# COL729 Assignment1 Report

Manish Yadav
2015CS10460

## 1 LLVM IR bitcode

The -O0 doesn't consist of any optimization and hence has a corresponding instruction for each instruction in source code. Loop is implemented using different modules each having a branch instruction at the end. For every variable (including argc and argv), space is allocated on the stack first using alloca instruction, the value is stored in the stack using store and when we need the variable, it is loaded in a temporary variable %n using load (n is an integer).

In -O2, the compiler can detect empty loops and completely omit them in IR. The variables are used directly instead of first storing them in stack (e.g. `- icmp sgt i32 %argc, 1` uses argc directly). Some instructions are also changed. For instance, `argc >= 2` uses a `sgt` instruction instead of `sge` which is used in -O0. In cases where tail call optimizations can be done, it is usually done. It can also detect loops like sumn() which sums the first n integers and use the closed formula to compute the sum instead of iterating in the loop. phi instruction is used in many places.

### 1.1 emptyloop

#### 1.1.1 -O0

It creates space for variables on stack and stores the values in it intially using alloca and store instructions.

```
define i32 @emptyloop(i32 %argc, i8** %argv) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i8**, align 8
  %i = alloca i64, align 8
  %numiter = alloca i64, align 8
  store i32 %argc, i32* %1, align 4
  store i8** %argv, i8*** %2, align 8
  store i64 2147483646, i64* %numiter, align 8
  %3 = load i32* %1, align 4
  %4 = icmp sge i32 %3, 2
  br i1 %4, label %5, label %11
```

Temporary variables %1 and %2 are created to hold the values for argc and argv(pointer) in stack. %3 is created to hold the temporary value which is loaded from stack(it too contains argc). `%4 = icmp sge i32 %3, 2` is the corresponding instruction for `argc >= 2` used in if statement. %numiter stores 2147483646 which is the value of INT_MAX - 1 indicating that it directly computes the value without using a separate `sub` instruction for it. Depending on the result, it branches off to labels %5 or %11.

```
; <label>:5                                       ; preds = %0
  %6 = load i8*** %2, align 8
  %7 = getelementptr inbounds i8** %6, i64 1
  %8 = load i8** %7, align 8
  %9 = call i32 @atoi(i8* %8) #2
  %10 = sext i32 %9 to i64
  store i64 %10, i64* %numiter, align 8
  br label %11
```

The above code shows bitcode for label %5. It executes the statement `numiter = atoi(argv[1])`. First the vales of argv (i.e. the address of the first element) is loaded in %6. getelemenetptr instruction is used to find the *address* of the 1st element of the array which is stored in %7. %8 then stores the value of argv[1] using that address. The `atoi()` function is called on argv[1] and the result is stored in %9. `sext (sign extends)` typecasts the result from i32 to i64 which is finally stored in %numiter. Then it branches off to label %11 which executes the assignment statement of for loop.

```
; <label>:11                                      ; preds = %5, %0
  store i64 0, i64* %i, align 8
  br label %12
```

The above code executes the `i = 0` statement of for loop.

```
; <label>:12                                      ; preds = %23, %11
  %13 = load i64* %i, align 8
  %14 = load i64* %numiter, align 8
  %15 = icmp ult i64 344865, %14
  br i1 %15, label %16, label %18
```

Label %12 executes the compare statement of for loop i.e. `i < MAX(MAGIC_NUMBER,numiter)`. %13 stores i from stack, %14 stores numiter, which is then compared with MAGIC_NUMBER which is 344865. It branches off depending on the compare result. `%15 = icmp ult i64 344865, %14` is essentially executing the `MAX(MAGIC_NUMBER,numiter)` statement.

```
; <label>:16                                      ; preds = %12
  %17 = load i64* %numiter, align 8
  br label %19

; <label>:18                                      ; preds = %12
  br label %19

; <label>:19                                      ; preds = %18, %16
  %20 = phi i64 [ %17, %16 ], [ 344865, %18 ]
  %21 = icmp ult i64 %13, %20
  br i1 %21, label %22, label %26
```

The above code is a continuation of the compare statement of for loop. Depending on the result of the MAX() function, %20 stores either MAGIC_NUMBER or numiter. `%21 = icmp ult i64 %13, %20` is the compare statement of for loop.

```
; <label>:22                                      ; preds = %19
  br label %23

; <label>:23                                      ; preds = %22
  %24 = load i64* %i, align 8
  %25 = add i64 %24, 1
  store i64 %25, i64* %i, align 8
  br label %12
```

Label %23 executes the update statement of for loop and then branches back to the beginning of the loop (label %12). `%25 = add i64 %24, 1` is the `i++` statement.

```
; <label>:26                                      ; preds = %19
  ret i32 0
```

This is the final statement of the code which returns 0;

### 1.1.2 -O2

```
define i32 @emptyloop(i32 %argc, i8** nocapture readonly %argv) #0
   {
  %1 = icmp sgt i32 %argc, 1
  br i1 %1, label %2, label %6

; <label>:2                                    ; preds = %0
  %3 = getelementptr inbounds i8** %argv, i64 1
  %4 = load i8** %3, align 8, !tbaa !1
  %5 = tail call i64 @strtol(i8* nocapture %4, i8** null, i32 10) #2
  br label %6

; <label>:6                                    ; preds = %2, %0
  ret i32 0
```

The above code shows the whole IR code using -O2 flag. We see that variables are not stored on stack but used directly as in the statement `%1 = icmp sgt i32 %argc, 1` where %argc is used directly. Also, the compiler has detected empty loop and omitted it from the IR. IR only consists of the if-statement. It's also using `strtol()` function instead of `atoi()` as mentioned in the source code. Using this, it doesn't require the sext instruction.

## 1.2 fib

### 1.2.1 -O0

```
define i32 @fib(i32 %n) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %n, i32* %2, align 4
  %3 = load i32* %2, align 4
  %4 = icmp slt i32 %3, 2
  br i1 %4, label %5, label %6
```

Initialising of variables on stack takes place. %1 is used to store the return value. %2 stores n on stack. `%4 = icmp slt i32 %3, 2` executes `n < 2` statement inside if(). It then branches depending on the result.

```
; <label>:5                                    ; preds = %0
  store i32 1, i32* %1
  br label %14
```

This stores 1 as the return value in case n ¡ 2.

```
; <label>:6                                       ; preds = %0
  %7 = load i32* %2, align 4
  %8 = sub nsw i32 %7, 1
  %9 = call i32 @fib(i32 %8)
  %10 = load i32* %2, align 4
  %11 = sub nsw i32 %10, 2
  %12 = call i32 @fib(i32 %11)
  %13 = add nsw i32 %9, %12
  store i32 %13, i32* %1
  br label %14
```

This executes the recursive part of the program. %8 stores n - 1. %9 stores the result for *fib(n-1)*. Similarly, %12 stores *fib(n-2)*. The results are added in %13 and stored as the return value in %1 pointer.

```
; <label>:14                                      ; preds = %6, %5
  %15 = load i32* %1
  ret i32 %15
```

This returns from the code using the return value stored at %1.

### 1.2.2 -O2

```
define i32 @fib(i32 %n) #0 {
  %1 = icmp slt i32 %n, 2
  br i1 %1, label %tailrecurse.\_crit\_edge, label
      %tailrecurse.preheader
```

No variable initialisation on stack. It creates new modules %tailrecurse, %tailrecurse._crit_edge etc. and branches to them directly based on the comparison result.

```
tailrecurse.preheader:                            ; preds = %0
  br label %tailrecurse

tailrecurse:                                      ; preds =
    %tailrecurse, %tailrecurse.preheader
  %n.tr2 = phi i32 [ %4, %tailrecurse ], [ %n,
      %tailrecurse.preheader ]
  %accumulator.tr1 = phi i32 [ %5, %tailrecurse ], [ 1,
      %tailrecurse.preheader ]
  %2 = add nsw i32 %n.tr2, -1
  %3 = tail call i32 @fib(i32 %2)
  %4 = add nsw i32 %n.tr2, -2
  %5 = add nsw i32 %3, %accumulator.tr1
  %6 = icmp slt i32 %4, 2
  br i1 %6, label %tailrecurse._crit_edge.loopexit, label
      %tailrecurse
```

Here we can see the tail recursion optimization. The compiler uses tail call instead of call instruction for this. This avoids creating a new stack frame of the function (which happens in -O0) and uses a constant stack space instead. %accumulator.tr1 keeps track of the accumulated sum so far and %n.tr2 keeps track of the current value of n which keeps on updating in `%4 = add nsw i32 %n.tr2, -2`. For example, in case of factorial, the usual approach would be -

```
(fact 3)
(* 3 (fact 2))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 (* 1 (fact 0))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

Whereas using tail-recursion, it would be -

```
(fact 3)
(fact-tail 3 1)
(fact-tail 2 3)
(fact-tail 1 6)
(fact-tail 0 6)
6
```

Here we keep track of the accumulated product so far (1, 3, 6, 6) and the current value of n (3, 2, 1, 0). phi instruction is used to keep track of the previous n.

```
tailrecurse._crit_edge.loopexit:             ; preds = %tailrecurse
  %.lcssa = phi i32 [ %5, %tailrecurse ]
  br label %tailrecurse._crit_edge

tailrecurse._crit_edge:                      ; preds =
    %tailrecurse._crit_edge.loopexit, %0
  %accumulator.tr.lcssa = phi i32 [ 1, %0 ], [ %.lcssa,
      %tailrecurse._crit_edge.loopexit ]
  ret i32 %accumulator.tr.lcssa
```

loopexit is reached from the tailrecurse module and stores final result in %.lcssa. It then goes to _crit_edge. _crit_edge is reached either from loopexit or when the initial n ¡ 2 condition is failed. It gets the final result depending on the previous module using phi and finally returns it.

## 1.3 fibo_iter

### 1.3.1 -O0

```
define i64 @fibo_iter(i32 %n) #0 {
  %1 = alloca i64, align 8
  %2 = alloca i32, align 4
  %fibo_cur = alloca i64, align 8
  %fibo_prev = alloca i64, align 8
  %i = alloca i32, align 4
  %tmp = alloca i32, align 4
  store i32 %n, i32* %2, align 4
  %3 = load i32* %2, align 4
  %4 = icmp ult i32 %3, 3
  br i1 %4, label %5, label %6
```

Variables are allocated and stored on stack. %1 stores the pointer to the result. %2 stores pointer to n. `%4 = icmp ult i32 %3, 3` executes n ¡ 3. The next br statement executes the if statement, branching depending on the result of compare (stored in %4).

```
; <label>:5                                       ; preds = %0
  store i64 1, i64* %1
  br label %24
```

Stores 1 in the final return value (%1 is the pointer to the final return value) in case n ¡ 3.

```
; <label>:6                                       ; preds = %0
  store i64 1, i64* %fibo_cur, align 8
  store i64 1, i64* %fibo_prev, align 8
  store i32 3, i32* %i, align 4
  br label %7
```

The above module is reached of n ¿= 3. It only initialises fibo_cur, fibo_prev and i.

```
; <label>:7                                       ; preds = %19, %6
  %8 = load i32* %i, align 4
  %9 = load i32* %2, align 4
  %10 = icmp ule i32 %8, %9
  br i1 %10, label %11, label %22
```

`%10 = icmp ule i32 %8, %9` statement executes the comparison statement of for loop (i.e. i ¡= n). It then branches depending on the result.

```
; <label>:11                                      ; preds = %7
  %12 = load i64* %fibo_cur, align 8
  %13 = trunc i64 %12 to i32
  store i32 %13, i32* %tmp, align 4
  %14 = load i64* %fibo_prev, align 8
  %15 = load i64* %fibo_cur, align 8
  %16 = add i64 %15, %14
  store i64 %16, i64* %fibo_cur, align 8
  %17 = load i32* %tmp, align 4
  %18 = zext i32 %17 to i64
  store i64 %18, i64* %fibo_prev, align 8
  br label %19
```

This is the main body of the loop. Note that trunc is used to typecast unsigned long to unsigned int and zext(zero extend) is used to typecast unsigned int to unsigned long. `store i32 %13, i32* %tmp, align 4` executes `temp = fibo_cur`. %16 adds fibo_cur and fibo_prev. `store i64 %16, i64* %fibo_cur, align 8` stores the result of addition back in fibo_cur. The next 3 lines execute `fibo_prev = tmp`.

```
; <label>:19                                      ; preds = %11
  %20 = load i32* %i, align 4
  %21 = add i32 %20, 1
  store i32 %21, i32* %i, align 4
  br label %7
```

This executes i++ statement of for loop and then branches off to label 7 where the comparison statement of for loop takes place.

```
; <label>:22                                      ; preds = %7
  %23 = load i64* %fibo_cur, align 8
  store i64 %23, i64* %1
  br label %24
```

```
; <label>:24                                      ; preds = %22, %5
  %25 = load i64* %1
  ret i64 %25
```

Label %22 stores the final result in the pointer pointing to return value (i.e. %1*). Label %24 returns the final value.

### 1.3.2 -O2

```
define i64 @fibo_iter(i32 %n) #0 {
  %1 = icmp ult i32 %n, 3
  br i1 %1, label %.loopexit, label %.lr.ph.preheader
```

This again creates new modules lr.ph, loopexit etc. for optimization. `%1 = icmp ult i32 %n, 3` executes the `if(n < 3)` statement.

```
.lr.ph.preheader:                                 ; preds = %0
  br label %.lr.ph

.lr.ph:                                           ; preds = %.lr.ph,
    %.lr.ph.preheader
  %i.03 = phi i32 [ %4, %.lr.ph ], [ 3, %.lr.ph.preheader ]
  %fibo_prev.02 = phi i64 [ %3, %.lr.ph ], [ 1, %.lr.ph.preheader ]
  %fibo_cur.01 = phi i64 [ %2, %.lr.ph ], [ 1, %.lr.ph.preheader ]
  %2 = add i64 %fibo_prev.02, %fibo_cur.01
  %3 = and i64 %fibo_cur.01, 4294967295
  %4 = add i32 %i.03, 1
  %5 = icmp ugt i32 %4, %n
  br i1 %5, label %.loopexit.loopexit, label %.lr.ph
```

The above code shows the main body of the loop. We see that the tmp variable has been omitted out. fibo_cur is being updated in the line `%2 = add i64 %fibo_prev.02, %fibo_cur.01`. To update fibo_prev, the compiler uses the number 4294967295 which is UINT_MAX. This is done because tmp and fibo_prev are of different types and hence integer overflow can occur. Adding UINT_MAX produces a result which consists of only those bits which tmp can store. The results are stored in temporary variables %2 and %3 and not updated directly. They are updated in the next iteration when these values are initialised in using phi instruction. Hence the value of fibo_cur in `%3 = and i64 %fibo_cur.01, 4294967295` remains fibo_cur and is not fibo_cur + fibo_prev. If tmp were of type unsigned long, then we wouldn't have to add UINT_MAX and could've directly used fibo_cur.01 instead.

```
.loopexit.loopexit:                               ; preds = %.lr.ph
  %.lcssa = phi i64 [ %2, %.lr.ph ]
  br label %.loopexit

.loopexit:                                        ; preds =
    %.loopexit.loopexit, %0
  %.0 = phi i64 [ 1, %0 ], [ %.lcssa, %.loopexit.loopexit ]
  ret i64 %.0
```

This is the final part which returns the return value adter exiting from the loop. `%.0 = phi i64 [ 1, %0 ], [ %.lcssa, %.loopexit.loopexit ]` chooses the return value based on whether we reached here after exiting the loop or directly after the initial if statement of n ¡ 3.

## 1.4 gcd

### 1.4.1 -O0

```
define i32 @gcd1(i32 %a, i32 %b) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %a, i32* %2, align 4
  store i32 %b, i32* %3, align 4
  %4 = load i32* %3, align 4
  %5 = icmp ne i32 %4, 0
  br i1 %5, label %8, label %6

; <label>:6                                       ; preds = %0
  %7 = load i32* %2, align 4
  store i32 %7, i32* %1
  br label %14

; <label>:8                                       ; preds = %0
  %9 = load i32* %3, align 4
  %10 = load i32* %2, align 4
  %11 = load i32* %3, align 4
  %12 = srem i32 %10, %11
  %13 = call i32 @gcd1(i32 %9, i32 %12)
  store i32 %13, i32* %1
  br label %14
```

```
; <label>:14                                      ; preds = %8, %6
  %15 = load i32* %1
  ret i32 %15
```

The above code defined the IR for gcd1(). Initially the variables are allocated on stack. `%5 = icmp ne i32 %4, 0` executes !b statement. This and the next br statement implement the if statement. Label 6 is reached b is 0. It then stores a as the return value (in *%1). Otherwise it goes to label 8 where it recursively calls itself in the line `%13 = call i32 @gcd1(i32 %9, i32 %12)`. Label 14 returns the final value.

```
define i32 @gcd2(i32 %a, i32 %b) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %a, i32* %1, align 4
  store i32 %b, i32* %2, align 4
  br label %3

; <label>:3                                       ; preds = %19, %0
  %4 = load i32* %1, align 4
  %5 = load i32* %2, align 4
  %6 = icmp ne i32 %4, %5
  br i1 %6, label %7, label %20

; <label>:7                                       ; preds = %3
  %8 = load i32* %1, align 4
  %9 = load i32* %2, align 4
  %10 = icmp sgt i32 %8, %9
  br i1 %10, label %11, label %15

; <label>:11                                      ; preds = %7
  %12 = load i32* %2, align 4
  %13 = load i32* %1, align 4
  %14 = sub nsw i32 %13, %12
  store i32 %14, i32* %1, align 4
  br label %19

; <label>:15                                      ; preds = %7
  %16 = load i32* %1, align 4
  %17 = load i32* %2, align 4
  %18 = sub nsw i32 %17, %16
  store i32 %18, i32* %2, align 4
  br label %19
```

```
; <label>:19                                      ; preds = %15, %11
  br label %3

; <label>:20                                      ; preds = %3
  %21 = load i32* %1, align 4
  ret i32 %21
```

The above code shows the implementation for gcd2(). After storing a and b on stack, we branch to label 3 where we check whether a is equal to b or not in the line `%6 = icmp ne i32 %4, %5`. If they are equal, we branch off to label 20 which returns a. Otherwise we branch to label 7 where we then check if a ¿ b or not in line `%10 = icmp sgt i32 %8, %9`. If a ¿ b, we branch to label 11 where we execute `a -= b`. Similarly, we a ¡= b, we branch to label 15 where we execute `b -= a`. At the end of both label 11 and 15, we branch to label 19 and then to 3 which marks the beginning of the while loop.

```
define i32 @gcd3(i32 %a, i32 %b) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %temp = alloca i32, align 4
  store i32 %a, i32* %1, align 4
  store i32 %b, i32* %2, align 4
  br label %3

; <label>:3                                       ; preds = %6, %0
  %4 = load i32* %2, align 4
  %5 = icmp ne i32 %4, 0
  br i1 %5, label %6, label %12

; <label>:6                                       ; preds = %3
  %7 = load i32* %2, align 4
  store i32 %7, i32* %temp, align 4
  %8 = load i32* %1, align 4
  %9 = load i32* %2, align 4
  %10 = srem i32 %8, %9
  store i32 %10, i32* %2, align 4
  %11 = load i32* %temp, align 4
  store i32 %11, i32* %1, align 4
  br label %3

; <label>:12                                      ; preds = %3
```

```
  %13 = load i32* %1, align 4
  ret i32 %13
```

The above code represents gcd3(). After initialising a, b, and temp on stack, we branch to label 3 where we check if b == 0 or not in line `%5 = icmp ne i32 %4, 0`. In case b == 0, we branch to label 12 where we return a. Else we branch to label 6 which is the main body of the function. %7 loads b and the next statement stores it in temp. Similarly, %8 and %9 stores a and b, and %10 stores a % b using `%10 = srem i32 %8, %9`. %11 stores temp and then stores in in a in the next statement. Finally we branch to label 3 which is the beginning of our while loop.

### 1.4.2 -O2

```
define i32 @gcd1(i32 %a, i32 %b) #0 {
  %1 = icmp eq i32 %b, 0
  br i1 %1, label %tailrecurse._crit_edge, label
      %tailrecurse.preheader

tailrecurse.preheader:                        ; preds = %0
  br label %tailrecurse

tailrecurse:                                  ; preds =
    %tailrecurse, %tailrecurse.preheader
  %b.tr2 = phi i32 [ %2, %tailrecurse ], [ %b,
      %tailrecurse.preheader ]
  %a.tr1 = phi i32 [ %b.tr2, %tailrecurse ], [ %a,
      %tailrecurse.preheader ]
  %2 = srem i32 %a.tr1, %b.tr2
  %3 = icmp eq i32 %2, 0
  br i1 %3, label %tailrecurse._crit_edge.loopexit, label
      %tailrecurse

tailrecurse._crit_edge.loopexit:              ; preds = %tailrecurse
  %b.tr2.lcssa = phi i32 [ %b.tr2, %tailrecurse ]
  br label %tailrecurse._crit_edge

tailrecurse._crit_edge:                       ; preds =
    %tailrecurse._crit_edge.loopexit, %0
  %a.tr.lcssa = phi i32 [ %a, %0 ], [ %b.tr2.lcssa,
      %tailrecurse._crit_edge.loopexit ]
  ret i32 %a.tr.lcssa
```

The above code represents gcd1(). We check if b is zero or not in line `%1 = icmp eq i32 %b, 0`. If it's zero, we branch to tailrecurse._crit_edge and return a. Otherwise we branch to tailrecurse.preheader and then to tailrecurse. In tailrecurse, we get the values of a.tr1(first term of function) and b.tr2(second term of function) using phi instruction (depending on the previous module from which we entered it). %2 stores a % b which is then used as the second term in next iteration (i.e. b). If it's zero, (checked in `%3 = icmp eq i32 %2, 0`), then we exit the loop. Else we branch to tailrecurse again. After exiting the loop, we store the first term (i.e. a) as the return value in b.tr2.lcssa. The final return value is choosen depending on whether we reached there after recursing or directly after the first if loop (`%a.tr.lcssa = phi i32 [ %a, %0 ], [ %b.tr2.lcssa, %tailrecurse._crit_edge.loopexit]`).

```
define i32 @gcd2(i32 %a, i32 %b) #0 {
  %1 = icmp eq i32 %a, %b
  br i1 %1, label %.outer._crit_edge, label %.lr.ph.preheader

.lr.ph.preheader:                                 ; preds = %0
  br label %.lr.ph

.lr.ph:                                           ; preds = %.outer,
    %.lr.ph.preheader
  %.0.ph6 = phi i32 [ %.03.lcssa, %.outer ], [ %b,
     %.lr.ph.preheader ]
  %.01.ph5 = phi i32 [ %7, %.outer ], [ %a, %.lr.ph.preheader ]
  br label %4

; <label>:2                                       ; preds = %4
  %3 = icmp eq i32 %.01.ph5, %6
  br i1 %3, label %.outer._crit_edge.loopexit, label %4

; <label>:4                                       ; preds = %2, %.lr.ph
  %.03 = phi i32 [ %.0.ph6, %.lr.ph ], [ %6, %2 ]
  %5 = icmp slt i32 %.03, %.01.ph5
  %6 = sub nsw i32 %.03, %.01.ph5
  br i1 %5, label %.outer, label %2

.outer:                                           ; preds = %4
  %.03.lcssa = phi i32 [ %.03, %4 ]
  %7 = sub nsw i32 %.01.ph5, %.03.lcssa
  %8 = icmp eq i32 %7, %.03.lcssa
  br i1 %8, label %.outer._crit_edge.loopexit13, label %.lr.ph
```

```
.outer._crit_edge.loopexit:                         ; preds = %2
  %.01.ph5.lcssa = phi i32 [ %.01.ph5, %2 ]
  br label %.outer._crit_edge

.outer._crit_edge.loopexit13:                       ; preds = %.outer
  %.03.lcssa.lcssa = phi i32 [ %.03.lcssa, %.outer ]
  br label %.outer._crit_edge

.outer._crit_edge:                                  ; preds =
    %.outer._crit_edge.loopexit13, %.outer._crit_edge.loopexit, %0
  %.01.ph.lcssa = phi i32 [ %a, %0 ], [ %.01.ph5.lcssa,
      %.outer._crit_edge.loopexit ], [ %.03.lcssa.lcssa,
      %.outer._crit_edge.loopexit13 ]
  ret i32 %.01.ph.lcssa
```

The above code shows the IR for gcd2(). After comparing a and b in `%1 = icmp eq i32 %a, %b`, we branch to either outer._crit_edge (exit the loop and return a) or to lr.ph.header(enter the while loop). From lr.ph.header, we go to lr.ph which initialises 0.ph6 to b and 01.ph5 to a. Then we branch to label 4. Label 4 compares if a ¿ b or not (`%5 = icmp slt i32 %.03, %.01.ph5`). If a ¿ b, we branch to .outer where we store a - b in %7(`%7 = sub nsw i32 %.01.ph5, %.03.lcssa`). Since this will be our new a, we check here only if it's equal to b or not and exit the loop incase it is (`%8 = icmp eq i32 %7, %.03.lcssa`). If it's not, we branch to lr.ph again where the new a is initialised as %7 (which had a - b). Similarly, in case a ¡= b, we branch to label 2, where we check if a is equal to %6 (which stores b - a from .outer). If it is, we break from the loop and branch to outer._crit_edge. Else, we go to label 4 again where the new b is initialised as %6 (i.e. b - a). Since we exit the loop from many locations, we need to correctly initilise the return value. It'll either be the new value of a (i.e. a - b) stored in some temprary variable from previous module or it'll be a itself(in case the previous module had changed b i.e. executed b -= a). This is done using phi instruction. The final value is then returned.

## 1.5 loops

### 1.5.1 -O0

```
define zeroext i1 @is_sorted(i32* %a, i32 %n) #0 {
  %1 = alloca i1, align 1
  %2 = alloca i32*, align 8
  %3 = alloca i32, align 4
  %i = alloca i32, align 4
  store i32* %a, i32** %2, align 8
  store i32 %n, i32* %3, align 4
  store i32 0, i32* %i, align 4
  br label %4

; <label>:4                                       ; preds = %24, %0
  %5 = load i32* %i, align 4
  %6 = load i32* %3, align 4
  %7 = sub nsw i32 %6, 1
  %8 = icmp slt i32 %5, %7
  br i1 %8, label %9, label %27

; <label>:9                                       ; preds = %4
  %10 = load i32* %i, align 4
  %11 = sext i32 %10 to i64
  %12 = load i32** %2, align 8
  %13 = getelementptr inbounds i32* %12, i64 %11
  %14 = load i32* %13, align 4
  %15 = load i32* %i, align 4
  %16 = add nsw i32 %15, 1
  %17 = sext i32 %16 to i64
  %18 = load i32** %2, align 8
  %19 = getelementptr inbounds i32* %18, i64 %17
  %20 = load i32* %19, align 4
  %21 = icmp sgt i32 %14, %20
  br i1 %21, label %22, label %23

; <label>:22                                      ; preds = %9
  store i1 false, i1* %1
  br label %28

; <label>:23                                      ; preds = %9
  br label %24

; <label>:24                                      ; preds = %23
  %25 = load i32* %i, align 4
  %26 = add nsw i32 %25, 1
  store i32 %26, i32* %i, align 4
  br label %4

; <label>:27                                      ; preds = %4
```

```
  store i1 true, i1* %1
  br label %28

; <label>:28                                      ; preds = %27, %22
  %29 = load i1* %1
  ret i1 %29
```

---

The above code represents the is_sorted() function. Variables are initialized on stack. %1 stores the pointer to the return value, %2 stores pointer to a[], %3 stores pointer to n. We then branch to label 4 after storing their values. Label 4 performs the comparison check of for loop. %7 stores n - 1 and %5 stores i. They are compared in `%8 = icmp slt i32 %5, %7`. Depending on the result we branch to different labels. If i ¿= n-1, we branch to label 27 which stores true as the return value and then branches to label 28 which returns this value. Otherwise we branch to label 9 which is the inside of the loop. We load the stack values in temprary variables. %13 stores address of a[i], %14 stores value of a[i], %16 stores i + 1, %19 stores address of a[i+1], %20 stores value of a[i+1]. We compare a[i] and a[i+1] using `%21 = icmp sgt i32 %14, %20`. If a[i] ¿ a[i + 1], we branch to label 22 which stores false as the return value and then branches to label 28 which returns this value. Otherwise we branch to label 23 and then to 24 where the update part of for loop takes place (i.e. i++). `%26 = add nsw i32 %25, 1` does i++ and stores the result back in stack before branching to label 4 which is the beginning of the loop.

---

```
define void @add_arrays(i32* %a, i32* %b, i32* %c, i32 %n) #0 {
  %1 = alloca i32*, align 8
  %2 = alloca i32*, align 8
  %3 = alloca i32*, align 8
  %4 = alloca i32, align 4
  %i = alloca i32, align 4
  store i32* %a, i32** %1, align 8
  store i32* %b, i32** %2, align 8
  store i32* %c, i32** %3, align 8
  store i32 %n, i32* %4, align 4
  store i32 0, i32* %i, align 4
  br label %5

; <label>:5                                       ; preds = %25, %0
  %6 = load i32* %i, align 4
  %7 = load i32* %4, align 4
  %8 = icmp slt i32 %6, %7
  br i1 %8, label %9, label %28

; <label>:9                                       ; preds = %5
  %10 = load i32* %i, align 4
  %11 = sext i32 %10 to i64
  %12 = load i32** %1, align 8
  %13 = getelementptr inbounds i32* %12, i64 %11
  %14 = load i32* %13, align 4
  %15 = load i32* %i, align 4
  %16 = sext i32 %15 to i64
  %17 = load i32** %2, align 8
  %18 = getelementptr inbounds i32* %17, i64 %16
  %19 = load i32* %18, align 4
  %20 = add nsw i32 %14, %19
  %21 = load i32* %i, align 4
  %22 = sext i32 %21 to i64
  %23 = load i32** %3, align 8
  %24 = getelementptr inbounds i32* %23, i64 %22
  store i32 %20, i32* %24, align 4
  br label %25

; <label>:25                                      ; preds = %9
  %26 = load i32* %i, align 4
  %27 = add nsw i32 %26, 1
  store i32 %27, i32* %i, align 4
  br label %5

; <label>:28                                      ; preds = %5
  ret void
```

The above represents the IR for add_arrays function. Again the variables are assigned space on stack and then initialised. %1, %2, %3 are pointers which stores address for arrays a, b, and c. %4 is a pointer to n. After initialising variables, we branch to label 5, where we compare i with n (`%8 = icmp slt i32 %6, %7`) i.e. the comparison statement of for loop. If i ¿= n, we exit the loop by branching to label 28 which returns void. Otherwise we branch to label 9 which is the main body of the loop. Temporary variables are initialised again. %14 and %19 store the values of a[i] and b[i]. We store their sum in %20 (`%20 = add nsw i32 %14, %19`). %24 stores the address for c[i]. The sum is then stored at this location. (`store i32 %20, i32* %24, align 4`). We then branch to label 25 which updates the value of i i.e. i++(`%27 = add nsw i32 %26, 1`) and then stores it in stack(`store i32

`%27, i32* %i, align 4`). We then branch to label 5 which is the beginning of the loop.

```
define i32 @sum(i8* %a, i32 %n) #0 {
  %1 = alloca i8*, align 8
  %2 = alloca i32, align 4
  %ret = alloca i32, align 4
  %i = alloca i32, align 4
  store i8* %a, i8** %1, align 8
  store i32 %n, i32* %2, align 4
  store i32 0, i32* %ret, align 4
  store i32 0, i32* %i, align 4
  br label %3

; <label>:3                                   ; preds = %16, %0
  %4 = load i32* %i, align 4
  %5 = load i32* %2, align 4
  %6 = icmp slt i32 %4, %5
  br i1 %6, label %7, label %19

; <label>:7                                   ; preds = %3
  %8 = load i32* %i, align 4
  %9 = sext i32 %8 to i64
  %10 = load i8** %1, align 8
  %11 = getelementptr inbounds i8* %10, i64 %9
  %12 = load i8* %11, align 1
  %13 = zext i8 %12 to i32
  %14 = load i32* %ret, align 4
  %15 = add nsw i32 %14, %13
  store i32 %15, i32* %ret, align 4
  br label %16

; <label>:16                                  ; preds = %7
  %17 = load i32* %i, align 4
  %18 = add nsw i32 %17, 1
  store i32 %18, i32* %i, align 4
  br label %3

; <label>:19                                  ; preds = %3
  %20 = load i32* %ret, align 4
  ret i32 %20
```

The above shows IR for sum() function. Variables are initialised on stack.

%1 is the pointer to array a, %2 is a pointer to n, %ret and %i are pointers to return value and i respectively. We branch to label 3 adter initialization. In label 3, we perform the loop comparison operation i.e. i ¡ n(`%6 = icmp slt i32 %4, %5`). If i ¿= n, we exit the loop by branching to label 19 which returns the final sum. Otherwise we branch to label 7 which is the main body of the loop. We store addressess and values in temprary variables. %10 stores address of a[i], %12 stores value of a[i]. zext(zero extends) is used to typecast. %14 stores the current value of ret. We add a[i] and ret using `%15 = add nsw i32 %14, %13` and store it back in stack (`store i32 %15, i32* %ret, align 4`). We then branch to label 16 where we update the value of i. %17 stores the current value of i. `%18 = add nsw i32 %17, 1` preforms i++ and `store i32 %18, i32* %i, align 4` stores it back in the stack. We branch back to label 3 which is the beginning of the loop.

```
define i32 @sumn(i32 %n) #0 {
  %1 = alloca i32, align 4
  %ret = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  store i32 0, i32* %ret, align 4
  store i32 0, i32* %i, align 4
  br label %2

; <label>:2                                       ; preds = %10, %0
  %3 = load i32* %i, align 4
  %4 = load i32* %1, align 4
  %5 = icmp slt i32 %3, %4
  br i1 %5, label %6, label %13

; <label>:6                                       ; preds = %2
  %7 = load i32* %ret, align 4
  %8 = load i32* %i, align 4
  %9 = add nsw i32 %7, %8
  store i32 %9, i32* %ret, align 4
  br label %10

; <label>:10                                      ; preds = %6
  %11 = load i32* %i, align 4
  %12 = add nsw i32 %11, 1
  store i32 %12, i32* %i, align 4
  br label %2

; <label>:13                                      ; preds = %2
```

```
  %14 = load i32* %ret, align 4
  ret i32 %14
```

The above code shows the IR for sumn() function. The variables are initialised on stack. %1 is pointer to n, %ret is pointer to n, %i is pointer to i. After initialising them on stack, we branch to label 2 where we check the comparison statement of for loop i.e. if i ¡ n or not(`%5 = icmp slt i32 %3, %4`). If i ¿= n, we branch to label 13 and return the return value. Otherwise we branch to label 6 which is the main body of the loop. %7 stores the current value of ret and %8 stores i. We add then (`%9 = add nsw i32 %7, %8`) and store it back in stack(`store i32 %9, i32* %ret, align 4`). We then branch to label 10 which updates the value of i(i.e. i++) before branching to label 2 again which is the beginning of the for loop.

### 1.5.2 -O2

```
define zeroext i1 @is_sorted(i32* nocapture readonly %a, i32 %n)
    #0 {
  %1 = add nsw i32 %n, -1
  %2 = sext i32 %1 to i64
  br label %3

; <label>:3                                       ; preds = %5, %0
  %indvars.iv = phi i64 [ %indvars.iv.next, %5 ], [ 0, %0 ]
  %4 = icmp slt i64 %indvars.iv, %2
  br i1 %4, label %5, label %11

; <label>:5                                       ; preds = %3
  %6 = getelementptr inbounds i32* %a, i64 %indvars.iv
  %7 = load i32* %6, align 4, !tbaa !1
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %8 = getelementptr inbounds i32* %a, i64 %indvars.iv.next
  %9 = load i32* %8, align 4, !tbaa !1
  %10 = icmp sgt i32 %7, %9
  br i1 %10, label %11, label %3

; <label>:11                                      ; preds = %5, %3
  %.0 = phi i1 [ false, %5 ], [ true, %3 ]
  ret i1 %.0
```

Above code shows IR for is_sorted function with -O2 flag. %1 stores n - 1. %2 stores n - 1 signed extended to unsigned long. We branch to label 3.

%indvars.iv stores the value of i, and %indvars.iv.next (in label 6) stores i + 1. %indvars.iv is initialised to 0 originally and then updated from label 6's value. We compare i with n-1 using `%4 = icmp slt i64 %indvars.iv, %2` and then branch accordingly. If i ¡ n-1, we branch to label 5 which is the main body of the loop, else we branch to label 11 where we return the final value. In label 5, %6 stores address for a[i], %7 stores value of a[i], %indvars.iv.next stores i + 1, %8 stores address for a[i + 1], %9 stores value of a[i+1]. We compare a[i] and a[i+1] using `%10 = icmp sgt i32 %7, %9`. If a[i] ¿ a[i + 1], we exit the loop by branching to label 11 and returning false. Else after the loop is done, we branch to label 11 and return true.

---

```
define void @add_arrays(i32* nocapture readonly %a, i32* nocapture
    readonly %b, i32* nocapture %c, i32 %n) #1 {
  %1 = icmp sgt i32 %n, 0
  br i1 %1, label %.lr.ph, label %._crit_edge

.lr.ph:                                           ; preds = %0
  %2 = add i32 %n, -1
  %3 = zext i32 %2 to i64
  %4 = add nuw nsw i64 %3, 1
  %end.idx = add nuw nsw i64 %3, 1
  %n.vec = and i64 %4, 8589934584
  %cmp.zero = icmp eq i64 %n.vec, 0
  %5 = add i32 %n, -1
  %6 = zext i32 %5 to i64
  %scevgep = getelementptr i32* %c, i64 %6
  br i1 %cmp.zero, label %middle.block, label %vector.memcheck

vector.memcheck:                                  ; preds = %.lr.ph
  %scevgep5 = getelementptr i32* %a, i64 %6
  %bound0 = icmp uge i32* %scevgep5, %c
  %bound1 = icmp uge i32* %scevgep, %a
  %found.conflict = and i1 %bound0, %bound1
  %scevgep8 = getelementptr i32* %b, i64 %6
  %bound010 = icmp uge i32* %scevgep8, %c
  %bound111 = icmp uge i32* %scevgep, %b
  %found.conflict12 = and i1 %bound010, %bound111
  %conflict.rdx = or i1 %found.conflict, %found.conflict12
  br i1 %conflict.rdx, label %middle.block, label
      %vector.body.preheader

vector.body.preheader:                            ; preds =
    %vector.memcheck
  br label %vector.body

vector.body:                                      ; preds =
    %vector.body, %vector.body.preheader
  %index = phi i64 [ %index.next, %vector.body ], [ 0,
      %vector.body.preheader ]
  %7 = getelementptr inbounds i32* %a, i64 %index
  %8 = bitcast i32* %7 to <4 x i32>*
  %wide.load = load <4 x i32>* %8, align 4, !tbaa !1
  %.sum23 = or i64 %index, 4
  %9 = getelementptr i32* %a, i64 %.sum23
  %10 = bitcast i32* %9 to <4 x i32>*
  %wide.load14 = load <4 x i32>* %10, align 4, !tbaa !1
  %11 = getelementptr inbounds i32* %b, i64 %index
  %12 = bitcast i32* %11 to <4 x i32>*
  %wide.load15 = load <4 x i32>* %12, align 4, !tbaa !1
  %.sum24 = or i64 %index, 4
  %13 = getelementptr i32* %b, i64 %.sum24
  %14 = bitcast i32* %13 to <4 x i32>*
  %wide.load16 = load <4 x i32>* %14, align 4, !tbaa !1
  %15 = add nsw <4 x i32> %wide.load15, %wide.load
  %16 = add nsw <4 x i32> %wide.load16, %wide.load14
  %17 = getelementptr inbounds i32* %c, i64 %index
  %18 = bitcast i32* %17 to <4 x i32>*
  store <4 x i32> %15, <4 x i32>* %18, align 4, !tbaa !1
  %.sum25 = or i64 %index, 4
  %19 = getelementptr i32* %c, i64 %.sum25
  %20 = bitcast i32* %19 to <4 x i32>*
  store <4 x i32> %16, <4 x i32>* %20, align 4, !tbaa !1
  %index.next = add i64 %index, 8
  %21 = icmp eq i64 %index.next, %n.vec
  br i1 %21, label %middle.block.loopexit, label %vector.body,
      !llvm.loop !5

middle.block.loopexit:                            ; preds = %vector.body
  br label %middle.block

middle.block:                                     ; preds =
    %middle.block.loopexit, %vector.memcheck, %.lr.ph
  %resume.val = phi i64 [ 0, %.lr.ph ], [ 0, %vector.memcheck ], [
      %n.vec, %middle.block.loopexit ]
  %cmp.n = icmp eq i64 %end.idx, %resume.val
  br i1 %cmp.n, label %._crit_edge, label %scalar.ph.preheader
```

```
scalar.ph.preheader:                              ; preds =
   %middle.block
  %22 = trunc i64 %resume.val to i32
  %23 = sub i32 %n, %22
  %24 = add i32 %n, -1
  %25 = sub i32 %24, %22
  %xtraiter = and i32 %23, 1
  %lcmp.mod = icmp ne i32 %xtraiter, 0
  br i1 %lcmp.mod, label %scalar.ph.prol, label
     %scalar.ph.preheader.split

scalar.ph.prol:                                   ; preds =
   %scalar.ph.preheader
  %26 = getelementptr inbounds i32* %a, i64 %resume.val
  %27 = load i32* %26, align 4, !tbaa !1
  %28 = getelementptr inbounds i32* %b, i64 %resume.val
  %29 = load i32* %28, align 4, !tbaa !1
  %30 = add nsw i32 %29, %27
  %31 = getelementptr inbounds i32* %c, i64 %resume.val
  store i32 %30, i32* %31, align 4, !tbaa !1
  %indvars.iv.next.prol = add nuw nsw i64 %resume.val, 1
  %lftr.wideiv.prol = trunc i64 %resume.val to i32
  %exitcond.prol = icmp eq i32 %lftr.wideiv.prol, %2
  br label %scalar.ph.preheader.split

scalar.ph.preheader.split:                        ; preds =
   %scalar.ph.prol, %scalar.ph.preheader
  %indvars.iv.unr = phi i64 [ %resume.val, %scalar.ph.preheader ],
     [ %indvars.iv.next.prol, %scalar.ph.prol ]
  %32 = icmp ult i32 %25, 1
  br i1 %32, label %._crit_edge.loopexit, label
     %scalar.ph.preheader.split.split

scalar.ph.preheader.split.split:                  ; preds =
   %scalar.ph.preheader.split
  br label %scalar.ph

scalar.ph:                                        ; preds = %scalar.ph,
   %scalar.ph.preheader.split.split
  %indvars.iv = phi i64 [ %indvars.iv.unr,
     %scalar.ph.preheader.split.split ], [ %indvars.iv.next.1,
     %scalar.ph ]
  %33 = getelementptr inbounds i32* %a, i64 %indvars.iv
  %34 = load i32* %33, align 4, !tbaa !1
  %35 = getelementptr inbounds i32* %b, i64 %indvars.iv
  %36 = load i32* %35, align 4, !tbaa !1
  %37 = add nsw i32 %36, %34
  %38 = getelementptr inbounds i32* %c, i64 %indvars.iv
  store i32 %37, i32* %38, align 4, !tbaa !1
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv to i32
  %39 = getelementptr inbounds i32* %a, i64 %indvars.iv.next
  %40 = load i32* %39, align 4, !tbaa !1
  %41 = getelementptr inbounds i32* %b, i64 %indvars.iv.next
  %42 = load i32* %41, align 4, !tbaa !1
  %43 = add nsw i32 %42, %40
  %44 = getelementptr inbounds i32* %c, i64 %indvars.iv.next
  store i32 %43, i32* %44, align 4, !tbaa !1
  %indvars.iv.next.1 = add nuw nsw i64 %indvars.iv.next, 1
  %lftr.wideiv.1 = trunc i64 %indvars.iv.next to i32
  %exitcond.1 = icmp eq i32 %lftr.wideiv.1, %2
  br i1 %exitcond.1, label %._crit_edge.loopexit.unr-lcssa, label
     %scalar.ph, !llvm.loop !8

._crit_edge.loopexit.unr-lcssa:                 ; preds = %scalar.ph
  br label %._crit_edge.loopexit

._crit_edge.loopexit:                           ; preds =
    %._crit_edge.loopexit.unr-lcssa, %scalar.ph.preheader.split
  br label %._crit_edge

._crit_edge:                                    ; preds =
    %._crit_edge.loopexit, %middle.block, %0
  ret void
```

The above code shows the add_arrays function. We initially check if n ¿ 0 or not. If not, we directly branch to ._crit_edge and return void. Else we branch to label .lr.ph. Here, %2 stores n - 1, and %3 stores the same thing typecasted to unsigned long. %4 and %end.ind stores n. %n.vec also stores n initially (8589934584 is the max possible value that can be stored in unsigned long). %5 and %6 are the same as %2 and %3 (duplicated possibly because we create separate variables for separate arrays). %scevgep stores the address for c[i]. ............................

```
define i32 @sum(i8* nocapture readonly %a, i32 %n) #0 {
  %1 = icmp sgt i32 %n, 0
  br i1 %1, label %.lr.ph, label %._crit_edge

.lr.ph:                                           ; preds = %0
  %2 = add i32 %n, -1
  %3 = zext i32 %2 to i64
  %4 = add nuw nsw i64 %3, 1
  %end.idx = add nuw nsw i64 %3, 1
  %n.vec = and i64 %4, 8589934584
  %cmp.zero = icmp eq i64 %n.vec, 0
  br i1 %cmp.zero, label %middle.block, label %vector.body.preheader

vector.body.preheader:                            ; preds = %.lr.ph
  br label %vector.body

vector.body:                                      ; preds =
    %vector.body, %vector.body.preheader
  %index = phi i64 [ %index.next, %vector.body ], [ 0,
      %vector.body.preheader ]
  %vec.phi = phi <4 x i32> [ %11, %vector.body ], [
      zeroinitializer, %vector.body.preheader ]
  %vec.phi4 = phi <4 x i32> [ %12, %vector.body ], [
      zeroinitializer, %vector.body.preheader ]
  %5 = getelementptr inbounds i8* %a, i64 %index
  %6 = bitcast i8* %5 to <4 x i8>*
  %wide.load = load <4 x i8>* %6, align 1, !tbaa !9
  %.sum16 = or i64 %index, 4
  %7 = getelementptr i8* %a, i64 %.sum16
  %8 = bitcast i8* %7 to <4 x i8>*
  %wide.load5 = load <4 x i8>* %8, align 1, !tbaa !9
  %9 = zext <4 x i8> %wide.load to <4 x i32>
  %10 = zext <4 x i8> %wide.load5 to <4 x i32>
  %11 = add nsw <4 x i32> %9, %vec.phi
  %12 = add nsw <4 x i32> %10, %vec.phi4
  %index.next = add i64 %index, 8
  %13 = icmp eq i64 %index.next, %n.vec
  br i1 %13, label %middle.block.loopexit, label %vector.body,
      !llvm.loop !10

middle.block.loopexit:                            ; preds = %vector.body
  %.lcssa18 = phi <4 x i32> [ %12, %vector.body ]
  %.lcssa17 = phi <4 x i32> [ %11, %vector.body ]
  br label %middle.block
```

```
middle.block:                                     ; preds =
   %middle.block.loopexit, %.lr.ph
  %resume.val = phi i64 [ 0, %.lr.ph ], [ %n.vec,
     %middle.block.loopexit ]
  %rdx.vec.exit.phi = phi <4 x i32> [ zeroinitializer, %.lr.ph ], [
     %.lcssa17, %middle.block.loopexit ]
  %rdx.vec.exit.phi12 = phi <4 x i32> [ zeroinitializer, %.lr.ph ],
     [ %.lcssa18, %middle.block.loopexit ]
  %bin.rdx = add <4 x i32> %rdx.vec.exit.phi12, %rdx.vec.exit.phi
  %rdx.shuf = shufflevector <4 x i32> %bin.rdx, <4 x i32> undef, <4
     x i32> <i32 2, i32 3, i32 undef, i32 undef>
  %bin.rdx13 = add <4 x i32> %bin.rdx, %rdx.shuf
  %rdx.shuf14 = shufflevector <4 x i32> %bin.rdx13, <4 x i32>
     undef, <4 x i32> <i32 1, i32 undef, i32 undef, i32 undef>
  %bin.rdx15 = add <4 x i32> %bin.rdx13, %rdx.shuf14
  %14 = extractelement <4 x i32> %bin.rdx15, i32 0
  %cmp.n = icmp eq i64 %end.idx, %resume.val
  br i1 %cmp.n, label %._crit_edge, label %scalar.ph.preheader

scalar.ph.preheader:                              ; preds =
   %middle.block
  %15 = trunc i64 %resume.val to i32
  %16 = sub i32 %n, %15
  %17 = add i32 %n, -1
  %18 = sub i32 %17, %15
  %xtraiter = and i32 %16, 3
  %lcmp.mod = icmp ne i32 %xtraiter, 0
  br i1 %lcmp.mod, label %scalar.ph.prol, label
     %scalar.ph.preheader.split

scalar.ph.prol:                                   ; preds =
   %scalar.ph.prol, %scalar.ph.preheader
  %indvars.iv.prol = phi i64 [ %indvars.iv.next.prol,
     %scalar.ph.prol ], [ %resume.val, %scalar.ph.preheader ]
  %ret.01.prol = phi i32 [ %22, %scalar.ph.prol ], [ %14,
     %scalar.ph.preheader ]
  %prol.iter = phi i32 [ %xtraiter, %scalar.ph.preheader ], [
     %prol.iter.sub, %scalar.ph.prol ]
  %19 = getelementptr inbounds i8* %a, i64 %indvars.iv.prol
  %20 = load i8* %19, align 1, !tbaa !9
  %21 = zext i8 %20 to i32
  %22 = add nsw i32 %21, %ret.01.prol
  %indvars.iv.next.prol = add nuw nsw i64 %indvars.iv.prol, 1
  %lftr.wideiv.prol = trunc i64 %indvars.iv.prol to i32
  %exitcond.prol = icmp eq i32 %lftr.wideiv.prol, %2
  %prol.iter.sub = sub i32 %prol.iter, 1
  %prol.iter.cmp = icmp ne i32 %prol.iter.sub, 0
  br i1 %prol.iter.cmp, label %scalar.ph.prol, label
     %scalar.ph.preheader.split, !llvm.loop !11

scalar.ph.preheader.split:                     ; preds =
   %scalar.ph.prol, %scalar.ph.preheader
  %.lcssa.unr = phi i32 [ 0, %scalar.ph.preheader ], [ %22,
     %scalar.ph.prol ]
  %indvars.iv.unr = phi i64 [ %resume.val, %scalar.ph.preheader ],
     [ %indvars.iv.next.prol, %scalar.ph.prol ]
  %ret.01.unr = phi i32 [ %14, %scalar.ph.preheader ], [ %22,
     %scalar.ph.prol ]
  %23 = icmp ult i32 %18, 3
  br i1 %23, label %._crit_edge.loopexit, label
     %scalar.ph.preheader.split.split

scalar.ph.preheader.split.split:               ; preds =
   %scalar.ph.preheader.split
  br label %scalar.ph

scalar.ph:                                     ; preds = %scalar.ph,
   %scalar.ph.preheader.split.split
  %indvars.iv = phi i64 [ %indvars.iv.unr,
     %scalar.ph.preheader.split.split ], [ %indvars.iv.next.3,
     %scalar.ph ]
  %ret.01 = phi i32 [ %ret.01.unr, %scalar.ph.preheader.split.split
     ], [ %39, %scalar.ph ]
  %24 = getelementptr inbounds i8* %a, i64 %indvars.iv
  %25 = load i8* %24, align 1, !tbaa !9
  %26 = zext i8 %25 to i32
  %27 = add nsw i32 %26, %ret.01
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv to i32
  %28 = getelementptr inbounds i8* %a, i64 %indvars.iv.next
  %29 = load i8* %28, align 1, !tbaa !9
  %30 = zext i8 %29 to i32
  %31 = add nsw i32 %30, %27
  %indvars.iv.next.1 = add nuw nsw i64 %indvars.iv.next, 1
  %lftr.wideiv.1 = trunc i64 %indvars.iv.next to i32
  %32 = getelementptr inbounds i8* %a, i64 %indvars.iv.next.1
  %33 = load i8* %32, align 1, !tbaa !9
  %34 = zext i8 %33 to i32
  %35 = add nsw i32 %34, %31
  %indvars.iv.next.2 = add nuw nsw i64 %indvars.iv.next.1, 1
  %lftr.wideiv.2 = trunc i64 %indvars.iv.next.1 to i32
  %36 = getelementptr inbounds i8* %a, i64 %indvars.iv.next.2
  %37 = load i8* %36, align 1, !tbaa !9
  %38 = zext i8 %37 to i32
  %39 = add nsw i32 %38, %35
  %indvars.iv.next.3 = add nuw nsw i64 %indvars.iv.next.2, 1
  %lftr.wideiv.3 = trunc i64 %indvars.iv.next.2 to i32
  %exitcond.3 = icmp eq i32 %lftr.wideiv.3, %2
  br i1 %exitcond.3, label %._crit_edge.loopexit.unr-lcssa, label
     %scalar.ph, !llvm.loop !13

._crit_edge.loopexit.unr-lcssa:              ; preds = %scalar.ph
  %.lcssa.ph = phi i32 [ %39, %scalar.ph ]
  br label %._crit_edge.loopexit

._crit_edge.loopexit:                        ; preds =
   %._crit_edge.loopexit.unr-lcssa, %scalar.ph.preheader.split
  %.lcssa = phi i32 [ %.lcssa.unr, %scalar.ph.preheader.split ], [
     %.lcssa.ph, %._crit_edge.loopexit.unr-lcssa ]
  br label %._crit_edge

._crit_edge:                                 ; preds =
   %._crit_edge.loopexit, %middle.block, %0
  %ret.0.lcssa = phi i32 [ 0, %0 ], [ %14, %middle.block ], [
     %.lcssa, %._crit_edge.loopexit ]
  ret i32 %ret.0.lcssa
```

The above code shows the IR for sum() function.

```
define i32 @sumn(i32 %n) #2 {
  %1 = icmp sgt i32 %n, 0
  br i1 %1, label %.lr.ph, label %11

.lr.ph:                                      ; preds = %0
  %2 = add i32 %n, -1
  %3 = zext i32 %2 to i33
  %4 = add i32 %n, -2
  %5 = zext i32 %4 to i33
  %6 = mul i33 %3, %5
  %7 = lshr i33 %6, 1
  %8 = trunc i33 %7 to i32
  %9 = add i32 %8, %n
  %10 = add i32 %9, -1
  br label %11

; <label>:11                                      ; preds = %.lr.ph, %0
  %ret.0.lcssa = phi i32 [ %10, %.lr.ph ], [ 0, %0 ]
  ret i32 %ret.0.lcssa
```

The above code shows IR for sumn() function. We initially check if n is zero or not. If yes, then we branch to label 11 and return 0. Else we branch to label .lr.ph which is the main body of the loop. Instead of iterating the loop, it calculates the return value using the formula for the sum of first n elements i.e.

$$\sum_{i=1}^{n} i = (n * (n + 1))/2$$

. %3 stores n - 1 (zero extended), and %5 stores n - 2(zero extended). %6 stored (n - 1) * (n - 2). `%7 = lshr i33 %6, 1` shifts the product right by 1 bit essentially dividing it by 2. %8 truncates it back to int from long. Till here, %8 contains

$$((n - 1) * (n - 2))/2$$

which is the sum of first n - 2 integers. But we want the sum of first n - 1 integers. Hence we add n - 1 to it using `%9 = add i32 %8, %n` and `%10 = add i32 %9, -1`. Finally we branch to label 11 and return the result.

## 1.6 print_arg

### 1.6.1 -O0

```
define i32 @print_arg(i32 %argc, i8** %argv) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8**, align 8
  store i32 %argc, i32* %2, align 4
  store i8** %argv, i8*** %3, align 8
  %4 = load i32* %2, align 4
  %5 = icmp ne i32 %4, 2
  br i1 %5, label %6, label %7

; <label>:6                                       ; preds = %0
```

```
  store i32 -1, i32* %1
  br label %12

; <label>:7                                    ; preds = %0
  %8 = load i8*** %3, align 8
  %9 = getelementptr inbounds i8** %8, i64 1
  %10 = load i8** %9, align 8
  %11 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3
      x i8]* @.str, i32 0, i32 0), i8* %10)
  store i32 0, i32* %1
  br label %12

; <label>:12                                   ; preds = %7, %6
  %13 = load i32* %1
  ret i32 %13
```

The above code shows the IR code for print_arg function. %1 stores pointer to the return value. %2 and %3 stores pointers to argc and argv[]. We check argc != 2 using `%5 = icmp ne i32 %4, 2`. If they're not equal, we branch to label 6 where the return value is initialised to -1. From there we branch to label 12 and return. Otherwise we branch to label 7 where the print statement takes place. We use getelementptr to find the address for argv[1] and load it's value in %10 (`%10 = load i8** %9, align 8`). We then call the print function using `%11 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i8* %10)`. The return value is initialised to 1 before branching to label 12 where the the return value is returned.

### 1.6.2 -O2

```
define i32 @print_arg(i32 %argc, i8** nocapture readonly %argv) #0
    {
  %1 = icmp eq i32 %argc, 2
  br i1 %1, label %2, label %6

; <label>:2                                    ; preds = %0
  %3 = getelementptr inbounds i8** %argv, i64 1
  %4 = load i8** %3, align 8, !tbaa !1
  %5 = tail call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
      ([3 x i8]* @.str, i64 0, i64 0), i8* %4) #2
  br label %6
```

```
; <label>:6                                    ; preds = %2, %0
  %.0 = phi i32 [ 0, %2 ], [ -1, %0 ]
  ret i32 %.0
```

The above code shows IR for print_arg using -O2 optimization. We compare argc with 2 in `%1 = icmp eq i32 %argc, 2`. If they're equal, we branch to label 2. Otherwise we branch to label 6 fomr where we return. In label 2, %3 stores the address for argv[1], %4 stores the value for argv[1]. Instead of calling printf firectly, we use tail call printf for tail call optimization. Finally we branch to label 6. In label 6, the final return value is selected using phi instruction depending on the previous module from which we entered here(i.e. 0 if we entered from label 2 or -1 if we entered from label 0).

## 2 x86 Assembly

Similar to llvm IR, it initialises stack space for variables and arguments. Tail call optimizations take place wherever possible and empty loops are discarded. For sumn, the sum is computed directly using closed formula. Return values may not be stored on stack and may be directly be stored in accumulator registers if possible.

### 2.1 emptyloop

#### 2.1.1 -O0

```
emptyloop:                              # @emptyloop
  .cfi_startproc
# BB#0:
  pushl %ebp
.Ltmp0:
  .cfi_def_cfa_offset 8
.Ltmp1:
  .cfi_offset %ebp, -8
  movl  %esp, %ebp
.Ltmp2:
  .cfi_def_cfa_register %ebp
  pushl %ebx
  pushl %esi
  subl  $64, %esp
.Ltmp3:
  .cfi_offset %esi, -16
```

```
.Ltmp4:
  .cfi_offset %ebx, -12
  movl  12(%ebp), %eax
  movl  8(%ebp), %ecx
  movl  %ecx, -12(%ebp)
  movl  %eax, -16(%ebp)
  movl  $0, -28(%ebp)
  movl  $2147483646, -32(%ebp) # imm = 0x7FFFFFFE
  cmpl  $2, -12(%ebp)
  jl  .LBB0_2
# BB#1:
  movl  -16(%ebp), %eax
  movl  4(%eax), %eax
  movl  %esp, %ecx
  movl  %eax, (%ecx)
  calll atoi
  movl  %eax, %ecx
  sarl  $31, %ecx
  movl  %eax, -32(%ebp)
  movl  %ecx, -28(%ebp)
.LBB0_2:
  movl  $0, -20(%ebp)
  movl  $0, -24(%ebp)
.LBB0_3:                           # =>This Inner Loop Header:
    Depth=1
  movl  -24(%ebp), %eax
  movl  -20(%ebp), %ecx
  movl  -32(%ebp), %edx
  movl  -28(%ebp), %esi
  subl  $344866, %edx        # imm = 0x54322
  setb  %bl
  movl  %eax, -36(%ebp)      # 4-byte Spill
  xorl  %eax, %eax
  testl %esi, %esi
  movb  %al, %bh
  movl  %edx, -40(%ebp)      # 4-byte Spill
  movl  %ecx, -44(%ebp)      # 4-byte Spill
  movb  %bh, -45(%ebp)       # 1-byte Spill
  movb  %bl, -46(%ebp)       # 1-byte Spill
  je  .LBB0_11
# BB#10:                           #   in Loop: Header=BB0_3
    Depth=1
  movb  -45(%ebp), %al       # 1-byte Reload
  movb %al, -46(%ebp)        # 1-byte Spill
.LBB0_11:                           #   in Loop: Header=BB0_3
    Depth=1
  movb -46(%ebp), %al        # 1-byte Reload
  testb %al, %al
  jne .LBB0_5
  jmp .LBB0_4
.LBB0_4:                            #   in Loop: Header=BB0_3
    Depth=1
  movl -32(%ebp), %eax
  movl -28(%ebp), %ecx
  movl %eax, -52(%ebp)       # 4-byte Spill
  movl %ecx, -56(%ebp)       # 4-byte Spill
  jmp .LBB0_6
.LBB0_5:                            #   in Loop: Header=BB0_3
    Depth=1
  xorl %eax, %eax
  movl $344865, %ecx         # imm = 0x54321
  movl %ecx, -52(%ebp)       # 4-byte Spill
  movl %eax, -56(%ebp)       # 4-byte Spill
  jmp .LBB0_6
.LBB0_6:                            #   in Loop: Header=BB0_3
    Depth=1
  movl -56(%ebp), %eax       # 4-byte Reload
  movl -52(%ebp), %ecx       # 4-byte Reload
  movl -36(%ebp), %edx       # 4-byte Reload
  subl %ecx, %edx
  setae %bl
  movl -44(%ebp), %ecx       # 4-byte Reload
  subl %eax, %ecx
  setae %bh
  movl %ecx, -60(%ebp)       # 4-byte Spill
  movl %edx, -64(%ebp)       # 4-byte Spill
  movb %bh, -65(%ebp)        # 1-byte Spill
  movb %bl, -66(%ebp)        # 1-byte Spill
  je  .LBB0_13
# BB#12:                            #   in Loop: Header=BB0_3
    Depth=1
  movb -65(%ebp), %al        # 1-byte Reload
  movb %al, -66(%ebp)        # 1-byte Spill
.LBB0_13:                           #   in Loop: Header=BB0_3
    Depth=1
  movb -66(%ebp), %al        # 1-byte Reload
  testb %al, %al
  jne .LBB0_9
  jmp .LBB0_7
.LBB0_7:                              #   in Loop: Header=BB0_3
    Depth=1
  jmp .LBB0_8
.LBB0_8:                              #   in Loop: Header=BB0_3
    Depth=1
  movl  -24(%ebp), %eax
  movl  -20(%ebp), %ecx
  addl  $1, %eax
  adcl  $0, %ecx
  movl  %eax, -24(%ebp)
  movl  %ecx, -20(%ebp)
  jmp .LBB0_3
.LBB0_9:
  xorl  %eax, %eax
  addl  $64, %esp
  popl  %esi
  popl  %ebx
  popl  %ebp
  retl
.Ltmp5:
  .size emptyloop, .Ltmp5-emptyloop
  .cfi_endproc
```

### 2.1.2 -O2

```
emptyloop:                            # @emptyloop
  .cfi_startproc
# BB#0:
  subl  $12, %esp
.Ltmp0:
  .cfi_def_cfa_offset 16
  cmpl  $2, 16(%esp)
  jl  .LBB0_2
# BB#1:
  movl  20(%esp), %eax
  movl  4(%eax), %eax
  movl  %eax, (%esp)
  movl  $10, 8(%esp)
  movl  $0, 4(%esp)
  calll strtol
.LBB0_2:
  xorl  %eax, %eax
  addl  $12, %esp
  retl
.Ltmp1:
  .size emptyloop, .Ltmp1-emptyloop
  .cfi_endproc
```

The above code shows assembly program for emptyloop. It subtracts 12 from esp and sets the offset as 16. It compares argc with 2 using `cmpl $2, 16(%esp)`. In case $argc < 2$, it branches to .LBB0_2. There is initialises the return value to 0 using `xorl %eax, %eax`(eax stores the return value ans xoring a register with itself makes it zero). It then restores the stack to it's original state using `addl $12, %esp` before finally returning. In case argc ¿= 2, it gets the address of argc and stores in eax (`movl 20(%esp), %eax`). It stores argc[1] in eax using `movl 4(%eax), %eax`. Then it stores this value in the stack(`movl %eax, (%esp)`) as well as the other parameters that'll be required by the strtol function. It then calls the function (for implementing atoi()).(`calll strtol`). Finally it returns from the function. In other words, the loop portion has been omitted here also.

## 2.2 fib

### 2.2.1 -O0

```
fib:                                    # @fib
  .cfi_startproc
# BB#0:
  pushl %ebp
.Ltmp0:
  .cfi_def_cfa_offset 8
.Ltmp1:
  .cfi_offset %ebp, -8
  movl  %esp, %ebp
```

It begins by pushing the current ebp on stack and defining offset. Then pushes the current esp in ebp to keep track the start of the current function stack.

```
.Ltmp2:
  .cfi_def_cfa_register %ebp
  subl  $24, %esp
  movl  8(%ebp), %eax
  movl  %eax, -8(%ebp)
  cmpl  $2, -8(%ebp)
  jge .LBB0_2
```

It compares n with 2 in line `cmpl $2, -8(%ebp)`. In case n ¿= 2, it jumps to .LBB0_2.

```
# BB#1:
  movl  $1, -4(%ebp)
  jmp .LBB0_3
```

If $n < 2$, it moves 1 to ebp - 4 (i.e. stores 1 as the return value on stack), and jumps to .LBB0_3.

```
.LBB0_2:
  movl  -8(%ebp), %eax
  subl  $1, %eax
  movl  %eax, (%esp)
  calll fib
  movl  -8(%ebp), %ecx
  subl  $2, %ecx
  movl  %ecx, (%esp)
  movl  %eax, -12(%ebp)       # 4-byte Spill
  calll fib
  movl  -12(%ebp), %ecx       # 4-byte Reload
  addl  %eax, %ecx
  movl  %ecx, -4(%ebp)
.LBB0_3:
  movl  -4(%ebp), %eax
  addl  $24, %esp
  popl  %ebp
  retl
.Ltmp3:
  .size fib, .Ltmp3-fib
  .cfi_endproc
```

In case n ¿= 2, it stores n in eax(`movl -8(%ebp), %eax`) and subtracts 1 from it(`subl $1, %eax`). It moves it to stack and calls fib(which will now take the topmost value of stack i.e. n-1 as it's argument). Similarly, it then stores n-2 in ecx and pushes it to stack. It then calls fib again(i.i. fib(n-2)). It adds the two results(`addl %eax, %ecx`) and moves it to ebp - 4(`movl %ecx,`

-4(%ebp)). This result is then stored in eax(which stores the return value) (movl -4(%ebp), %eax), stack is returned to it's original state and ebp is popped before returning.

The above code shows the assembly code for fib() function.

### 2.2.2 -O2

```
fib:                                    # @fib
  .cfi_startproc
# BB#0:
  pushl %edi
.Ltmp0:
  .cfi_def_cfa_offset 8
  pushl %esi
.Ltmp1:
  .cfi_def_cfa_offset 12
  pushl %eax
.Ltmp2:
  .cfi_def_cfa_offset 16
.Ltmp3:
  .cfi_offset %esi, -12
.Ltmp4:
  .cfi_offset %edi, -8
  movl  16(%esp), %edi
  movl  $1, %esi
  cmpl  $2, %edi
  jl  .LBB0_2
  .align  16, 0x90
.LBB0_1:                                # %tailrecurse
                                        # =>This Inner Loop Header:
                                           Depth=1
  leal  -1(%edi), %eax
  movl  %eax, (%esp)
  calll fib
  addl  $-2, %edi
  addl  %eax, %esi
  cmpl  $1, %edi
  jg  .LBB0_1
.LBB0_2:                                # %tailrecurse._crit_edge
  movl  %esi, %eax
  addl  $4, %esp
  popl  %esi
  popl  %edi
  retl
.Ltmp5:
  .size fib, .Ltmp5-fib
  .cfi_endproc
```

Above code shows assembly code for fib() using -O2 flag. After initialising stack and setting offsets, it moves n to edi (`movl 16(%esp), %edi`) and 1 to edi(which stores the return value in this case). It then compares n with 2(`cmpl $2, %edi`). In case $n < 2$, it jumps to .LBB0_2. Else, it stores n - 1 on stack and calls fib. Result is stored in eax(i.e. eax contains fib(n-1)). n becomes n - 2 (`addl $-2, %edi`). Return value is updated (`addl %eax, %esi`). It compares n - 2 with 1). In case $n-2 > 1$ (or $n-2 >= 2$), it jumps back to the start of .LBB0_1 to compute fib(n-2) (which it'll accumulate in esi). Finally, the return value is moved to eax in .LBB0_2 (`movl %esi, %eax`). The stack is returned to it's former state before returning. In this case also we see tail recursion as it doesn't call fib(n-2) recursively, instead jumps back to an inner tail recursive loop.

## 2.3 fibo_iter

### 2.3.1 -O0

```
fibo_iter:                                # @fibo_iter
  .cfi_startproc
# BB#0:
  pushl %ebp
.Ltmp0:
  .cfi_def_cfa_offset 8
.Ltmp1:
  .cfi_offset %ebp, -8
  movl %esp, %ebp
.Ltmp2:
  .cfi_def_cfa_register %ebp
  pushl %esi
  subl $44, %esp
.Ltmp3:
  .cfi_offset %esi, -12
  movl 8(%ebp), %eax
  movl %eax, -20(%ebp)
  cmpl $3, -20(%ebp)
  jae .LBB0_2
# BB#1:
  movl  $0, -12(%ebp)
  movl  $1, -16(%ebp)
  jmp .LBB0_7
.LBB0_2:
  movl  $0, -28(%ebp)
  movl  $1, -32(%ebp)
  movl  $0, -36(%ebp)
  movl  $1, -40(%ebp)
  movl  $3, -44(%ebp)
.LBB0_3:                                # =>This Inner Loop Header:
    Depth=1
  movl  -44(%ebp), %eax
  cmpl  -20(%ebp), %eax
  ja  .LBB0_6
# BB#4:                                 #   in Loop: Header=BB0_3
    Depth=1
  movl  -32(%ebp), %eax
  movl  %eax, -48(%ebp)
  movl  -40(%ebp), %eax
  movl  -36(%ebp), %ecx
  movl  -32(%ebp), %edx
  movl  -28(%ebp), %esi
  addl  %eax, %edx
  adcl  %ecx, %esi
  movl  %edx, -32(%ebp)
  movl  %esi, -28(%ebp)
  movl  -48(%ebp), %eax
  movl  %eax, -40(%ebp)
  movl  $0, -36(%ebp)
# BB#5:                                 #   in Loop: Header=BB0_3
    Depth=1
  movl  -44(%ebp), %eax
  addl  $1, %eax
  movl  %eax, -44(%ebp)
  jmp .LBB0_3
.LBB0_6:
  vmovsd -32(%ebp), %xmm0
  vmovsd %xmm0, -16(%ebp)
.LBB0_7:
  movl  -16(%ebp), %eax
  movl  -12(%ebp), %edx
  addl  $44, %esp
  popl  %esi
  popl  %ebp
  retl
.Ltmp4:
  .size fibo_iter, .Ltmp4-fibo_iter
  .cfi_endproc
```

The above code shows the assembly for fibo_iter() function.

### 2.3.2 -O2

```
fibo_iter:                           # @fibo_iter
  .cfi_startproc
# BB#0:
  pushl %ebp
.Ltmp0:
  .cfi_def_cfa_offset 8
  pushl %ebx
.Ltmp1:
  .cfi_def_cfa_offset 12
  pushl %edi
.Ltmp2:
  .cfi_def_cfa_offset 16
  pushl %esi
.Ltmp3:
  .cfi_def_cfa_offset 20
.Ltmp4:
  .cfi_offset %esi, -20
.Ltmp5:
  .cfi_offset %edi, -16
.Ltmp6:
  .cfi_offset %ebx, -12
.Ltmp7:
  .cfi_offset %ebp, -8
  movl  20(%esp), %ecx
  xorl  %esi, %esi
  movl  $1, %ebx
  cmpl  $3, %ecx
  jae .LBB0_2
# BB#1:
  movl  $1, %eax
  xorl  %edx, %edx
  jmp .LBB0_4
.LBB0_2:
  movl  $3, %edi
  movl  $1, %ebp
  xorl  %edx, %edx
  .align  16, 0x90
.LBB0_3:                              # %.lr.ph
                                      # =>This Inner Loop Header:
                                          Depth=1
  movl  %ebx, %eax
  addl  %ebp, %eax
  adcl  %esi, %edx
  addl  $1, %edi
  movl  %ebp, %ebx
  movl  %eax, %ebp
  cmpl  %ecx, %edi
  jbe .LBB0_3
.LBB0_4:                              # %.loopexit
  popl  %esi
  popl  %edi
  popl  %ebx
  popl  %ebp
  retl
.Ltmp8:
  .size fibo_iter, .Ltmp8-fibo_iter
  .cfi_endproc
```

The above code shows the assembly for fibo_iter() using -O2 flag.

## 2.4 print_arg

### 2.4.1 -O0

```
print_arg:                            # @print_arg
  .cfi_startproc
# BB#0:
  pushl %ebp
.Ltmp0:
  .cfi_def_cfa_offset 8
.Ltmp1:
  .cfi_offset %ebp, -8
  movl  %esp, %ebp
.Ltmp2:
  .cfi_def_cfa_register %ebp
  subl  $40, %esp
  movl  12(%ebp), %eax
  movl  8(%ebp), %ecx
  movl  %ecx, -8(%ebp)
  movl  %eax, -16(%ebp)
  cmpl  $2, -8(%ebp)
  je  .LBB0_2
# BB#1:
  movl  $-1, -4(%ebp)
  jmp .LBB0_3
.LBB0_2:
  leal  .L.str, %eax
  movl  -16(%ebp), %ecx
  movl  4(%ecx), %ecx
  movl  %eax, (%esp)
  movl  %ecx, 4(%esp)
  calll printf
  movl  $0, -4(%ebp)
  movl  %eax, -20(%ebp)       # 4-byte Spill
.LBB0_3:
  movl  -4(%ebp), %eax
  addl  $40, %esp
  popl  %ebp
  retl
.Ltmp3:
  .size print_arg, .Ltmp3-print_arg
  .cfi_endproc
```

Above code shows x86 assembly for print_arg() function. After moving the variables to stack and initialising it, it compares n with 2(`cmpl $2, -8(%ebp)`). If equal, it jumps to .LBB0_3 (where it restores the stack and sets the return value in eax before returning). Otherwise, it moves %̈s̈to eax, argv[1] to ecx, moves both of them to the top of stack using `movl %eax, (%esp)` and `movl %ecx, 4(%esp)` and calls the printf function. It changes the return value to 0(`movl $0, -4(%ebp)`).Finally it returns as before (changing the stack to it's original state etc.).

### 2.4.2 -O2

```
print_arg:                            # @print_arg
  .cfi_startproc
# BB#0:
  subl  $12, %esp
```

```
.Ltmp0:
  .cfi_def_cfa_offset 16
  movl  $-1, %eax
  cmpl  $2, 16(%esp)
  jne .LBB0_2
# BB#1:
  movl  20(%esp), %eax
  movl  4(%eax), %eax
  movl  %eax, 4(%esp)
  movl  $.L.str, (%esp)
  calll printf
  xorl  %eax, %eax
.LBB0_2:
  addl  $12, %esp
  retl
.Ltmp1:
  .size print_arg, .Ltmp1-print_arg
  .cfi_endproc
```

Above code shows x86 assembly for print_arg() function using -02 flag. After initialising stack and setting offsets, it stores -1 as the return value in eax(`movl $-1, %eax`). It compares 2 to argc in line `cmpl $2, 16(%esp)`. In case 2 != argc, it jumps to .LBB0_2(where it simply restores the stack and returns).If argc == 2, it stores value of argv[0] in eax(`movl 20(%esp), %eax`), then stores value of argv[1] in eax(`movl 4(%eax), %eax`). It moves this value to the top of the stack. Then moves .L.str to the top of the stack(.L.str contains ¨%s¨). It then calls the printf function. Finally it makes the return value as 0 (`xorl %eax, %eax`) and then restores the stack to it's original state before returning.

## 3 Performance comparison across optimization levels

The following graphs show comparison between different optimization levels in llvm bitcode -

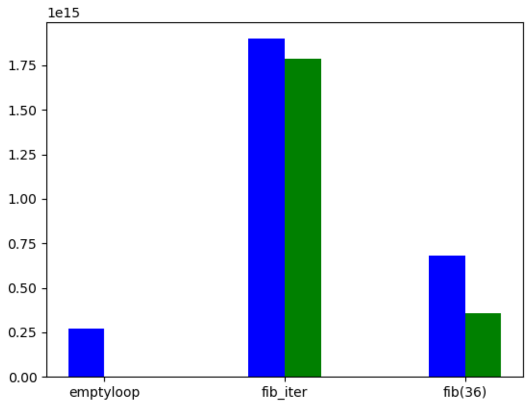

Figure 1: Comparison between different llvm optimizations

We see that -O3 performs slightly better than it's -O2 counterpart.

The following graph show comparison between different optimization levels in x86 assembly code -

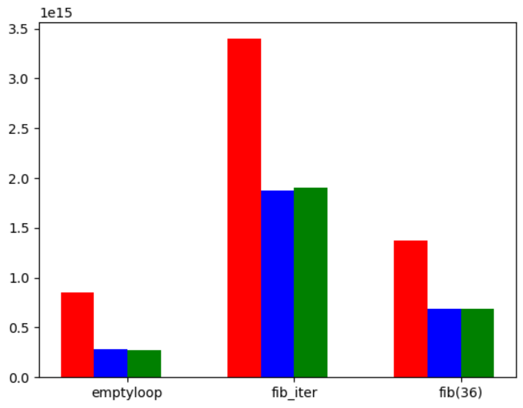

Figure 2: Comparison between different x86 assembly optimizations

We see that there's not uch difference between -O2 and -O3 levels with -O2 sometimes performing better than -O3 (e.g. in fib_iter). However, there's a huge difference between -O0 and -O2/-O3.