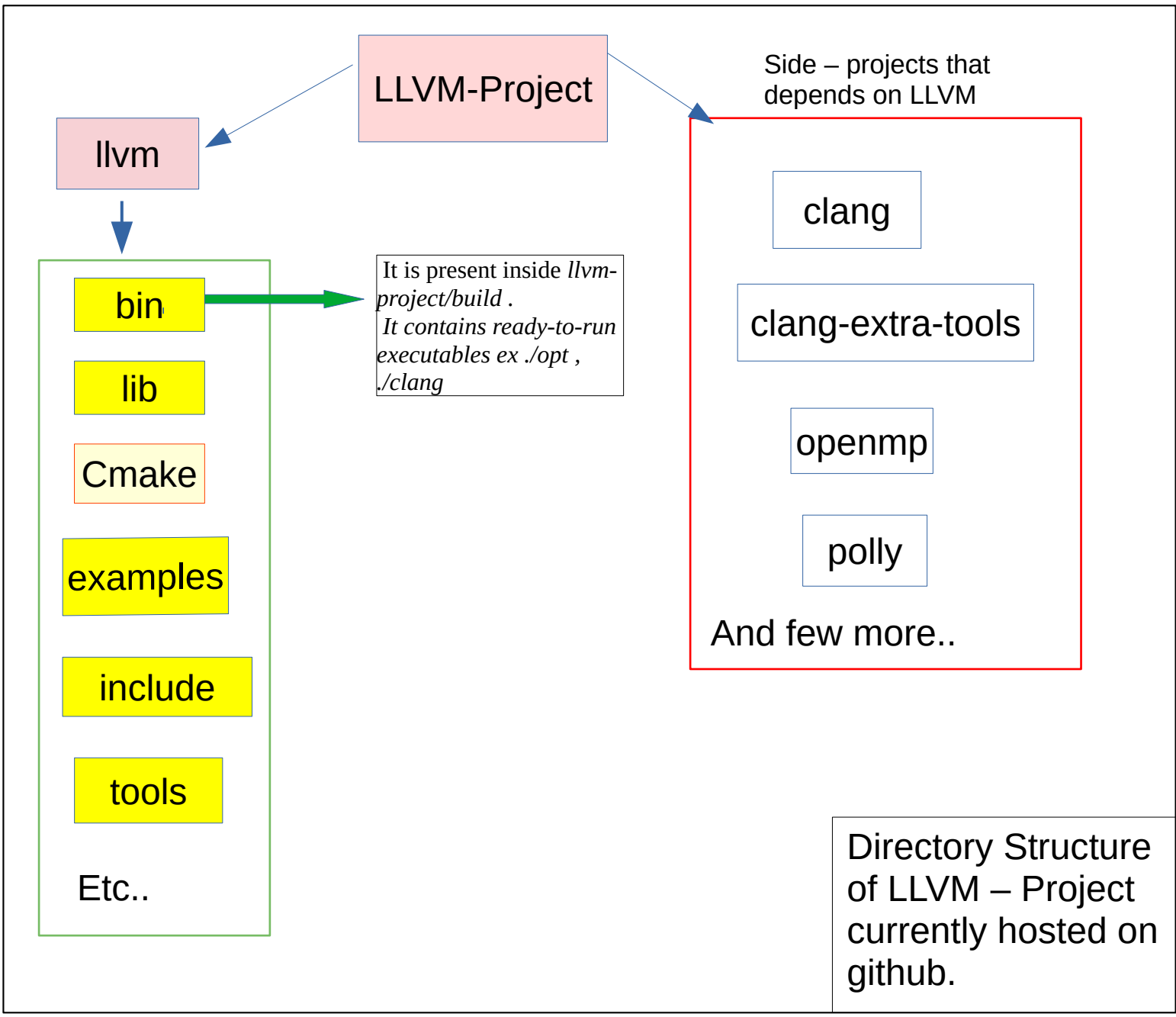
1) Understanding of LLVM Directory structure:



Clang:- Frontend for c/c++ and objetive c, WHEN USED WITH LLVM INFRASTUCTURE. It also in itself is a full fledged compiler. It has many flags and attributes just like GCC to control it. It also does program understanding / static analysis.

OpenMP:- aka Open Multi-Processing API : provides compiler directives for various architectures for c/c+ + ,Fortran.It works over shared memory programming model where multiple threads can access same memory location.

Polly:- Pollyhedral optimization infrastructure that uses affine transformations over loops and data locations.

/llvm /

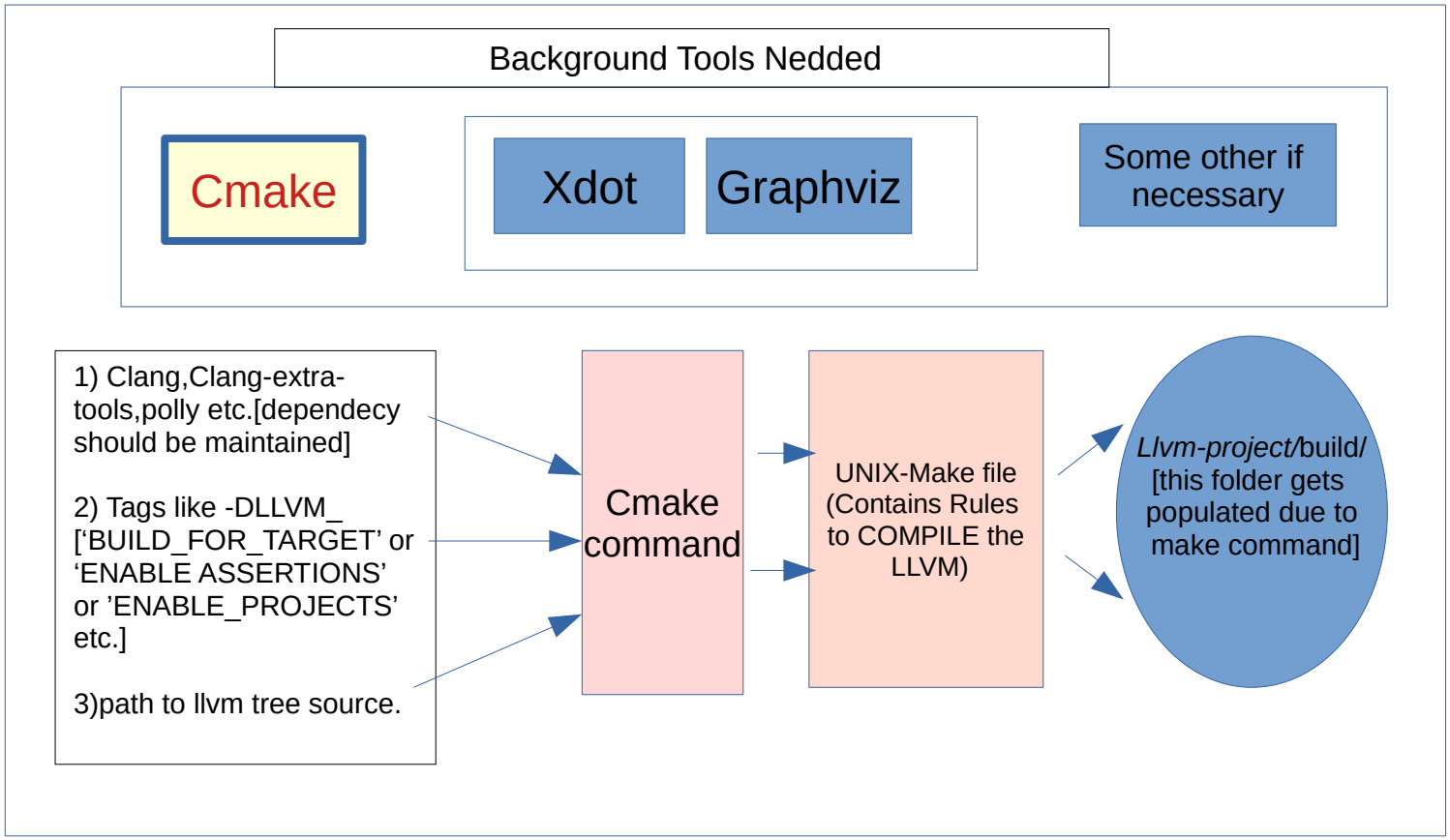
lib:- contains the .cpp files and correponding cmake files in a folder which will be compiled via Make. Further these executables will give facility to run various optimization passes , Linking to generate final excutable etc.

Examples:- it contains the snippets showing how to use LLVM as a compiler for a custom language - including lowering, optimization, and code generation.
Lowering:- it's a mapping of more comlex PL constructs into the simpler ones . Ex: In a PL called Dlang , all the do while and while loops are converted into for loops.

Include:- it contains all the necessary header files needed by code files inside /lib or support libraries not specific to llvm etc.

Tools:- it contains assembler(llvm-as),linker(llvm-link or llvm-lld),backend compiler(llc), JIT based interpreter (lli) , dissasembler (llvm-dis) , library archiver(llvm-ar), opt [this as told by rama sir contains those tranformation passes that are benchmarked to produce better code].

1.a) How to build LLVM and Other needed tools from source



Cmake: Before discussing about cmake first let's see what is make file. For unix-like OS a make file is a sequence of shell commands [hence which shell you are using that is imp.] that serves as a rule to compile (recompile) a series of files. Make is a utility that executes this makefile.
One very important point about Make is keeps track of the last time files (normally object files) were updated and only updates those files which are required (ones containing changes) to keep the sourcefile up-to-date. And this has actually saved the memory problem or compilation error and process getting terminated while building the llvm.

Now Cmake is a system that runs before the actual build system(ex unix-Makefiles, or Ninja-Build). For the softwares like llvm which extensively large, complex and ever – evolving with respect to cross compilation, new passes, new side projects (IR2Vec maybe) it is intarctable to hand-generate makefiles for all OS, Architecture combinations hence Cmake via special files called CmakeLists.txt and a Language of its own generates the needed file for underlying build system.

Xdot & Graphviz: Many analysis pass generate visual debugs like domtree, CFG, call graph , dominace frontier etc. Hence these files have special extension .dot . To view these we need Xdot which used Graphviz to render the needed structure.

Other needed: other build systems like ninja- build , or visual code build .

2) LLVM-IR Understanding

SOME INTRO:- it is a turing complete language thus any computable function can be represented hence very suitable to be an IR for languages like C/C++ . It follows SSA form which enables many efficient and fast optimization to crunch out the code and "try to" make it " better" . It's statements follows three-address code form which is "at max 1 operator, 2 operands on RHS and 1 operand on LHS where operands could be GPR's or memory locations.

One impotant point about LLVM Code representation is lossless translation between below three formats

LLVM IR == LLVM Bit code == LLVM Assembly

it's significance 1) easy translation into machine dependent asm 2) many modules (like opt) inside llvm can use either of these files (opt works on .ll or .bc)

LLVM Bitcode:- Basically a code representaion which is persistent as it is in bit-stream form residing on disk which is not human redable.

SOME COMMON INFO PRESENT IN A .LL FILE

1) Module ID: name of the compiled source file when compiling from source through the clang front end.

2) Source Filename: equal to Module ID.

3) target datalayout: this is a large list of tags . This directive basically specifies data layout in memory for the program . It controls Little endian/Big endian orientation , pointer size, allignment of data objects, etc.

4) target triple: Describes the target host ex. "x86_64-unknown-linux-gnu" which conforms to

ARCHITECTURE-VENDOR-OPERATING_SYSTEM-ENVIRONMENT

SOME COMMON LANGUAGE CONSTRUCTS (with respect to 5 Programs)

5) @str= private unnamed_addr constant [21 x i8] c"Multiple of three %d\00", align 1: defines an anonymous variable to contain string constant (which will be passed in printf invocation). Here str is global varaible or alias

6) Arrays are represented as [n*i<s>] <type> where n is the no. Of element , <s> size of each element . <type> = base data type

7) define dso_local i32 @main() #0 : define a function called main whose return type is 32bit integer

8) #0 or #1: In LLVM IR each function uses set of attributes thus it may happen that multiple function refers to same attribute set . So this #<n>, is a reference to one such set. Attributes can be: noninline(no function def expansion), optnone(-O0) , nounwind(maybe it means that no exception handling).

9) entry: entry point for the function. Or entry Basic block.

10) %a.addr = alloca i32, align 4 : alloca-> LLVM IR assumes a stack memory model thus this instrn allocates the memory along with data type to be stored(i means integer) , 32(width of data type) , align 4(store this variable with base adress being multiple of 4)

11) store i32 %a, i32* %a.addr, align 4: store instrn is used to write to memory at a location pointed by 2nd operand of this instrn, with align

12) %0 = load i32, i32* %a.addr, align 4: ina temporary register value is being loaded from abstract stack . It is necessary cause llvm ir do operations over such temp registers.

13) %mul = mul nsw i32 %0, %1 or %add = add nsw i32 %mul, %div : mul or add means usual thing, nsw or nuw are flags that means “no signed wrap” or “no unsigned wrap” and if they are present then it accounts for poison value which means errorneous operation.

14) ret i32 %3: return directive ; type of value and a constant or register ref.

15) @func2(i32* %a, i32* %b): function prototype with pointer arguments(single pointers). Global identifiers began with @. local identifier began with ‘%’.

16) %call = call i32 @func1(i32 %1, i32 %3): a func whose return type is i32 and accepts two i32 values is called.

17) %conv = sitofp i32 %10 to double: implicit type conversion often needed in expressions. Direcive is i32 , with current type, rgister ref.,target type.

18) %mul = fmul double %conv, 5.660000e+00: this is imp. To understand . This exp involves not integer datatype . Notice that no width is alligned with double because in c X86_64 bit arch. Double is 64 byte and float is 32 bytes also the argument “5.66” is clearly defined .

19) rem = srem i32 %mul, 26: srem instrn returns modulus .

20) declare dso_local i32 @printf(i8*, ...) #1 : global declaration of printf.

21) %call1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), double %call): this one statement you will find in all the interactive c programs this is actually how a call to printf is resolved. First let us see getelementptr : it provides the register ref of the value inside an aggregate data structure . Inbounds: it srelation is with bounds of the refrenced structure and can return poison value . (i8*,..): it signifies that this function uses VarArgs and takes atleast one pointer argument to a char. 3 x i8], [3 x i8]* @.str, i64 0, i64 0: these ar arguments to work . First is the type of aggergate data structure it self, 2nd is the refrence to that structure, 3rd and 4th are two int constant they actually signifies the indexing offset into the structure. Since the [3*i8]* @.str is actually "%f\00" hence at the base location itself we get target value.

22) br label %for.cond: unconditional branch to secified label

23) for loops: it is interesting to see how a for loop is translated into llvm-IR. Since we know that a for loop has 3 parts 1) condition(for.cond:) 2) body(for.body:) 3) increement 4) what aboud initialization actually that is done seperately(in a basic block prior to above three basic blocks) 5) end (for.end) . Now these 5 parts are 5 seperate BB. Each doing some tsk and then branching to another one of these BB.

24) !llvm.loop !2 !2 = distinct !{!2, !3} !3 = !{!"llvm.loop.mustprogress"}	First line is loop meta data and further lines define loop property This "mustprogress" basically means that loop must do some work byinteractimg with enviroment else it will be removed.

24) ; preds = %for.body : provides list of predescessor to basic block

25) if.then: or if.else: label of blocks specific for if-else laddder.

26) %cmp = icmp sle i32 %0, 0 :integer comparison directive returns boolean; sle means <=

27) i did'nt found any difference between normal function call and recursive call

28) @__const.main.arr = private unnamed_addr constant [5 x i32] [i32 1, i32 2, i32 3, i32 4, i32 5], align 16 : this may look daunting but it simply means int arr[]={1,2,3,4,5} an array of 5 int constants ; const.main.arr: this might mean that arr is a constant Data structure inside main.(i maybe wrong here)

29) %0 = bitcast [5 x i32]* %arr to i8* : this instruction is often seen when bit to bit copy is needed and the value is to be casted . Here value is refrence to array arr(base address) , its type is[5*i32]* and it is to be converted to i8* which pointer to a byte

30)
```
%arrayidx = getelementptr inbounds [5 x i32], [5 x i32]* %arr, i64 0, i64 4
%1 = load i32, i32* %arrayidx, align 16
 store i32 %1, i32* %temp, align 4
```
Explain:
First line calculates refrence to 5th element in array [5*i32]
2nd line loads the value at 5th position or 4th index in register
3rd line stores this value in temp which is on stack
Basically it is temp=array[4]

31) !llvm.module.flags = !{!0}
!llvm.ident = !{!1}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.1 (https://github.com/llvm/llvm-project.git bla bla}

METADATA

3) Name Mangling : A technique to club the original user defined function identifier and its arguments along with some more info into a unique name . This is often used by c++ compilers or languages that support function overloading due to which one function name is used many times hence it is necessary to do name mangling for correct invocation Clang/Frontend do this and is reflected in generated llvm.ll file . As far as .c source files are concerned -> a simple '@' is appended forward with the identifier name

For Study of name mangling Below is a .cpp source which simply defines a function as " func1 " getting overloaded in 3 forms: func1(int a, int b)--> @_Z5func1ii ; func1(int a)-->@_Z5func1i; func1(double a, float b)-->@_Z5func1df;

Source File

```
// functio overloading
// to see the effects of name mangling(m
 generated name that has coded informatio
#include <iostream>
using namespace std;

int func1(int a, int b );
int func1(int a);
double func1(double a , float b);

int func1(int a, int b)
{
    return a+b;
}

int func1(int a)
{
return a*a*a;
}
```

.ll file generated by clang using -S – emit- llvm.

```
; Function Attrs: noinline nounwind optnone uwtable mustprogress
define dso_local i32 @_Z5func1ii(i32 %a, i32 %b) #4 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}

; Function Attrs: noinline nounwind optnone uwtable mustprogress
define dso_local i32 @_Z5func1i(i32 %a) #4 {
entry:
  %a.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %a.addr, align 4
  %mul = mul nsw i32 %0, %1
  %2 = load i32, i32* %a.addr, align 4
  %mul1 = mul nsw i32 %mul, %2
  ret i32 %mul1
}

; Function Attrs: noinline nounwind optnone uwtable mustprogress
define dso_local double @_Z5func1df(double %a, float %b) #4 {
entry:
  %a.addr = alloca double, align 8
  %b.addr = alloca float, align 4
  store double %a, double* %a.addr, align 8
  store float %b, float* %b.addr, align 4
  %0 = load double, double* %a.addr, align 8
  %1 = load float, float* %b.addr, align 4
```

4) Uses of Various Flags of Compiler(Clang),LLVM-OPT & LLVM TOOLS

For this section of study i have used a different set of programs since the earlier program were simple cause their aim was to make LLVM-IR understand. These programs are complex then above ones (dynamic memory allocate, phi nodes in IR , multiple loops , switch case etc.)

Clang Flags

1) -E : to generate preprocessed file from .c source ; all the # inlude or other preprocessor directives are resolved and replaced by respective header files + the all original function definitions

2) -fsyntaxonly: using this , clang will do preprocessing+ parsing+semantic analysis and generate AST for the program.

3) -S: this flag will cause above steps+ compilation and .bc file or bitcode generation.

4) -C : this flag will ensure complete compilation till assembly generation(machine dependent)

5) -emit -llvm: used to generate .ll file instead of .bc file since .ll is human readable.

6) -Xclang: very important . Used to pass xtra command line args to clang

7) - ast -view : generate .dot files for corresponding AST.

8) -disable O0 optnone: if you want to run various opt passes then optnone has to be disabled.

I also tried certain other flags using clang -cc1 –help like – emit – llvm – bc to get IR just after AST creation or – emit – llvm – only since the .ll file generated by -S – emit – llvm has effect of optimization stages also but using the above mentioned tags i am not able to generate output files.

LLVM – opt

1) **Analyses Passes**: --view -cfg , --view -dom , --view -callgraph. : these passes generated visual debug info.

2) **Transform Passes** : -mem2Reg, **-** dce, - sccp. : in this

a) **mem2Reg** : it promotes all the memory refrences in IR to register refrence. Also this pass is assumed by both dce and sccp. This pass is crucial. Even though the llvm-ir is in ssa form , it has lot of redundancy with respect to alloca, load, store instrn and there is no phi node (necessary for choosing correct value out of different value for same memory variable). Thus it creates something called as “pruned” ssa.

b) **dce** : **dead code elimination** : to remove unrechable or irrelevant instrns,

c) **sccp : sparse conditional constant propagation** : this pass works after mem2reg has happened cause this pass evaluates the subexpression invloving constants. This pass proves the values returned by expressions to be constants and replace them with the same. Thus causing dead sdefinitions lying around

d) **sccp followed dce**.

Now Below i have used two programs over several Transforms Passes which are mentioned above to evaluate effects on their intial IR.

Programs used are a) **10012021_1.c:**

Note: Some part of a module file i.e .ll file will remian unaffected.

```
; ModuleID = '/home/shikharj/CS5863_Program_Analysis_work/Assign_1/Programs_Observing_C_Constructs/10022021_1.c'
source_filename = "/home/shikharj/CS5863_Program_Analysis_work/Assign_1/Programs_Observing_C_Constructs/10022021_1.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-unknown-linux-gnu"

@.str = private unnamed_addr constant [21 x i8] c"a is %c and c is %d \00", align 1
@.str.1 = private unnamed_addr constant [21 x i8] c"a is not %c c is %d \00", align 1
@.str.2 = private unnamed_addr constant [5 x i8] c"oops\00", align 1

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

attributes #0 = { noinline nounwind uwtable "disable-tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { "disable-tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.1 (https://github.com/llvm/llvm-project.git fed41342a82f5a3a9201819a82bf7a48313e296b)"}
```

LLVM IR ▾ Tab Width: 8 ▾ Ln 6, Col 84 ▾ INS

here you should not find optnone

Also i noticed that basic blocks with there label remianed intact. Although there is no use of them. May be pass like dead block elimination will remove them.

1) effect by mem2Reg pass: All alloca ,store , load instrns got removed and the named identifiers which were actually getting defined by these instructions are now replaced by there constant values.

```
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %aa = alloca i32, align 4
  %bb = alloca i32, align 4
  %cc = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 10, i32* %aa, align 4
  store i32 20, i32* %bb, align 4
  %0 = load i32, i32* %aa, align 4
  %1 = load i32, i32* %bb, align 4
  %cmp = icmp sgt i32 %0, %1
  br i1 %cmp, label %if.then, label %if.else
```

```
define dso_local i32 @main() #0 {
entry:
  %cmp = icmp sgt i32 10, 20
  br i1 %cmp, label %if.then, label %if.else

```

Another major add-on done by this pass is inclusion of phi nodes. Now it is imporatnat to note that the original IR generated by -S -emit -llvm flags does has phi nodes . Actually that ir is like miiror image of our actual .c source file . But simpler instrns. For Ex:-

```
if.then:                                          ; preds = %entry
  %2 = load i32, i32* %aa, align 4
  %3 = load i32, i32* %bb, align 4
  %mul = mul nsw i32 %2, %3
  %add = add nsw i32 %mul, 100
  store i32 %add, i32* %cc, align 4
  br label %if.end

if.else:                                          ; preds = %entry
  %4 = load i32, i32* %aa, align 4
  %5 = load i32, i32* %bb, align 4
  %sub = sub nsw i32 %4, %5
  store i32 %sub, i32* %cc, align 4
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %6 = load i32, i32* %aa, align 4
  %add1 = add nsw i32 %6, 75
  %rem = srem i32 %add1, 26
  %conv = trunc i32 %rem to i8
  %conv2 = sext i8 %conv to i32
  store i32 %conv2, i32* %aa, align 4
  %7 = load i32, i32* %aa, align 4
  switch i32 %7, label %sw.default [
    i32 75, label %sw.bb
    i32 65, label %sw.bb3
  ]
```

In the above IR for if.end we can see that it has preds or predecessor if.then and if.else and from both of these blocks definition of ‘cc’ is flowing down to if.end . And since mem2Reg uses dominance frontier to calculate phi node so neither of if.then or if.else block is a dominator to if.end .hence a phi intstrn for ‘c’ is added . See below.

```
if.then:                                          ; preds = %entry
  %mul = mul nsw i32 10, 20
  %add = add nsw i32 %mul, 100
  br label %if.end

if.else:                                          ; preds = %entry
  %sub = sub nsw i32 10, 20
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %cc.0 = phi i32 [ %add, %if.then ], [ %sub, %if.else ]
  %add1 = add nsw i32 10, 75
  %rem = srem i32 %add1, 26
  %conv = trunc i32 %rem to i8
  %conv2 = sext i8 %conv to i32
  switch i32 %conv2, label %sw.default [
    i32 75, label %sw.bb
    i32 65, label %sw.bb3
  ]
```

Note: sext -> sign extension as here 8bit int is getting converted to 32bit int
trunc -> truncation of largr width.

2) effect of dce: No effect as no dead definition is found.

3) effect of sccp: this pass further optmized the code by evaluating any subexpression invoving constants. You can see how much the entry BB in main has reduced in size . Also effect on branch calculation in all if’s BB and in switch case BB can be seen.

```
; Function Attrs: nofree norecurse nosync nounwind uwtable
define dso_local i32 @main() #0 {
entry:
  br i1 false, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  br label %if.end

if.else:                                          ; preds = %entry
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %rem = srem i32 85, 26
  %conv = trunc i32 %rem to i8
  %0 = zext i8 %conv to i32
  switch i32 %0, label %sw.default [
    i32 75, label %sw.bb
    i32 65, label %sw.bb3
  ]

sw.bb:                                            ; preds = %if.end
  br label %sw.bb3

sw.bb3:                                           ; preds = %sw.bb, %if.end
  br label %sw.default

sw.default:                                       ; preds = %sw.bb3, %if.end
  %call5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str.2, i64 0, i64 0))
  br label %sw.epilog

sw.epilog:                                        ; preds = %sw.default
  ret i32 0
}
```

4) sccp->dce : no more optimized.

Some more new IR constructs :- switch -> switch case block, need a constant integer expression, label to default case block, maintains a key – value pair table where key= integer constant ; value= label of a BB corresponding to each case case inside switch block.

b) 10022021_4.c:

In the .ll of this file we see some more new IR construct : 1) global c-structures: using struct instrn which declares structure name and type of values it will store.

2) @malloc: just like printf, its definition is provided by linker .

3) @llvm.memcopy intrinsic : Intrinsic are functions known to compiler whch it applies in a well optimized manner. There need is to alleviate need of new instructions in LLVM-IR. Basically they are block of llvm ir code with predefined metadata and accepts arguments. Now coming unto this particular intrinsic, its work is to copy certain number of bytes of data from one location to other . It is intersting to note that the allocation of linear block of memory happens globally and a place other than function's scope ; when a array variable is declared while processing that function then this intrinsic is call to copy N no. of bytes. Why this happens, i dont understand.

1)Effect of mem2Reg: same as discussed fir first program, but the alloca intructions for struct type and array remain intact. Even the store instrn remian there for array.

2)Effect of dce: no change caused

3)Effect of sccp: Nothing extra to note;

4)sccp+dce: same as sccp pass.

LLVM-Tools

Note : certain tools are used in above activities but some are used only for there demonstration.

1)llvm-dis: very important tool cause it converts .bc file generated by opt passes back to human radable .ll files due to which i was able to compare the effects of passes.

2)lli: an interpreter which uses just in time compilation. Basically on giving a .bc file input it diretly executes program witout generating any intermediate file and dump output on terminal.

3) llvm-link: after you have final optimized .bc file . you need this tool to give .out file which is the executable. This tool links all the needed modules into one .

4) llvm -as: assembler that takes in .ll to generate .bc file

5) llc : backend llvm compiler . Taken in .ll or .bc to genrate a not so machine specific assembly file with .s extension.

Refrences:

1) For LLVM-IR: https://llvm.org/docs/LangRef.html
2) For LLVM Directory Structure :http://llvm.org/docs/GettingStarted.html#directory-layout
3) For Clang : https://clang.llvm.org/docs/ClangCommandLineReference.html and -cc1 –help on terminal

4) For LLVM Tools there –help options and their own websites

5) And last but should be first is the tutorial lecture taken by Both TA's that actually created a knowledge base for me and made this report possible.