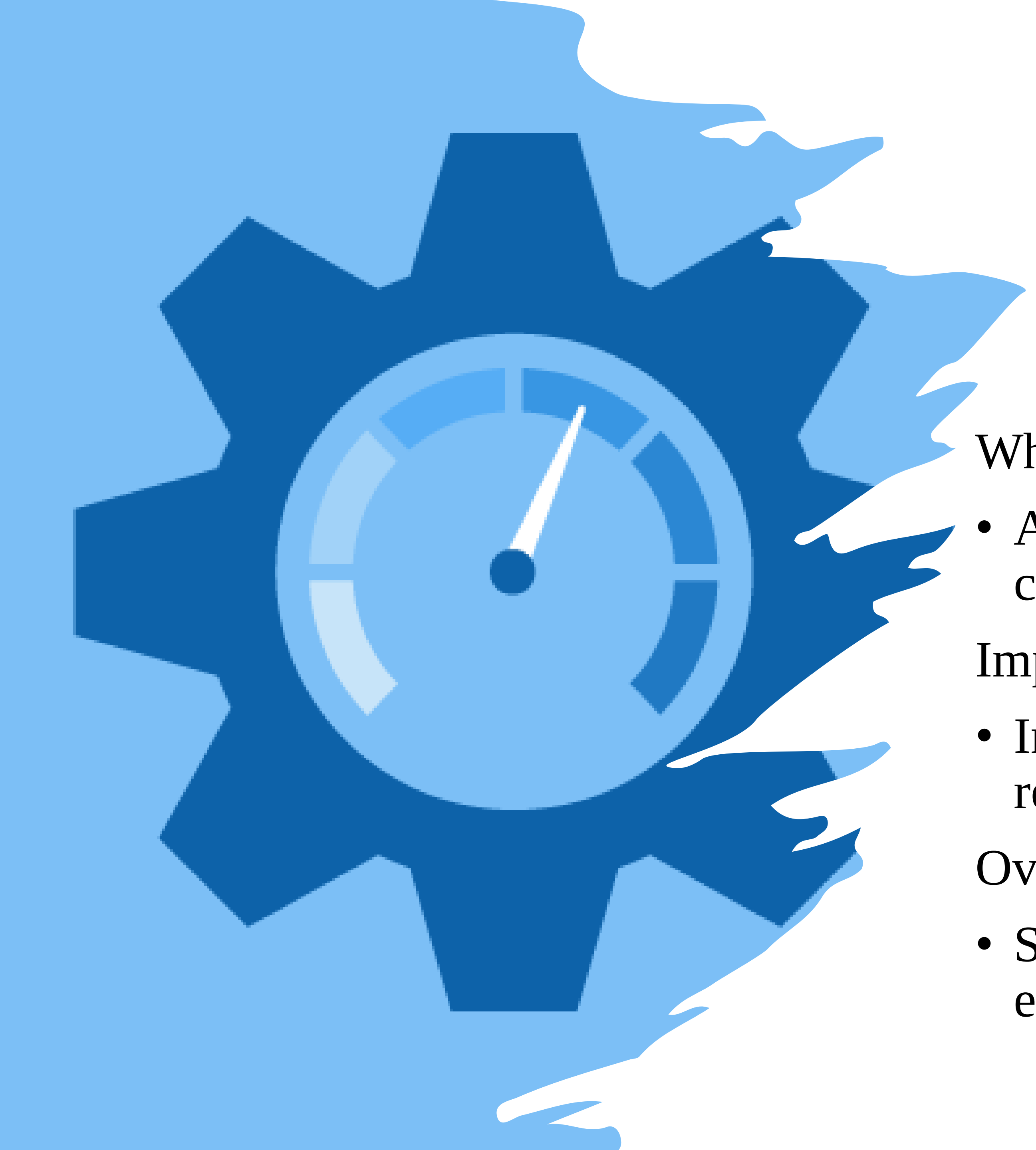
UNIVERSITY OF BELGRADE,
SERBIA
FACULTY OF MATHEMATICS

LLVM Algebraic Optimization Passes

Luka Filipović 80/2020

Sara Kalinić 387/2021

Construction of Compilers



Introduction

What is LLVM?

- A compiler infrastructure for building compilers.

Importance of optimization in compilation?

- Improves runtime performance and reduces resource usage.

Overview of algebraic optimizations:

- Simplifying mathematical expressions to enhance efficiency

Overview of Optimization Passes

- A technique to improve the performance of code during compilation.

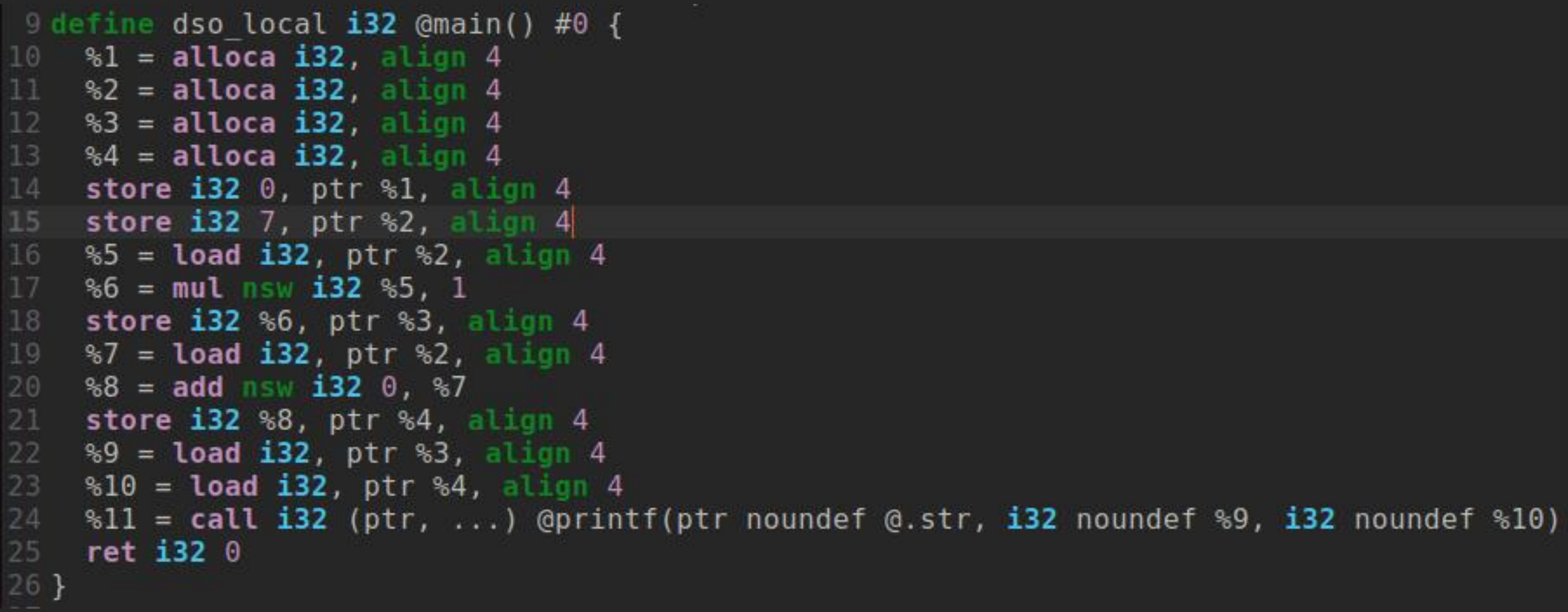
a. Add Mul Pass

b. Common Subexpression Pass

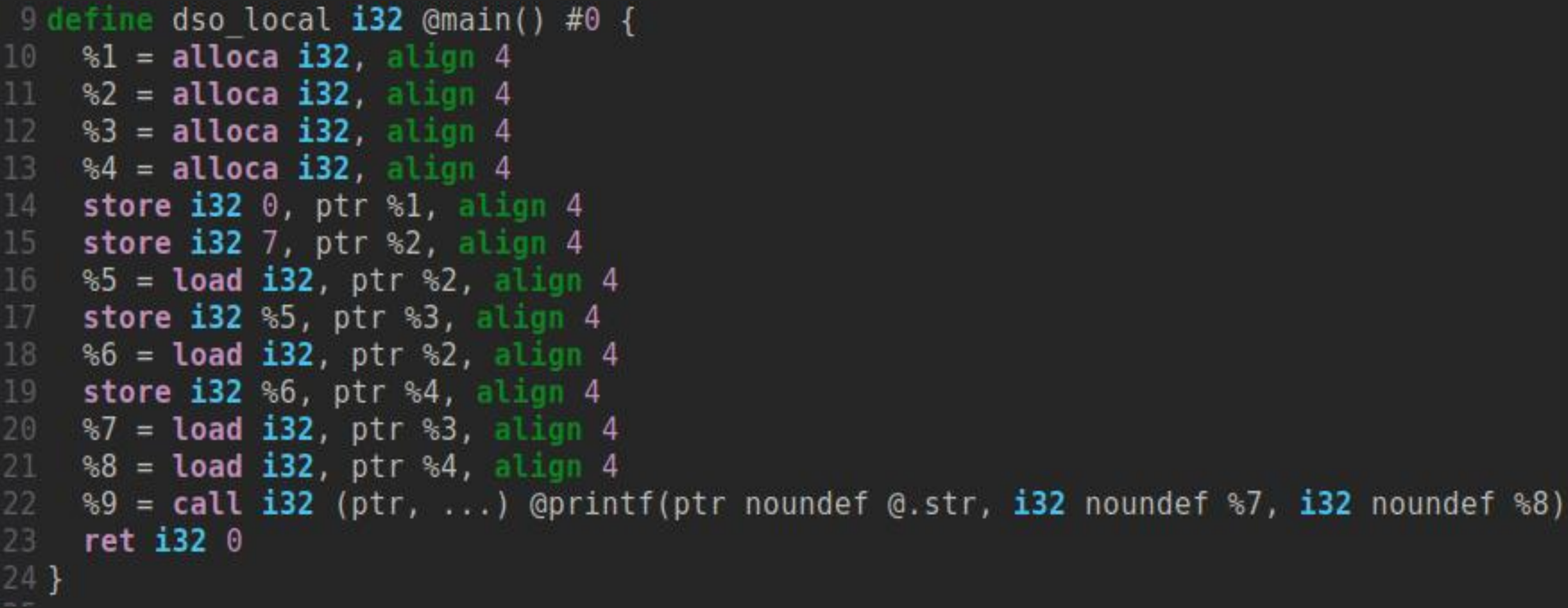
c. Power Optimization Pass

Add Mul Pass

- Simplifies expressions involving addition and multiplication.
- Examples
  - $x + 0 = x$
  - $x * 1 = x$
- Benefits:
  - Reduces unnecessary calculations.
  - Improves overall performance.



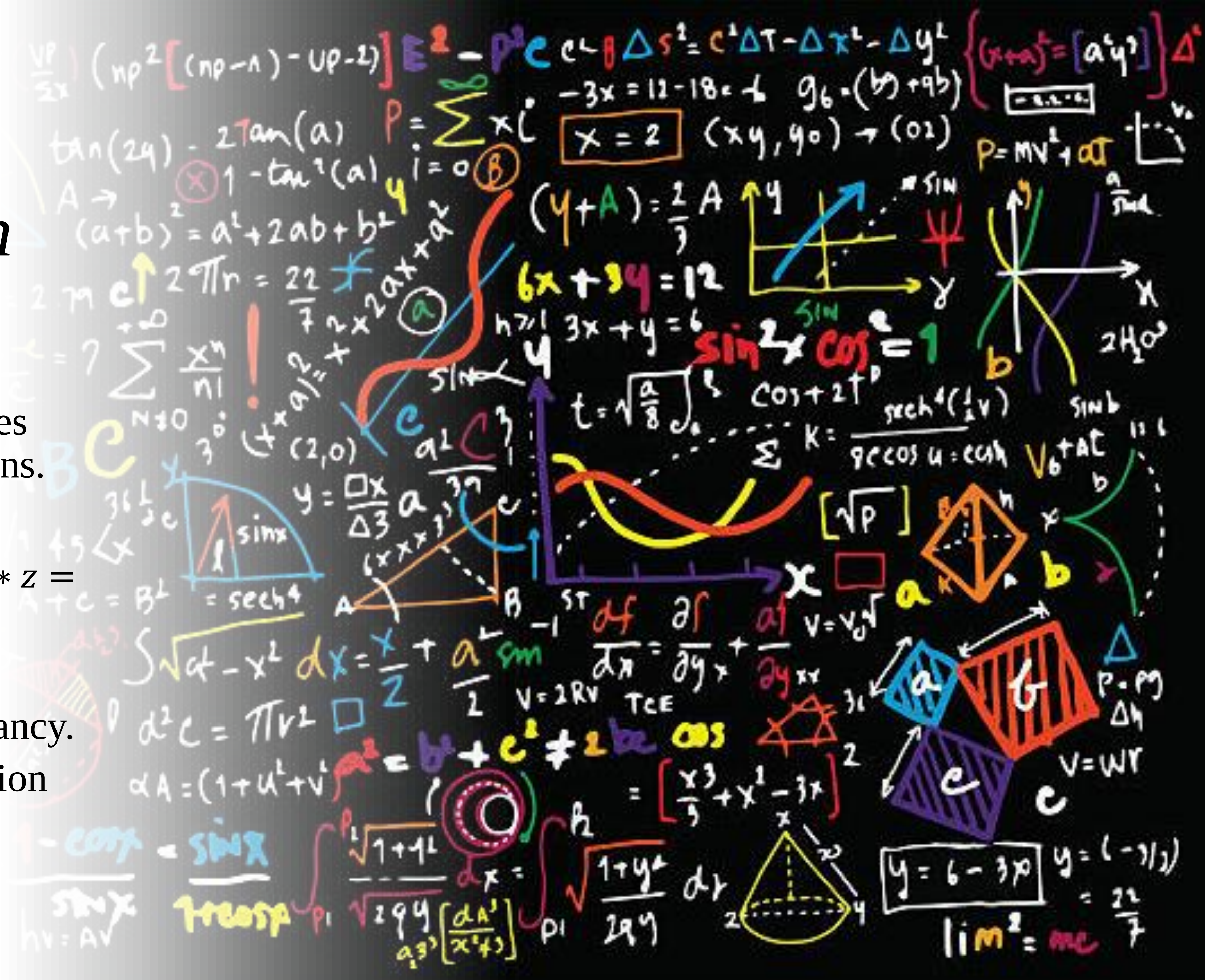
```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 7, ptr %2, align 4
  %5 = load i32, ptr %2, align 4
  %6 = mul nsw i32 %5, 1
  store i32 %6, ptr %3, align 4
  %7 = load i32, ptr %2, align 4
  %8 = add nsw i32 0, %7
  store i32 %8, ptr %4, align 4
  %9 = load i32, ptr %3, align 4
  %10 = load i32, ptr %4, align 4
  %11 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %9, i32 noundef %10)
  ret i32 0
}
```

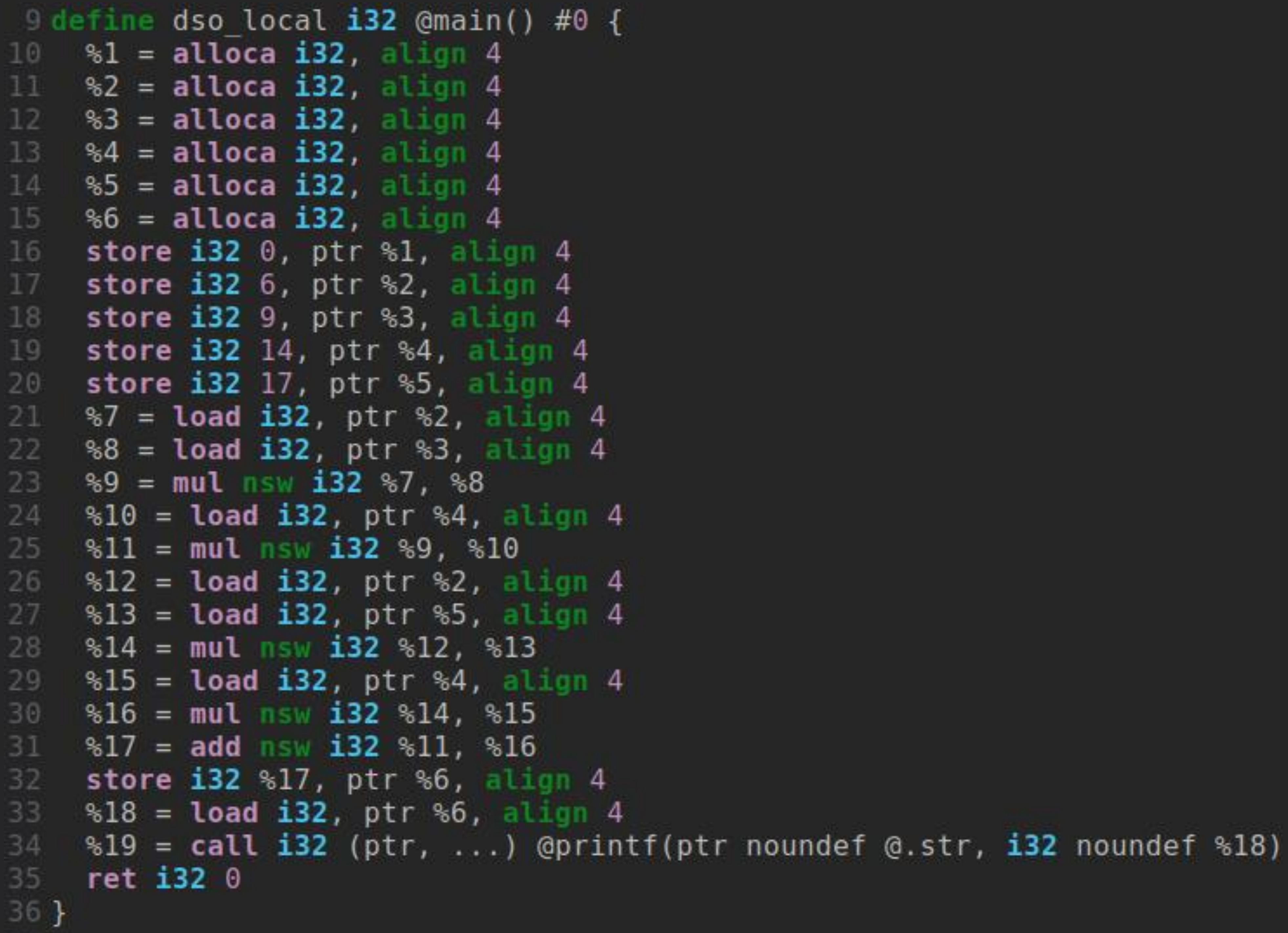


```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 7, ptr %2, align 4
  %5 = load i32, ptr %2, align 4
  store i32 %5, ptr %3, align 4
  %6 = load i32, ptr %2, align 4
  store i32 %6, ptr %4, align 4
  %7 = load i32, ptr %3, align 4
  %8 = load i32, ptr %4, align 4
  %9 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %7, i32 noundef %8)
  ret i32 0
}
```

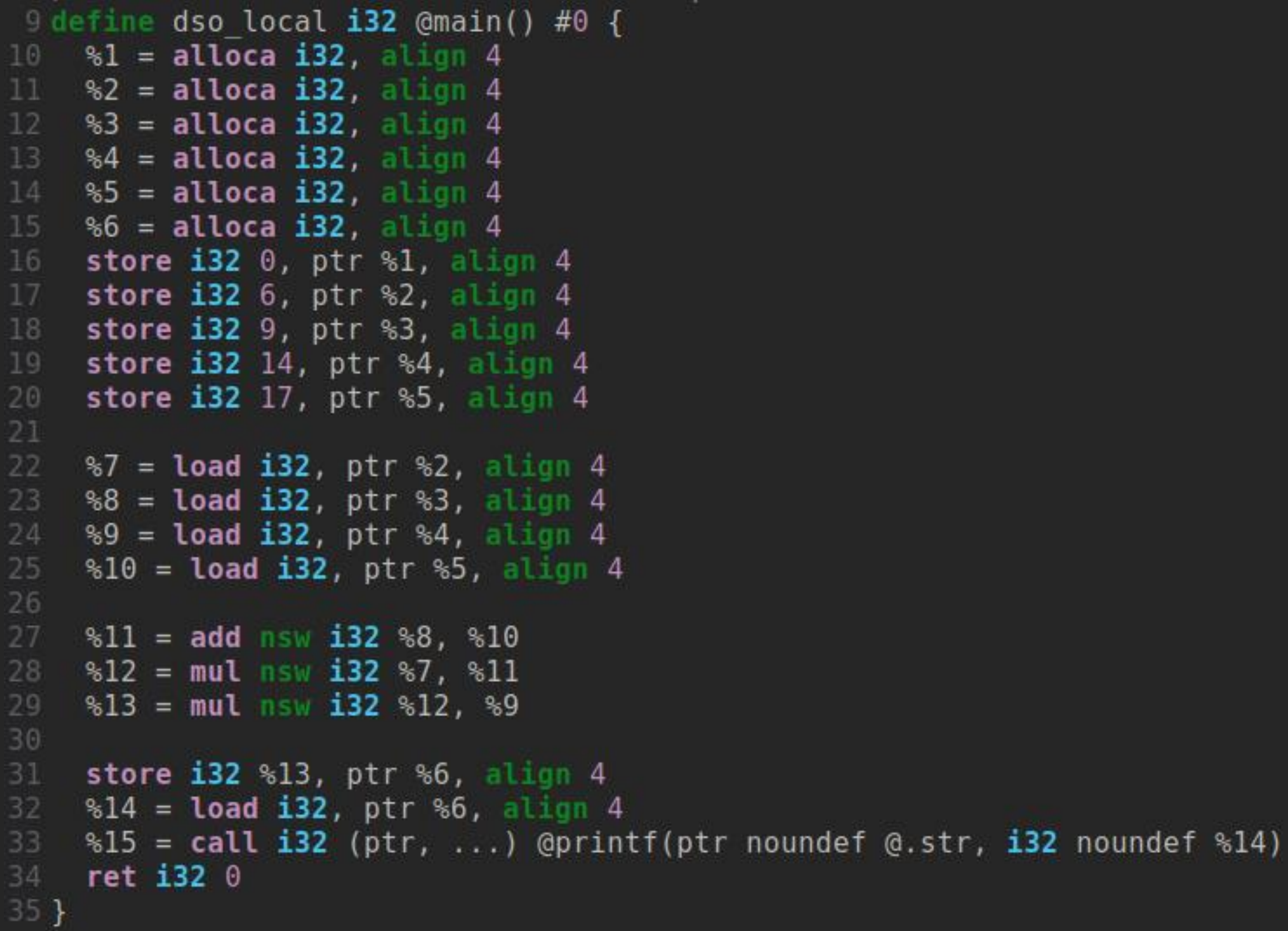
Common Subexpression Pass

- Identifies and factorizes common subexpressions.
- Example
  - $x * y * z + x * a * z = x * z * (y + a)$
- Benefits
  - Decreases redundancy.
  - Enhances calculation efficiency.





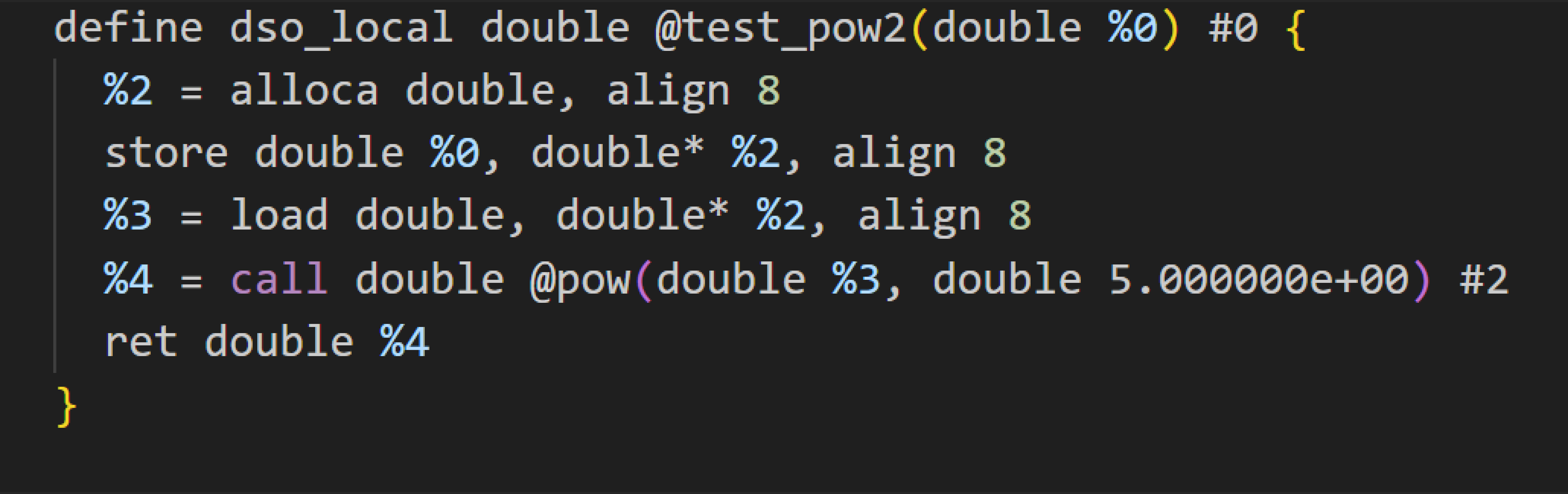
```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 6, ptr %2, align 4
  store i32 9, ptr %3, align 4
  store i32 14, ptr %4, align 4
  store i32 17, ptr %5, align 4
  %7 = load i32, ptr %2, align 4
  %8 = load i32, ptr %3, align 4
  %9 = mul nsw i32 %7, %8
  %10 = load i32, ptr %4, align 4
  %11 = mul nsw i32 %9, %10
  %12 = load i32, ptr %2, align 4
  %13 = load i32, ptr %5, align 4
  %14 = mul nsw i32 %12, %13
  %15 = load i32, ptr %4, align 4
  %16 = mul nsw i32 %14, %15
  %17 = add nsw i32 %11, %16
  store i32 %17, ptr %6, align 4
  %18 = load i32, ptr %6, align 4
  %19 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %18)
  ret i32 0
}
```



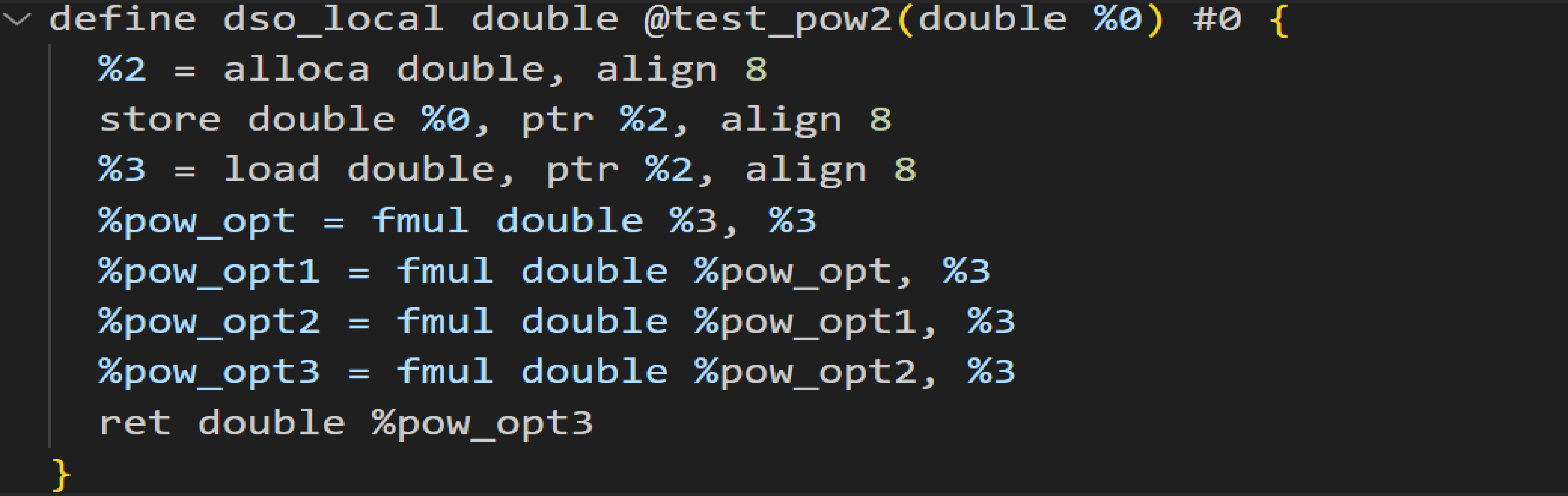
```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 6, ptr %2, align 4
  store i32 9, ptr %3, align 4
  store i32 14, ptr %4, align 4
  store i32 17, ptr %5, align 4

  %7 = load i32, ptr %2, align 4
  %8 = load i32, ptr %3, align 4
  %9 = load i32, ptr %4, align 4
  %10 = load i32, ptr %5, align 4

  %11 = add nsw i32 %8, %10
  %12 = mul nsw i32 %7, %11
  %13 = mul nsw i32 %12, %9

  store i32 %13, ptr %6, align 4
  %14 = load i32, ptr %6, align 4
  %15 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %14)
  ret i32 0
}
```

Power Optimization Pass

- Transforms exponentiation into multiplication.
- Examples
  - $x^4 = x * x * x * x$
- Benefits:
  - Improves efficiency for specific operations.
  - Reduces computational overhead.



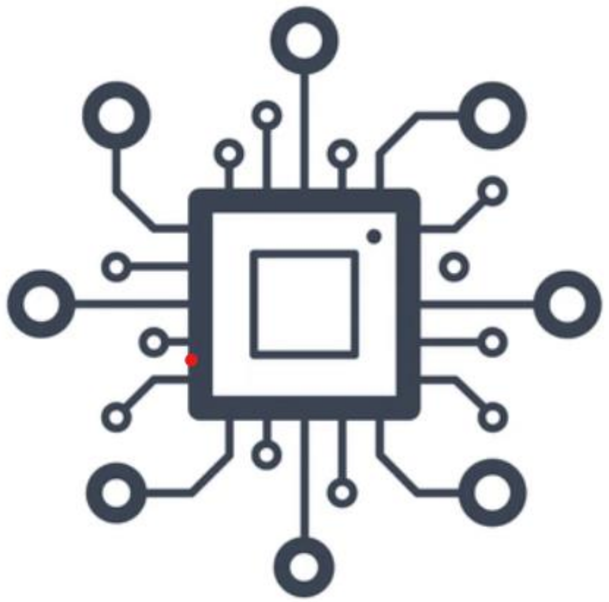
```
define dso_local double @test_pow2(double %0) #0 {
  %2 = alloca double, align 8
  store double %0, double* %2, align 8
  %3 = load double, double* %2, align 8
  %4 = call double @pow(double %3, double 5.000000e+00) #2
  ret double %4
}
```



```
define dso_local double @test_pow2(double %0) #0 {
  %2 = alloca double, align 8
  store double %0, ptr %2, align 8
  %3 = load double, ptr %2, align 8
  %pow_opt = fmul double %3, %3
  %pow_opt1 = fmul double %pow_opt, %3
  %pow_opt2 = fmul double %pow_opt1, %3
  %pow_opt3 = fmul double %pow_opt2, %3
  ret double %pow_opt3
}
```

Conclusion

- Algebraic optimizations are essential in modern compiler design as they help improve runtime performance and resource utilization.
- Each of these passes plays a crucial role in simplifying expressions, reducing computational overhead, and ultimately enhancing the performance of the compiled code.



Thank you for the attention!