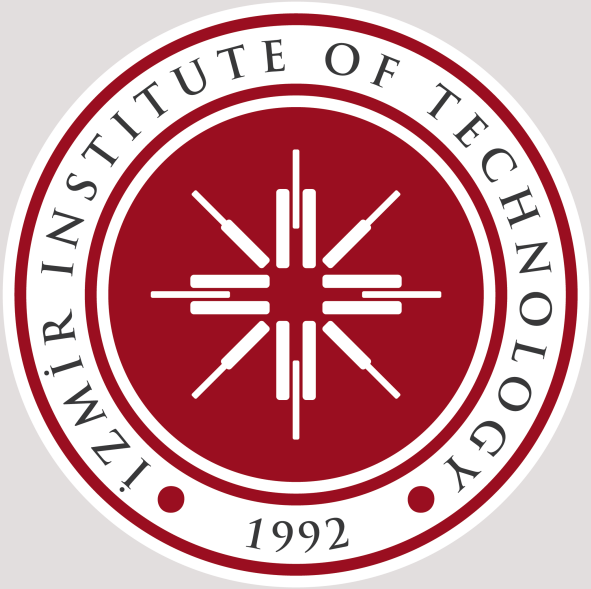
Izmir Institute of Technology
Computer Engineering Department
CENG513 Midterm Exam Spring 2024
Question 1

Student Name: Gökay Gülsoy Student No: 270201072

April 19, 2024



Question 1

a-) My hand-coded LLVM IR for the `prime_number.c` source code is as follows:

```
; ModuleID = 'is_prime'
source_filename = "prime_number.c"

define i32 @is_prime(i32 %n) {
entry:
    %cmp1 = icmp sle i32 %n, 1 ; check if n is <= 1
    br i1 %cmp1, label %not_prime, label %loop_entrance

loop_entrance:
    %i = add i32 0, 2  ; initialize i as 2
    br label %loop_condition

loop_condition:
    %is_less_than_n = icmp slt i32 %i, %n ; compare if i is less than n
    br i1 %is_less_than_n, label %loop_body, label %prime

loop_body:
    %remainder = srem i32 %n, %i ; check if n is divisible by i
    %is_divisible = icmp eq i32 %remainder, 0
    br i1 %is_divisible, label %not_prime, label %incr_i

incr_i:
    %i = add i32 %i, 1 ; increment i
    br label %loop_condition

not_prime:
    ret i32 0 ; if number is not prime return 0

prime:
    ret i32 1 ; if number is prime return 1
}
```

Figure 1: is_prime_hand_v1.ll

Screenshot for the error related to SSA restriction is as follows:

```
bash MEM: 28.03% | 5/15GB 24ms
16:51 | Question1
opt -S -verify-each is_prime_hand_v1.ll
opt: is_prime_hand_v1.ll:24:5: error: multiple definition of local value named 'i'
    %i = add i32 %i, 1
    ^
```

Figure 2: is_prime_hand_v1.ll SSA error

b-) Fixed LLVM IR by using phi function is as follows:

```
; ModuleID = 'is_prime_hand_phi.ll'
source_filename = "prime_number.c"

define i32 @is_prime(i32 %n) {
entrance:
  %is_prime = icmp sle i32 %n, 1 ; if n is less than or equal to 1 n is not prime
  br i1 %is_prime, label %not_prime, label %for_body

for_body:
  %i = phi i32 [ 2, %entrance ], [ %next_i, %loop_inc ] ; phi function
  %next_i = add i32 %i, 1
  %remainder = srem i32 %n, %i
  %is_divisible = icmp eq i32 %remainder, 0
  br i1 %is_divisible, label %not_prime, label %loop_inc

loop_inc:
  %is_prime = icmp ne i32 %next_i, %n
  br i1 %is_prime, label %for_body, label %prime

not_prime:
  ret i32 0 ; return 0 if n is not prime

prime:
  ret i32 1 ; return 1 if n is prime
}
```

Figure 3: is_prime_hand_phi.ll

Screenshot for the LLVM IR opt command output using phi function without error is as follows:

executed command: **opt -S -verify-each is_prime_hand_phi.ll**

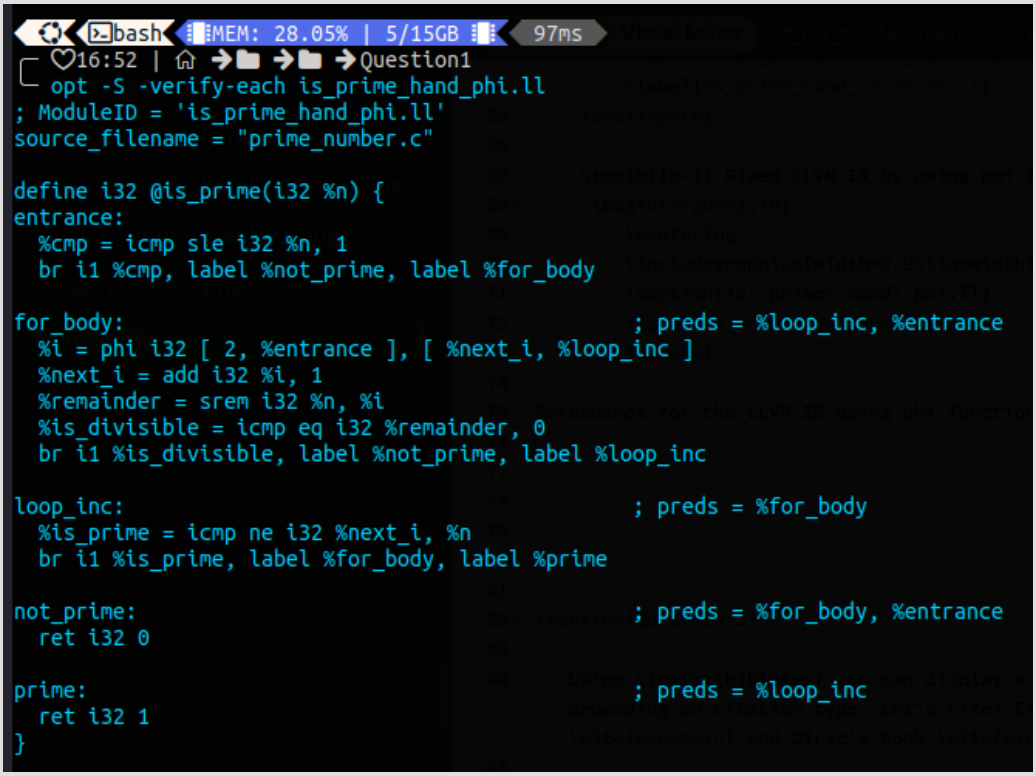


Figure 4: opt command output for is_prime_hand_phi.ll

Fixed LLVM IR by using alloca function is as follows:

```
; ModuleID = 'is_prime_hand_alloca.ll'
source_filename = "prime_number.c"

define i32 @is_prime(i32 %n) {
entrance:
  %is_prime = icmp sle i32 %n, 1 ; if n is less than or equal to 1 n is not prime
  br i1 %is_prime, label %not_prime, label %initialize

initialize:
  %i_addr = alloca i32   ; allocate memory for i
  store i32 2, i32* %i_addr ;store the initial value of i in allocated address
  br label %loop_cond

loop_cond:
  %i = load i32, i32* %i_addr ; reload the value at memory pointed by i_addr
  %is_loop_end = icmp sle i32 %i, %n
  br i1 %is_loop_end, label %for_body, label %prime

for_body:
  %remainder = srem i32 %n, %i
  %is_divisible = icmp eq i32 %remainder, 0
  br i1 %is_divisible, label %not_prime, label %loop_inc

loop_inc:
  %i_new = add i32 %i, 1 ; create a variable i_new to hold the new value
  store i32 %i_new, i32* %i_addr ; store the new value of i at memory pointed by i_addr
  br label %loop_cond

not_prime:
  ret i32 0


prime:
  ret i32 1
}
```

Figure 5: is_prime_hand_alloca.ll

Screenshot for the LLVM IR opt command output using alloca function without error is as follows:

executed command: **opt -S -verify-each is_prime_hand_alloca.ll**

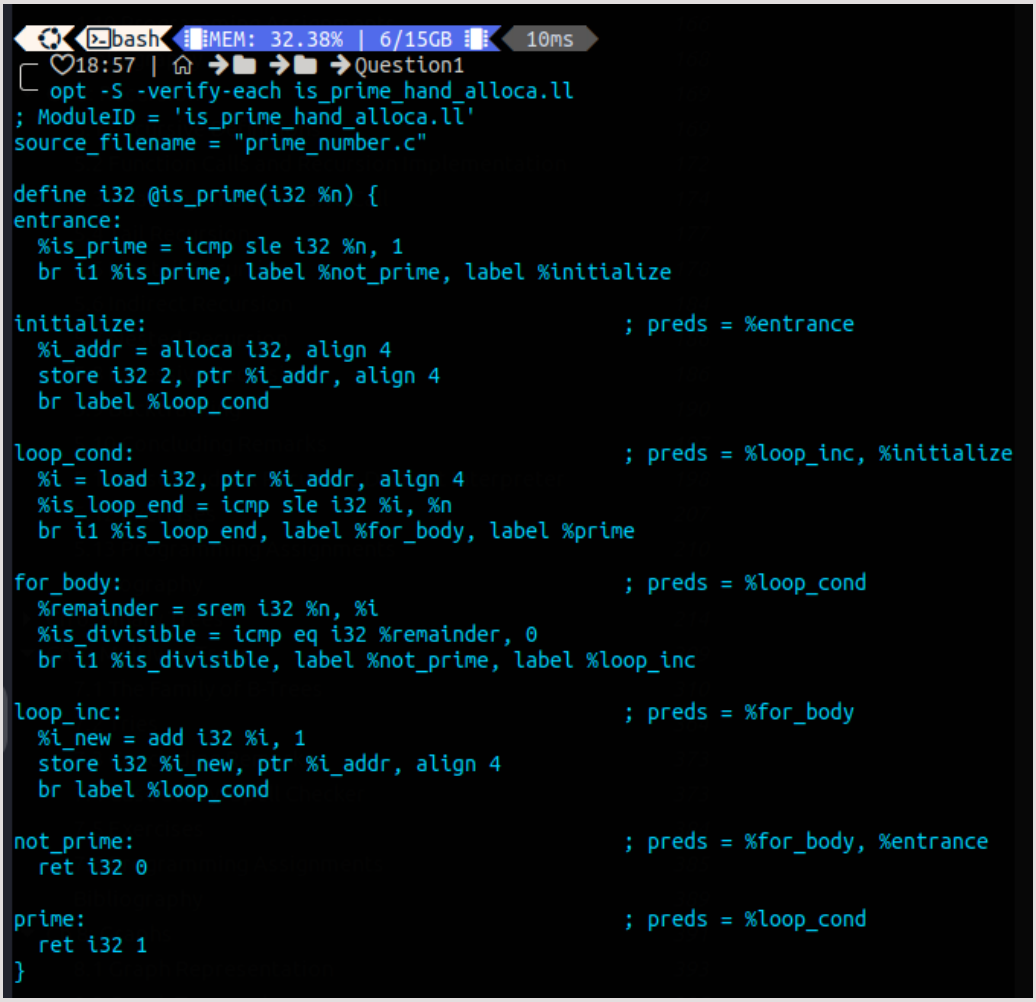


Figure 6: opt command output for is_prime_hand_alloca.ll

Part of LLVM IR generated with clang -emit-llvm for `is_prime_gen_O0.ll` is as follows:

executed command: **clang -S -emit-llvm -O0 prime_number.c -o is_prime_gen_O0.ll**

```
; ModuleID = 'prime_number.c'
source_filename = "prime_number.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @is_prime(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4
  %5 = load i32, ptr %3, align 4
  %6 = icmp sle i32 %5, 1
  br i1 %6, label %7, label %8

7:                                                ; preds = %1
  store i32 0, ptr %2, align 4
  br label %24

8:                                                ; preds = %1
  store i32 2, ptr %4, align 4
  br label %9

9:                                                ; preds = %20, %8
  %10 = load i32, ptr %4, align 4
  %11 = load i32, ptr %3, align 4
  %12 = icmp slt i32 %10, %11
  br i1 %12, label %13, label %23

13:                                               ; preds = %9
  %14 = load i32, ptr %3, align 4
  %15 = load i32, ptr %4, align 4
  %16 = srem i32 %14, %15
  %17 = icmp eq i32 %16, 0
  br i1 %17, label %18, label %19

18:                                               ; preds = %13
  store i32 0, ptr %2, align 4
  br label %24

19:                                               ; preds = %13
  br label %20

20:                                               ; preds = %19
  %21 = load i32, ptr %4, align 4
  %22 = add nsw i32 %21, 1
  store i32 %22, ptr %4, align 4
  br label %9, !llvm.loop !6

23:                                               ; preds = %9
  store i32 1, ptr %2, align 4
  br label %24
```

Figure 7: is_prime_genO0.ll

Part of LLVM IR generated with clang -emit-llvm for `is_prime_gen_O1.ll` is as follows:

executed command: **clang -S -emit-llvm -O1 prime_number.c -o is_prime_gen_O1.ll**

```
; ModuleID = 'prime_number.c'
source_filename = "prime_number.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: nofree norecurse nosync nounwind memory(none) uwtable
define dso_local i32 @is_prime(i32 noundef %0) local_unnamed_addr #0 {
  %2 = icmp slt i32 %0, 2
  br i1 %2, label %19, label %3

3:                                                ; preds = %1
  %4 = icmp eq i32 %0, 2
  %5 = and i32 %0, 1
  %6 = icmp eq i32 %5, 0
  br i1 %6, label %16, label %7

7:                                                ; preds = %3, %11
  %8 = phi i32 [ %9, %11 ], [ 2, %3 ]
  %9 = add nuw nsw i32 %8, 1
  %10 = icmp eq i32 %9, %0
  br i1 %10, label %14, label %11, !llvm.loop !5

11:                                               ; preds = %7
  %12 = urem i32 %0, %9
  %13 = icmp eq i32 %12, 0
  br i1 %13, label %14, label %7, !llvm.loop !5

14:                                               ; preds = %7, %11
  %15 = icmp sge i32 %9, %0
  br label %16

16:                                               ; preds = %14, %3
  %17 = phi i1 [ %4, %3 ], [ %15, %14 ]
  %18 = zext i1 %17 to i32
  br label %19

19:                                               ; preds = %16, %1
  %20 = phi i32 [ 0, %1 ], [ %18, %16 ]
  ret i32 %20
}
```

Figure 8: is_prime_genO1.ll

Commands to generate the .svg files are as follows:

opt -passes="dot-cfg" `is_prime_hand_phi.ll` **then** dot -Tsvg `is_prime.dot` -o `is_prime_hand_phi.svg`

opt -passes="dot-cfg" `is_prime_hand_alloca.ll` **then** dot -Tsvg `is_prime.dot` -o `is_prime_hand_alloca.svg`

opt -passes="dot-cfg" `is_prime_gen_O0.ll` **then** dot -Tsvg `is_prime.dot` -o `is_prime_gen_O0.svg`

opt –passes="dot-cfg" `is_prime_gen_O1.ll` **then** dot -Tsvg `is_prime.dot` -o `is_prime_gen_O1.svg`

When I compare the CFGs and IRs which are generated by clang and wirtten by myself following are the main similarities and differences:

`is_prime_gen_O0.ll` which is generated by clang with the command clang -S -emit-llvm -00 is using alloca functions, loads and stores. In that way it resembles to my hand-coded IR `is_prime_hand_alloca.ll` because it is generated at lowest optimization level (-O0) that is the reason it contains a lot of load and store operations, but it contains extra fields in the header part and at the end of the file defining architectural details of the computer system in which IR is generated. `is_prime_gen_O1.ll` file generated by clang with the command clang -S -emit-llvm -O1 is using phi functions instead of alloca functions loads, and stores different from `is_prime_gen_O0.ll` because it was generated at optimization level (-O1). In that respect `is_prime_gen_O1.ll` is similar to my hand-coded `is_prime_hand_phi.ll` which also uses phi function. When I compare the CFG for `is_prime_gen_O0.ll` with my hand-coded `is_prime_hand_alloca.ll` IR, clang generated version consists of more labels and instructions than my hand-coded version, so I can say that CFG for clang generated `is_prime_gen_O0.ll` IR is more complex than my hand-coded `is_prime_hand_alloca.ll` version. When I compare the CFG for `is_prime_gen_O1.ll` with my hand-coded `is_prime_hand_phi.ll` IR, due to optimization level set to (-O1) complexity of the CFG produced for the `is_prime_gen_O1.ll` is reduced even though it contains some different instructions compared to my-handcoded version.