CS3423: Compilers - II

Mini-Assignment #4

Soumi Chakraborty

ES19BTECH11017

Question 1

For answering this questions, we will consider two cases:

- y = 1 from the beginning
- y != 1 from the beginning

1.1 y = 1 initially

In this case, the condition x != 1 will evaluate to false and thus the control never reaches P2.

Thus, assignments at P1:

- x = y
- y = 1

And assignments at P2:

- None, since the control doesn't reach here

1.2 y != 1 initially

In this case, the condition x != 1 will evaluate to true and thus both P1 and P2 will have at least 4 assignments each.

Thus, assignments at P1:

- x = y
- y = 1

And,

- y = x * y
- x = x - 1

until x becomes 1.

And assignments at P2:

- x = y
- y = 1

And,

- y = x * y
- x = x - 1

until x becomes 1.

Question 2

Command to generate the AST:

```
clang -Xclang -ast-dump -fsyntax-only filename.c
```

2.A Number of alias sets

- foo(): 3 alias sets for 4 pointer values
- bar(): 4 alias sets for 8 pointer values
- main(): 7 alias sets for 15 pointer values

2.B

This has got to do with strict aliasing.

2.C

Answer: No.

Question 3

3.A scev-aa

- scev comes from ScalarEvolution which refers to the change in the value of a variable over the execution of a loop. In this kind of alias analysis, expressions are converted to scalar evolution expressions.

3.B globals-aa

- This pass takes care of all the global variables in a program.

3.C external-aa

- This pass takes care of all the external variables in a program.

3.D tbaa

- This pass stands for type-based alias analysis and it provides aliasing information to the optimiser.

Question 4

Syntax to compare the CFG's of a program before and after a consthoist pass:

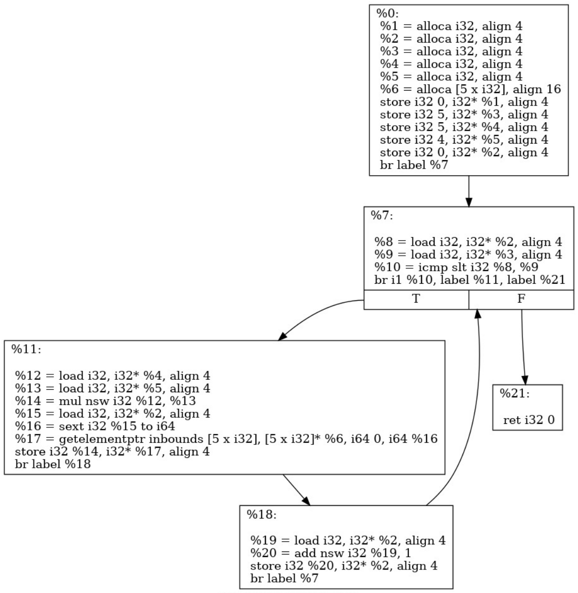
- `clang -emit-llvm -S filename.c -o filename.ll`
- `opt --dot-cfg filename.ll`
- `dot -Tpng cfg.func.dot -o filename.png`
- `opt --dce filename.ll>filename_pass.ll`
- `opt --dot-cfg filename_pass.ll`
- `dot -Tpng cfg.func.dot -o filename_pass.png`

4.1 --consthoist

This pass stands for constant hoisting and hoists any constant expressions from within a loop to outside it to prevent recomputation. Thus, constant hoisting optimises the code to compute constant expressions only once before the loop and then reuses the value in every iteration.

4.1.1 Output

Before the pass:



```
%0:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
 %5 = alloca i32, align 4
 %6 = alloca [5 x i32], align 16
 store i32 0, i32* %1, align 4
 store i32 5, i32* %3, align 4
 store i32 5, i32* %4, align 4
 store i32 4, i32* %5, align 4
 store i32 0, i32* %2, align 4
 br label %7
```

```
%7:

 %8 = load i32, i32* %2, align 4
 %9 = load i32, i32* %3, align 4
 %10 = icmp slt i32 %8, %9
 br i1 %10, label %11, label %21
```

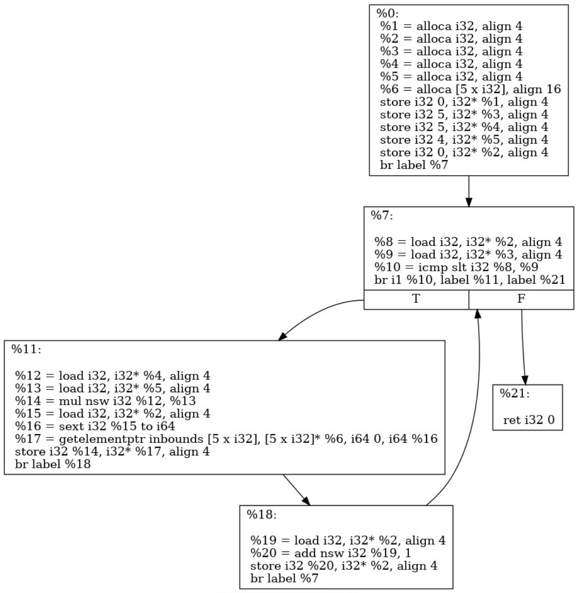
```
%11:

 %12 = load i32, i32* %4, align 4
 %13 = load i32, i32* %5, align 4
 %14 = mul nsw i32 %12, %13
 %15 = load i32, i32* %2, align 4
 %16 = sext i32 %15 to i64
 %17 = getelementptr inbounds [5 x i32], [5 x i32]* %6, i64 0, i64 %16
 store i32 %14, i32* %17, align 4
 br label %18
```

```
%21:

 ret i32 0
```

```
%18:

 %19 = load i32, i32* %2, align 4
 %20 = add nsw i32 %19, 1
 store i32 %20, i32* %2, align 4
 br label %7
```

CFG for 'main' function

After the pass:



```
%0:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
 %5 = alloca i32, align 4
 %6 = alloca [5 x i32], align 16
 store i32 0, i32* %1, align 4
 store i32 5, i32* %3, align 4
 store i32 5, i32* %4, align 4
 store i32 4, i32* %5, align 4
 store i32 0, i32* %2, align 4
 br label %7
```

```
%7:

 %8 = load i32, i32* %2, align 4
 %9 = load i32, i32* %3, align 4
 %10 = icmp slt i32 %8, %9
 br i1 %10, label %11, label %21
```

```
%11:

 %12 = load i32, i32* %4, align 4
 %13 = load i32, i32* %5, align 4
 %14 = mul nsw i32 %12, %13
 %15 = load i32, i32* %2, align 4
 %16 = sext i32 %15 to i64
 %17 = getelementptr inbounds [5 x i32], [5 x i32]* %6, i64 0, i64 %16
 store i32 %14, i32* %17, align 4
 br label %18
```

```
%21:

 ret i32 0
```

```
%18:

 %19 = load i32, i32* %2, align 4
 %20 = add nsw i32 %19, 1
 store i32 %20, i32* %2, align 4
 br label %7
```

CFG for 'main' function

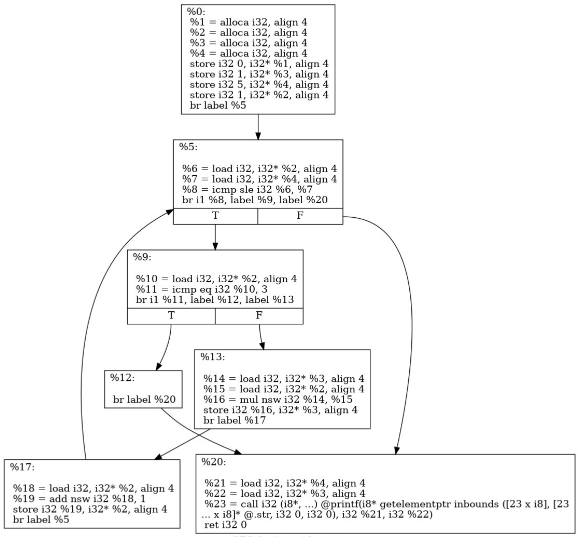
Surprisingly, there are no differences between the CFGs before and after the pass despite the presence of lines that can be hoisted outside the loop.

4.2 --loop-simplify

This pass simplifies loops to whatever extent possible. To compare CFG's before and after this pass, a code file which included a for loop with a break statement in it was used.

4.2.1 Output

Before the pass:



```
%0:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 1, i32* %3, align 4
 store i32 5, i32* %4, align 4
 store i32 1, i32* %2, align 4
 br label %5
```

```
%5:

 %6 = load i32, i32* %2, align 4
 %7 = load i32, i32* %4, align 4
 %8 = icmp sle i32 %6, %7
 br i1 %8, label %9, label %20
```

```
%9:

 %10 = load i32, i32* %2, align 4
 %11 = icmp eq i32 %10, 3
 br i1 %11, label %12, label %13
```

```
%12:

 br label %20
```

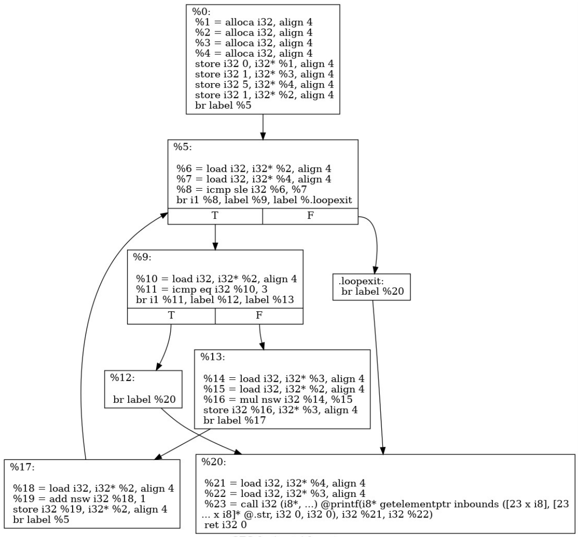
```
%13:

 %14 = load i32, i32* %3, align 4
 %15 = load i32, i32* %2, align 4
 %16 = mul nsw i32 %14, %15
 store i32 %16, i32* %3, align 4
 br label %17
```

```
%17:

 %18 = load i32, i32* %2, align 4
 %19 = add nsw i32 %18, 1
 store i32 %19, i32* %2, align 4
 br label %5
```

```
%20:

 %21 = load i32, i32* %4, align 4
 %22 = load i32, i32* %3, align 4
 %23 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([23 x i8], [23
... x i8]* @.str, i32 0, i32 0), i32 %21, i32 %22)
 ret i32 0
```

CFG for 'main' function

After the pass:



```
%0:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 1, i32* %3, align 4
 store i32 5, i32* %4, align 4
 store i32 1, i32* %2, align 4
 br label %5
```

```
%5:

 %6 = load i32, i32* %2, align 4
 %7 = load i32, i32* %4, align 4
 %8 = icmp sle i32 %6, %7
 br i1 %8, label %9, label %.loopexit
```

```
%9:

 %10 = load i32, i32* %2, align 4
 %11 = icmp eq i32 %10, 3
 br i1 %11, label %12, label %13
```

```
.loopexit:
 br label %20
```

```
%12:

 br label %20
```

```
%13:

 %14 = load i32, i32* %3, align 4
 %15 = load i32, i32* %2, align 4
 %16 = mul nsw i32 %14, %15
 store i32 %16, i32* %3, align 4
 br label %17
```

```
%17:

 %18 = load i32, i32* %2, align 4
 %19 = add nsw i32 %18, 1
 store i32 %19, i32* %2, align 4
 br label %5
```

```
%20:

 %21 = load i32, i32* %4, align 4
 %22 = load i32, i32* %3, align 4
 %23 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([23 x i8], [23
... x i8]* @.str, i32 0, i32 0), i32 %21, i32 %22)
 ret i32 0
```

CFG for 'main' function

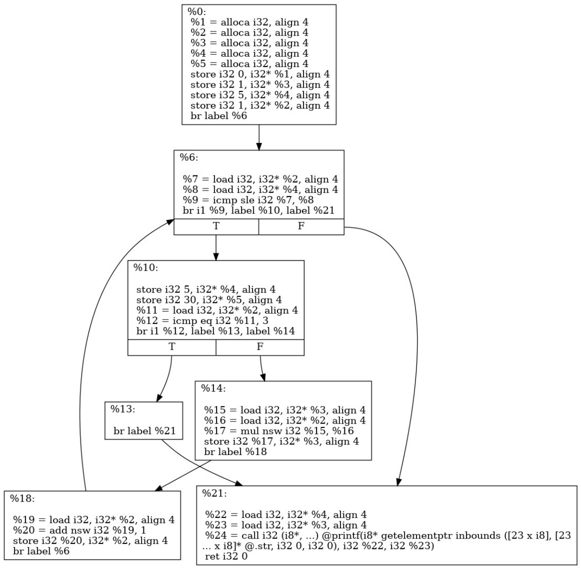
As we can see, there's a difference between the two CFG's which is due to the break statement inside the loop.

4.3 --licm

This pass stands for loop invariant code motion. This pass tries to clean up the code inside a loop as much as possible by doing constant hoisting or removing redundant expressions. To compare CFG's before and after this pass, a code file which included a for loop with a break statement in it was used.

4.3.1 Output

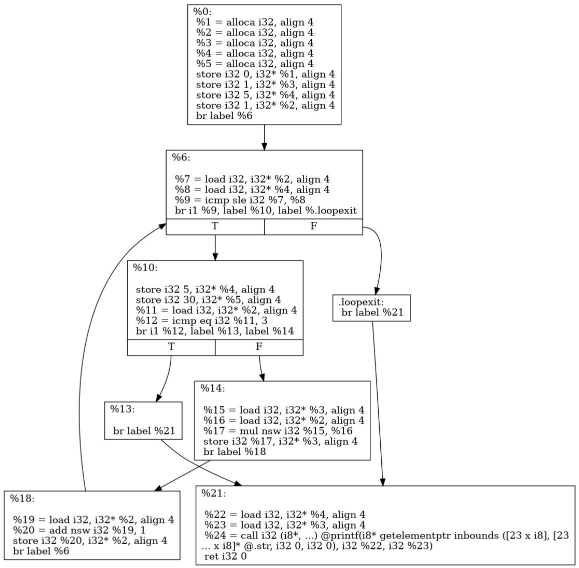
Before the pass:



```
%0:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
 %5 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 1, i32* %3, align 4
 store i32 5, i32* %4, align 4
 store i32 1, i32* %2, align 4
 br label %6

%6:

 %7 = load i32, i32* %2, align 4
 %8 = load i32, i32* %4, align 4
 %9 = icmp sle i32 %7, %8
 br i1 %9, label %10, label %21

%10:

 store i32 5, i32* %4, align 4
 store i32 30, i32* %5, align 4
 %11 = load i32, i32* %2, align 4
 %12 = icmp eq i32 %11, 3
 br i1 %12, label %13, label %14

%13:

 br label %21

%14:

 %15 = load i32, i32* %3, align 4
 %16 = load i32, i32* %2, align 4
 %17 = mul nsw i32 %15, %16
 store i32 %17, i32* %3, align 4
 br label %18

%18:

 %19 = load i32, i32* %2, align 4
 %20 = add nsw i32 %19, 1
 store i32 %20, i32* %2, align 4
 br label %6

%21:

 %22 = load i32, i32* %4, align 4
 %23 = load i32, i32* %3, align 4
 %24 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([23 x i8], [23
... x i8]* @.str, i32 0, i32 0), i32 %22, i32 %23)
 ret i32 0
```

CFG for 'main' function

After the pass:



```
%0:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
 %5 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 1, i32* %3, align 4
 store i32 5, i32* %4, align 4
 store i32 1, i32* %2, align 4
 br label %6

%6:

 %7 = load i32, i32* %2, align 4
 %8 = load i32, i32* %4, align 4
 %9 = icmp sle i32 %7, %8
 br i1 %9, label %10, label %.loopexit

%10:

 store i32 5, i32* %4, align 4
 store i32 30, i32* %5, align 4
 %11 = load i32, i32* %2, align 4
 %12 = icmp eq i32 %11, 3
 br i1 %12, label %13, label %14

.loopexit:
 br label %21

%13:

 br label %21

%14:

 %15 = load i32, i32* %3, align 4
 %16 = load i32, i32* %2, align 4
 %17 = mul nsw i32 %15, %16
 store i32 %17, i32* %3, align 4
 br label %18

%18:

 %19 = load i32, i32* %2, align 4
 %20 = add nsw i32 %19, 1
 store i32 %20, i32* %2, align 4
 br label %6

%21:

 %22 = load i32, i32* %4, align 4
 %23 = load i32, i32* %3, align 4
 %24 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([23 x i8], [23
... x i8]* @.str, i32 0, i32 0), i32 %22, i32 %23)
 ret i32 0
```

CFG for 'main' function

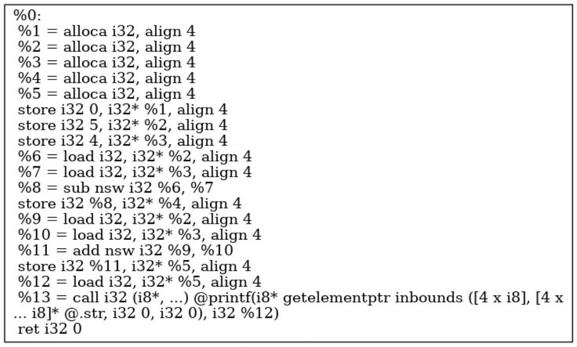
As we can see, there's a difference between the two CFG's which is due to the break statement inside the loop and the redundant lines.

4.4 --dce

This pass stands for dead code elimination. It is expected to remove any redundant or un-important lines of code to optimise execution. Un-important lines are those whose existence or lack thereof has no effect on the final output of the program.

4.4.1 Output

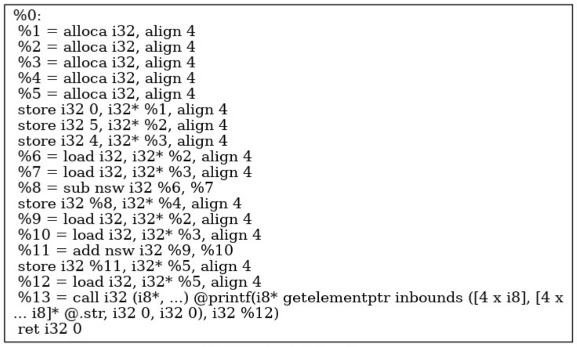
Before the pass:



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

CFG for 'main' function

After the pass:



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

CFG for 'main' function

Surprisingly, there are no differences between the CFGs before and after the pass despite the presence of redundant lines (unused variables, multiple return statements).

Question 5

SCoP stands for Static Control Part, which is like a Control Flow Graph but it only has static control flow and it has only one entry and exit option.

I tried the tiling option on a simple fibonacci program. Please not that I tried other options like parallelisation on other files as well but this combination was the only one to return a different .ll file, and hence this submission.

Syntax:

- `clang -emit-llvm -S filename.c -o filename.ll`
- `clang -emit-llvm -S -O3 -mllvm -polly-tiling  filename.c -o filename_tiling.ll`