UE19CS351 - Compiler Design

Session: Jan 2022 – May 2022

MANUAL

Semester:	VI
Problem Statement :	LLVM Exercises
Course Anchor :	Prof. Preet Kanwal
Teaching Assistant :	Anirudh H M
	Sowmya Prasad

Table of Contents

1 Introduction

This document contains a few guided exercises to familiarize you with the LLVM language and tooling. In the following tasks, you will learn to use various tools in the LLVM infrastructure to perform tasks in the compilation pipeline, and learn to write simple programs in LLVM IR code.

This manual is created on the Linux platform, and appropriate adjustments must be made for other platforms. It is assumed that you have the LLVM toolchain set up on your system. Most of the tasks will also use the C programming language, so basic knowledge of it is required.

2 Using the LLVM tools

In this section, we will use various LLVM tools to compile a program in stages, and observe the outputs of each stage. For the following tasks, we will be using a simple C program, split across two files -

```c
// main.c

#include <stdio.h>
int sum(int x, int y);
int main() {
  int r = sum(3, 4);
  printf("r = %d\n", r);
  return 0;
}
```

```c
// sum.c

int sum(int x, int y) {
  return x + y;
}
```

As a quick refresher, this program can be compiled using the `clang` compiler as follows -

- `clang -c sum.c -o sum.o && clang -c main.c -o main.o && clang main.o sum.o -o sum`
- In a single step: `clang main.c sum.c -o sum`

2.1 Generating LLVM bitcode and IR code

Since `clang` is a compiler frontend,we can use it to generate LLVM bitcode and IR code from our programs as follows -

- To generate bitcode:

```
clang -O0 -emit-llvm -c main.c -o main.bc
clang -O0 -emit-llvm -c sum.c -o sum.bc
```

Explanation of options:

`-O0` Do not run optimization on code

`-emit-llvm` Output LLVM IR code

`-c` Generate LLVM bitcode

This will generate binary files `main.bc` and `sum.bc`, which contain LLVM IR code in bitcode representation

- To generate human-readable IR code:

```
clang -O0 -emit-llvm -S -c main.c -o main.ll
clang -O0 -emit-llvm -S -c sum.c -o sum.ll
```

Explanation of options:

`-S` Generate LLVM human-readable code

This will generate human-readable text files `main.ll` and `sum.ll`, which contain LLVM IR code in human readable representation

Note: The `-S` and `-c` flags, when used without the `-emit-llvm` flag, generate machine-dependent assembly code and object files respectively.

2.1.1 Understanding the generated LLVM code

From the previous step, we have generated two LLVM IR code files. Reading through `main.ll`, we can observe the following features - (a sample output is given below for reference)

```
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [8 x i8] c"r = %d\0A\00", align 1

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %3 = call i32 @sum(i32 3, i32 4)
  store i32 %3, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = call i32
  ,→(i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]* @.str, i64 0, i64 0), i32 %4)
  ret i32 0
}

declare i32 @sum(i32, i32) #1

declare i32 @printf(i8*, ...) #1

attributes #0 = { noinline nounwind optnone sspstrong uwtable "frame-pointer"="all" "min-legal-vector
    width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
,→
    "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
,→
attributes #1 = { "frame-pointer"="all"
    "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
,→
    "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
,→

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"clang version 13.0.0"}
```

- The first three lines contain metadata about the program
  - `source_filename` defines the name of the source file from which this code was generated
  - `target datalayout` defines how data is stored in memory on the current

system

- `target triple` defines the target platform on which this code was generated

- `@.str` defines a global variable named `.str`, which contains the output format string we defined in the program

- The `main` function is defined, returning an `i32` and having one basic block in its body. Note that the basic block was automatically numbered with `#0`

- The body of the `main` function contains memory allocations, memory storage and retrieval and function calls. Temporary variables are automatically created, starting from `%0`. Note that some instructions are redundant, and will be eliminated during the optimization stages

- The `sum` and `printf` functions are declared. These functions are defined in other files, and the definitions are retrieved during the linking stage of compilation

- The rest of the file contains metadata about LLVM configuration and platform-specific information, which is used for optimization

Exercise: Add the `-fno-discard-value-names` flag while generating human-readable code and note the differences. This flag instructs the compiler to retain variable names from the source code instead of replacing them with unnamed temporaries.

2.2 Converting bitcode to human-readable code and vice-versa

We can convert LLVM bitcode to human-readable assembly (also known as **disassembly**), and human-readable assembly to LLVM bitcode (also known as **assembly**), using the `llvm-dis` and `llvm-as` tools respectively. To convert the files created in the previous task -

- Convert to bitcode:

```
llvm-as main.ll -o conv_main.bc
llvm-as sum.ll -o conv_sum.bc
```

- Convert to human-readable assembly: `llvm-dis main.bc -o conv_main.ll && llvm-dis sum.bc -o conv_sum.ll`

```
llvm-dis main.bc -o conv_main.ll
llvm-dis sum.bc -o conv_sum.ll
```

Compare the respective files to their `conv_` variants. You will observe that there is no difference between the two, demonstrating that the two representations of LLVM code are interchangeable i.e, no data is added/removed during conversion between the two formats *Exercise*: There is *one* difference between the files generated by `clang` and those generated through conversion by `llvm-as/dis`. Find out what it is, and why it happens (Ans: The module name changes from `main.c/sum.c` to `main.bc/sum.bc`, since the file from which the code is generated changes)

2.3 Linking and converting bitcode to object files

Once LLVM bitcode is generated, it can be used to generate machine-specific assembly code, or native object files (`.o` files, which we have seen earlier). For this, we use the `llc` static compiler

- To generate machine-dependent assembly:

```
llc -filetype=asm main.bc -o main.s
llc -filetype=asm sum.bc -o sum.s
```

- To generate native object files:

```
llc -filetype=obj main.bc -o main.o
llc -filetype=obj sum.bc -o sum.o
```

To verify that this works as expected, we can link the object files together to obtain the final executable:

```
clang main.o sum.o -o sum
./sum
```

Note: The linker may display some warnings at this stage, which can be safely ignored

While the linker can resolve function definitions across object files, we can also perform linking at the IR level using the `llvm-link` tool. This tool takes several LLVM bitcode files and generates a single bitcode file that contains resolved symbols. This file can then be compiled and executed as usual. We can use the tool as follows -

- To create the combined bitcode file:

```
llvm-link main.bc sum.bc -o combined.bc
```

- To compile and run it:

```
llc -filetype=obj combined.bc -o combined.o
clang combined.o -o sum
./sum
```

Exercise: `ld` is the standard GNU linker, used for linking object files. However, we choose to use `clang` for linking instead of `ld`, and using `ld` raises an error. Why does this happen? (Answer: The source code uses `printf`, which is defined in the C standard library. This library also needs to be linked, which does not happen while using `ld`).

2.4 Conclusion

In this section, we have learnt how to use the LLVM tools to

1. Convert source code into various LLVM intermediate formats
2. Change the representation of LLVM intermediate code
3. Convert LLVM intermediate code to executables

3 Writing programs in LLVM IR

In this section, we will learn more about the LLVM IR code by modifying and writing some simple programs in the LLVM IR language. As observed in the previous section, the `clang` command adds various metadata fields to the intermediate code, which are required for compilation. However, we do not want to manually add these fields. Thus, we will be using another LLVM tool, `lli`, to execute our intermediate code without compiling it to a native executable. This tool can be used as follows - (assuming the human-readable code is in `main.ll`)

```
llvm-as main.ll -f -o - | lli
```

Explanation of flags:

`-f` Force writing of binary data to stdout

`-o -` Write output to stdout (does not work without `-f`)

3.1 Hello world ++

In this exercise, we will be modifying the LLVM IR code generated for a simple “hello world” program to change its behavior. We will add a new string to the program, printing out our name. The reference program is given below -
[alt]

```
@.str = constant [13 x i8] c"Hello world\0A\00"

define dso_local i32 @main() {
  %printstr = getelementptr [13 x i8], [13 x i8]* @.str, i64 0, i64 0
  %1 = call i32 (i8*, ...) @printf(i8* %printstr)
  ret i32 0
}

declare i32 @printf(i8*, ...)
```

This can be modified as follows -

- First, we declare a new string variable with the string to print out. Be sure to change the length of the vector holding the string accordingly after inserting your name into the string
- Next, we call the `printf` function with the appropriate signature. Recall that `printf` returns an integer denoting the number of characters printed to screen, and takes in a string and a variable number of arguments
  - `getelementptr` is a LLVM instruction to compute the address of a pointer. It works as follows -

1. The first argument (`[13 x i8]`) is the type of the value to which. Here, we are computing the pointer to a value of type vector, with 13 `i8` elements

2. The second argument (`[13 x i8]* @.str`) is the base pointer, from which offsets are computed. Here, it is a pointer to the location of the global value `@.str`, the string to be printed

3. The third argument (`i64 0`) is the offset from the base pointer. Here, we are getting the element at an offset 0 from the base pointer (i.e, at the location of the base pointer)

4. The fourth argument (`i64 0`) is the offset within the element referred to by the previous offset. Here, we are getting the first element within the vector of `i8`

Finally, we get a pointer of type `i8*`, which points to the first character of the string we want to print
Note: Why can't we directly use `@.str`? `@.str` is a pointer to a vector of `i8`, not a pointer to `i8`. Thus, the types do not match

The final LLVM code appears as follows, and can be executed as described above.

```
@.helloworld = constant [13 x i8] c"Hello world\0A\00"
; Declare new string
@.myname = constant [29 x i8] c"My name is <your name here>\0A\00"
define dso_local i32 @main() {
  %hello_world_pointer = getelementptr [13 x i8], [13 x i8]* @.helloworld, i64 0, i64
  0 %my_name_pointer = getelementptr [29 x i8], [29 x i8]* @.myname, i64 0, i64 0
  %1 = call i32 (i8*, ...) @printf(i8* %hello_world_pointer)
  %2 = call i32 (i8*, ...) @printf(i8* %my_name_pointer)
  ret i32 0
}

declare i32 @printf(i8*, ...)
```

Exercise: Print out your name and age, separated by a horizontal tab character (`\t`). Note that you cannot directly use `\t`, since LLVM does not support C-like escape characters

3.2 Counting C's

In this simple program, we will be writing a program to count the number of times the character 'c' occurs in a string. We will define an LLVM function to perform the computation, and call it from our main function. We will be working off the following template - (note that it does not run as is)

```
@.targetstr = constant [22 x i8] c"counting c's is cool!\00"
@.formatstr = constant [22 x i8] c"Number of c's is: %d\0A\00"

define dso_local i32 @main() {
  %targetptr = getelementptr [22 x i8], [22 x i8]* @.targetstr, i32 0, i32 0
  %1 = call i32 @countc(i8* %targetptr, i32 22)
  %formatptr = getelementptr [22 x i8], [22 x i8]* @.formatstr, i64 0, i64 0
  %2 = call i32 (i8*, ...) @printf(i8* %formatptr, i32 %1)
  ret i32 0
}

define dso_local i32 @countc(i8* %string, i32 %string_length) {
  ; To be completed
}

declare i32 @printf(i8*, ...)
```

To complete the `countc` function, we will be using a few important instructions -

`alloca` Dynamically allocaesmemory fora variable. Ex:`%x = alloca i32`(allocates memory for an i32 variable)

`load` Loads a value from a memory location. Ex: `%x = load i32, i32* %ptr` (loads a value of type `i32` from the location pointed to by `%ptr`)

`store` Stores a value into a memory location. Ex: `store i32 %x, i32* %ptr` (stores a value of type `i32` from `%x` to the location pointed to by `%ptr`)

icmp Compares two values, using an opcode to determine the type of comparison. Ex: `%cond = icmp slt i32 %x, i32 %y` (checks if the value of `%x` is less than the value of `%y`. `slt` stands for 'signed less than')

`br` Branch instruction. Can be either a conditional or unconditional branch. Ex: `br i1 %cond, label %1, label %2` (checks `%cond`, jumps to label `%1` if true and label `%2` if false)

The function may be implemented as follows -

Note: This is only one possible implementation, it is recommended to try and implement this function yourselves in different ways .

```
define dso_local i32 @countc(i8* %string, i32 %string_length) {
  %c_count = alloca i32
  %index = alloca i32
  store i32 0, i32* %c_count
  store i32 0, i32* %index
  br label %1

1:
  %idx_1 = load i32, i32* %index
  %cmp_1 = icmp slt i32 %idx_1, %string_length
  br i1 %cmp_1, label %2, label %5

2:
  %idx_2 = load i32, i32* %index
  %charptr = getelementptr i8, i8* %string, i32 %idx_2
  %char = load i8, i8* %charptr
  %cmp_2 = icmp eq i8 %char, 99
  br i1 %cmp_2, label %3, label %4

3:
  %cnt = load i32, i32* %c_count
  %cnt_inc = add i32 %cnt, 1
  store i32 %cnt_inc, i32* %c_count
  br label %4

4:
  %idx_3 = load i32, i32* %index
  %idx_3_inc = add i32 %idx_3, 1
  store i32 %idx_3_inc, i32* %index
  br label %1

5:
  %retval = load i32, i32* %c_count
  ret i32 %retval
}
```

Note that this function consists of multiple basic blocks, and uses labels and branch instructions to create a loop construct.

3.3 Fibonacci sequence

In this exercise, you will implement an LLVM function to compute the nth element of the Fibonacci sequence. Invalid indices, such as negative numbers and 0, need not be handled. The sequence is expected to start with 0 i.e, the sequence runs as <0,1,1,2,3,5...>. The following template can be used -

```
@.formatstr = constant [28 x i8] c"%d Fibonacci number is: %d\0A\00"
@.elem = constant i32 5 ; Find 5th fibonacci number

define dso_local i32 @main() {
  %elem = load i32, i32* @.elem
  %fibnum = call i32 @fib(i32 %elem)
  %formatptr = getelementptr [28 x i8], [28 x i8]* @.formatstr, i64 0, i64 0
  %1 = call i32 (i8*, ...) @printf(i8* %formatptr, i32 %elem, i32 %fibnum)
  ret i32 0
}
define dso_local i32 @fib(i32 %elem) {
  ; To be completed
}

declare i32 @printf(i8*, ...)
```

This can be implemented using the looping construct used in the previous exercise. A possible implementation could be -
Note: This is only one possible implementation, it is recommended to try and implement this function yourselves in different ways

```
define dso_local i32 @fib(i32 %elem) {
  %cur_val = alloca i32
  %prev_val = alloca i32
  %index = alloca i32
  store i32 0, i32* %prev_val
  store i32 1, i32* %cur_val
  store i32 2, i32* %index
  br label %1

1:
  %cmp_1 = icmp eq i32 %elem, 1
  br i1 %cmp_1, label %2, label %3

2:
  %retval_1 = load i32, i32* %prev_val
  ret i32 %retval_1

3:
  %cmp_2 = icmp eq i32 %elem, 2
  br i1 %cmp_2, label %6, label %4

4:
  %idx_1 = load i32, i32* %index
  %cmp_3 = icmp slt i32 %idx_1, %elem
  br i1 %cmp_3, label %5, label %6

5:
  %old_cur = load i32, i32* %cur_val
  %old_prev = load i32, i32* %prev_val
```

```
  %new_cur = add i32 %old_prev, %old_cur
  store i32 %new_cur, i32* %cur_val
  store i32 %old_cur, i32* %prev_val

  %idx_2 = load i32, i32* %index
  %idx_2_inc = add i32 %idx_2, 1
  store i32 %idx_2_inc, i32* %index
  br label %1

6:
  %retval_final = load i32, i32* %cur_val
  ret i32 %retval_final
}
```

3.4 Conclusion

In this section, we have learnt how to read, modify and write LLVM IR code by implementing three simple programs. This should help you become more comfortable in reading and understanding LLVM IR code.

4 Optimization using LLVM

In this section, we will learn about how LLVM performs optimizations on LLVM IR code, and use LLVM tools to execute these optimizations.

4.1 Introduction

- In LLVM, optimizations are implemented as **passes** that traverse some portion of a program to either collect information or transform the program

- Passes are divided into three categories -

  Analysis passes Compute information that other passes can use or for debugging or program visualization

  Transform passes Can use (or invalidate) the analysis passes. Mutate the program in some way

  Utility passes Provide some miscellaneous utility that does not fit other categories. For example, passes to extract functions to bitcode or write a module to bitcode are neither analysis nor transform passes

- `opt` is a modular LLVM analyzer and optimizer. It takes the input LLVM IR code file and runs the optimization or analysis passes specified on the command line. In general, the tool is used as follows -

  ```
  opt <options> [input_file]
  ```

  – Input can be a **binary code file** (with a .bc extension), or an **LLVM IR file** (with a .ll extension). In this exercise we will only work with LLVM IR files as input

- *Note*: In March 2021, LLVM introduced a new pass manager, which changes how `opt` is used to run passes. In this manual, we will prefer to use the syntax of the new pass manager. However, if you would like to use the old pass manager, you can add the flag `-enable-new-pm=0` and use the old syntax

4.2 Analysis pass

For this exercise, we will use a simple C program to replace all occurrences of a word in a string with a replacement word

```c
// word.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char *replaceWord(const char *s, const char *oldW, const char *newW) {
  char *result;
  int i, cnt = 0;
  int newWlen = strlen(newW);
  int oldWlen = strlen(oldW);

  for (i = 0; s[i] != '\0'; i++) {
    if (strstr(&s[i], oldW) == &s[i]) {
      cnt++;
      i += oldWlen - 1;
    }
  }

  result = (char *)malloc(i + cnt * (newWlen - oldWlen) + 1);
  i = 0;
  while (*s) {
    if (strstr(s, oldW) == s) {
      strcpy(&result[i], newW);
      i += newWlen;
      s += oldWlen;
    } else {
      result[i++] = *s++;
    }
  }
  result[i] = '\0';
  return result;
}

// Driver Program
int main() {
  char str[] = "A jellyfish stinged me on the beach.";
  char c[] = "stinged";
  char d[] = "stung";

  char *result = NULL;
  printf("Old string: %s\n", str);
  result = replaceWord(str, c, d);
  printf("New String: %s\n", result);
  free(result);
  return 0;
}
```

As in the previous exercises, we can generate human-readable IR code with

```
clang word.c -S -emit-llvm -Xclang -disable-O0-optnone -fno-discard-value-names
clang word.c -S -emit-llvm -Xclang -disable-O0-optnone -fno-discard-value-names
```

Explanation of options:

`-Xclang -disable-O0-optnone` Pass option to clang, indicating that optimizations should not be performed during code generation, but manual optimizations can be performed on the intermediate code later

4.2.1 Loop information

This analysis is used to identify natural loops and determine the loop depth of various nodes of the CFG. To run it using `opt` -

```
# With old PM: opt -enable-new-pm=0 -analyze -loops word.ll
opt -passes='function(print<loops>)' -disable-output word.ll
```

A reference output is given below -

```
Loop at depth 1 containing: %while.cond<header><exiting>,%while.body,%if.else,%if.then21,%if.end29<latch>
Loop at depth 1 containing: %for.cond<header><exiting>,%for.body,%if.then,%if.end,%for.inc<latch>
```

As expected, this shows that all the loops in our program are one level deep.

4.2.2 Call graph

This analysis option prints the call graph to standard error in a human-readable form.

```
# With old PM: opt -enable-new-pm=0 -analyze -print-callgraph word.ll
opt -passes='module(print-callgraph)' -disable-output word.ll
```

A reference output is given below -

```
Call graph node <<null function>><<0x16a8370>> #uses=0
CS<0x0> calls function 'replaceWord'
CS<0x0> calls function 'strlen'
CS<0x0> calls function 'strstr'
CS<0x0> calls function 'malloc'
CS<0x0> calls function 'strcpy'
CS<0x0> calls function 'main'
CS<0x0> calls function 'llvm.memcpy.p0i8.p0i8.i64'
CS<0x0> calls function 'printf'
CS<0x0> calls function 'free'
Call graph node for function: 'free'<<0x16b28f0>> #uses=2
CS<0x0> calls external node
Call graph node for function: 'llvm.memcpy.p0i8.p0i8.i64'<<0x16b29a0>>
#uses=1

Call graph node for function: 'main'<<0x16b2850>> #uses=1
CS<0x16e2a58> calls function 'printf'
CS<0x16e2e40> calls function 'replaceWord'
CS<0x16e3088> calls function 'printf'
CS<0x16e34e0> calls function 'free'
Call graph node for function: 'malloc'<<0x16b26e0>> #uses=2
CS<0x0> calls external node
Call graph node for function: 'printf'<<0x16b28c0>> #uses=3
CS<0x0> calls external node
Call graph node for function: 'replaceWord'<<0x16b2150>> #uses=2
```

```
CS<0x16dc0d0> calls function 'strlen'
CS<0x16dc500> calls function 'strlen'
CS<0x16dd398> calls function 'strstr'
CS<0x16debb0> calls function 'malloc'
CS<0x16df298> calls function 'strstr'
CS<0x16e0038> calls function 'strcpy'
Call graph node for function: 'strcpy'<<0x16b2750>> #uses=2
CS<0x0> calls external node
Call graph node for function: 'strlen'<<0x16b21f0>> #uses=3
CS<0x0> calls external node
Call graph node for function: 'strstr'<<0x16b2220>> #uses=3
CS<0x0> calls external node
```

This shows the hierarchy of function calls within the program. Note that this also includes standard library functions(`printf, strcpy` etc.) and LLVM intrinsic functions(`llvm.memcpy`), not only the user-defined ones.

Exercise: Use the `helloworld` pass in a similar way to the above options and observe the output. What does it do? (Ans: It prints the names of all functions in the IR code)

4.3 Transform pass

`opt` uses the same set of optimization flags found in the Clang compiler driver: `-O0, -O1, -O2, -O3, -Os, -Oz`. These flags can be used to run a predefined set of optimization passes on any code, same as clang.

```
opt <filename.ll> -O3 -o <outputfilename.ll>
```

4.3.1 Mem2reg and time-passes

In this exercise, we will explore the `mem2reg` pass. This pass promotes `alloca` instructions to LLVM local values, converting them to use the SSA form if they receive multiple assignments when converted into a local value using the `phi` function. `mem2reg` is included at `O1` optimization level and above.

We will use the following code for this section.

```c
// sum.c
#include <stdio.h>

int main() {
  int x = 1, y = 2, z, sum;
  sum = x + y;
  printf("%d", sum);
  return 0;
}
```

```
clang sum.c -S -emit-llvm -Xclang -disable-O0-optnone -fno-discard-value-names
```

Before running `mem2reg`, the IR code appears as follows (metadata is removed for easier reading) -

```
@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1

define dso_local i32 @main() {
entry:
  %retval = alloca i32, align 4
  %x = alloca i32, align 4
  %y = alloca i32, align 4
  %z = alloca i32, align 4
  %sum = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 1, i32* %x, align 4
  store i32 2, i32* %y, align 4
  %0 = load i32, i32* %x, align 4
  %1 = load i32, i32* %y, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %sum, align 4
  %2 = load i32, i32* %sum, align 4
  %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
  [3 x i8]* @.str, i64 0, i64 0), i32 %2)
  ret i32 0
}

declare dso_local i32 @printf(i8*, ...)
```

To run the `mem2reg` optimization pass,

```
# With old PM: opt -enable-new-pm=0 -S sum.ll -mem2reg -o sumMemReg.ll
opt -S sum.ll -passes='mem2reg' -o sumMemReg.ll
```

After running `mem2reg`, the optimized code appears as follows (metadata is removed for easier reading) -

```
@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1

define dso_local i32 @main() {
entry:
  %add = add nsw i32 1, 2
  %call = call i32 (i8*,
  ↪...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32 %add)
  ret i32 0
}

declare dso_local i32 @printf(i8*, ...)
```

Notice that now there are only two instructions(one `add` and one `ret` instruction) along with a function call in the body of `main`. All the `load`, `store` and `alloca` instructions have been removed.

We will now use the `-time-passes` flag to show the amount of time spent on each pass

```
# With old PM: opt -enable-new-pm=0 -S sum.ll -mem2reg -time-passes
-disable-output
opt -S sum.ll -passes='mem2reg' -time-passes -disable-output
```

Your output should look something like this:

```
  ===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
 ===-------------------------------------------------------------------------===
  Total Execution Time: 0.0002 seconds (0.0002 wall clock)

   ---User Time--- --System Time-- --User+System-- ---Wall Time--- --- Name --- 0.0000 (
   39.3%) 0.0000 ( 39.2%) 0.0001 ( 39.3%) 0.0001 ( 39.6%) PrintModulePass 0.0000 ( 22.2%)
   0.0000 ( 22.8%) 0.0000 ( 22.4%) 0.0000 ( 22.0%) VerifierPass 0.0000 ( 17.1%) 0.0000 (
   17.7%) 0.0000 ( 17.3%) 0.0000 ( 18.1%) PromotePass 0.0000 ( 15.4%) 0.0000 ( 15.2%) 0.0000
   ( 15.3%) 0.0000 ( 14.0%) VerifierAnalysis 0.0000 ( 3.4%) 0.0000 ( 2.5%) 0.0000 ( 3.1%)
   0.0000 ( 3.6%) DominatorTreeAnalysis 0.0000 ( 2.6%) 0.0000 ( 2.5%) 0.0000 ( 2.6%) 0.0000
   ( 2.7%) AssumptionAnalysis 0.0001 (100.0%) 0.0001 (100.0%) 0.0002 (100.0%) 0.0002
   (100.0%) Total

 ===-------------------------------------------------------------------------===
                           LLVM IR Parsing
 ===-------------------------------------------------------------------------===
  Total Execution Time: 0.0003 seconds (0.0003 wall clock)

   ---User Time--- --System Time-- --User+System-- ---Wall Time--- --- Name ---
   0.0002 (100.0%) 0.0001 (100.0%) 0.0003 (100.0%) 0.0003 (100.0%) Parse IR
   0.0002 (100.0%) 0.0001 (100.0%) 0.0003 (100.0%) 0.0003 (100.0%) Total
```

Exercise: Use the `instcount` and `stat` flags with the above instruction. This will indicate the number of instructions, basic blocks and other useful statistics about the generated IR code. Note that you will most probably need a debug build of LLVM to run this.

4.4 Utility pass

In this exercise, we will generate and view the control flow graph (CFG) for a program. We will be using the `word.ll` IR code generated earlier for the analysis pass exercise. For this exercise, we recommend installing the `xdot` package, which helps visualize dot graphs.

To generate the CFG in dot format -

```
# With old PM: opt -enable-new-pm=0 -analyze -view-cfg word.ll
opt word.ll -passes='function(view-cfg)' -disable-output
```

This command generates the CFG in dot format, and opens the temporary files with the default application for dot files (ideally `xdot`). This will produce a control flow graph with 13 basic blocks, in line with our code.

Exercise: Use the `-cfg-func-name=<substring>` flag to filter the functions that are displayed. All functions that contain the specified `substring` will be displayed.

4.5 Conclusion

In this section, we have learnt about how LLVM handles optimization of intermediate code through passes, and we have used the `opt` tool to manually run LLVM passes on IR code. You should now be familiar with LLVM IR code and the commonly used tools in the LLVM ecosystem.