CSE4100: Compilers **Fall 2015**

Homework 0: Getting Started with LLVM

Out Date: 09/08/2015
Due Date: 09/15/2015

Objectives

The purpose of this homework is to get you started with LLVM, our target language. You will be writing several simple program in LLVM assembly, compiling and testing them.

The actual algorithms should be completely straightforward. Your objectives are

1. Get familiar with LLVM IR `http://llvm.org/docs/LangRef.html`
2. Get familiar with the LLVM compiler tools
3. Learn the basics of SSA, functions, typed assembly, blocks, ϕ-nodes.
4. Write a few simple functions
5. Systematically test your code.

Tools

You will be using the following tools:

- Linux or MacOS as an operating system. Use the lab machines if you have to.
- `llvm-as` the LLVM source language assembler.
- `opt` the LLVM byte-code optimizer
- `llc` the LLVM byte-code assembler
- `clang` the LLVM assembler (the C compiler used to assemble)
- `clang` the LLVM linker (the C compiler used to invoke the linker)

Handout

We are distributing a zip file with the empty shell of the source code where you will do your work as well as a `Makefile` to compile the whole thing! Beware, avoid copy paste of the Makefile in the PDF. It would contain non-printable unicode characters (invisible characters) guaranteed to drive the make command crazy. Use the handout!

Handin

Please, respect these directions to the letter. You must create a zip file for your submission. The zip file should contain a single directory named `hw1_<YourFullName>`. For instance, if I wanted to turn in a submission for myself, I would create a zip file of a folder named `hw1_Laurent_Michel`. That folder is supposed to contain all your source files (`.llvm` files) which is just one file for this assignment.

1 LLVM pipeline

As pointed out in class, compilers are pipelines that progressively transform the source into the target language producing some intermediate representation along the way. The LLVM source language can be turned into an executable by manually executing all the stages of this pipeline. The pipeline works as follows

LLVM Assemble the file `source.llvm` into a byte code representation `source.bc`

OPT Optionally, optimize the bytecode

COMPILE Compile the bytecode file `source.bc` into a native assembly file `source.s`

ASSEMBLE Assemble the native assembly `source.s` into a native object file `source.o`

LINK Link the object file with system libraries into an executable. This last stage is OS dependent. The example below is what works on a MacOS system. Linux is *slightly* different.

The fragment below shows the execution of all the command to turn a file containting the Fibonacci function implementation into an executable named fib.

```
/Users/ldm/llvm/bin/llvm-as fib.llvm -o fib.bc
/Users/ldm/llvm/bin/llc fib.bc -o fib.s
/Users/ldm/llvm/bin/clang fib.s -o fib.o
/Users/ldm/llvm/bin/clang fib.o -o fib
rm fib.bc fib.s
```

Typing all this command again and again is not all that fun. You can use the simple Makefile in Figure 1 to do the job. Here is a transcript, and you can find a pastable version here `http://pastie.org/5938193` as well as on moodle. As before, don't forget to fix the `CPATH` definition for your system.

2 LLVM Briefly

The documentation of LLVM is available here `http://llvm.org/docs/LangRef.html`. The LLVM language is a very friendly assembly language that is, to a large extent, simpler than both $MIPS$ and $Intel$ native languages that you might already know. A good part of this assignment is to read the LLVM documentation. You should understand most of the sections 1 through 9 in the document linked above (you can ignore all the intrinsics for now). A few observations are in order to get you started as quickly as possible.

1. LLVM is a *typed* assembly language. You must associate a type with every operand you manipulate.
2. LLVM, like all other assembly languages, mandates that you understand pointers thoroughly.
3. LLVM is a high-level assembly hiding the details of the calling convention for instance. Indeed, calling conventions (whether arguments are passed in registers, on the stack, in what order, etc...) are always platform and OS dependent, so these concerns are handled when assembling LLVM files into *native* assembly files.

```
OFILES= fib.o
OFILES1= fact1.o
OFILES2= fact2.o
CPATH=~/llvm/bin/
LLVMASY=$(CPATH)llvm-as
LLVMOPT=$(CPATH)opt
LLVMLC=$(CPATH)llc
AS=clang -c
LD=clang

all: fib fact1 fact2

fib: $(OFILES)
	$(LD) $(OFILES) $(LIBS) -o $@

fact1: $(OFILES1)
	$(LD) $(OFILES1) $(LIBS) -o $@

fact2: $(OFILES2)
	$(LD) $(OFILES2) $(LIBS) -o $@

%.o : %.s
	$(AS) $< -o $@

%.s : %.bc
	$(LLVMLC) $< -O=3 -tailcallopt -o $@

%.bc : %.llvm
	$(LLVMASY) $< -o $@

clean:
	rm *.o *.s *.bc fib fact1 fact2 *~
```

Figure 1: Makefile for LLVM

4. LLVM does not model registers. Instead, you have an *infinite* supply of temporaries.

5. LLVM uses static single assignment (*SSA* form). The core idea behind SSA (and unlike all the other assembly languages you might know) is that a *temporary* can only be written to **once**. This does not refer to a write to a temporary in a loop body. Instead, it refers to the simple fact that you cannot have two distinct instructions that both write a value to the same temporary.

6. To address the restriction of SSA, LLVM uses ϕ *nodes* to merge the output values coming from different path (branches) in a program. The necessity of ϕ nodes is obvious when you consider that temporaries can only be written to once as you cannot have two paths in the program writing to the same temporary! We will see an example shortly and it is essential for writing branching or looping programs.

```
1  ; ModuleID = "max.llvm"
2  declare i32 @printf(i8*,...) ; import the prototype of printf
3
4  @msg = constant [9 x i8] c"got: %d\0A\00"
5
6  define i32 @max(i32 %a,i32 %b) {
7  entry:
8      %t0 = icmp sle i32 %a,%b
9      br i1 %t0, label %then, label %else
10 then:
11     %t1 = and i32 %b,%b
12     br label %exit
13 else:
14     %t2 = and i32 %a,%a
15     br label %exit
16 exit:
17     %t3 = phi i32 [%t1,%then],[%t2,%else]
18     ret i32 %t3
19 }
20
21 define i32 @main() {
22     %t1 = call i32 @max(i32 5,i32 10)
23     %t2 = call i32(i8*,...)* @printf(i8* bitcast ([9 x i8]* @msg to i8*),i32 %t1)
24     ret i32 %t1
25 }
```

Figure 2: Computing a max in LLVM.

Simple Program

Consider the following C function for computing a maximum and a corresponding test program:

```
int max(int a, int b) {
    if (a < b)
        return b;
    else return a;
}
int main() {
    extern int printf(char* format,...);
    int x = max(5,10);
    printf("max is %d\n",x);
    return 0;
}
```

Its purpose is to implement a maximum function and to test it. Now consider the equivalent LLVM program shown in Figure 2 and whose source can be pasted from `http://www.pastie.org/1509023`

walk-through

The program is easy to follow. Line 1 shows a comment (starting with a semi-colon).

Line 2 declares the name of an external symbol (`printf`) living in a library. Notice how it specifies the return type ($i32$, i.e., a 32-bit wide integer) as well as the type of the first argument ($i8*$, a pointer to an 8-bit wide piece of data – a pointer to a byte –).

Line 4 declares a `global` constant named *msg*. Again the constant is typed, it is an array of 9 8-bit wide pieces of data and the constant value follows. Notice how special characters like the C newline are given as hexadecimal sequences.

Line 6-19 define the max function (named @max since all globals names start with @). The function returns a 32-bit wide integer and takes as input two 32-bit wide integers named %a and %b (note how % is used to refer to a the name of a local).

The function has several labeled *blocks* starting with *entry* and continuing with *then*, *else* and *exit*. The block names have no special significance.

Line 8 starts the entry block that performs a comparison of two 32-bit wide integers (%a and %b) using the "signed less than or equal to" operator. The result is 1-bit wide and is stored in the temporary %t0.

Line 9 branches to a block (either %then or %else) based on the value of the one bit stored in %t0. The %then block starts on line 10 and copies %b into a fresh temporary %t1.

Line 12 then jumps to the exit block. The %else block is defined similarly. The %exit block uses the ϕ node instruction on line 17. Notice how the output of each block is in different temporaries (%t1 and %t2) since one cannot write to a temporary more than once. The purpose of the ϕ node is to let the output of the correct block flow into a fresh temporary %t3. It states the type of the result ($i32$) and list as many bracketed pairs as there are blocks leading to %exit. Each pair specifies the value flowing out of a block and the identity of the block. To paraphrase, the instruction says that the value in %t3 is either %t1 or %t3 depending on whether the execution went through the %then block or the %else block.

Lines 21-25 define the main routine (named, of course, @main). Line 22 places a call to @max, passing two 32-bit wide operands (5 and 10). The $i32$ type in front of the call is the type of the result which is stored in the fresh temporary %t1. Clearly, the temporaries are local to a function and the %t1 here is completely different from the %t1 appearing in the @max implementation.

Line 23 places a call to the C library function `printf`. The call is slightly more complex given that `printf` takes a variable number of arguments and requires some casting. In essence, the type after the `call` keyword is the type of the function itself, i.e., it is a pointer to a function. Here `printf` takes two arguments. The first is a pointer to the formatting string. The second is the temporary %t1. The last bit of complexity comes from the necessity to use a cast to convert the type of the argument to the type of the formal. Indeed, the argument (@msg) is an array of 9 bytes whereas the type of the first formal in `printf` is a *pointer* to a byte. As you surely know, The address of an array is nothing but the address of its first element. Therefore @msg is not only the name of an array of 9 bytes, it is also the address of the array of 9 bytes and therefore it is a pointer to its first byte. One simply needs to inform LLVM of that fact with a *bitcast* instruction which says: "the operand @msg is a pointer to an array of

9 bytes `[9 x i8*]*` that we *convert* into a pointer to a byte `i8*`". In practice, this conversion does absolutely nothing. Its sole purpose is to tell LLVM that you really meant to impose that interpretation of the pointer.

3 Assignment

Now that we have all the details down, you must write three test programs in LLVM.

FACT1 This program computes the factorial function for the numerical input 5 (hard-coded), relies on an iterative implementation and prints out the result via `printf`. The output is on a single line and should use the format string "got: %d`\n`" where %d refers to the computed value. (Naturally, printing "got: 120" is *cheating*. You must write an iterative program that truly computes the result.)

FACT2 This program computes the factorial function for the numerical input 5 (hard-coded) with a recursive implementation and prints out the result via `printf`. The output is on a single line and should use the format string "got: %d`\n`" where %d refers to the computed value. (Naturally, printing "got: 120" is *cheating*. You must write an iterative program that truly computes the result.)

FIB This program computes the fibonacci function for the numerical input 6 (hard-coded) and prints out the result via `printf`. Once again, the output should be a single line and use the format string "got: %d`\n`" where %d refers to the computed value.

Don't forget that you must not only write the code, but read all the documentation and thoroughly test your code.

The outputs of the three programs should be:

```
slivewl:hw1 ldm$ ./fact1
got: 120
slivewl:hw1 ldm$ ./fact2
got: 120
slivewl:hw1 ldm$ ./fib
got: 8
```

where `slivewl:hw1 ldm$` is the Operating System prompt at the terminal and `./fact1` is the command I typed to run the first program (the same applies for the next two programs).

Have fun!