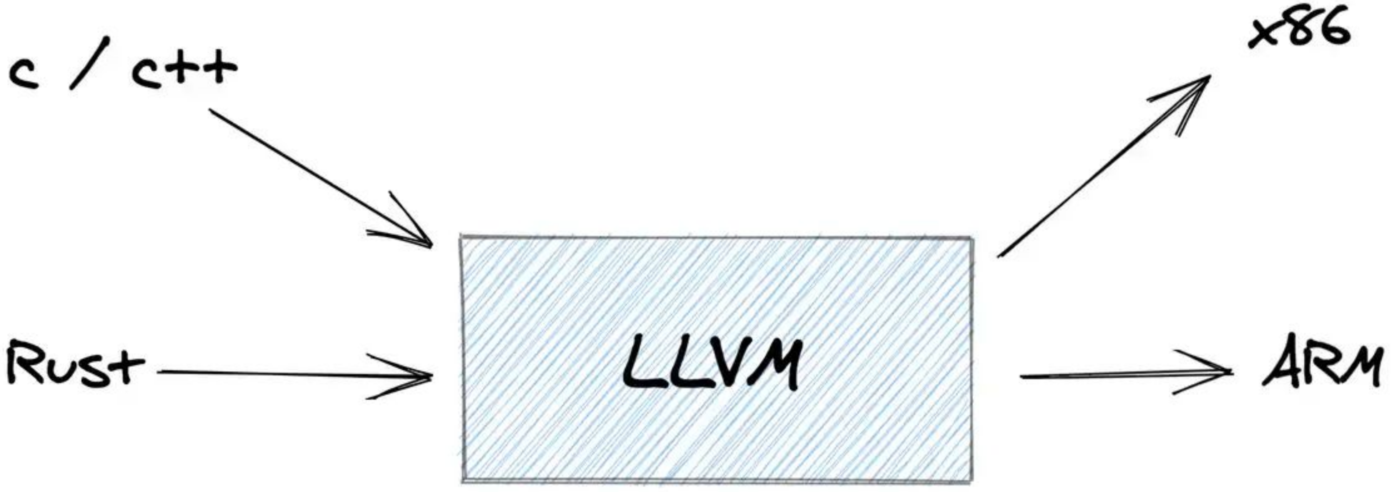


CMPE 230 - Spring 2023

Gökçe Uludoğan

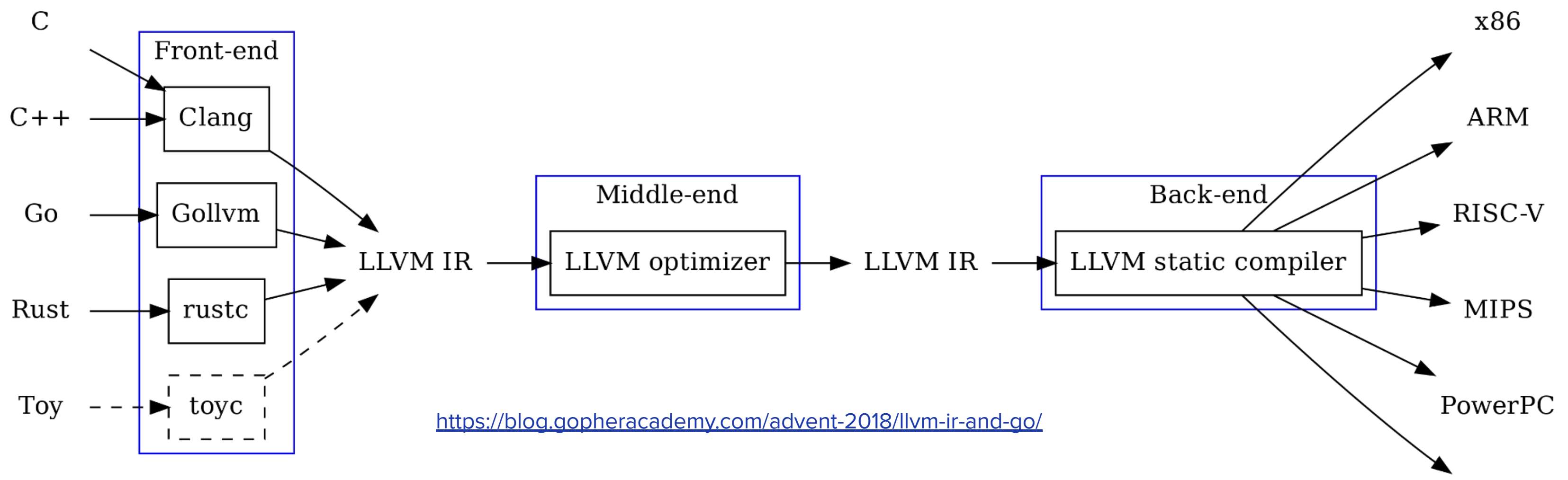
https://mukulrathi.com/create-your-own-programming-language/llvm-ir-cpp-api-tutorial/

LLVM (Low Level Virtual Machine)

- LLVM is a compiler infrastructure!
    - a collection of tools that provide the framework for building compilers
    - modular components to perform different tasks in the compilation process
        - Parsing
        - Optimization
        - Code generation
        - Linking
    - Highly configurable
        - to create custom compilers for specific programming languages or target architectures

LLVM IR

- Intermediate representation in the compilation process
- Low-level & Typed
- Looks like a more readable form of assembly
- Machine-independent



https://blog.gopheracademy.com/advent-2018/llvm-ir-and-go/

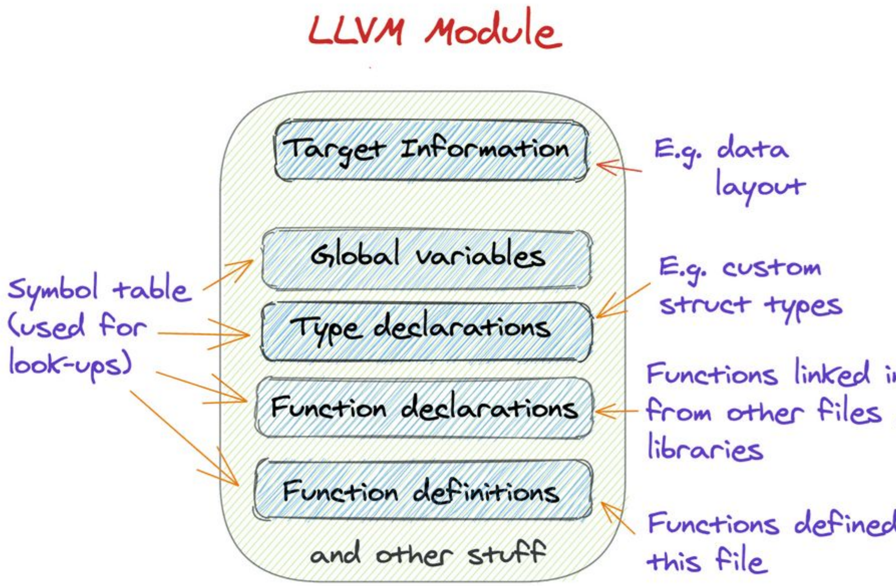
LLVM IR

- Static Single Assignment (SSA)
  - each variable is assigned only once and is defined before it is used
  - `x = x+1` ➜ `x2 = x1 + 1`

- Why SSA?
  - Makes optimization easier!
  - Unambiguous representation of variables and their usages
  - A strict assignment of variables to registers

LLVM IR at a glance

- Each file is a Module
- Each Module is comprised of
  - Global variables
  - A set of Functions which are comprised of
    - A set of basic blocks which are comprised of
      - A set of instructions

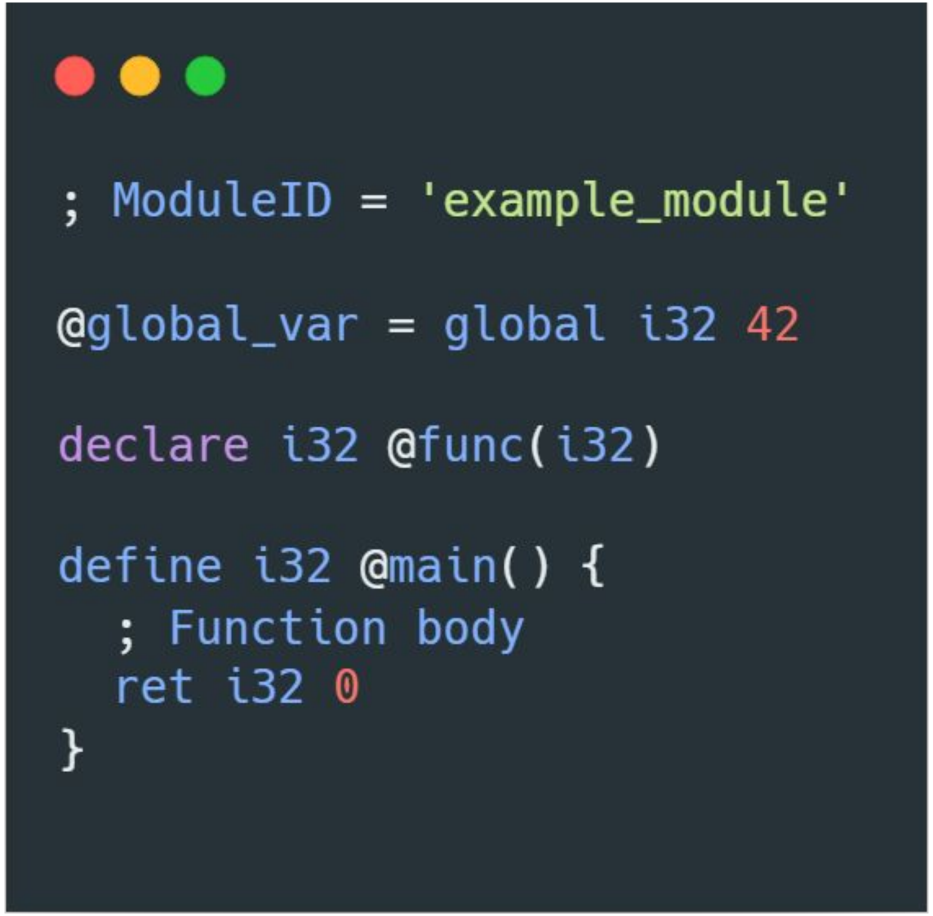


LLVM IR at a glance

C language	**LLVM IR**
Scope: file, function	module, function
Type: bool, char, int, struct{int, char}	i1, i8, i32, {i32, i8}
A statement with multiple expressions	A sequence of instructions each of which is in a form of “x = y op z”.
Data-flow: a sequence of reads/writes on variables	1. load the values of memory addresses (variables) to registers; 2. compute the values in registers; 3. store the values of registers to memory addresses
Control-flow in a function: if, for, while, do while, switch-case,...	A set of basic blocks each of which ends with a conditional jump (or return)

Example Program

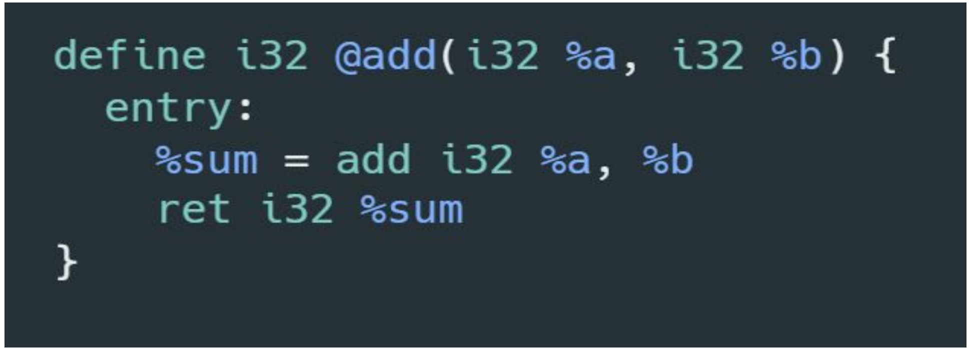
- a module with the ID "example_module"
- Global symbols begin with an at sign (@).
- a function prototype **`@func`** which takes an **`i32`** argument and returns an **`i32`** value
- **`@main`** function which returns an **`i32`** value of 0.



```
; ModuleID = 'example_module'

@global_var = global i32 42

declare i32 @func(i32)

define i32 @main() {
  ; Function body
  ret i32 0
}
```

Functions

- The building blocks of LLVM IR.
- Functions are defined using the **define** keyword, followed by the function signature, a set of basic blocks, and a return statement.
- Contains a set of basic blocks which ends with **terminator instructions**



```
define i32 @add(i32 %a, i32 %b) {
  entry:
    %sum = add i32 %a, %b
    ret i32 %sum
}
```

Instruction

Terminator instructions:

- return

  ```
  ret <type> <value> | ret void
  ```

- branches
  - Unconditional

    ```
    br label <dest>
    ```

  - Conditional

    ```
    br i1 <cond>, label <thenbb>, label <elsebb>
    ```

Comparison

- `<res> = icmp <cmp> <ty> <op1>, <op2>`

Returns either true or false (i1) based on comparison of two variables (op1 and op2) of the same type (ty)

cmp: comparison option

eq (equal), ne (not equal), ugt (unsigned greater than),

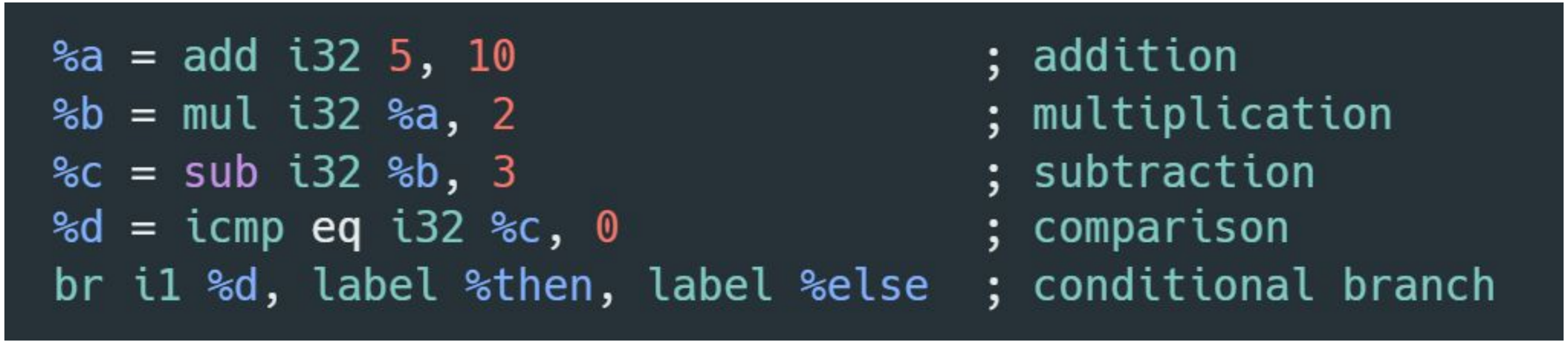
uge (unsigned greater or equal), ult (unsigned less than),

ule (unsigned less or equal), sgt (signed greater than),

sge (signed greater or equal), slt (signed less than), sle (signed less or equal)

Instructions

- Typed operations that operate on values
- Operations for arithmetic, logical, memory, control flow, and other operations
- A syntax that resembles an assembly-like language, with an opcode followed by operands.
- Example instructions

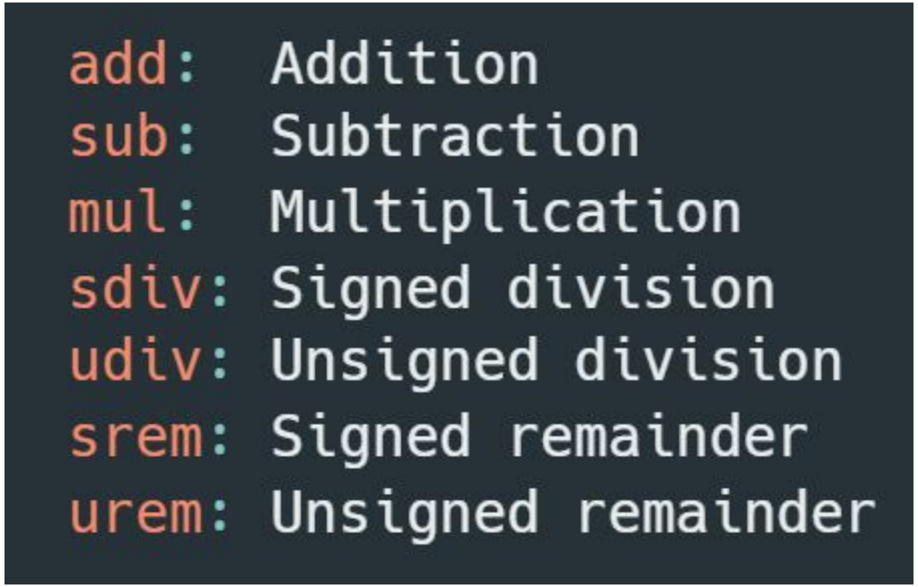


```
%a = add i32 5, 10                           ; addition
%b = mul i32 %a, 2                           ; multiplication
%c = sub i32 %b, 3                           ; subtraction
%d = icmp eq i32 %c, 0                       ; comparison
br i1 %d, label %then, label %else           ; conditional branch
```

LLVM IR is a typed language, which means that operands and instructions have explicit types.

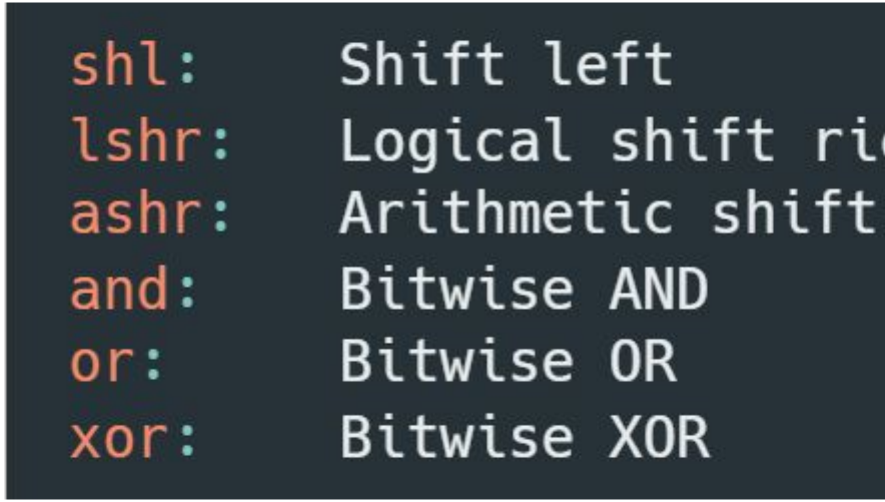
Instructions

Arithmetic Instructions



```
add:  Addition
sub:  Subtraction
mul:  Multiplication
sdiv: Signed division
udiv: Unsigned division
srem: Signed remainder
urem: Unsigned remainder
```

Bitwise Instructions



```
shl:     Shift left
lshr:    Logical shift ri
ashr:    Arithmetic shift
and:     Bitwise AND
or:      Bitwise OR
xor:     Bitwise XOR
```

Registers and Memory variables

Registers

- Virtual CPU registers
- Used to store intermediate results and temporary values during program execution.
- Infinite number of registers
- By default, registers are numbered (%0, %1, %2), but you can give them custom names.

-
  ```
  %reg = add i32 4, 2
  ```

Memory Variables

- Represent data that is stored in memory and accessed using memory addresses
- Accessed using pointers
  - alloca: yields a pointer

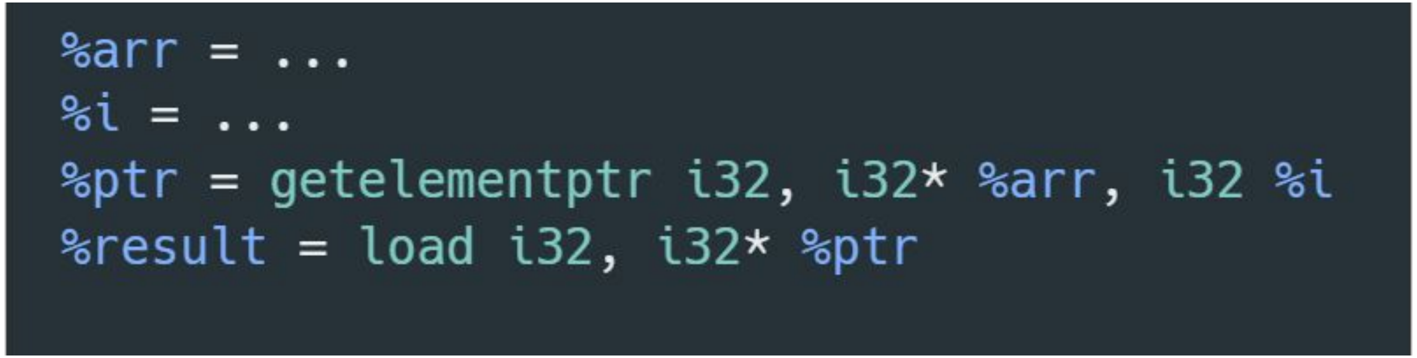
    ```
    %var = alloca i32
    ```
  - load

    ```
    %result = load i32, i32* %var
    ```
  - store

    ```
    store i32 3, i32* %var
    ```

getelementptr

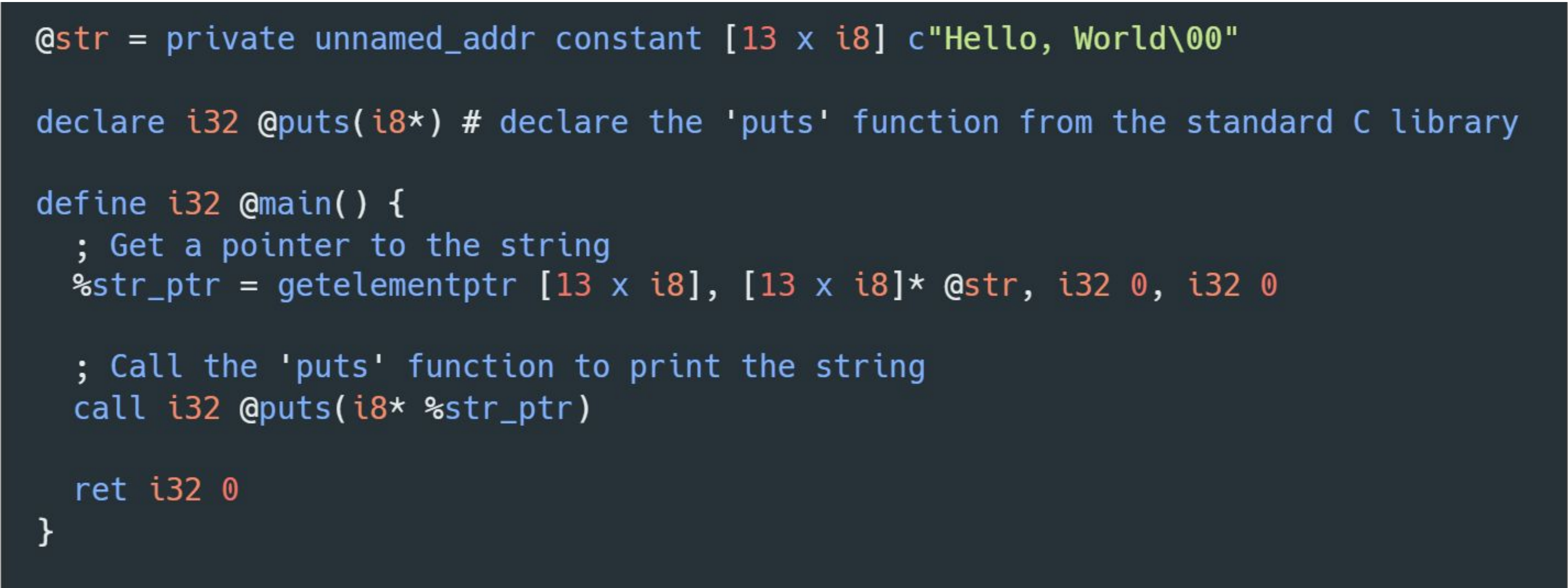
- To calculate a pointer to an element in an array or a structure.
- Takes a base pointer and one or more indices as operands, and calculates the pointer to the desired element by adding the appropriate offset.
- Loading i-th element of **arr**



```
%arr = ...
%i = ...
%ptr = getelementptr i32, i32* %arr, i32 %i
%result = load i32, i32* %ptr
```

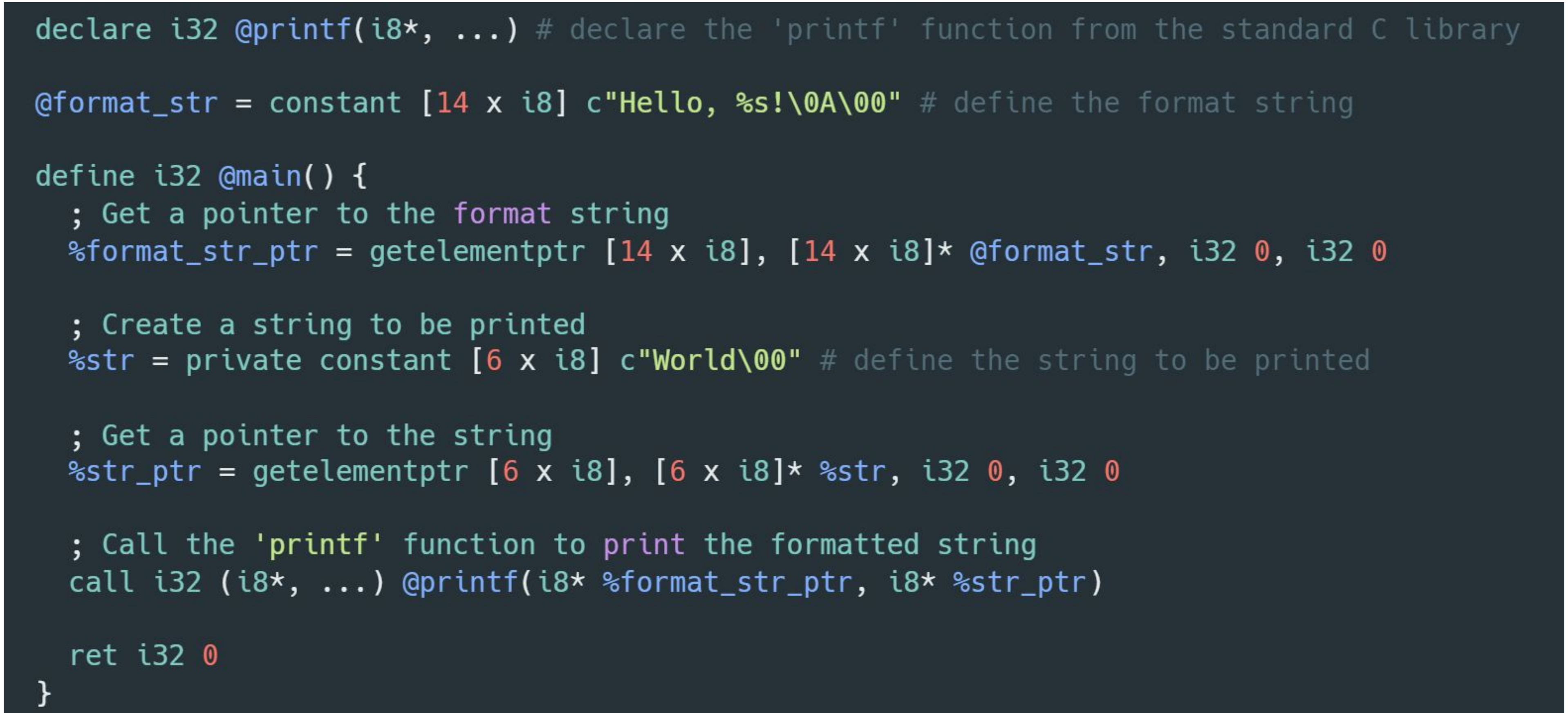
Output

- Define the string as a global constant.
- Call instruction to invoke a standard library function that can print the string to the console or other output.



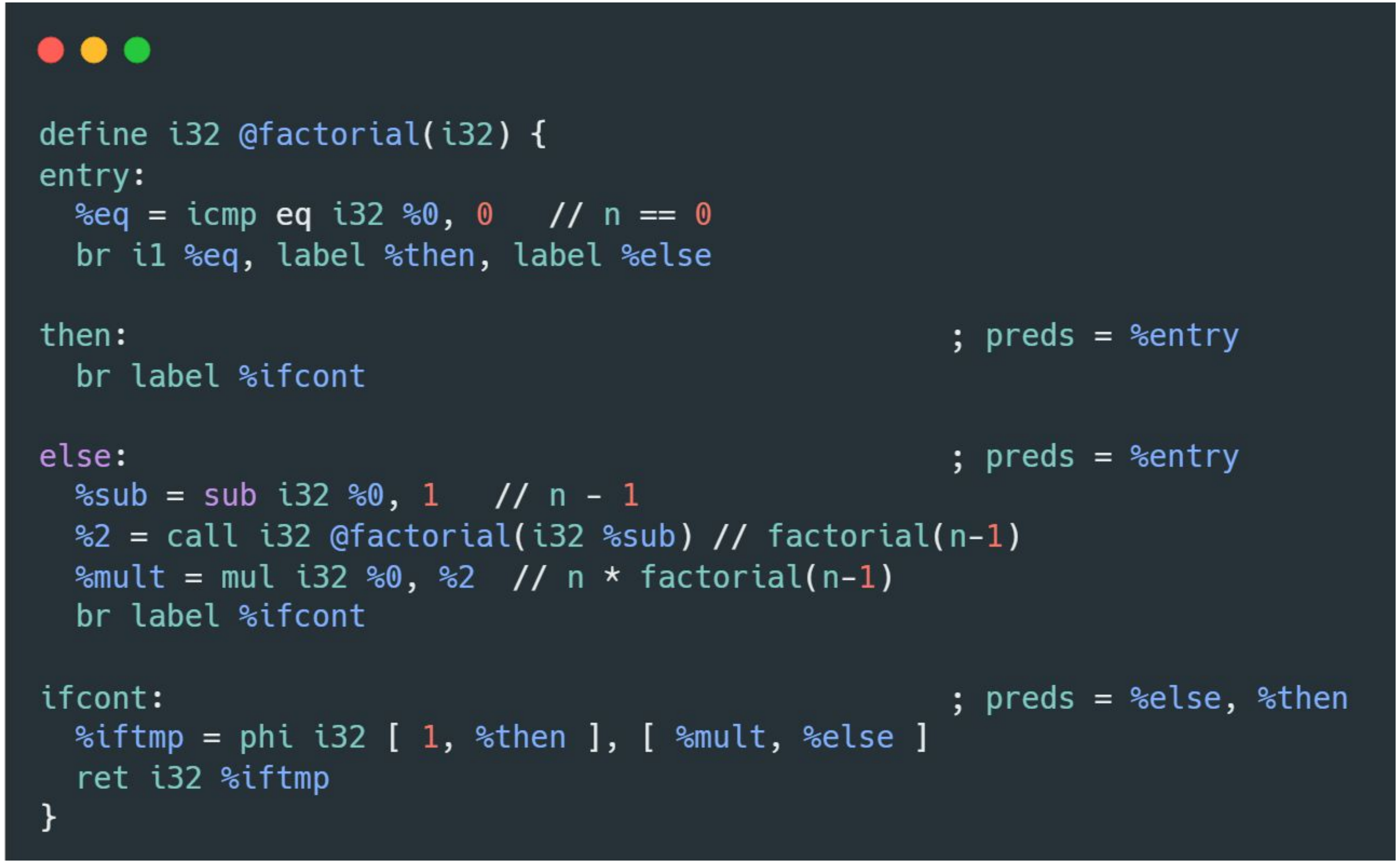
```
@str = private unnamed_addr constant [13 x i8] c"Hello, World\00"

declare i32 @puts(i8*) # declare the 'puts' function from the standard C library

define i32 @main() {
  ; Get a pointer to the string
  %str_ptr = getelementptr [13 x i8], [13 x i8]* @str, i32 0, i32 0

  ; Call the 'puts' function to print the string
  call i32 @puts(i8* %str_ptr)

  ret i32 0
}
```

Output



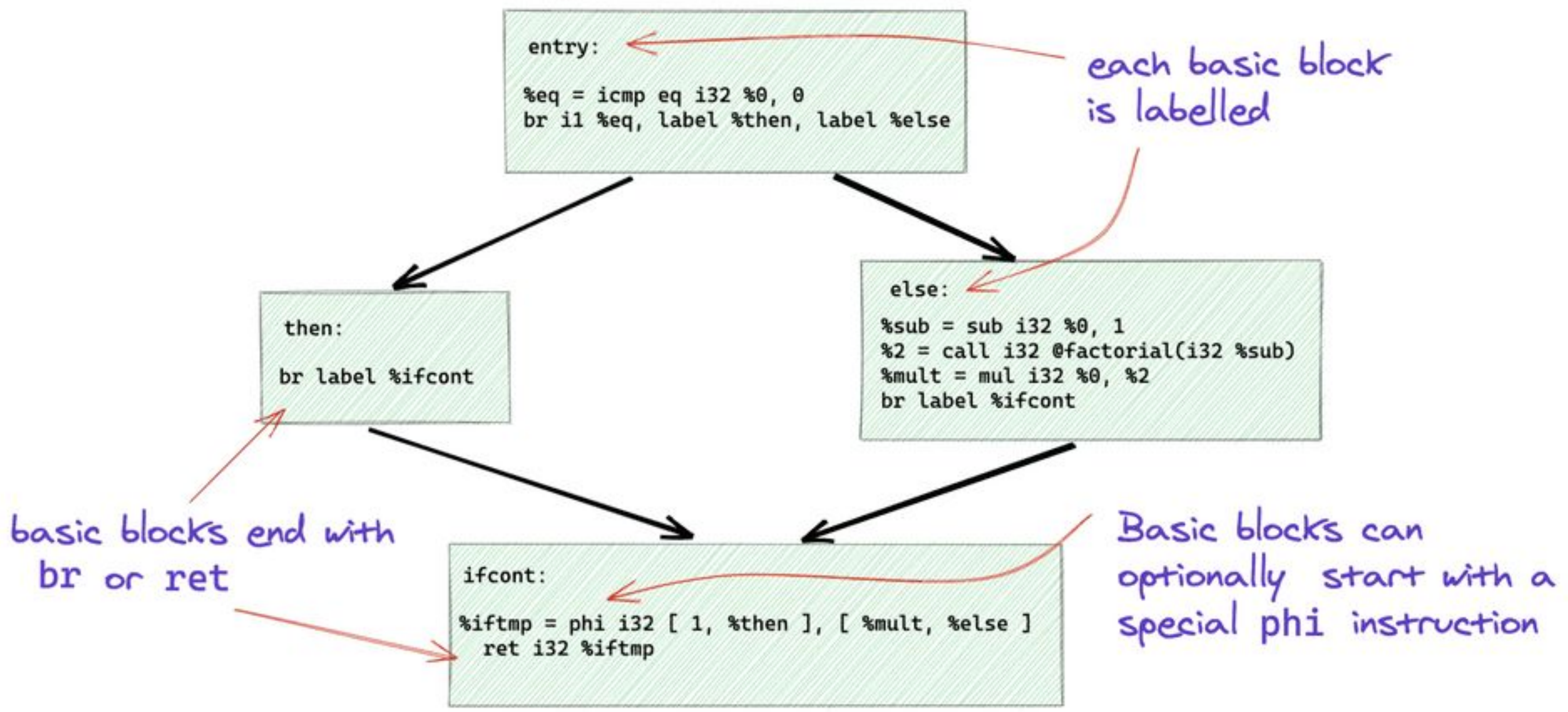
```
declare i32 @printf(i8*, ...) # declare the 'printf' function from the standard C library

@format_str = constant [14 x i8] c"Hello, %s!\0A\00" # define the format string

define i32 @main() {
  ; Get a pointer to the format string
  %format_str_ptr = getelementptr [14 x i8], [14 x i8]* @format_str, i32 0, i32 0

  ; Create a string to be printed
  %str = private constant [6 x i8] c"World\00" # define the string to be printed

  ; Get a pointer to the string
  %str_ptr = getelementptr [6 x i8], [6 x i8]* %str, i32 0, i32 0

  ; Call the 'printf' function to print the formatted string
  call i32 (i8*, ...) @printf(i8* %format_str_ptr, i8* %str_ptr)

  ret i32 0
}
```

Example: Factorial



```
define i32 @factorial(i32) {
entry:
  %eq = icmp eq i32 %0, 0   // n == 0
  br i1 %eq, label %then, label %else

then:                                                  ; preds = %entry
  br label %ifcont

else:                                                  ; preds = %entry
  %sub = sub i32 %0, 1   // n - 1
  %2 = call i32 @factorial(i32 %sub) // factorial(n-1)
  %mult = mul i32 %0, %2  // n * factorial(n-1)
  br label %ifcont

ifcont:                                                ; preds = %else, %then
  %iftmp = phi i32 [ 1, %then ], [ %mult, %else ]
  ret i32 %iftmp
}
```

Control flow graphs in LLVM IR



```
entry:

%eq = icmp eq i32 %0, 0
br i1 %eq, label %then, label %else
```

```
then:

br label %ifcont
```

```
else:

%sub = sub i32 %0, 1
%2 = call i32 @factorial(i32 %sub)
%mult = mul i32 %0, %2
br label %ifcont
```

```
ifcont:

%iftmp = phi i32 [ 1, %then ], [ %mult, %else ]
  ret i32 %iftmp
```

each basic block is labelled
basic blocks end with br or ret
Basic blocks can optionally start with a special phi instruction

In LLVM, everything that can have a name but does not is assigned a number

Commands

- C to LLVM IR
  - `clang -S -emit-llvm test.c`
- LLVIM interpreter
  - `lli test.ll`
- Assembly & Machine Code
  - `llc test.ll -o test.s`
  - `clang file.s-o myexec && ./myexec`