MyJIT: Documentation

version 0.9.0.x

Petr Krajča, 2015

Dept. Computer Science
Palacký University Olomouc

Contents

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

1

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.

```
movi  reg, imm        O1 := O2
movr  reg, reg        O1 := O2


fmovr freg, freg      O1 := O2
fmov  freg, fimm      O1 := O2
```

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.

```
addr   reg, reg, reg     O1 := O2 + O3
addi   reg, reg, imm     O1 := O2 + O3
addxr  reg, reg, reg     O1 := O2 + (O3 + carry)
addxi  reg, reg, imm     O1 := O2 + (O3 + carry)
addcr  reg, reg, reg     O1 := O2 + O3, set carry
addci  reg, reg, imm     O1 := O2 + O3, set carry
```

```
subr    reg, reg, reg      O1 := O2 - O3
subi    reg, reg, imm      O1 := O2 - O3
subxr   reg, reg, reg      O1 := O2 - (O3 + carry)
subxi   reg, reg, imm      O1 := O2 - (O3 + carry)
subcr   reg, reg, reg      O1 := O2 - O3, set carry
subci   reg, reg, imm      O1 := O2 - O3, set carry
rsbr    reg, reg, reg      O1 := O3 - O2
rsbi    reg, reg, imm      O1 := O3 - O2


mulr    reg, reg, reg      O1 := O2 * O3
muli    reg, reg, imm      O1 := O2 * O3
hmulr   reg, reg, reg      O1 := high bits of O2 * O3
hmuli   reg, reg, imm      O1 := high bits of O2 * O3


divr    reg, reg, reg      O1 := O2 / O3
divi    reg, reg, imm      O1 := O2 / O3
modr    reg, reg, reg      O1 := O2 % O3
modi    reg, reg, imm      O1 := O2 % O3


andr    reg, reg, reg      O1 := O2 & O3
andi    reg, reg, imm      O1 := O2 & O3
orr     reg, reg, reg      O1 := O2 | O3
ori     reg, reg, imm      O1 := O2 | O3
xorr    reg, reg, reg      O1 := O2 ^ O3
xori    reg, reg, imm      O1 := O2 ^ O3


lshr    reg, reg, reg      O1 := O2 << O3
lshi    reg, reg, imm      O1 := O2 << O3
rshr    reg, reg, reg      O1 := O2 >> O3
rshi    reg, reg, imm      O1 := O2 >> O3
rshr_u  reg, reg, reg      O1 := O2 >> O3  (unsigned variant)
rshi_u  reg, reg, imm      O1 := O2 >> O3  (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     O1 := O2 + O3
faddi    freg, freg, fimm     O1 := O2 + O3
fsubr    freg, freg, freg     O1 := O2 - O3
```

```
fsubi   freg, freg, fimm      O1 := O2 - O3
frsbr   freg, freg, freg      O1 := O3 - O2
frsbi   freg, freg, fimm      O1 := O3 - O2
fmulr   freg, freg, freg      O1 := O2 * O3
fmuli   freg, freg, fimm      O1 := O2 * O3
fdivr   freg, freg, freg      O1 := O2 / O3
fdivi   freg, freg, fimm      O1 := O2 / O3
```

2.3.3 Unary Arithmetic Operations

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

```
negr  reg     O1 := -O2
notr  reg     O1 := ~O2
fnegr freg    O1 := -O2
```

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          O1 := *O2
ldi     reg, imm, size          O1 := *O2
ldr_u   reg, reg, size          O1 := *O2        (unsigned variant)
ldi_u   reg, imm, size          O1 := *O2        (unsigned variant)


ldxr    reg, reg, reg, size     O1 := *(O2 + O3)
ldxi    reg, reg, imm, size     O1 := *(O2 + O3)
ldxr_u  reg, reg, reg, size     O1 := *(O2 + O3) (unsigned variant)
ldxi_u  reg, reg, imm, size     O1 := *(O2 + O3) (unsigned variant)


fldr    freg, reg, fsize        O1 := *O2
fldi    freg, imm, fsize        O1 := *O2


fldxr   freg, reg, reg, fsize   O1 := *(O2 + O3)
fldxi   freg, reg, imm, fsize   O1 := *(O2 + O3)
```

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              *O1 := O2
sti     imm, reg, size              *O1 := O2


stxr    reg, reg, reg, size         *(O1 + O2) := O3
stxi    imm, reg, reg, size         *(O1 + O2) := O3


fstr    reg, freg, fsize            *O1 := O2
fsti    imm, freg, fsize            *O1 := O2


fstxr   reg, reg, freg, fsize       *(O1 + O2) := O3
fstxi   imm, reg, freg, fsize       *(O1 + O2) := O3
```

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       O1 := (O2 <  O3)
lti     reg, reg, imm       O1 := (O2 <  O3)
ltr_u   reg, reg, reg       O1 := (O2 <  O3)  (unsigned variant)
lti_u   reg, reg, imm       O1 := (O2 <  O3)  (unsigned variant)


ler     reg, reg, reg       O1 := (O2 <= O3)
lei     reg, reg, imm       O1 := (O2 <= O3)
ler_u   reg, reg, reg       O1 := (O2 <= O3)  (unsigned variant)
lei_u   reg, reg, imm       O1 := (O2 <= O3)  (unsigned variant)


gtr     reg, reg, reg       O1 := (O2 >  O3)
gti     reg, reg, imm       O1 := (O2 >  O3)
gtr_u   reg, reg, reg       O1 := (O2 >  O3)  (unsigned variant)
gti_u   reg, reg, imm       O1 := (O2 >  O3)  (unsigned variant)


ger     reg, reg, reg       O1 := (O2 >= O3)
```

```
gei    reg, reg, imm       O1 := (O2 >= O3)
ger_u  reg, reg, reg       O1 := (O2 >= O3)  (unsigned variant)
gei_u  reg, reg, imm       O1 := (O2 >= O3)  (unsigned variant)


eqr    reg, reg, reg       O1 := (O2 == O3)
eqi    reg, reg, imm       O1 := (O2 == O3)


ner    reg, reg, reg       O1 := (O2 != O3)
nei    reg, reg, imm       O1 := (O2 != O3)
```

2.3.7 Conversions

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

```
extr    freg, reg       O1 := O2
truncr  reg, freg       O1 := trunc(O2)
ceilr   reg, freg       O1 := ceil(O2)
floorr  reg, freg       O1 := floor(O2)
roundr  reg, freg       O1 := round(O2)
```

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

7

- 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

8

2.3.10 Jumps

Operations `jmpi` and `jmpr` 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 jmpi/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 `jmpi` 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 = jmpi 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.

```
bltr      imm, reg, reg      if (O2 <  O3) goto O1
blti      imm, reg, imm      if (O2 <  O3) goto O1
bltr_u    imm, reg, reg      if (O2 <  O3) goto O1
blti_u    imm, reg, imm      if (O2 <  O3) goto O1

bler      imm, reg, reg      if (O2 <= O3) goto O1
blei      imm, reg, imm      if (O2 <= O3) goto O1
bler_u    imm, reg, reg      if (O2 <= O3) goto O1
blei_u    imm, reg, imm      if (O2 <= O3) goto O1

bgtr      imm, reg, reg      if (O2 >  O3) goto O1
bgti      imm, reg, imm      if (O2 >  O3) goto O1
bgtr_u    imm, reg, reg      if (O2 >  O3) goto O1
```

```
bgti_u     imm, reg, imm      if (O2 >  O3) goto O1


bger       imm, reg, reg      if (O2 >= O3) goto O1
bgei       imm, reg, imm      if (O2 >= O3) goto O1
bger_u     imm, reg, reg      if (O2 >= O3) goto O1
bgei_u     imm, reg, imm      if (O2 >= O3) goto O1


beqr       imm, reg, reg      if (O2 == O3) goto O1
beqi       imm, reg, imm      if (O2 == O3) goto O1
bner       imm, reg, reg      if (O2 != O3) goto O1
bnei       imm, reg, imm      if (O2 != O3) goto O1


bmsr       imm, reg, reg      if (O2 &  O3) goto O1
bmsi       imm, reg ,imm      if (O2 &  O3) goto O1
bmcr       imm, reg ,reg      if !(O2 & O3) goto O1
bmci       imm, reg ,imm      if !(O2 & O3) goto O1


boaddr     imm, reg, reg      O2 += O3, goto O1 on overflow
boaddi     imm, reg, imm      O2 += O3, goto O1 on overflow
bnoaddr    imm, reg, reg      O2 += O3, goto O1 on not overflow
bnoaddi    imm, reg, imm      O2 += O3, goto O1 on not overflow


bosubr     imm, reg, reg      O2 -= O3, goto O1 on overflow
bosubi     imm, reg, imm      O2 -= O3, goto O1 on overflow
bnosubr    imm, reg, reg      O2 -= O3, goto O1 on not overflow
bnosubi    imm, reg, imm      O2 -= O3, goto O1 on not overflow


fbltr      imm, freg, freg    if (O2 <  O3) goto O1
fblti      imm, freg, fimm    if (O2 <  O3) goto O1
fbler      imm, freg, freg    if (O2 <= O3) goto O1
fblei      imm, freg, fimm    if (O2 <= O3) goto O1


fbgtr      imm, freg, freg    if (O2 >  O3) goto O1
fbgti      imm, freg, fimm    if (O2 >  O3) goto O1
fbger      imm, freg, freg    if (O2 >= O3) goto O1
fbgei      imm, freg, fimm    if (O2 >= O3) goto O1


fbeqr      imm, freg, freg    if (O2 == O3) goto O1
fbeqi      imm, freg, fimm    if (O2 == O3) goto O1
```

```
fbner    imm, freg, freg    if (O2 != O3) goto O1
fbnei    imm, freg, fimm    if (O2 != O3) goto O1
```

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.

```c
#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

MyJIT contains several tools simplifying application development. The first one 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));
```

MyJIT provides several means for code analysis. It 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`)

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.

```
JIT_DEBUG_OPS
------------


prolog      0xbfe62858
declarg     integer, 0x4
getarg      r0, 0x0
addi        r1, r0, 0x1
retr        r1
```

```
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

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
```

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

```
comment imm
```

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

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_ASSIGNEMENTS` -- 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 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.