Marras, Colli, Lazzarini

LINGUAGGI E COMPILATORI - ASSIGNMENT 1

Esercizio 1) -> Loop.c

```c
int g;

int g_incr(int c) {
  g += c;
  return g;
}
```

SORGENTE

```
; Function Attrs: mustprogress nofree norecurse nosync nounwind uwtable willreturn
define dso_local i32 @g_incr(i32 noundef %0) local_unnamed_addr #0 {
  %2 = load i32, i32* @g, align 4, !tbaa !5
  %3 = add nsw i32 %2, %0
  store i32 %3, i32* @g, align 4, !tbaa !5
  ret i32 %3
}
```

LLVM IR

```c
int loop(int a, int b, int c) {
  int i, ret = 0;

  for (i = a; i < b; i++) {
    g_incr(c);
  }

  return ret + g;
}
```

SORGENTE

```
; Function Attrs: nofree norecurse nosync nounwind uwtable
define dso_local i32 @loop(i32 noundef %0, i32 noundef %1,
                           i32 noundef %2) local_unnamed_addr #1 {
  %4 = load i32, i32* @g, align 4, !tbaa !5
  %5 = icmp sgt i32 %1, %0
  br i1 %5, label %6, label %10          BB1

T
6:                                       ; preds = %3
  %7 = sub i32 %1, %0
  %8 = mul i32 %7, %2
  %9 = add i32 %4, %8
  store i32 %9, i32* @g, align 4, !tbaa !5
  br label %10                           BB2

F
FT
10:                                      ; preds = %6, %3
  %11 = phi i32 [ %9, %6 ], [ %4, %3 ]
  ret i32 %11                            BB3
}
```

LLVM IR

Cos'è successo al loop?

Il loop viene esplicitamente rimosso dalla versione ottimizzata poiché viene calcolato a compile time il numero di volte che sarebbe stato eseguito, e quindi sostituito da una moltiplicazione tra %7 e %2.

Provate a rigenerare la IR con l'opzione -O0 al comando di invocazione di clang.

```c
int g;

int g_incr(int c) {
  g += c;
  return g;
}
```

SORGENTE

```
@g = dso_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @g_incr(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = load i32, i32* @g, align 4
  %5 = add nsw i32 %4, %3
  store i32 %5, i32* @g, align 4
  %6 = load i32, i32* @g, align 4
  ret i32 %6
}
```

BB1

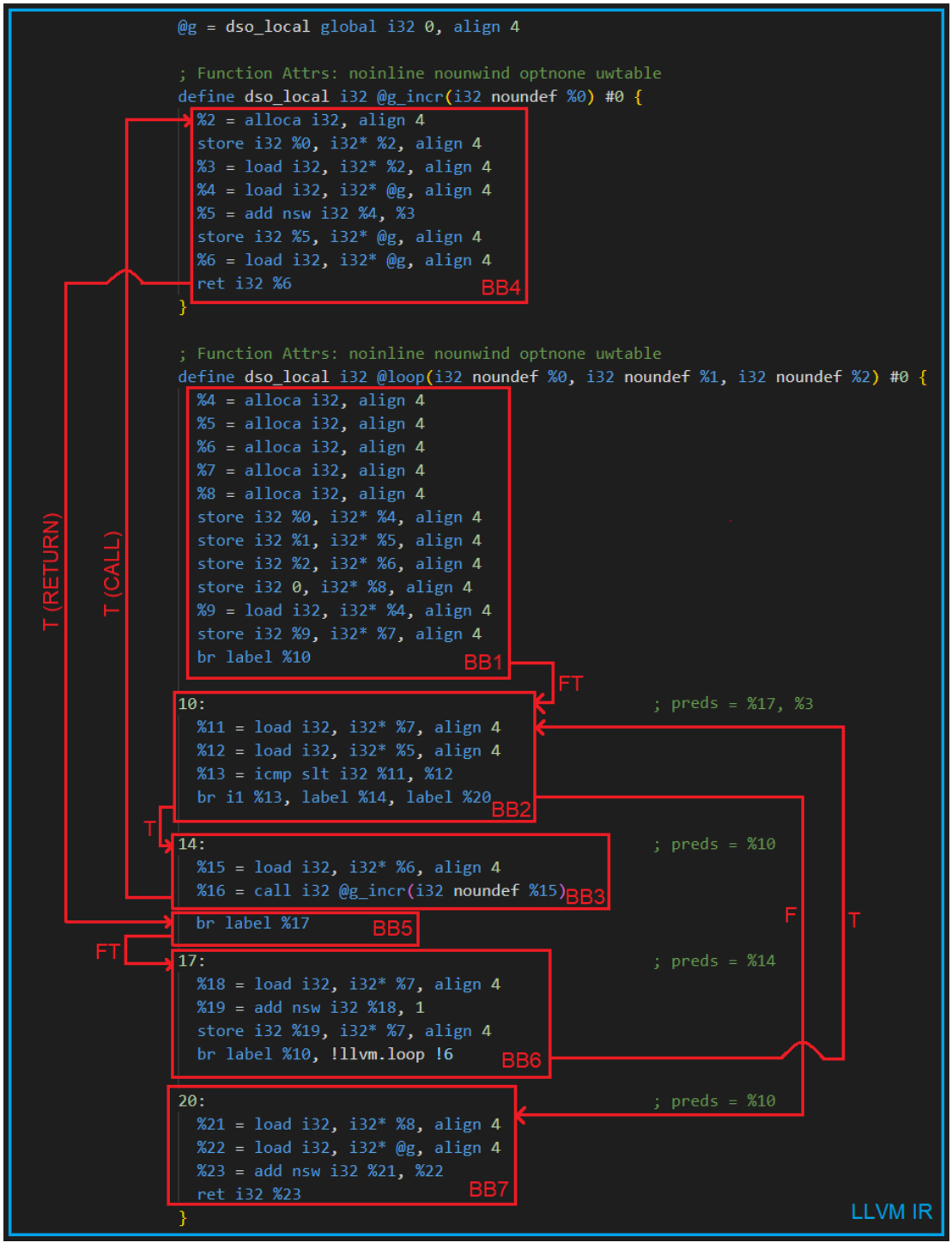
LLVM IR

```c
int loop(int a, int b, int c) {
  int i, ret = 0;

  for (i = a; i < b; i++) {
    g_incr(c);
  }

  return ret + g;
}
```

SORGENTE



Cosa cambia nell'intermedio?

Nel codice intermedio della versione ottenuta con l'opzione -O0 sono state disabilitate tutte le ottimizzazioni, tra cui:

- *Inlining:* precedentemente utilizzato per la funzione g_incr();
- *Dead Code Elimination:* precedentemente utilizzato per la variabile "ret".

Di conseguenza, il codice non ottimizzato risulta più prolisso, dal momento che utilizza un maggior numero di registri e di istruzioni.

Provate ad aggiungere il flag *-Rpass=.** al comando di invocazione di clang (con –O2)

Si ottiene la lista delle ottimizzazioni che sono state effettuate.

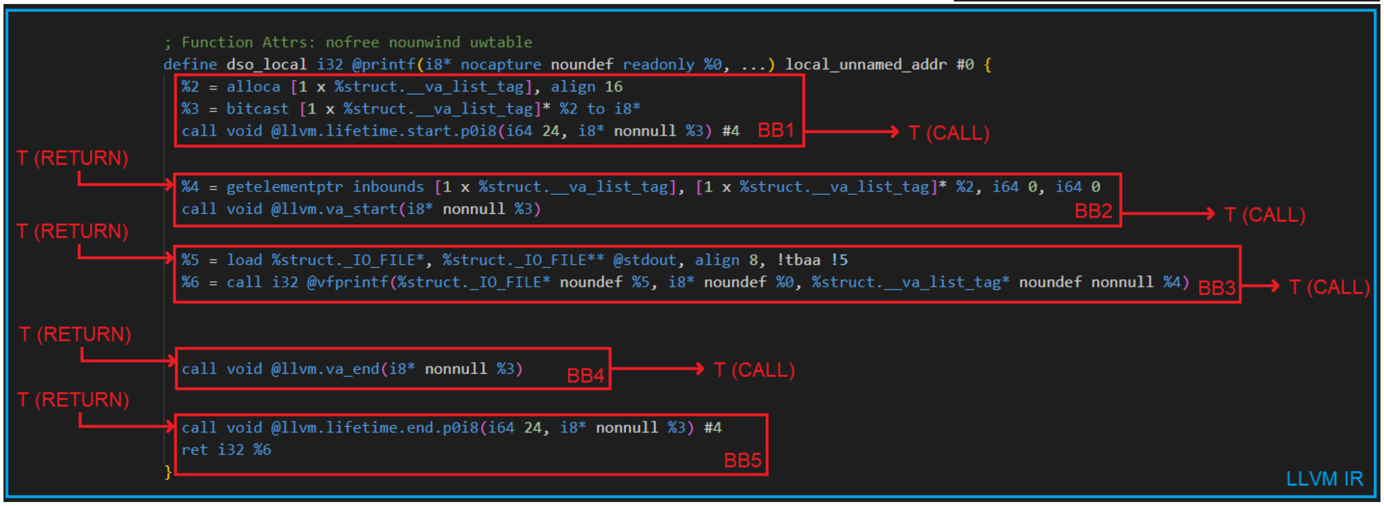
```
marrase@DESKTOP-P63EG9I:/mnt/c/C++/Linguaggi-e-Compilatori-2022-2023/Tutorial-01/TestPass$ clang -O2 -emit-llvm -S -c test/L
test/Loop.c:34:5: remark: 'g_incr' inlined into 'loop' with (cost=-20, threshold=337) at callsite loop:4:5; [-Rpass=inline]
    g_incr(c);
    ^
test/Loop.c:21:5: remark: Moving accesses to memory location out of the loop [-Rpass=licm]
  g += c;
    ^
test/Loop.c:21:5: remark: Moving accesses to memory location out of the loop [-Rpass=licm]
test/Loop.c:33:3: remark: Loop deleted because it is invariant [-Rpass=loop-delete]
  for (i = a; i < b; i++) {
  ^
test/Loop.c:37:16: remark: load of type i32 eliminated [-Rpass=gvn]
  return ret + g;
               ^
```

Esercizio 1) -> Fibonacci.c

```c
int printf(const char *format, ...) {
  int ret;
  va_list args;
  va_start(args, format);
  ret = vfprintf(stdout, format, args);
  va_end(args);

  return ret;
}
```

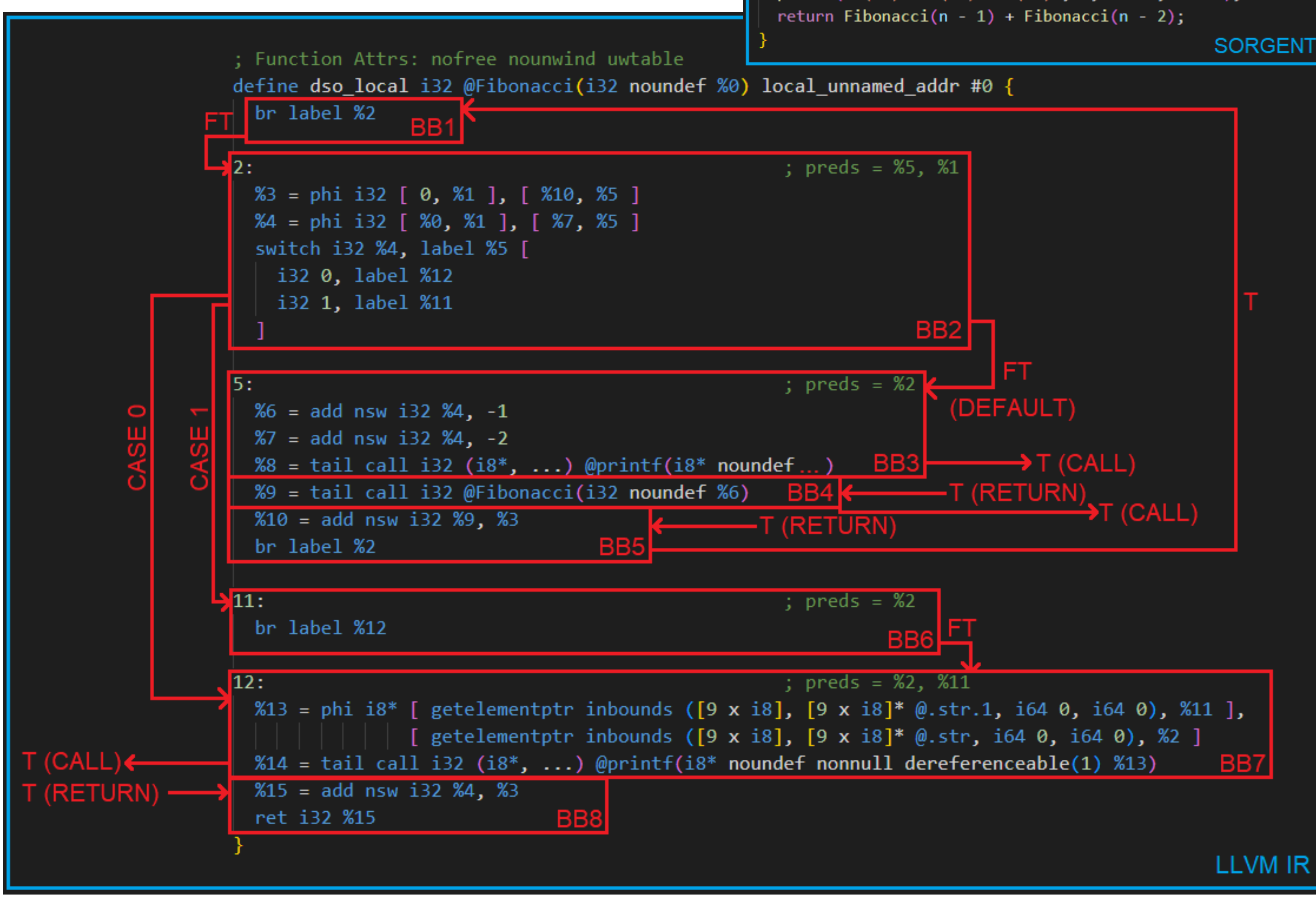
SORGENTE



; Function Attrs: nofree nounwind uwtable
define dso_local i32 @printf(i8* nocapture noundef readonly %0, ...) local_unnamed_addr #0 {
%2 = alloca [1 x %struct.__va_list_tag], align 16
%3 = bitcast [1 x %struct.__va_list_tag]* %2 to i8*
call void @llvm.lifetime.start.p0i8(i64 24, i8* nonnull %3) #4
BB1
T (CALL)
T (RETURN)
%4 = getelementptr inbounds [1 x %struct.__va_list_tag], [1 x %struct.__va_list_tag]* %2, i64 0, i64 0
call void @llvm.va_start(i8* nonnull %3)
BB2
T (CALL)
T (RETURN)
%5 = load %struct._IO_FILE*, %struct._IO_FILE** @stdout, align 8, !tbaa !5
%6 = call i32 @vfprintf(%struct._IO_FILE* noundef %5, i8* noundef %0, %struct.__va_list_tag* noundef nonnull %4)
BB3
T (CALL)
T (RETURN)
call void @llvm.va_end(i8* nonnull %3)
BB4
T (CALL)
T (RETURN)
call void @llvm.lifetime.end.p0i8(i64 24, i8* nonnull %3) #4
ret i32 %6
BB5
}
LLVM IR


```c
int Fibonacci(const int n) {
  if (n == 0) {
    printf("f(0) = 0");
    return 0;
  }
  if (n == 1) {
    printf("f(1) = 1");
    return 1;
  }
  printf("f(%d) = f(%d) + f(%d)", n, n - 1, n - 2);
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

SORGENTE



; Function Attrs: nofree nounwind uwtable
define dso_local i32 @Fibonacci(i32 noundef %0) local_unnamed_addr #0 {
FT
br label %2
BB1
2: ; preds = %5, %1
%3 = phi i32 [0, %1], [%10, %5]
%4 = phi i32 [%0, %1], [%7, %5]
switch i32 %4, label %5 [
i32 0, label %12
i32 1, label %11
]
BB2
T
FT
(DEFAULT)
5: ; preds = %2
%6 = add nsw i32 %4, -1
%7 = add nsw i32 %4, -2
%8 = tail call i32 (i8*, ...) @printf(i8* noundef ...)
BB3
T (CALL)
%9 = tail call i32 @Fibonacci(i32 noundef %6)
BB4
T (RETURN)
T (CALL)
%10 = add nsw i32 %9, %3
br label %2
BB5
T (RETURN)
CASE 0
CASE 1
11: ; preds = %2
br label %12
BB6
FT
12: ; preds = %2, %11
%13 = phi i8* [getelementptr inbounds ([9 x i8], [9 x i8]* @.str.1, i64 0, i64 0), %11],
[getelementptr inbounds ([9 x i8], [9 x i8]* @.str, i64 0, i64 0), %2]
%14 = tail call i32 (i8*, ...) @printf(i8* noundef nonnull dereferenceable(1) %13)
BB7
T (CALL)
T (RETURN)
%15 = add nsw i32 %4, %3
ret i32 %15
BB8
}
LLVM IR

Provate a rigenerare la IR con l'opzione -O0 al comando di invocazione di clang.

```c
int printf(const char *format, ...) {
  int ret;
  va_list args;
  va_start(args, format);
  ret = vfprintf(stdout, format, args);
  va_end(args);

  return ret;
}
```

SORGENTE

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @printf(i8* noundef %0, ...) #0 {
  %2 = alloca i8*, align 8
  %3 = alloca i32, align 4
  %4 = alloca [1 x %struct.__va_list_tag], align 16
  store i8* %0, i8** %2, align 8
  %5 = getelementptr inbounds [1 x %struct.__va_list_tag], [1 x %struct.__va_list_tag]* %4, i64 0, i64 0
  %6 = bitcast %struct.__va_list_tag* %5 to i8*
  call void @llvm.va_start(i8* %6)
```

BB1

T (CALL)

T (RETURN)

```
  %7 = load %struct._IO_FILE*, %struct._IO_FILE** @stdout, align 8
  %8 = load i8*, i8** %2, align 8
  %9 = getelementptr inbounds [1 x %struct.__va_list_tag], [1 x %struct.__va_list_tag]* %4, i64 0, i64 0
  %10 = call i32 @vfprintf(%struct._IO_FILE* noundef %7, i8* noundef %8, %struct.__va_list_tag* noundef %9)
```

BB2

T (CALL)

T (RETURN)

```
  store i32 %10, i32* %3, align 4
  %11 = getelementptr inbounds [1 x %struct.__va_list_tag], [1 x %struct.__va_list_tag]* %4, i64 0, i64 0
  %12 = bitcast %struct.__va_list_tag* %11 to i8*
  call void @llvm.va_end(i8* %12)
```

BB3

T (CALL)

T (RETURN)

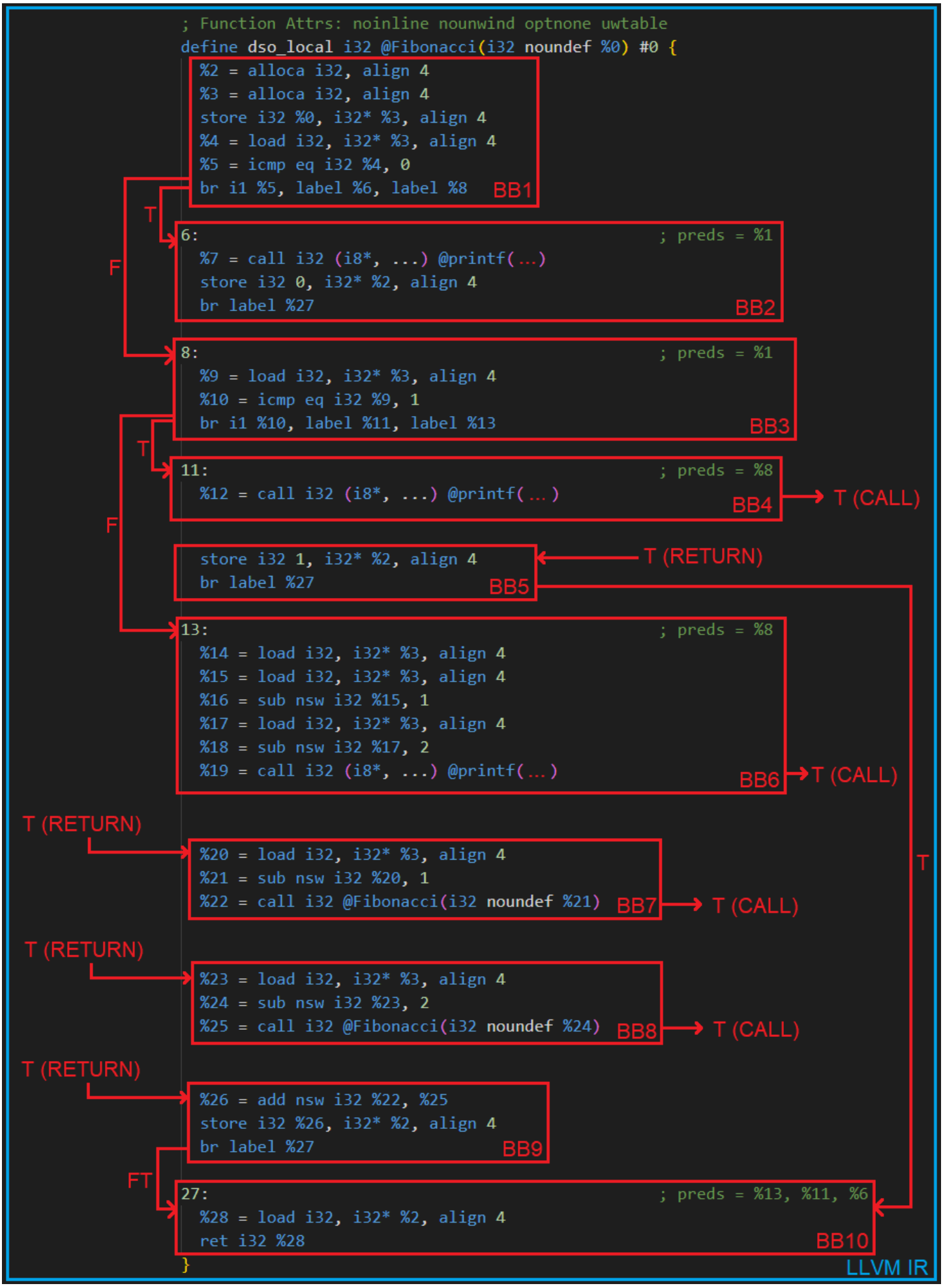
```
  %13 = load i32, i32* %3, align 4
  ret i32 %13
}
```

BB4

LLVM IR

```c
int Fibonacci(const int n) {
  if (n == 0) {
    printf("f(0) = 0");
    return 0;
  }
  if (n == 1) {
    printf("f(1) = 1");
    return 1;
  }
  printf("f(%d) = f(%d) + f(%d)", n, n - 1, n - 2);
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

SORGENTE


; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @Fibonacci(i32 noundef %0) #0 {
%2 = alloca i32, align 4
%3 = alloca i32, align 4
store i32 %0, i32* %3, align 4
%4 = load i32, i32* %3, align 4
%5 = icmp eq i32 %4, 0
br i1 %5, label %6, label %8
BB1
T
F
6: ; preds = %1
%7 = call i32 (i8*, ...) @printf(...)
store i32 0, i32* %2, align 4
br label %27
BB2
8: ; preds = %1
%9 = load i32, i32* %3, align 4
%10 = icmp eq i32 %9, 1
br i1 %10, label %11, label %13
BB3
T
F
11: ; preds = %8
%12 = call i32 (i8*, ...) @printf(...)
BB4
T (CALL)
store i32 1, i32* %2, align 4
br label %27
BB5
T (RETURN)
13: ; preds = %8
%14 = load i32, i32* %3, align 4
%15 = load i32, i32* %3, align 4
%16 = sub nsw i32 %15, 1
%17 = load i32, i32* %3, align 4
%18 = sub nsw i32 %17, 2
%19 = call i32 (i8*, ...) @printf(...)
BB6
T (CALL)
T (RETURN)
%20 = load i32, i32* %3, align 4
%21 = sub nsw i32 %20, 1
%22 = call i32 @Fibonacci(i32 noundef %21)
BB7
T (CALL)
T
T (RETURN)
%23 = load i32, i32* %3, align 4
%24 = sub nsw i32 %23, 2
%25 = call i32 @Fibonacci(i32 noundef %24)
BB8
T (CALL)
T (RETURN)
%26 = add nsw i32 %22, %25
store i32 %26, i32* %2, align 4
br label %27
BB9
FT
27: ; preds = %13, %11, %6
%28 = load i32, i32* %2, align 4
ret i32 %28
BB10
}
LLVM IR

Nella versione con il flag -O2 di Fibonacci.c è stata convertita la chiamata ricorsiva di coda della funzione Fibonacci() in un loop, al fine di migliorare le prestazioni non dovendo andare a gestire una nuova chiamata di funzione sullo stack.

Inoltre, a differenza della versione senza ottimizzazioni (-O0), la versione ottimizzata presenta una conversione degli if statement della funzione Fibonacci() in un singolo switch case.

Provate ad aggiungere il flag *-Rpass=.** al comando di invocazione di clang (con –O2)

```
marrase@DESKTOP-P63EG9I:/mnt/c/C++/Linguaggi-e-Compilatori-2022-2023/Tutorial-01/TestPass$ clang -O2 -emit-l
.*
test/Fibonacci.c:24:29: remark: transforming tail recursion into loop [-Rpass=tailcallelim]
  return Fibonacci(n - 1) + Fibonacci(n - 2);
                            ^
test/Fibonacci.c:24:29: remark: advising against unrolling the loop because it contains a call [-Rpass=TTI]
test/Fibonacci.c:24:29: remark: advising against unrolling the loop because it contains a call [-Rpass=TTI]
```