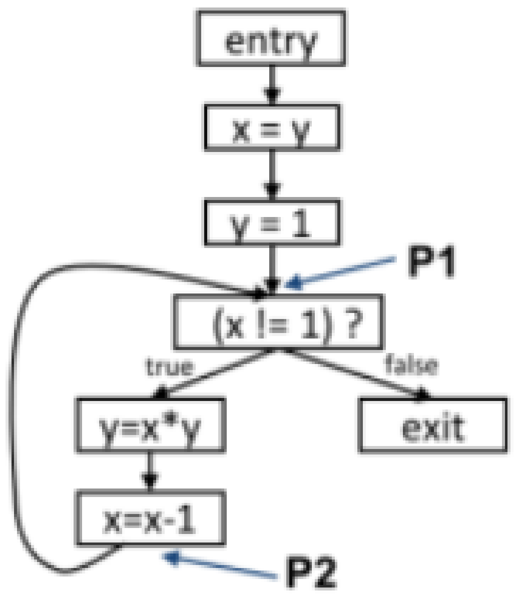
Mini Assignment #4

Cs19btech11020

Q1. Consider the given CFG and list down all the assignments that reach the program points P1 and P2.



Assignments reaching P1 are:
x=y
y=1
y=x*y
x=x-1
Assignment reaching P2 are:
y=x*y
x=x-1

Q2. Consider the following program:

A) Alias sets for function 'foo':
Alias Set Tracker: 3 alias sets for 4 pointer values
Alias sets for function 'bar':

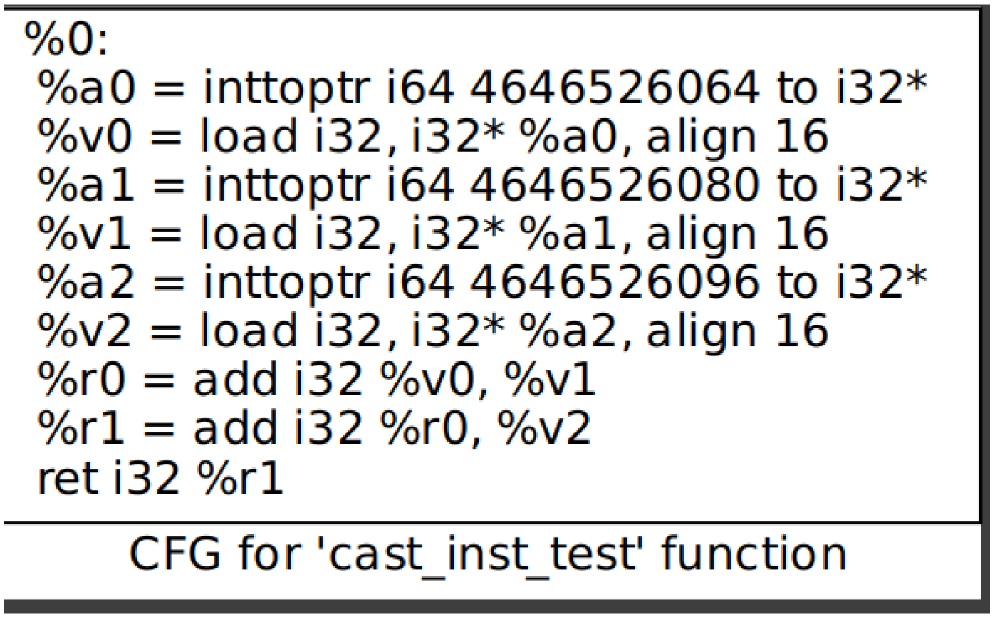
Alias Set Tracker: 4 alias sets for 8 pointer values
Alias sets for function 'main'':
Alias Set Tracker: 7 alias sets for 15 pointer values

B) The compiler needs to be so conservative so that it works correctly for the worst case scenario, to make sure IR is optimized and there are no leaks in the program, it will also be useful for the further analysis and evaluation of the program.
C) Yes pointers main.a0 and foo.a alias, as in main function we call foo and pass a0 as parameter which corresponds to a in foo , in foo we pass by reference so at that point they both must alias.

Q3)
A. scev-aa:
The -scev-aa pass implements AliasAnalysis queries by translating them into ScalarEvolution queries. This gives it a more complete understanding of getelementptr instructions and loop induction variables than other alias analyses have.
B. globals-aa:
This simple pass provides alias and mod/ref information for global values that do not have their address taken. if a global does not have its address taken, the pass knows that no pointers alias the global. This pass also keeps track of functions that it knows never access memory or never read memory.

D. tbaa:
Type-Based Alias Analysis. Alias analyses *based* on programming language types. The analysis uses type compatibility to determine aliases.

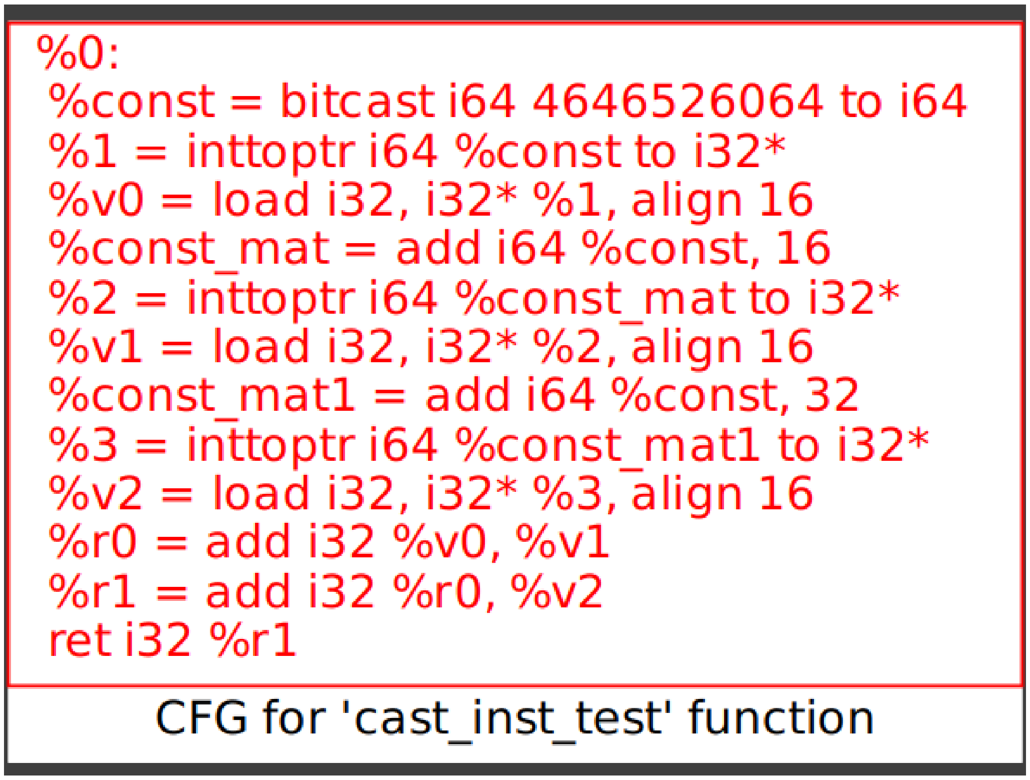
Q4)
1)--consthoist (Constant Hoisting):
Before optimization:



```
%0:
 %a0 = inttoptr i64 4646526064 to i32*
 %v0 = load i32, i32* %a0, align 16
 %a1 = inttoptr i64 4646526080 to i32*
 %v1 = load i32, i32* %a1, align 16
 %a2 = inttoptr i64 4646526096 to i32*
 %v2 = load i32, i32* %a2, align 16
 %r0 = add i32 %v0, %v1
 %r1 = add i32 %r0, %v2
 ret i32 %r1
```

CFG for 'cast_inst_test' function

After optimization pass:

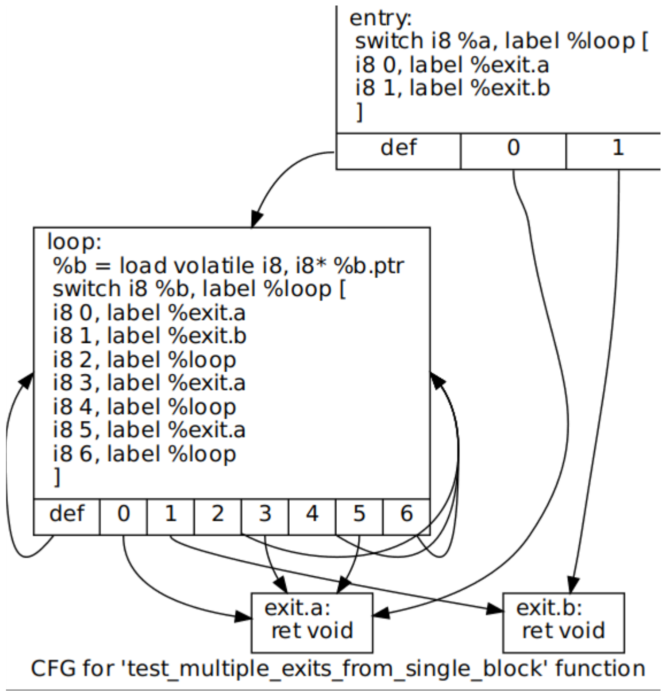


```
%0:
 %const = bitcast i64 4646526064 to i64
 %1 = inttoptr i64 %const to i32*
 %v0 = load i32, i32* %1, align 16
 %const_mat = add i64 %const, 16
 %2 = inttoptr i64 %const_mat to i32*
 %v1 = load i32, i32* %2, align 16
 %const_mat1 = add i64 %const, 32
 %3 = inttoptr i64 %const_mat1 to i32*
 %v2 = load i32, i32* %3, align 16
 %r0 = add i32 %v0, %v1
 %r1 = add i32 %r0, %v2
 ret i32 %r1
```

CFG for 'cast_inst_test' function

2) --loop-simplify: Canonicalize natural loops
The LoopSimplify pass will detect the loop and ensure separate headers for the outer and inner loop. This pass guarantees that loops will have exactly one backedge. This pass transforms natural loops into a simpler form, to make subsequent analyses and transformations simpler and effective.

Before optimization:



```
entry:
 switch i8 %a, label %loop [
 i8 0, label %exit.a
 i8 1, label %exit.b
 ]
```

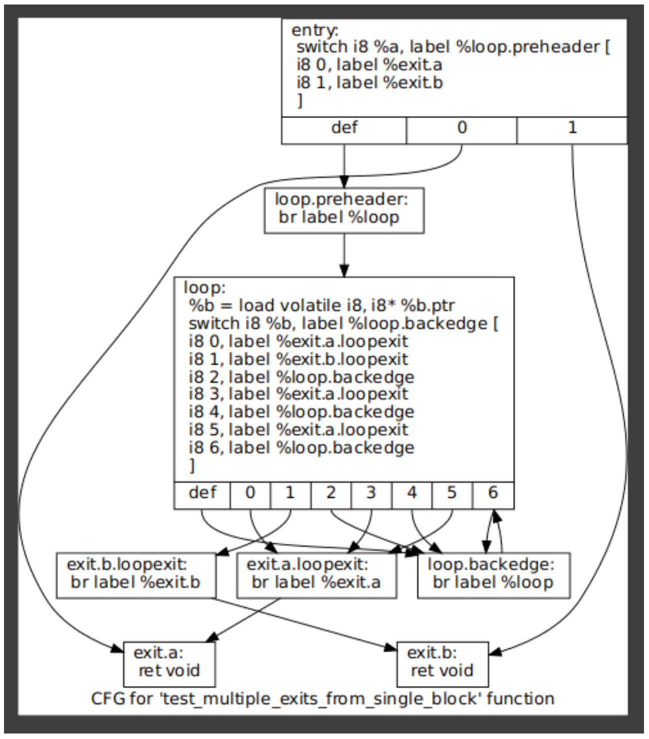
```
loop:
 %b = load volatile i8, i8* %b.ptr
 switch i8 %b, label %loop [
 i8 0, label %exit.a
 i8 1, label %exit.b
 i8 2, label %loop
 i8 3, label %exit.a
 i8 4, label %loop
 i8 5, label %exit.a
 i8 6, label %loop
 ]
```

```
exit.a:
 ret void
```

```
exit.b:
 ret void
```

CFG for 'test_multiple_exits_from_single_block' function

After optimization:



```
entry:
 switch i8 %a, label %loop.preheader [
 i8 0, label %exit.a
 i8 1, label %exit.b
 ]
```

```
loop.preheader:
 br label %loop
```

```
loop:
 %b = load volatile i8, i8* %b.ptr
 switch i8 %b, label %loop.backedge [
 i8 0, label %exit.a.loopexit
 i8 1, label %exit.b.loopexit
 i8 2, label %loop.backedge
 i8 3, label %exit.a.loopexit
 i8 4, label %loop.backedge
 i8 5, label %exit.a.loopexit
 i8 6, label %loop.backedge
 ]
```

```
exit.b.loopexit:
 br label %exit.b
```

```
exit.a.loopexit:
 br label %exit.a
```

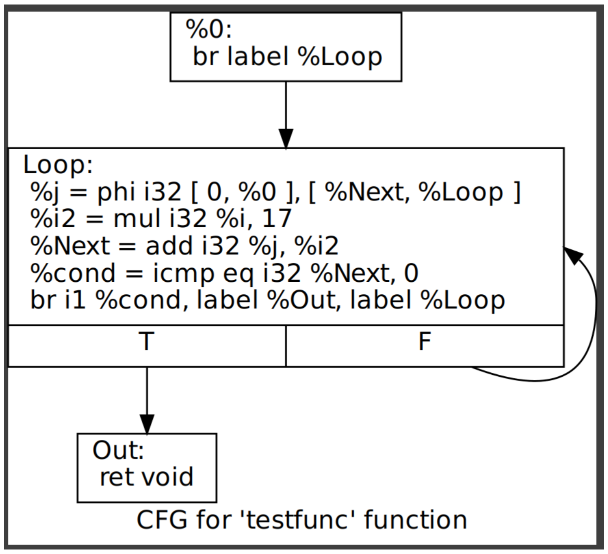
```
loop.backedge:
 br label %loop
```

```
exit.a:
 ret void
```

```
exit.b:
 ret void
```

CFG for 'test_multiple_exits_from_single_block' function

3) --licm (Loop Invariant Code Motion)
This pass performs loop invariant code motion, attempts to remove as much code from the body of a loop as possible. It either hoists code into the preheader block, or by sinks code to the exit blocks if it is safe. It also performs hoisting and sinking “invariant” loads and stores.
Before optimization:

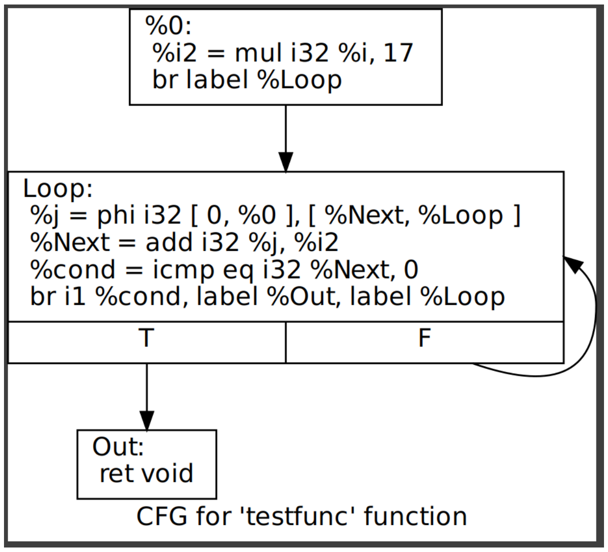


```
%0:
 br label %Loop

Loop:
 %j = phi i32 [ 0, %0 ], [ %Next, %Loop ]
 %i2 = mul i32 %i, 17
 %Next = add i32 %j, %i2
 %cond = icmp eq i32 %Next, 0
 br i1 %cond, label %Out, label %Loop

Out:
 ret void
```

After optimization:

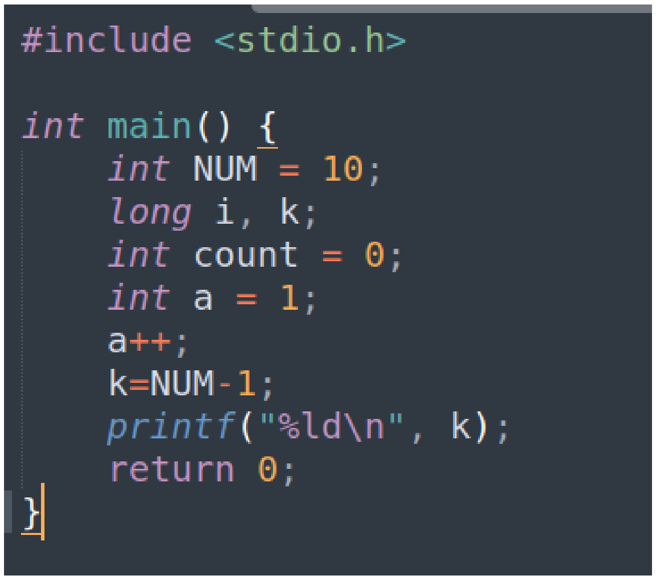


```
%0:
 %i2 = mul i32 %i, 17
 br label %Loop

Loop:
 %j = phi i32 [ 0, %0 ], [ %Next, %Loop ]
 %Next = add i32 %j, %i2
 %cond = icmp eq i32 %Next, 0
 br i1 %cond, label %Out, label %Loop

Out:
 ret void
```

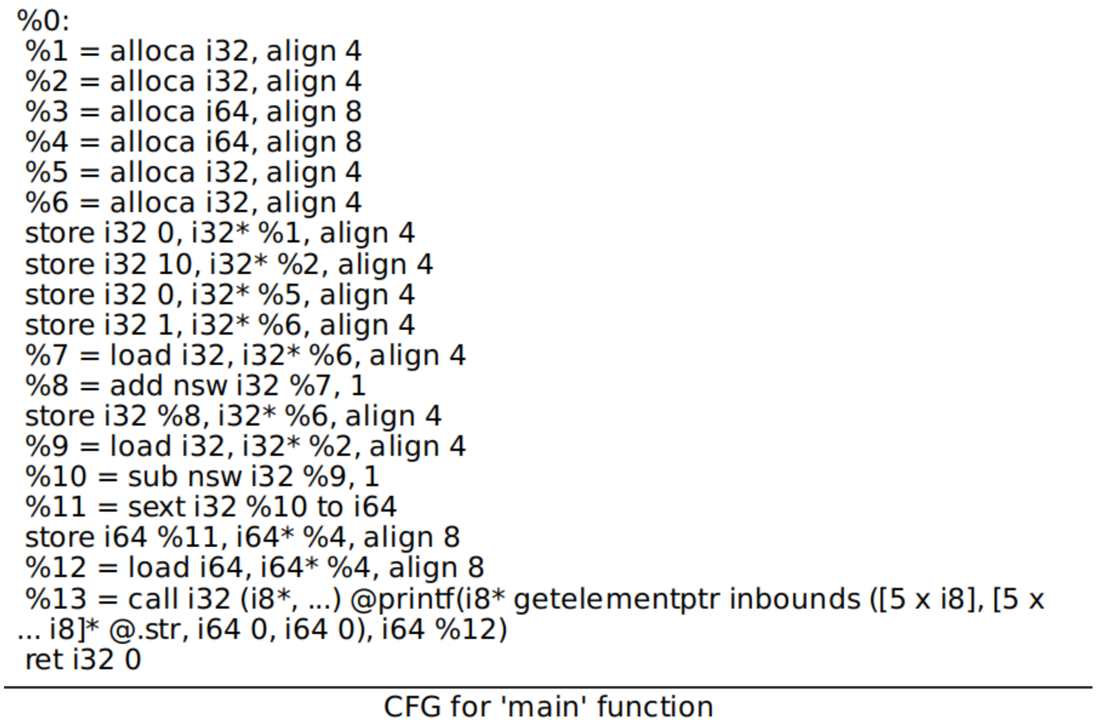
4) -dce: Dead Code Elimination
Dead code elimination performs passes over the function, removing instructions that are obviously dead and rechecks instructions that were used by removed instructions to see if they are newly dead.
Example1:
Code:



```c
#include <stdio.h>

int main() {
    int NUM = 10;
    long i, k;
    int count = 0;
    int a = 1;
    a++;
    k=NUM-1;
    printf("%ld\n", k);
    return 0;
}
```

CFG: before optimization

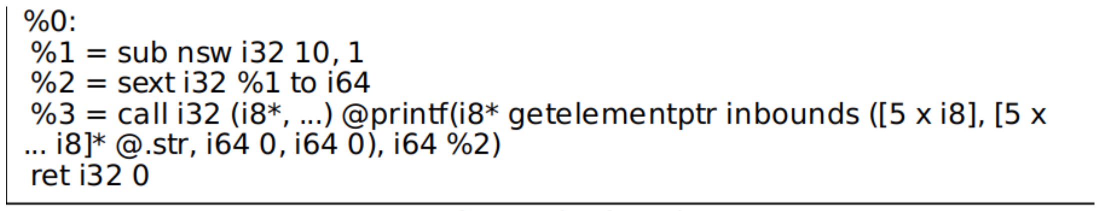


```
%0:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i64, align 8
 %4 = alloca i64, align 8
 %5 = alloca i32, align 4
 %6 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 10, i32* %2, align 4
 store i32 0, i32* %5, align 4
 store i32 1, i32* %6, align 4
 %7 = load i32, i32* %6, align 4
 %8 = add nsw i32 %7, 1
 store i32 %8, i32* %6, align 4
 %9 = load i32, i32* %2, align 4
 %10 = sub nsw i32 %9, 1
 %11 = sext i32 %10 to i64
 store i64 %11, i64* %4, align 8
 %12 = load i64, i64* %4, align 8
 %13 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x
... i8]* @.str, i64 0, i64 0), i64 %12)
 ret i32 0
```

CFG for 'main' function

CFG after optimization using DCE:

All the dead code is eliminated from the IR .



```
%0:
 %1 = sub nsw i32 10, 1
 %2 = sext i32 %1 to i64
 %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x
... i8]* @.str, i64 0, i64 0), i64 %2)
 ret i32 0
```

CFG for 'main' function

5)

c)
Polly-tiling: it enables loop tiling
Polly-parallel: Generates thread parallel code (isl codegen only)
Polly-pattern-matching-based-opts: does optimizations based on pattern matching.