Information lost necessitating more complex hardware

PYTHON

```
np.add(arr1, arr2)
```

C/C++

```
for(i = 0;i < n;i++)
 res[i] = arr1[i] + arr2[i]
```

ISA

```
.Loop:
lw    a5, 0(a2)       # *(arr1+i)
lw    a6, 0(a3)       # *(arr2+i)
add    a0, a5, a6
sw    a0, 0(a4)
# Bump pointers.
  addi   a2, a2, 4
  addi   a3, a3, 4
  addi   a4, a4, 4
  addi   a1, a1, 1
  bne    a1, a3, loop
```

Global reg

Load/Store Queues

Branch Predictor to find loop paralleism

Why ISAs suck ?

```
#pragma clang unroll_count(10)
for(int i = 0;i < 10;i++)
res[i] = arr1[i] + arr2[i];
}
```

```
res[0] = arr1[0] + arr2[0];
res[1] = arr1[1] + arr2[1];
…..
res[9] = arr1[9] + arr2[9];
```

Register naming introduced dependencies

Need register renaming hardware

```
lw  a6, 0(a0)
lw  a4, 0(a1)
lw  a5, 4(a0)
lw  a3, 4(a1)
add a4, a4, a6
sw  a4, 0(a2)
add a6, a3, a5
lw  a7, 8(a0)
lw  a5, 8(a1)
lw  a3, 12(a0)
lw  a4, 12(a1)
sw  a6, 4(a2)
add a5, a5, a7
sw  a5, 8(a2)
add a6, a4, a3
lw  a7, 16(a0)
lw  a5, 16(a1)
lw  a3, 20(a0)
lw  a4, 20(a1)
```

Intel tutorial

Assume **no** previous Intermediate Representation (IR) knowledge.

But this is not a lecture about compiler theory!

After the tutorial, you should:

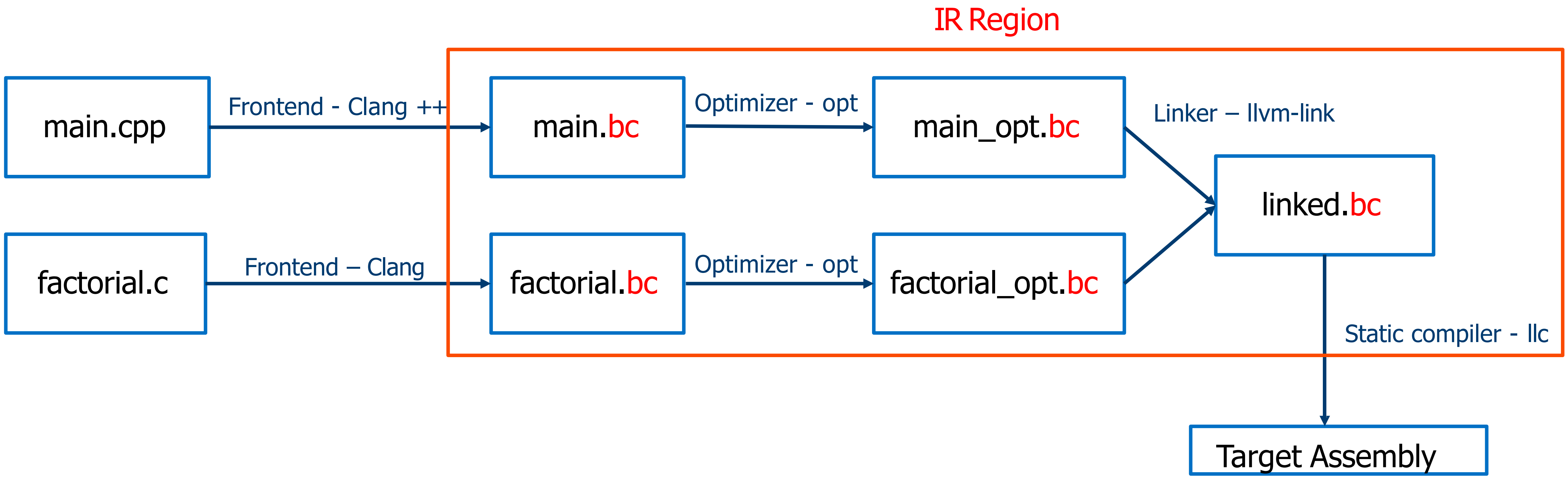
- Understand common LLVM tools.
- Be able to write simple IR.
- Be able to understand the language reference.
  - Use it to inspect compiler-generated IR.

What is the LLVM IR?

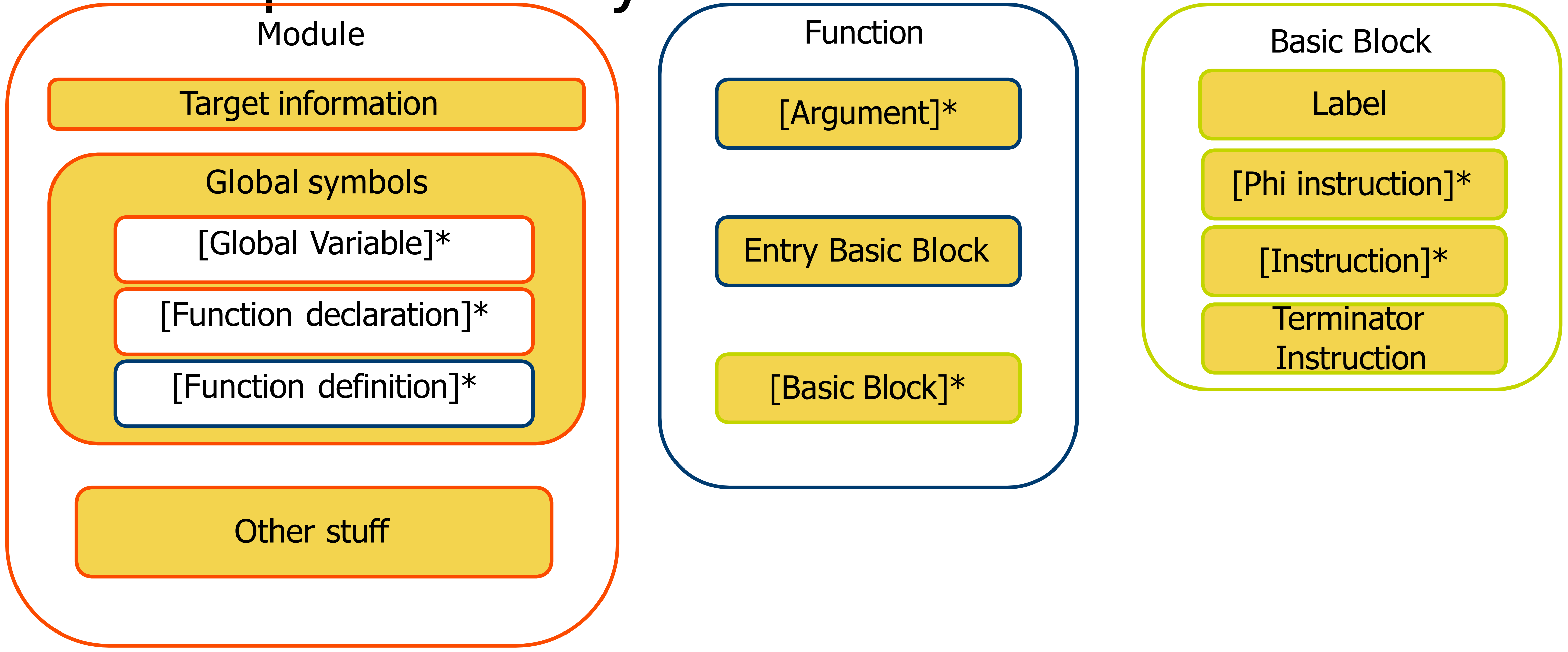
The LLVM **I**ntermediate **R**epresentation:

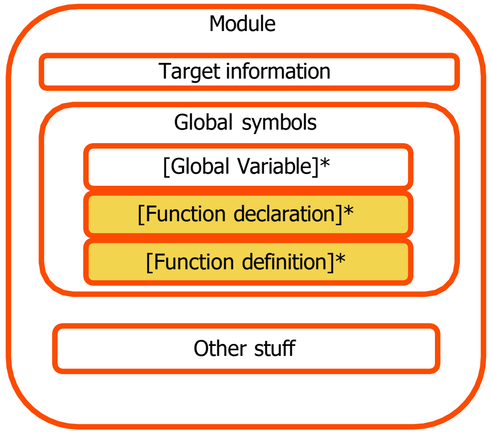
- Is a low level programming language
  - RISC-like instruction set
- … while being able to represent high-level ideas.
  - i.e. high-level languages can map cleanly to IR.
- Enables efficient code optimization

IR & the compilation process



Simplified IR layout





A basic main program

Hand-written IR for this program:

```c
int factorial(int val);

int main(int argc, char** argv)
{
  return factorial(2) * 7 == 42;
}
```

```llvm
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

% Virtual Registers %

Those are "local" variables.

"LLVM IR has infinite registers"

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

"LLVM IR has infinite registers"

Static Single Assignment (SSA)

Every variable is assigned *exactly* once.

Every variable is defined before it is used.

Information lost necessitating more complex hardware

PYTHON

```
np.add(arr1, arr2)
```

C/C++

```
for(i = 0;i < n;i++)
 res[i] = arr1[i] + arr2[i]
```

ISA

```
.Loop:
lw     a5, 0(a2)       # *(arr1+i)
lw     a6, 0(a3)       # *(arr2+i)
add    a0, a5, a6
sw     a0, 0(a4)
# Bump pointers.
  addi   a2, a2, 4
  addi   a3, a3, 4
  addi   a4, a4, 4
  addi   a1, a1, 1
  bne    a1, a3, loop
```

Global reg

Load/Store Queues

Branch Predictor to find loop paralleism

Why ISAs suck ?

```
#pragma clang unroll_count(10)
for(int i = 0;i < 10;i++)
res[i] = arr1[i] + arr2[i];
}
```

```
res[0] = arr1[0] + arr2[0];
res[1] = arr1[1] + arr2[1];
…..
res[9] = arr1[9] + arr2[9];
```

Register naming introduced dependencies

Need register renaming hardware

```
lw  a6, 0(a0)
lw  a4, 0(a1)
lw  a5, 4(a0)
lw  a3, 4(a1)
add a4, a4, a6
sw  a4, 0(a2)
add a6, a3, a5
lw  a7, 8(a0)
lw  a5, 8(a1)
lw  a3, 12(a0)
lw  a4, 12(a1)
sw  a6, 4(a2)
add a5, a5, a7
sw  a5, 8(a2)
add a6, a4, a3
lw  a7, 16(a0)
lw  a5, 16(a1)
lw  a3, 20(a0)
lw  a4, 20(a1)
```

Types, types everywhere!

Very much a typed language.

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

Types, types everywhere!

Very much a typed language.

```
        i32            i32

       i32       i32        i8**      ) {
            i32            i32
           i32
               i32
                 i1       i32
      i32
}
```

Types, types everywhere!

The instructions explicitly dictate the types expected.

Easy to figure out argument types.

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

Types, types everywhere!

The instructions explicitly dictate the types expected.

Easy to figure out argument types.

Easy to figure out return types (mostly)

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

Types, types everywhere!

No implicit conversions!

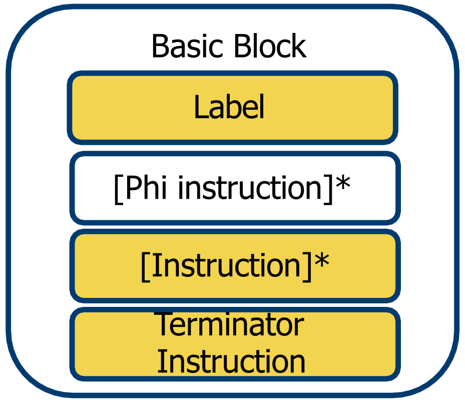
```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

Basic Blocks

List of non-terminator instructions ending with a terminator instruction:

- **Branch - "br"**
- **Return - "ret"**
- Switch – "switch"
- Unreachable – "unreachable"
- Exception handling instructions



```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:
  ret i32 1
recursive_case:
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %0)
  %3 = mul i32 %val, %1
  ret i32 %2
}
```

Basic Blocks

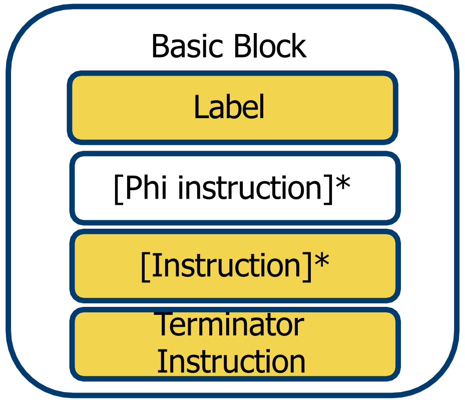
List of non-terminator instructions ending with a terminator instruction:

- Return - “ret”

Execution proceeds to:

- calling function

```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:
  ret i32 1
recursive_case:
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %0)
  %3 = mul i32 %val, %1
  ret i32 %2
}
```



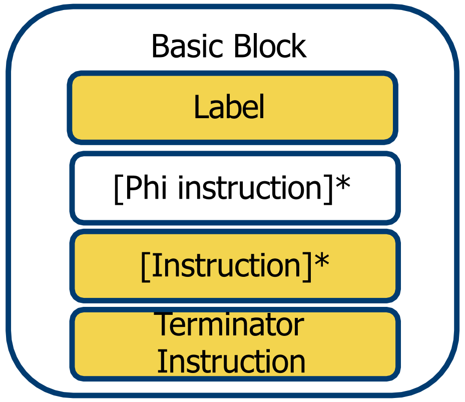
Basic Blocks

List of non-terminator instructions ending with a terminator instruction:

- Branch - “br”

Execution proceeds to:

- another Basic Block
  - It’s **successor**!



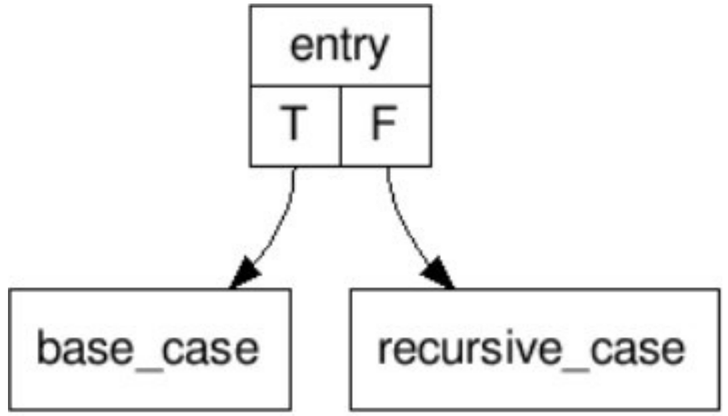
```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:
  ret i32 1
recursive_case:
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %0)
  %3 = mul i32 %val, %1
  ret i32 %2
}
```

Control Flow Graph (CFG)

The optimizer can generate the CFG in dot format:

```
opt –analyze –dot-cfg-only <input.ll>
```

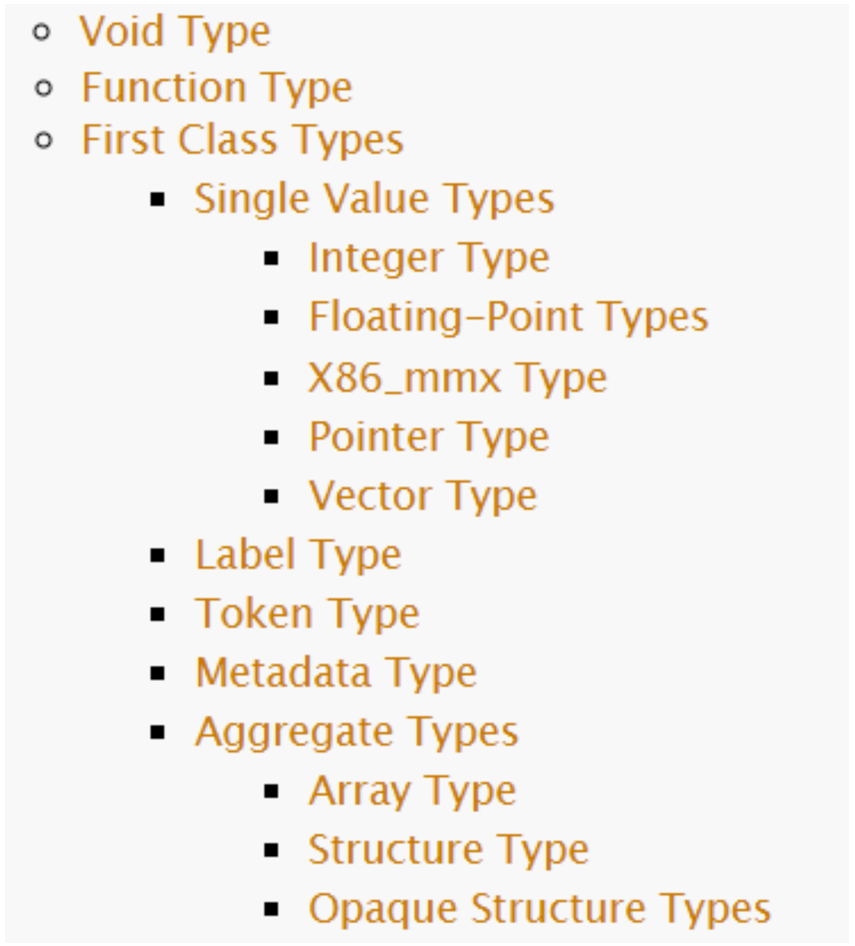
-dot-cfg-only = Generate .dot files. Don’t include instructions.



CFG for 'ir_factorial' function

LLVM’s type system

From the language reference:



- Void Type
- Function Type
- First Class Types
  - Single Value Types
    - Integer Type
    - Floating-Point Types
    - X86_mmx Type
    - Pointer Type
    - Vector Type
  - Label Type
  - Token Type
  - Metadata Type
  - Aggregate Types
    - Array Type
    - Structure Type
    - Opaque Structure Types

Aggregate types: arrays

Defined by:

- A constant size.
- An element type.
- [for GVs] an initializer

```
@array = global [17 x   ]
@array = global [17 x i8]
@array = global [17 x i8] zeroinitializer
```

Accessing arrays & manipulating pointers

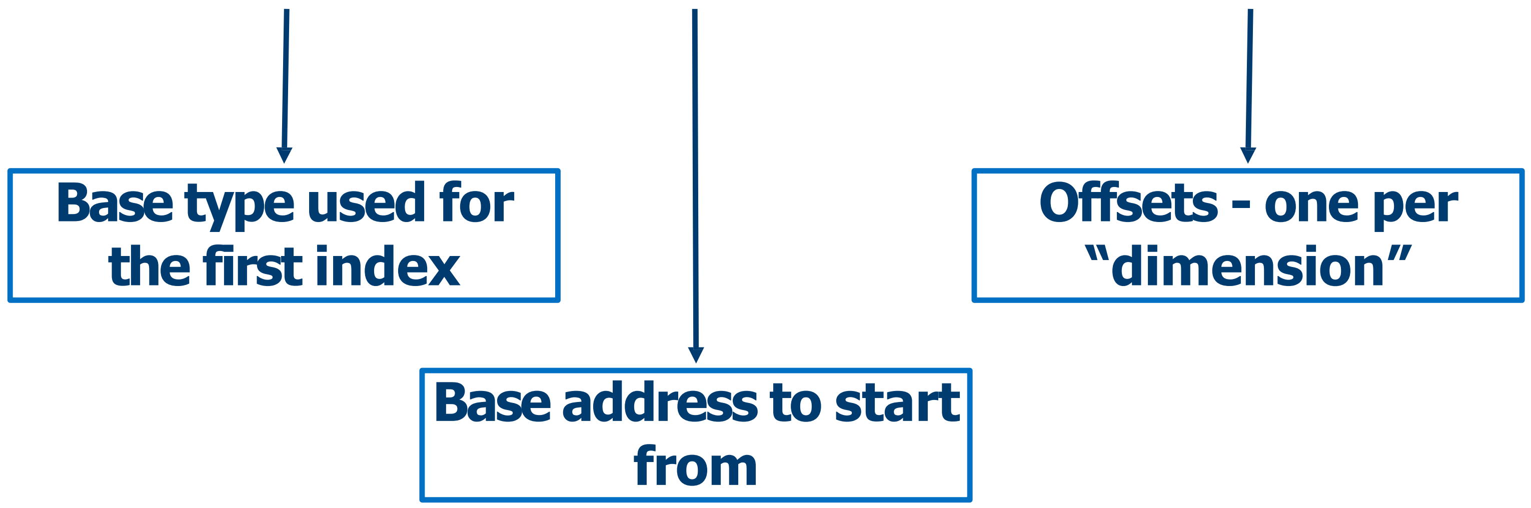
The Get Element Pointer (GEP) instruction:

- Provides a way to calculate pointer offsets.
- Abstracts away details like:
  - Size of types
  - Padding inside structs
- Intuitive to use…
  …once you understand a few basic principles.

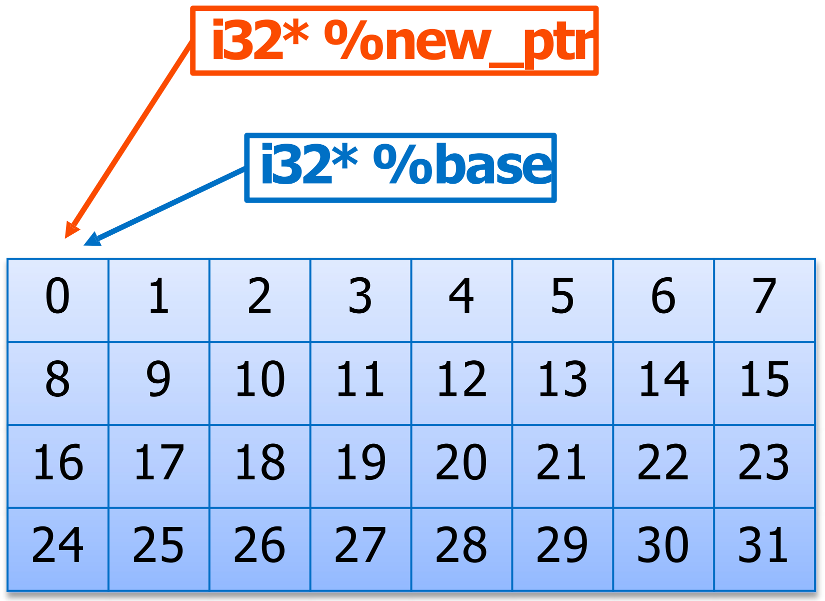
Manipulating pointers

The Get Element Pointer (GEP) instruction:

```
<result> = getelementptr <ty>, <ty>* <ptrval>,  [i32 <idx>]+
```


Base type used for the first index
Base address to start from
Offsets - one per "dimension"

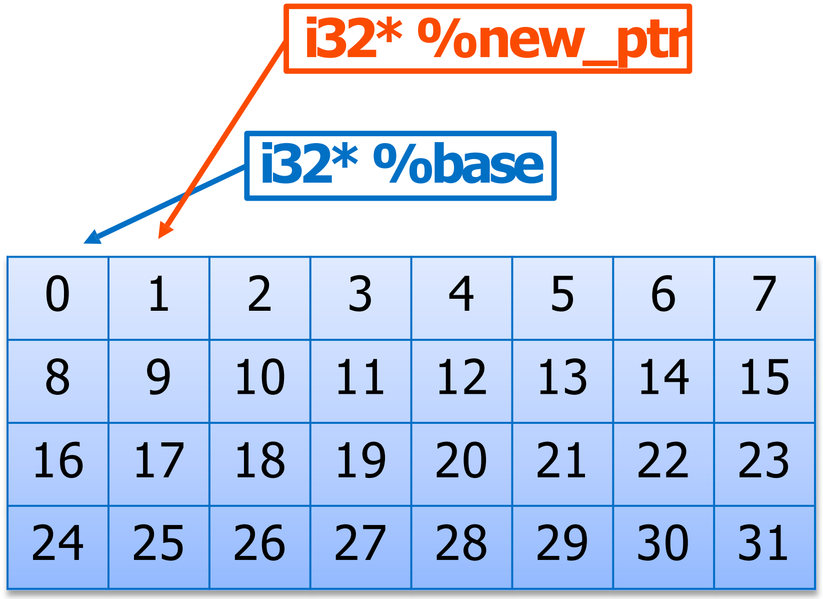
Manipulating pointers



```
%new_ptr = getelementptr i32, i32* %base, i32 0
```

"Offset by 0 **elements of the base type**"

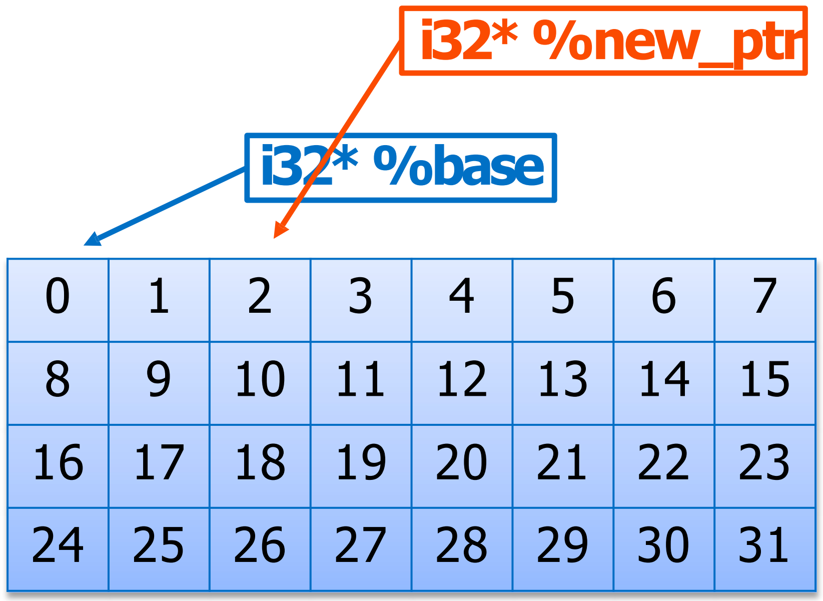
Manipulating pointers



```
%new_ptr = getelementptr i32, i32* %base, i32 1
```

"Offset by 1 **elements of the base type**"

Manipulating pointers



```
%new_ptr = getelementptr i32, i32* %base, i32 2
```

"Offset by 2 **elements of the base type**"