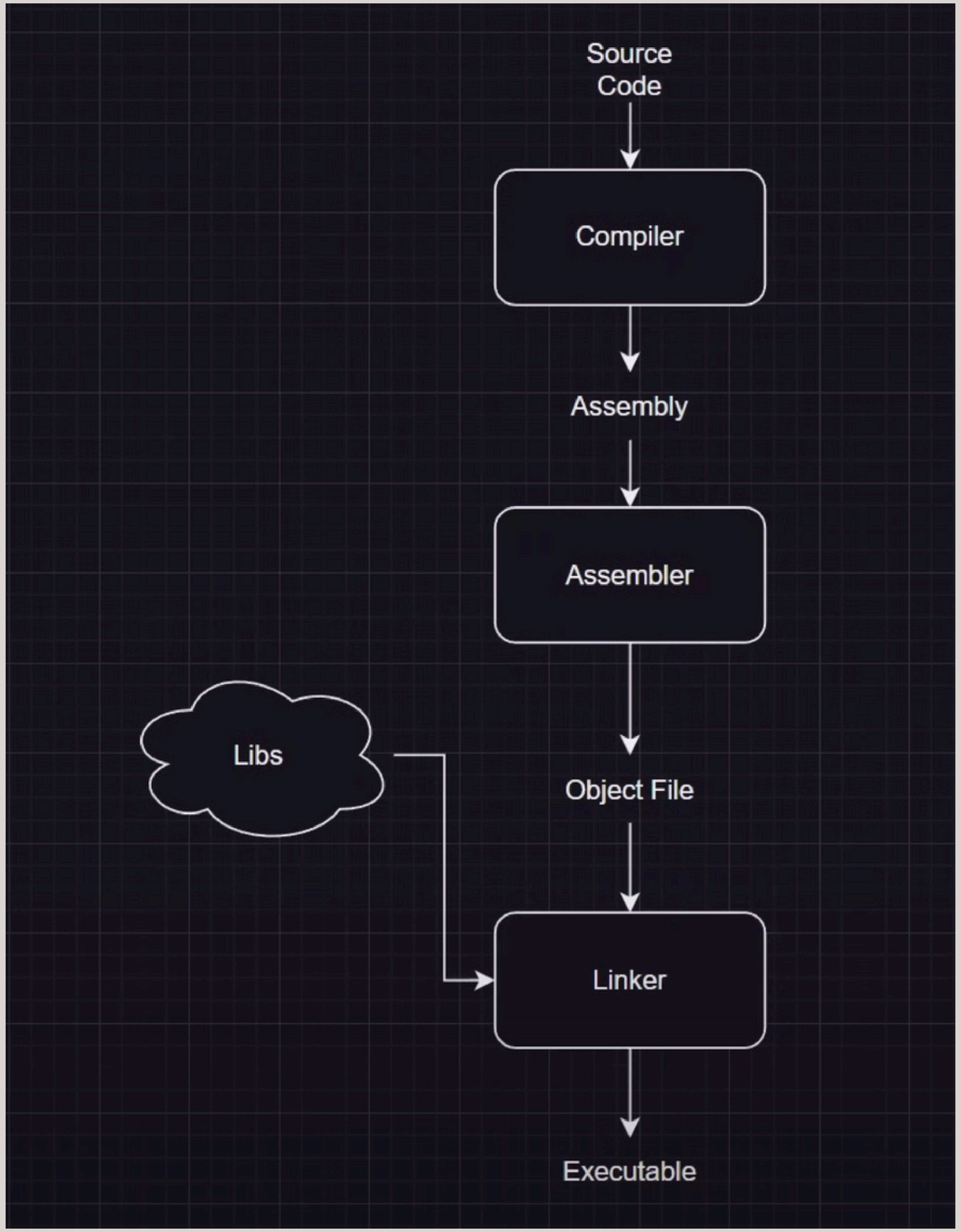
Kharagpur Open Source Society

THE INNER WORKINGS OF COMPILERS

By-
ABHINAV TIWARI

Compilation Process: From Source Code to Executable

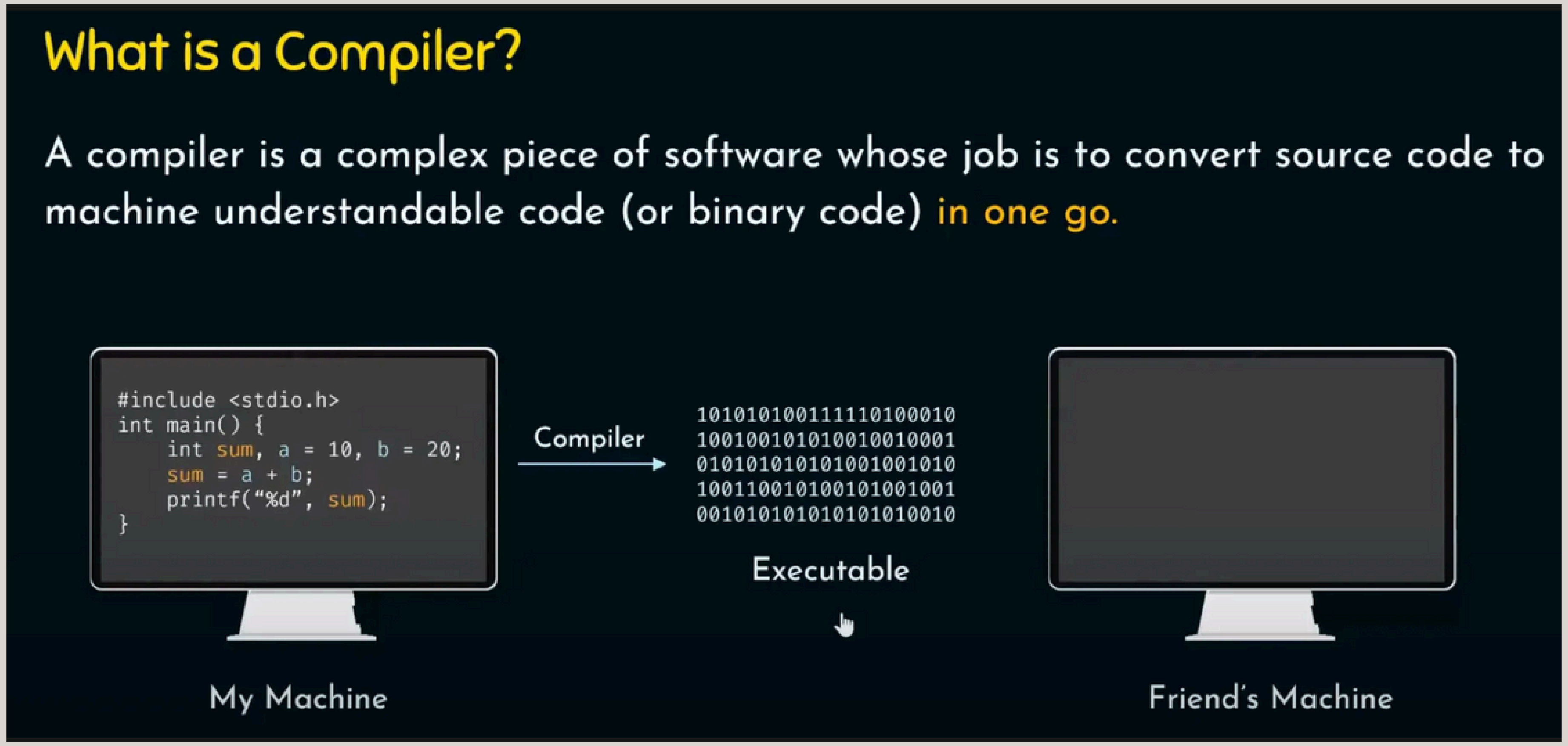


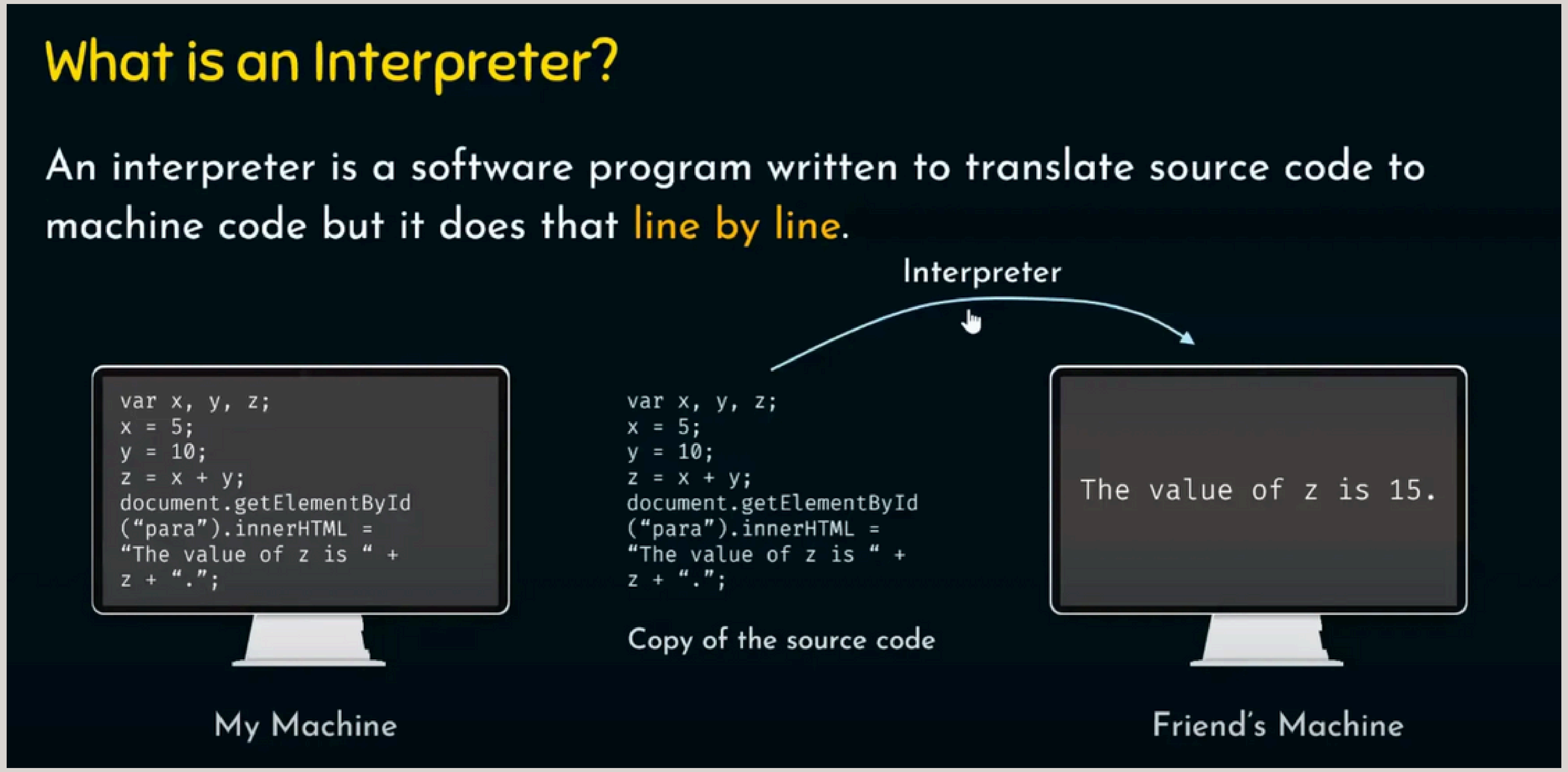
- Compilation: The compiler translates high-level code into assembly language, optimizing it for efficient execution.
- Assembly: The assembler converts the assembly code into a machine-readable object file.
- Linking: The linker merges object files and connects them with libraries to create the final executable program.
- Execution: The generated executable runs directly on the system without further translation, making compiled programs generally faster than interpreted ones.

COMPILERS vs INTERPRETERS

- Convert the entire source code into machine code before execution.
- The resulting executable file runs independently of the source code.
- Compiled programs run faster since they don't require translation at runtime.
- Compiled languages: C, C++, Rust.

- Process and execute code line by line, translating it into machine instructions at runtime.
- Interpreted programs run slower due to continuous translation.
- Interpreted languages: Python, JavaScript, Ruby.





COMPILERS

FRONT-END

The front end takes source code as input and transforms it into an intermediate representation (IR).

MIDDLE-END

The middle end optimizes the IR for performance, reducing execution time and memory usage.

BACK-END

The back end converts the optimized IR into machine code (assembly code) and prepares it for execution.

COMPILERS - FRONT END

1. Lexical Analysis : Breaks code into tokens

For Example :

```
int a = 10;
```

This gets tokenised as :

```
[int] [a] [=] [10] [;]
```

2. Syntax analysis: Verifies code structure and constructs a parse tree using grammar rules

For Example :

```
int a = 10;
```

Corresponding tree :

```
Assignment
├── Type: int
├── Variable: a
├── Value: 10
```

COMPILERS - FRONT END

3. Semantic analysis: Checks for logical errors and type consistency

For Example :

```
int x = "hello"; → Type error
```

4. Intermediate Code Generation : Converts code to low-level representation.

For Example :

```
a = b + c;
```

Intermediate Code :

```
t1 = b + c
a = t1
```

COMPILERS - MIDDLE END

What is an Intermediate Representation (IR)?

- IR is a low-level representation of the source code, but it is not yet machine code.
- Compilers use it to optimize and generate machine-specific instructions efficiently.
- It makes optimisation easier before final machine code generation.
- It is independent of the source language and target machine.

COMPILERS - MIDDLE END

Types of Intermediate Representations

1. Abstract Syntax Tree (AST) :
   a. A tree representation of the parsed source code.
   b. Example for a = b + c * d

```
   =
  / \
 a   +
    / \
   b   *
      / \
     c   d
```

2. Three-Address Code (TAC):
   a. Breaks down complex expressions into simpler three-operand statements.
   b. Example for a = b + c * d

```
t1 = c * d
t2 = b + t1
a = t2
```

COMPILERS - MIDDLE END

Code Optimisation: Improves efficiency independent of target architecture

1. Constant Folding:
    a. Replaces constant expressions at compile time
    b. Example for int x = 5 * 4 // Becomes: int x = 20
    c. Compiler **precomputes** 5 * 4 instead of computing it at runtime.

2. Dead Code Elimination:
    a. Removes unreachable or unnecessary code.
    b. The compiler removes x = 10; as it's overwritten immediately.

```
int x = 10;
x = 20;  // x = 10 is never used, so it's removed.
```

COMPILERS - MIDDLE END

Example: Unoptimized C Code

```c
int compute(int x) {
    int y = x * 2;  // Used
    int z = x * 5;  // Unused (dead code)
    return y;
}
```

Before Optimization

```
assembly
CopyEdit
imul eax, edi, 2  ; y = x * 2
imul ecx, edi, 5  ; z = x * 5 (DEAD CODE)
mov eax, eax      ; return y
```

Optimized Code after Dead Code Elimination

```c
int compute(int x) {
    return x * 2;  // Removed 'z' since it's never used
}
```

After Dead Code Elimination

Plain Text

```
assembly
CopyEdit
imul eax, edi, 2  ; Only y = x * 2 is kept
```

COMPILERS - MIDDLE END

3. Loop Unrolling:
   a. Expands loops to reduce branching overhead.
   b. The compiler removes x = 10; as it's overwritten immediately.

For Example:

```
for (i = 0; i < 4; i++) {arr[i] = 0; }
```

Converts to :

```
arr[0] = 0;
arr[1] = 0;
arr[2] = 0;
arr[3] = 0;
```

4. Strength Reduction:
   a. Replaces expensive operations with cheaper ones.
   b. Eg : $x = y * 2 \Rightarrow x = y << 1$ // Bitwise is faster than multplication

COMPILERS - MIDDLE END

```
for (int i = 0; i < 100; ++i)
{
    func(i * 1234);
}
```

Even on today's CPUs, multiplication is a little slower than simpler arithmetic, so the compiler will rewrite that loop to be something like

```
for (int iTimes1234 = 0; iTimes1234 < 100 * 1234; i += 1234)
{
    func(iTimes1234);
}
```

5. Common Subexpression Elimination (CSE)
    a. Removes redundant calculations.

```
int x = (a + b) * c + (a + b) * d;
```

→

```
int t = (a + b);
int x = t * c + t * d;
```

COMPILERS - MIDDLE END

5. Constant propagation:
   - The compiler tracks the provenance of values and takes advantage of knowing that certain values are constant for all possible executions.

6. Tail call removal:
   - A recursive function that ends in a call to itself can often be rewritten as a loop, reducing call overhead and reducing the chance of stack overflow.

BEFORE

```
int factorial(int n, int acc) {
    if (n == 0) return acc;
    return factorial(n - 1, acc * n);
}
```

AFTER

```
int factorial(int n) {
    int acc = 1;
    while (n > 0) {
        acc *= n;
        n--;
    }
    return acc;
}
```

COMPILERS - BACK END

The back end of a compiler is responsible for converting optimised intermediate representation (IR) into assembly code or machine code for execution. It consists of code selection, register allocation, instruction scheduling, and final assembly generation.

Steps in the Back End:

1. **Instruction Selection**
   - Converts IR into target CPU instructions.
   - Example: `x = a + b;` → `ADD R1, R2, R3`.
2. **Instruction Scheduling**
   - Reorders instructions to improve execution speed (pipelining).
3. **Register Allocation**
   - Assigns variables to CPU registers to minimize memory access.
4. **Code Generation**
   - Produces final **assembly or machine code** for execution.
5. **Code Linking & Relocation**
   - Combines object files, resolves function calls, and loads the program into memory.

SOME INTERESTING COMPILER OPTIMIZATIONS

1. Integer division by a constant :
   a. Integer Division is the most expensive thing you could do on a modern CPU

```
unsigned divideByThree(unsigned x)
{
    return x / 3;
}
```

```
divideByThree(unsigned int):
  mov eax, edi              ; eax = edi
  mov edi, 2863311531       ; edi = 0xaaaaaaab
  imul rax, rdi             ; rax = rax * 0xaaaaaaab
  shr rax, 33               ; rax >>= 33
  ret
```

- Since (div) is typically slow so it uses multiplication and bit shifting, which are much faster.
- 0xAAAAAAAB (2863311531 in decimal) is carefully chosen because it approximates 1/3 in fixed-point arithmetic, which is multiplied instead of division by 3.
- imul extends the multiplication to 64 bits, which has to be converted back to 32 bits by a right shift of 33 bits.
- The lower 32 bits represent the fractional part, which is not needed, and the upper 32 bits are only required.
- Thus, we shift the entire 64-bit result right by 33 bits by dividing the number by 2^33.

SOME INTERESTING COMPILER OPTIMIZATIONS

2. Counting set bits:

   a. The number of set bits (also called the Hamming weight or population count) in an integer is the count of 1s in its binary representation.

```c
int countSetBits(unsigned a)
{
    int count = 0;
    while (a != 0)
    {
        count++;
        a &= (a - 1); // clears the bottom set bit
    }
    return count;
}
```

```asm
countSetBits(unsigned int):
  xor eax, eax        ; count = 0
  test edi, edi       ; is a == 0?
  je .L4              ; if so, return
.L3:
  inc eax             ; count ++
  blsr edi, edi       ; a &= (a - 1);
  jne .L3             ; jump back to L3 if a != 0
  ret
.L4:
  Ret
```

- This method flips the rightmost set bit (1) to 0 and turns all lower bits to 1 (if any).
- Then, it removes this lowest set bit, reducing the number of set bits by 1 in each step.
- Example: n = 9 (Binary: 1001)
- At n = 0, we stop

```
n =   1001 (decimal 9)
n-1 = 1000 (decimal 8)
n &= (n - 1) → 1001 & 1000 = 1000   (1 bit cleared)
```

```
n =   1000 (decimal 8)
n-1 = 0111 (decimal 7)
n &= (n - 1) → 1000 & 0111 = 0000   (1 more bit cleared)
```

SOME INTERESTING COMPILER OPTIMIZATIONS

2. Counting set bits:
   a. The number of set bits (also called the Hamming weight or population count) in an integer is the count of 1s in its binary representation.

```
countSetBits(unsigned int):
  popcnt eax, edi      ; count = number of set bits in a
  ret
```

- The code generator recognizes this pattern and is able to choose the perfect instruction and does it in one go: POPCNT (population count).
- POPCNT (short for Population Count) is an inbuilt CPU instruction that efficiently counts the number of set bits (1s) in a binary number.

SOME INTERESTING COMPILER OPTIMIZATIONS

3. Chained conditionals:
   - To check if a character is whitespace or not.

```
bool isWhitespace(char c)
{
    return c == ' '
      || c == '\r'
      || c == '\n'
      || c == '\t';
}
```

```
Bit 32 -> Space (' ') = 1
Bit 13 -> '\r' = 1
Bit 10 -> '\n' = 1
Bit 9  -> '\t' = 1
```

```
isWhitespace(char):
  xor eax, eax              ; result = false
  cmp dil, 32               ; is c > 32
  ja .L4                    ; if so, exit with false
  movabs rax, 4294977024    ; rax = 0x100002600
  shrx rax, rax, rdi        ; rax >>= c
  and eax, 1                ; result = rax & 1
.L4:
  ret
```

- The number rax = 0x100002600 (in binary: 100000000000000000010011000000000) acts as a bitmask.
- Each bit position corresponds to an ASCII character, with 1s for whitespace characters.
- If c > 32, return false (not whitespace); this avoids unnecessary bit shifts for values outside our lookup range.
- Shift the lookup table right by c bits so that the lowest bit corresponds to c.
- When the order of comparisons is modified, GCC fails to apply the lookup table trick and instead generates a different sequence of conditional jumps.

INTERPRETERS

Interpreters execute source code line by line, translating and running each instruction sequentially.

The process typically involves:

1. Parsing the source code
2. Executing each instruction immediately

How an Interpreter Works

When you run a script using an interpreter (like Python or JavaScript), it follows these steps:

- Reads a statement from the source code => The Python interpreter reads the first line and moves to execution immediately.

- Checks for syntax errors => If there is a syntax error, Python stops immediately and does not execute the next lines.

- Converts the statement into an intermediate form => When you run a Python program, it does not execute the source code directly. Instead, Python first translates the human-readable source code into an intermediate representation called bytecode before executing it.

Bytecode

Bytecode is a **low-level** representation of your Python program, which is **not human-readable** but still not machine code. It is a set of instructions that the **Python Virtual Machine** can understand and execute.

Why Bytecode?

1. Portability: Bytecode can run on any system with a Python interpreter, unlike machine code, which is system-dependent.
2. Faster Execution: Parsing source code every time would be slow; bytecode speeds up execution.
3. Intermediate Representation: Acts as a bridge between source code and machine execution.

Let's take a simple Python function:

```
import dis

def example():
    x = 5
    y = x * 2
    print(y)

dis.dis(example)
```

Bytecode output

```
2           0 LOAD_CONST               1 (5)
            2 STORE_FAST               0 (x)

3           4 LOAD_FAST                0 (x)
            6 LOAD_CONST               2 (2)
            8 BINARY_MULTIPLY
           10 STORE_FAST               1 (y)

4          12 LOAD_GLOBAL              0 (print)
           14 LOAD_FAST                1 (y)
           16 CALL_FUNCTION            1
           18 POP_TOP
           20 RETURN_VALUE
```

Breaking down the Bytecode:

1. x=5 → Bytecode Representation

```
2           0 LOAD_CONST               1 (5)
            2 STORE_FAST               0 (x)
```

- LOAD_CONST 1 (5) → Loads the constant value 5 onto the stack.
- STORE_FAST 0 (x) → Stores the value from the stack into the variable x.

2. y = x * 2 → Bytecode Representation

```
3           4 LOAD_FAST                0 (x)
            6 LOAD_CONST               2 (2)
            8 BINARY_MULTIPLY
           10 STORE_FAST               1 (y)
```

- LOAD_FAST 0 (x) → Loads the value of x onto the stack.
- LOAD_CONST 2 (2) → Loads the constant 2 onto the stack.
- BINARY_MULTIPLY → Multiplies x and 2, leaving the result on the stack.
- STORE_FAST 1 (y) → Stores the result into variable y.

Breaking down the Bytecode:

3. print(y) → Bytecode Representation

```
  4           12 LOAD_GLOBAL              0 (print)
              14 LOAD_FAST                1 (y)
              16 CALL_FUNCTION            1
              18 POP_TOP
              20 RETURN_VALUE
```

- LOAD_GLOBAL 0 (print) → Loads the function print from global scope.
- LOAD_FAST 1 (y) → Loads the value of y onto the stack.
- CALL_FUNCTION 1 → Calls the print function with 1 argument (y).
- POP_TOP → Removes the function's return value (since print() returns None).
- RETURN_VALUE → Ends the function execution.

Challenges Encountered during the Development of Compilers and Interpreters

Problem:

- Different platforms (Windows, Linux, macOS) and hardware architectures (x86, ARM, RISC-V) have different instruction sets.
- A compiler must generate code that works across multiple platforms.

Solution:

- Use an Intermediate Representation (IR)
- To achieve platform independence, compilers use an Intermediate Representation (IR). IR is a machine-independent representation that can be later converted into target-specific machine code.

Challenges Encountered during the Development of Compilers and Interpreters

- Understanding the LLVM IR Code

```
define i32 @add(i32 %a, i32 %b) {
  %1 = add i32 %a, %b
  ret i32 %1
}
```

1. **Function Definition (`define i32 @add(i32 %a, i32 %b)`)**
   - `define i32 @add` → Defines a function named `add` that returns an **i32 (32-bit integer)**.
   - `(i32 %a, i32 %b)` → The function takes **two 32-bit integer arguments**, `%a` and `%b`.
2. **Addition (`%1 = add i32 %a, %b`)**
   - `%1` is a **temporary variable** (SSA register).
   - `add i32 %a, %b` → Adds `%a` and `%b` and stores the result in `%1`.
3. **Return Statement (`ret i32 %1`)**
   - `ret i32 %1` → Returns the result stored in `%1`.

Challenges Encountered during the Development of Compilers and Interpreters

Problem:

- Different CPUs have unique features like:
  - Vectorized Instructions (SIMD)
  - Parallel Execution (Multi-core Processing)
  - Different Register Sizes (32-bit vs. 64-bit)
- A compiler must optimize the generated code to fully utilize hardware capabilities.

```c
int square(int x) {
    return x * x;
}
```

- Generated x86 Assembly (Optimized for Intel Processors)

```
imul eax, eax   ; Uses the efficient "imul" multiplication instruction
```

- Generated ARM Assembly (Optimized for ARM Processors)

```
mul r0, r0, r0   ; Uses ARM's multiplication instruction
```

Disassembler to compare the compiled assembly code to higher-level code

To demonstrate how a compiler optimises register allocation using a disassembler, we will:

1. Write a simple C function and compile it.
2. Generate unoptimised (O0) and optimised (O3) assembly code using clang or GCC.
3. Analyse the differences using a disassembler (objdump).

```
#include <stdio.h>

int compute(int a, int b) {
    int x = a + b;  // x is stored in memory at -O0
    int y = x * 2;  // y is stored in memory at -O0
    return y;
}

int main() {
    printf("%d\n", compute(3, 5));
    return 0;
}
```

Disassembler to compare the compiled assembly code to higher-level code

Compile with Different Optimization Levels

We will compile this with no optimizations (-O0) and maximum optimizations (-O3).

```
gcc -O0 -S -masm=intel test.c -o test_O0.s  # No optimization
gcc -O3 -S -masm=intel test.c -o test_O3.s  # High optimization
```

- O0 → No optimization, keeps variables in memory.
- O3 → Aggressive optimization, stores values in registers instead of memory.
- S → Generates assembly code.
- masm=intel → Uses Intel syntax (easier to read than AT&T syntax).

Disassembler to compare the compiled assembly code to higher-level code

Disassemble and Compare the Assembly Code

Assembly Output with O0 (No Optimization)

```
compute:
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi   ; Store a in memory
    mov     DWORD PTR [rbp-8], esi   ; Store b in memory
    mov     eax, DWORD PTR [rbp-4]   ; Load a from memory
    add     eax, DWORD PTR [rbp-8]   ; Add b (memory access)
    mov     DWORD PTR [rbp-12], eax  ; Store x in memory
    mov     eax, DWORD PTR [rbp-12]  ; Load x from memory
    imul    eax, eax, 2              ; Multiply by 2
    mov     DWORD PTR [rbp-16], eax  ; Store y in memory
    mov     eax, DWORD PTR [rbp-16]  ; Load y from memory
    pop     rbp
    ret
```

Analysis (Unoptimized Code)

- Memory-heavy: Stores and loads every variable (a, b, x, y) from memory.
- Performance issue: CPU registers are not used efficiently.
- Multiple redundant memory accesses slow down execution.

Disassembler to compare the compiled assembly code to higher-level code

Disassemble and Compare the Assembly Code

Assembly Output with O3 (Optimized Code)

```
compute:
    lea     eax, [rdi + rsi]   ; Compute x = a + b directly into a register
    add     eax, eax           ; Multiply by 2 (y = x * 2)
    ret
```

Analysis (Optimized Code)

- No memory access: Everything is kept in registers (eax, rdi, rsi), eliminating slow memory reads/writes.
- Optimized arithmetic:
  - lea eax, [rdi + rsi] replaces mov and add (faster instruction).
  - add eax, eax replaces multiplication (cheaper operation).
- Shorter, more efficient assembly:
  - O3 eliminates redundant instructions.
  - Faster execution and better CPU utilization.

Kharagpur Open Source Society

THANK YOU