2.3.8	Function declaration	7
2.3.9	Function calls	8
2.3.10	Jumps	9
2.3.11	Branch Operations	9
2.3.12	Misc	11
3	Getting Started	13
4	Debugging	15
4.1	Debugging messages	15
4.2	Warnings	15
4.3	Code listing	16
4.3.1	Example of the IL listing (JIT_DEBUG_OPS)	16
4.3.2	Example of the machine code listing (JIT_DEBUG_CODE)	17
4.3.3	Example of the combined listing (JIT_DEBUG_COMBINED)	17
5	Optimizations	18
6	Download	19
6.1	Getting MyJIT	19
6.2	Documentation	19
6.3	License	19

1 About MyJIT

MyJIT is a library that allows to generate machine code at run-time and afterwards execute it. The project has started as a part of the Just-in-Time compiler of the Schemik (<http://schemik.sourceforge.net>) programming language and as a replacement for the GNU lightning library (<http://www.gnu.org/software/lightning/>) fixing some of its design issues. However, it has evolved into a more powerful tool. Therefore, MyJIT has a very similar instruction set as GNU lightning but it differs in some key aspects. The most important features which make MyJIT different or which should be highlighted are:

- support for an unlimited number of registers
- thread-safety
- support for the i386, AMD64, and SPARC processors
- convenient debugging
- easy to use and easy to extend design
- support for optimizations
- the library is fully embeddable
- particular backward compatibility with GNU lightning

2 Instruction Set

The instruction set of the MyJIT intermediate language is inspired by GNU lightning and in some aspects resembles architecture of RISC processors. Each operation has up to four operands which can be immediate values (numbers, constants) or registers. The number of available registers is virtually unlimited.

All general purpose registers and integer values are treated as signed integers and have the same size which corresponds to the register size of the CPU. Note that i386 and SPARC processors have 32 bit wide registers and AMD64 has 64 bit wide registers. In order to overcome this inconsistency, MyJIT provides the `jit_value` type which has the same size as the CPU's general purpose register. In specific cases, e.g., if smaller or unsigned value is needed and it is appropriate, you may specify the size or type of the value. This topic is discussed later.

All floating-point numbers and registers are internally treated as double precision values. However, if necessary, the value can be converted into a single precision value.

Typically, name of each instruction consists of three parts:

- name of the operation (e.g., `add` for addition, `mul` for multiplication, etc.)
- the name of the operation is often accompanied with the suffix `r` or `i` indicating whether the operation is taking merely registers or if it also takes an immediate value as its argument
- name of the operation can be equipped with additional flag delimited by the underscore (typically, this is used to identify operations handling unsigned numbers)

2.1 Registers

MyJIT supports arbitrary number of register. If the number of used registers is higher than the number of available hardware registers, MyJIT emulates them. Nevertheless, to achieve the best performance, it is a good practice not to use too many registers. All registers are denoted by positive integers including zero. To refer to these registers you should use macros `R(x)` and `FR(x)` identifying general purpose and floating point registers, respectively. Note that registers `R(x)` and `FR(x)` are completely different register and do not occupy the same space.

Besides, MyJIT has two special purpose registers---`R_FP` and `R_OUT`. `R_FP` serves as the frame pointer and is used to access dynamically allocated memory on the stack. The `R_OUT` can be used to handle the return values more efficiently. It can be used to read the return value right after the

return from the function. Otherwise, the value of the register is undefined. Furthermore, it can be used right before the return from the function to set the return value more efficiently. If the value is set earlier, it can lead to undefined behavior. However, in most cases register allocator can optimize this which makes this register almost obsolete.

2.2 Notation

In order to describe instruction set, we are using symbols `reg` and `freg` to denote general purpose and floating-point registers, respectively. Analogously, `imm` and `fimm` denote immediate integer values and floating-point values. Particular instructions (e.g., load and store operations) have an extra operand which specifies the size (number of bytes) of data they work with. This operand shall be denoted by `size` and `fsize`. The value passed by the operand `size` can be 1, 2, 4, or 8. However, only the AMD64 port supports operation processing 8 bytes long values. The value passed by the operand `fsize` can be either 4 or 8. In other words, `fsize` denotes precision of the value.

2.3 Instructions

2.3.1 Transfer Operations

These operations allow to assign a value into a register. The first operand is always a register and the second one can be either an immediate value or register.

<code>movi reg, imm</code>	<code>O1 := O2</code>
<code>movr reg, reg</code>	<code>O1 := O2</code>
<code>fmovr freg, freg</code>	<code>O1 := O2</code>
<code>fmov freg, fimm</code>	<code>O1 := O2</code>

2.3.2 Binary Arithmetic Operations

Each binary arithmetic operation has exactly three operands. First two operands are always registers and the last one can be an immediate value or register. These operations are fully compatible with those in the GNU lightning instruction set.

<code>addr reg, reg, reg</code>	<code>O1 := O2 + O3</code>
<code>addi reg, reg, imm</code>	<code>O1 := O2 + O3</code>
<code>addxr reg, reg, reg</code>	<code>O1 := O2 + (O3 + carry)</code>
<code>addxi reg, reg, imm</code>	<code>O1 := O2 + (O3 + carry)</code>
<code>addcr reg, reg, reg</code>	<code>O1 := O2 + O3, set carry</code>
<code>addci reg, reg, imm</code>	<code>O1 := O2 + O3, set carry</code>

subr	reg, reg, reg	01 := 02 - 03
subi	reg, reg, imm	01 := 02 - 03
subxr	reg, reg, reg	01 := 02 - (03 + carry)
subxi	reg, reg, imm	01 := 02 - (03 + carry)
subcr	reg, reg, reg	01 := 02 - 03, set carry
subci	reg, reg, imm	01 := 02 - 03, set carry
rsbr	reg, reg, reg	01 := 03 - 02
rsbi	reg, reg, imm	01 := 03 - 02
mulr	reg, reg, reg	01 := 02 * 03
muli	reg, reg, imm	01 := 02 * 03
hmulr	reg, reg, reg	01 := high bits of 02 * 03
hmuli	reg, reg, imm	01 := high bits of 02 * 03
divr	reg, reg, reg	01 := 02 / 03
divi	reg, reg, imm	01 := 02 / 03
modr	reg, reg, reg	01 := 02 % 03
modi	reg, reg, imm	01 := 02 % 03
andr	reg, reg, reg	01 := 02 & 03
andi	reg, reg, imm	01 := 02 & 03
orr	reg, reg, reg	01 := 02 03
ori	reg, reg, imm	01 := 02 03
xorr	reg, reg, reg	01 := 02 ^ 03
xori	reg, reg, imm	01 := 02 ^ 03
lshr	reg, reg, reg	01 := 02 << 03
lshi	reg, reg, imm	01 := 02 << 03
rshr	reg, reg, reg	01 := 02 >> 03
rshi	reg, reg, imm	01 := 02 >> 03
rshr_u	reg, reg, reg	01 := 02 >> 03 (unsigned variant)
rshi_u	reg, reg, imm	01 := 02 >> 03 (unsigned variant)

Operations subx and addx have to directly follow subc and addc otherwise the result is undefined. Note that you can use the unsigned flag with the rshr operation to propagate the first bit accordingly.

There are also equivalent operations for floating-point values.

faddr	freg, freg, freg	01 := 02 + 03
faddi	freg, freg, fimm	01 := 02 + 03
fsubr	freg, freg, freg	01 := 02 - 03

fsubi	freg, freg, fimm	01 := 02 - 03
frsbr	freg, freg, freg	01 := 03 - 02
frsbi	freg, freg, fimm	01 := 03 - 02
fmulr	freg, freg, freg	01 := 02 * 03
fmuli	freg, freg, fimm	01 := 02 * 03
fdivr	freg, freg, freg	01 := 02 / 03
fdivi	freg, freg, fimm	01 := 02 / 03

2.3.3 Unary Arithmetic Operations

These operations have two operands, both of which have to be registers.

negr	reg	01 := -02
notr	reg	01 := ~02
fnegr	freg	01 := -02

2.3.4 Load Operations

These operations transfer data from the memory into a register. Each operation has 3 or 4 operands. The last operand is an immediate value indicating the "size" of the data processed by this operation, i.e., a number of bytes copied from the memory to the register. It can be one of the following values: 1, 2, 4, or 8. Furthermore, the size cannot be larger than the size of the register. If the size of the data copied from the memory is smaller than the size of the register, the value is expanded to fit the entire register. Therefore, it may be necessary to specify additional sign flag.

ldr	reg, reg, size	01 := *02	
ldi	reg, imm, size	01 := *02	
ldr_u	reg, reg, size	01 := *02	(unsigned variant)
ldi_u	reg, imm, size	01 := *02	(unsigned variant)
ldxr	reg, reg, reg, size	01 := *(02 + 03)	
ldxi	reg, reg, imm, size	01 := *(02 + 03)	
ldxr_u	reg, reg, reg, size	01 := *(02 + 03)	(unsigned variant)
ldxi_u	reg, reg, imm, size	01 := *(02 + 03)	(unsigned variant)
fldr	freg, reg, fsize	01 := *02	
fldi	freg, imm, fsize	01 := *02	
fldxr	freg, reg, reg, fsize	01 := *(02 + 03)	
fldxi	freg, reg, imm, fsize	01 := *(02 + 03)	

2.3.5 Store Operations

These operations transfer data from the register into the memory. Each operation has 3 or 4 operands. The last operand is an immediate value and indicates the "size" of the data, see "Load Operations" for more details. The first operand can be either an immediate or register. Other operands must be registers.

str	reg, reg, size	*01 := 02
sti	imm, reg, size	*01 := 02
stxr	reg, reg, reg, size	*(01 + 02) := 03
stxi	imm, reg, reg, size	*(01 + 02) := 03
fstr	reg, freg, fsize	*01 := 02
fsti	imm, freg, fsize	*01 := 02
fstxr	reg, reg, freg, fsize	*(01 + 02) := 03
fstxi	imm, reg, freg, fsize	*(01 + 02) := 03

2.3.6 Compare Instructions

These operations compare last two operands and store one or zero (if the condition was met or not, respectively) into the first operand. All these operations have three operands. The first two operands have to be registers and the last one can be either a register or an immediate value.

ltr	reg, reg, reg	01 := (02 < 03)
lti	reg, reg, imm	01 := (02 < 03)
ltr_u	reg, reg, reg	01 := (02 < 03) (unsigned variant)
lti_u	reg, reg, imm	01 := (02 < 03) (unsigned variant)
ler	reg, reg, reg	01 := (02 <= 03)
lei	reg, reg, imm	01 := (02 <= 03)
ler_u	reg, reg, reg	01 := (02 <= 03) (unsigned variant)
lei_u	reg, reg, imm	01 := (02 <= 03) (unsigned variant)
gtr	reg, reg, reg	01 := (02 > 03)
gti	reg, reg, imm	01 := (02 > 03)
gtr_u	reg, reg, reg	01 := (02 > 03) (unsigned variant)
gti_u	reg, reg, imm	01 := (02 > 03) (unsigned variant)
ger	reg, reg, reg	01 := (02 >= 03)

<code>gei</code>	<code>reg, reg, imm</code>	<code>01 := (02 >= 03)</code>
<code>ger_u</code>	<code>reg, reg, reg</code>	<code>01 := (02 >= 03)</code> (unsigned variant)
<code>gei_u</code>	<code>reg, reg, imm</code>	<code>01 := (02 >= 03)</code> (unsigned variant)
<code>eqr</code>	<code>reg, reg, reg</code>	<code>01 := (02 == 03)</code>
<code>eqi</code>	<code>reg, reg, imm</code>	<code>01 := (02 == 03)</code>
<code>ner</code>	<code>reg, reg, reg</code>	<code>01 := (02 != 03)</code>
<code>nei</code>	<code>reg, reg, imm</code>	<code>01 := (02 != 03)</code>

2.3.7 Conversions

Register for integer and floating-point values are independent and in order to convert value from one type to another you have to use one of the following operations.

<code>extr</code>	<code>freg, reg</code>	<code>01 := 02</code>
<code>truncr</code>	<code>reg, freg</code>	<code>01 := trunc(02)</code>
<code>ceilr</code>	<code>reg, freg</code>	<code>01 := ceil(02)</code>
<code>floorr</code>	<code>reg, freg</code>	<code>01 := floor(02)</code>
<code>roundr</code>	<code>reg, freg</code>	<code>01 := round(02)</code>

The operation `truncr` rounds the value towards zero and is the fastest one. Operations `floorr` and `ceilr` rounds the value towards negative or positive infinity, respectively. `roundr` rounds the given value to the nearest integer.

2.3.8 Function declaration

The following operations and auxiliary macros help to create a function, read its arguments, and return value.

- Operation `prolog imm` has one operand which is an immediate value, which is a reference to a pointer of the function defined by the intermediate code. In other words, MyJIT generates machine code for a function which resides somewhere in the memory. The address of the functions is handed by this reference. See the "Getting started" section, for more details and for an illustrative example.
- Operations `retr reg`, `reti imm`, `fretr freg`, `fsize`, and `freti freg`, `fsize` set the return value and return control to the calling procedure (or function).
- Operation `declare_arg imm`, `imm` is not an actual operation but rather an auxiliary function which declares the type of the argument and its size (in this order); `declare_arg` can take the following types of arguments

- `JIT_SIGNED_NUM` -- signed integer number

- JIT_UNSIGNED_NUM -- unsigned integer number
 - JIT_FLOAT -- floating-point number
 - JIT_PTR -- pointer
- To read an argument there are `getarg reg, imm` and `getarg freg, imm` operations having two arguments. The destination register where the input argument will be stored and the immediate value which identifies position of the argument.
 - Operation `allocai imm` reserves space on the stack which has at least the size specified by its operand. Note that the stack space may be aligned to some higher value. The macro returns an integer number which is an *offset from the frame pointer R_FP!*

2.3.9 Function calls

Each function call is done in three steps. The call is initiated by the operation `prepare` having no argument. In the second step, arguments are passed to a function using `putarg` or `fputarg`. (The arguments are passed in the normal order not in reverse, cf. GNU Lightning.) Afterwards, the function is called with the `call` operation. To retrieve the returned value you can use operations `retval` or `fretval`.

Let us make few notes on function calls:

- If calling a function defined in the same instance of the compiler (e.g., recursive function), you cannot pass values through registers. Each function has its own set of registers.
- Only `putargr`, `putargi`, `fputargr`, and `fputargi` operations are allowed inside the `prepare-call` block, otherwise, the behavior of the library is unspecified.

List of operations related to function calls:

- `prepare` -- prepares function call (generic)
- `putargr reg` -- passes an argument to a function
- `putargi imm` -- passes an argument to a function
- `fputargr freg, fsize` -- passes the argument to a function
- `fputargi fimm, fsize` -- passes the argument to a function
- `call imm` -- calls a function
- `callr reg`
- `retval reg` -- reads return value
- `fretval freg, fsize` -- reads return value

2.3.10 Jumps

Operations `jmp` and `jmp` can be used to implement unconditional jumps. Both operations have one operand, an address to jump to. To obtain this address you can use the `get_label` operation or use the forward declaration along with the `patch` operation.

- `get_label` is not an actual operation; it is a function that returns a `jit_label` value--value which corresponds to the current position in the code. This value can be passed to `jmp`/call or to a branch operation.
- It may happen that one need to jump into a code which will be defined later. Therefore, one can use the forward declaration and set the address later. This means, one can declare that the operation `jmp` or a branch operations jumps to the place defined by the `JIT_FORWARD` macro and store the pointer to the operation into some `jit_op * value`. To set the address later, there is the `patch imm` operation with an argument which is the patched operation. The following code illustrates the situation.

```
op = jmp JIT_FORWARD
;
; some code
;
patch op
```

2.3.11 Branch Operations

Branch operations represent conditional jumps and all have three operands. The first operand is an immediate value and represents the address to jump to. The latter two are values to be compared. The last operand can be either an immediate value or register.

<code>bltr</code>	<code>imm, reg, reg</code>	<code>if (02 < 03) goto 01</code>
<code>blti</code>	<code>imm, reg, imm</code>	<code>if (02 < 03) goto 01</code>
<code>bltr_u</code>	<code>imm, reg, reg</code>	<code>if (02 < 03) goto 01</code>
<code>blti_u</code>	<code>imm, reg, imm</code>	<code>if (02 < 03) goto 01</code>
<code>bler</code>	<code>imm, reg, reg</code>	<code>if (02 <= 03) goto 01</code>
<code>blei</code>	<code>imm, reg, imm</code>	<code>if (02 <= 03) goto 01</code>
<code>bler_u</code>	<code>imm, reg, reg</code>	<code>if (02 <= 03) goto 01</code>
<code>blei_u</code>	<code>imm, reg, imm</code>	<code>if (02 <= 03) goto 01</code>
<code>bgtr</code>	<code>imm, reg, reg</code>	<code>if (02 > 03) goto 01</code>
<code>bgti</code>	<code>imm, reg, imm</code>	<code>if (02 > 03) goto 01</code>
<code>bgtr_u</code>	<code>imm, reg, reg</code>	<code>if (02 > 03) goto 01</code>

bgti_u	imm, reg, imm	if (02 > 03) goto 01
bger	imm, reg, reg	if (02 >= 03) goto 01
bgei	imm, reg, imm	if (02 >= 03) goto 01
bger_u	imm, reg, reg	if (02 >= 03) goto 01
bgei_u	imm, reg, imm	if (02 >= 03) goto 01
beqr	imm, reg, reg	if (02 == 03) goto 01
beqi	imm, reg, imm	if (02 == 03) goto 01
bner	imm, reg, reg	if (02 != 03) goto 01
bnei	imm, reg, imm	if (02 != 03) goto 01
bmsr	imm, reg, reg	if (02 & 03) goto 01
bmsi	imm, reg, imm	if (02 & 03) goto 01
bmcrr	imm, reg, reg	if !(02 & 03) goto 01
bmcir	imm, reg, imm	if !(02 & 03) goto 01
boaddr	imm, reg, reg	02 += 03, goto 01 on overflow
boaddi	imm, reg, imm	02 += 03, goto 01 on overflow
bnoaddr	imm, reg, reg	02 += 03, goto 01 on not overflow
bnoaddi	imm, reg, imm	02 += 03, goto 01 on not overflow
bosubr	imm, reg, reg	02 -= 03, goto 01 on overflow
bosubi	imm, reg, imm	02 -= 03, goto 01 on overflow
bnosubr	imm, reg, reg	02 -= 03, goto 01 on not overflow
bnosubi	imm, reg, imm	02 -= 03, goto 01 on not overflow
fbltr	imm, freg, freg	if (02 < 03) goto 01
fblti	imm, freg, fimm	if (02 < 03) goto 01
fbler	imm, freg, freg	if (02 <= 03) goto 01
fblei	imm, freg, fimm	if (02 <= 03) goto 01
fbgtr	imm, freg, freg	if (02 > 03) goto 01
fbgti	imm, freg, fimm	if (02 > 03) goto 01
fbger	imm, freg, freg	if (02 >= 03) goto 01
fbgei	imm, freg, fimm	if (02 >= 03) goto 01
fbeqr	imm, freg, freg	if (02 == 03) goto 01
fbeqi	imm, freg, fimm	if (02 == 03) goto 01

```
fbner    imm, freg, freg    if (02 != 03) goto 01
fbnei    imm, freg, fimm    if (02 != 03) goto 01
```

2.3.12 Misc

There is an operation that allows to emit raw bytes of data into a generated code:

```
data_byte imm
```

This operation emits only one byte to a generated code. For convenience there are auxiliary macros emitting a sequence of bytes, string of chars (including the trailing 0), empty area, and values of common sizes, respectively.

```
jit_data_bytes(struct jit *jit, int count, unsigned char *data)
jit_data_str(jit, str)
jit_data_emptyarea(jit, size)
jit_data_word(jit, a)
jit_data_dword(jit, a)
jit_data_qword(jit, a)
```

If you are emitting raw data into a code, it is your responsibility to properly align code. For this purpose there is an operation:

```
jit_align imm
```

This operation takes care of proper code alignment. Note that particular platforms have their specific requirements. On SPARC all instructions have to be aligned to 4 bytes, AMD64 favors alignment to 16 bytes, but it is not mandatory, etc. Safe bet is to use 16 as an operand of this operation.

To obtain reference to a data or code you can use two operations:

```
ref_data reg, imm
ref_code reg, imm
```

That loads address of the label (second operand) into a register. The `ref_data` operation is intended for addresses of data (emitted with `data_*` operations) and `ref_code` is for address within an ordinary code. Note that address obtained with `ref_code` can be used only for local jumps inside a function. If necessary, for instance, if a some sort of branch table is needed, it is possible to emit address as a data with two operations.

```
data_code imm
data_data imm
```

Note that mixing code and data may not be a generally good idea and may lead to various issues, e.g. poor performance, weird behavior, etc. Albeit this feature is part of the library, users are encouraged to place data to some specific part of code (for instance, to the end of code) or

use data that are not part of the code and are allocated elsewhere, for instance, with ordinary `malloc`.

3 Getting Started

We start with a really simple example---function returning its argument incremented by one. The source code of this example can be found in `demo1.c` which is part of the MyJIT package.

```
#include <stdlib.h>
#include <stdio.h>

// includes the header file
#include "myjit/jitlib.h"

// pointer to a function accepting one argument of type long and returning long value
typedef long (* plfl)(long);

int main()
{
    // creates a new instance of the compiler
    struct jit * p = jit_init();

    plfl foo;

    // the code generated by the compiler will be assigned to the function `foo'
    jit_prolog(p, &foo);

    // the first argument of the function
    jit_declare_arg(p, JIT_SIGNED_NUM, sizeof(long));

    // moves the first argument into the register R(0)
    jit_getarg(p, R(0), 0);

    // takes the value in R(0), increments it by one, and stores the result into the
    // register R(1)
    jit_addi(p, R(1), R(0), 1);
```



```

// returns from the function and returns the value stored in the register R(1)
jit_retr(p, R(1));

// compiles the above defined code
jit_generate_code(p);

// checks, if it works
printf("Check #1: %li\n", foo(1));
printf("Check #2: %li\n", foo(100));
printf("Check #3: %li\n", foo(255));

// if you are interested, you can dump the machine code
// this functionality is provided through the `gcc` and `objdump`
// jit_dump_ops(p, JIT_DEBUG_CODE);

// cleanup
jit_free(p);
return 0;
}

```

We assume that the code above is quite (self-)explanatory, and thus, we do not include more comments on this. However, let us make a note on compiling programs using MyJIT. To start with MyJIT, it is sufficient to copy the `myjit` subdirectory into your project. Programs using the MyJIT should include the `#include "myjit/jitlib.h"` header file. In order to link the application and build a proper executable file, it is necessary to also compile `"myjit/libjit-core.c"`.

For instance, to build a program with `gcc` you may use the following steps:

```

gcc -c -g -Winline -Wall -std=c99 -pedantic -D_XOPEN_SOURCE=600 demo1.c
gcc -c -g -Winline -Wall -std=c99 -pedantic -D_XOPEN_SOURCE=600 myjit/jitlib-core.c
gcc -o demo1 -g -Wall -std=c99 -pedantic demo1.o jitlib-core.o

```

The first command compiles the example, the second one compiles functions used by MyJIT, and the last one links the object files together and creates an execute file---`demo1`.

It should be emphasized that MyJIT conforms to the C99 standard and all MyJIT files should be compiled according to this standard.

We also recommend to check out the `demo2.c` and `demo3.c` examples which are also included in the MyJIT package.

4 Debugging

4.1 Debugging messages

MyJIT contains several tools simplifying development. One of them is the `msg` operation which prints out the given message or a value of the given register. The `msg` operation has one or two operands. The first one is always an immediate value which is the string to display. The second operand is optional and it must be a register. In this case the first string serves as the format string for `printf` and the value of the register is printed out using this string. The example of the `msg` operation usage:

```
jit_msg(jit, "Simple message\n");
jit_msgr(jit, "Reg 1: %l\n", R(1));
```

4.2 Warnings

One of the MyJIT's goals is to achieve maximal performance while emitting code. Thus, it does not do many checks while generating machine code from the intermediate language. Therefore, if the code in the intermediate language contains an error, it leads to a faulty machine code, and subsequently to a crash of the program. In order to avoid such errors, MyJIT contains a function:

```
void jit_check_code(struct jit *jit, int warnings);
```

Which can be called before code generation and which can point out to the most common errors. In the second argument you may specify if you want to be warned about all types of errors (`JIT_WARN_ALL`) or you can pick only some of them from the following list:

- `JIT_WARN_DEAD_CODE` -- detects unreachable code
- `JIT_WARN_OP_WITHOUT_EFFECT` -- displays warnings about operations without effect
- `JIT_WARN_INVALID_DATA_SIZE` -- displays warning if the size operand does not contain a valid value (i.e., 1, 2, 4, or 8)
- `JIT_WARN_UNINITIALIZED_REG` -- displays warning if an uninitialized register is used
- `JIT_WARN_REGISTER_TYPE_MISMATCH` -- displays warning if a general purpose register is used in place where the floating point register is expected, or vice versa

- `JIT_WARN_MISSING_PATCH` -- reports all jump operations with a `JIT_FORWARD` declaration but without corresponding patch
- `JIT_WARN_UNALIGNED_CODE` -- displays warning if the code follows data section without alignment
- `JIT_WARN_INVALID_CODE_REFERENCE` -- displays warning if `ref_code` or `data_code` is referring to a data and not to a valid code
- `JIT_WARN_INVALID_DATA_REFERENCE` -- displays warning if `ref_data` or `data_data` is referring to a code and not to a data
- `JIT_WARN_ALL` -- displays all warnings

4.3 Code listing

In real programs is MyJIT typically called from various functions and code is constructed in several steps, thus it is sometimes difficult to figure out, how the code looks like. Therefore, MyJIT provides several means allowing to inspect final code in the intermediate language as well as in the machine code. This functionality is provided through the `jit_dump_ops` function. In the second argument you may specify if you want to list:

- list of all operations in the intermediate language (`JIT_DEBUG_OPS`)
- generated machine code (`JIT_DEBUG_CODE`)
- combination of both -- MyJIT operations and machine code (`JIT_DEBUG_COMBINED`)

To make the navigation through the listing easier, we have included one auxiliary operation:

```
comment imm
```

Which has only one argument -- string which will appear only in the dumps.

NOTICE! Do not use debugging operations and functions in the production code. These operations are not efficient and may lead to a poor performance. You should rather call the `printf` function explicitly. The `jit_dump_ops` with the `JIT_DEBUG_CODE` is using `gcc` and `objdump` to disassemble the code, therefore, these two programs have to be present in the system, or, on OS X `clang` and `otool` are used. The `JIT_DEBUG_COMBINED` option requires `myjit-disasm` disassembler in the directory along with the debugged program, or the path to the disassembler has to be specified in the `MYJIT_DISASM` environment variable.

Examples of the outputs for the above mentioned source code.

4.3.1 Example of the IL listing (`JIT_DEBUG_OPS`)

```
prolog      0xbfe62858
declarg     integer, 0x4
```

```

getarg    r0, 0x0
addi      r1, r0, 0x1
retr      r1

```

4.3.2 Example of the machine code listing (JIT_DEBUG_CODE)

```

0000000000000000 <main>:
 0:  55                      push    rbp
 1:  48 8b ec                mov     rbp, rsp
 4:  48 83 ec 20             sub     rsp, 0x20
 8:  48 8b f7                mov     rsi, rdi
 b:  48 8d 46 01             lea     rax, [rsi+0x1]
 f:  48 8b e5                mov     rsp, rbp
12:  5d                      pop     rbp
13:  c3                      ret

```

4.3.3 Example of the combined listing (JIT_DEBUG_COMBINED)

```

prolog    0x7fffa0371db0

0000: 55                      push    rbp
0001: 48 8b ec                mov     rbp, rsp
0004: 48 83 ec 20             sub     rsp, 0x20

declare_arg integer, 0x8
getarg    r0, 0x0
0008: 48 8b f7                mov     rsi, rdi

addi      r1, r0, 0x1
000b: 48 8d 46 01             lea     rax, [rsi+0x1]

retr      r1
000f: 48 8b e5                mov     rsp, rbp
0012: 5d                      pop     rbp
0013: c3                      ret

```

5 Optimizations

Support for multiple optimizations is available since release 0.7. These optimizations may speed up your code but the code generation may take longer. Therefore, you can turn particular optimization off and on using `jit_disable_optimization` and `jit_enable_optimization` functions, respectively. Currently, there are available the following optimizations:

- `JIT_OPT_OMIT_UNUSED_ASSIGNMENTS` -- compiler skips unused assignments. (Turned off by default.)
- `JIT_OPT_JOIN_ADDMUL` -- if possible, compiler joins adjacent `mul` and `add` (or two `add`'s) into one `LEA` operation (Turned on by default.)
- `JIT_OPT_OMIT_FRAME_PTR` -- if possible, compiler skips prolog and epilogue of the function. This significantly speeds up small functions. (Turned on by default.)

The optimized code for above mentioned example looks like this:

```
00000000 <main>:
  0:  8b 4c 24 04      mov    ecx,DWORD PTR [esp+0x4]
  4:  8d 41 01         lea    eax,[ecx+0x1]
  7:  c3              ret
```

Or, like this:

```
0000000000000000 <main>:
  0:  48 8d 47 01      lea    rax,[rdi+1]
  4:  c3              ret
```

6 Download

6.1 Getting MyJIT

The source code including this documentation and examples is available at SourceForge (<http://sourceforge.net/projects/myjit/files>) as of other information (<http://sourceforge.net/projects/myjit>)

You can also checkout the latest release from the GIT repository:

```
git clone git://git.code.sf.net/p/myjit/maincode myjit-maincode).
```

6.2 Documentation

Documentation is available in the `doc/` directory and on the project's website as a text file, [PDF file](#), or as a [single HTML page](#).

6.3 License

MyJIT is distributed under the terms of GNU Lesser General Public License v.3 or later (at your option).

Despite the fact that MyJIT is very similar to GNU Lightning, it does not share any source code with this project. However, some files come from the Mono project by Novel. (<http://www.mono-project.com>)

7 Notes on Development

- The primary use of this library is in our compiler of the Schemik programming language and the development of this library is driven by requirements of this compiler. Nevertheless, MyJIT is a general purpose library and its functionality is not limited.
- The library is almost complete and each release undergoes extensive testing (hundreds of tests), therefore, we hope there are no serious bugs. If you found any, please, let us know.
- Despite the fact that the library is almost complete, it is still under development and the API may slightly (but not much) change in the near future.
- Only the i386, AMD64, and SPARC platforms are supported right now, however, port to other architecture should be easy and straightforward.
- At this moment, MyJIT has support for floating point arithmetics. However, i386 port supports only processors having SSE2 unit, i.e., floating-point operations won't work on legacy CPUs without this unit.
- If you are using this library or if you want to contribute some code (e.g., port to some architecture), please, let us know.
- The documentation lacks information on internals of MyJIT. This is purely intentional because the library is still developed. We expect that the quality of the documentation will improve with the maturity of the project. If in doubts, ask in the mailing list (myjit-devel@lists.sourceforge.net). If you would like to participate on documentation improvements, please let us know, we will really appreciate it.