Part 7

The Structure of Computation (IR Code)

Programs

- Earlier, you developed a "model" for the structure of programs (the AST)

```
print 2 + 3 * 4;
```

➡

```
PrintStatement(
    BinOp('+',
          Integer(2),
          BinOp('*',
                Integer(3),
                Integer(4)
               )
         )
)
```

- Focus is on expressing program structure in the domain of the source language (syntax)

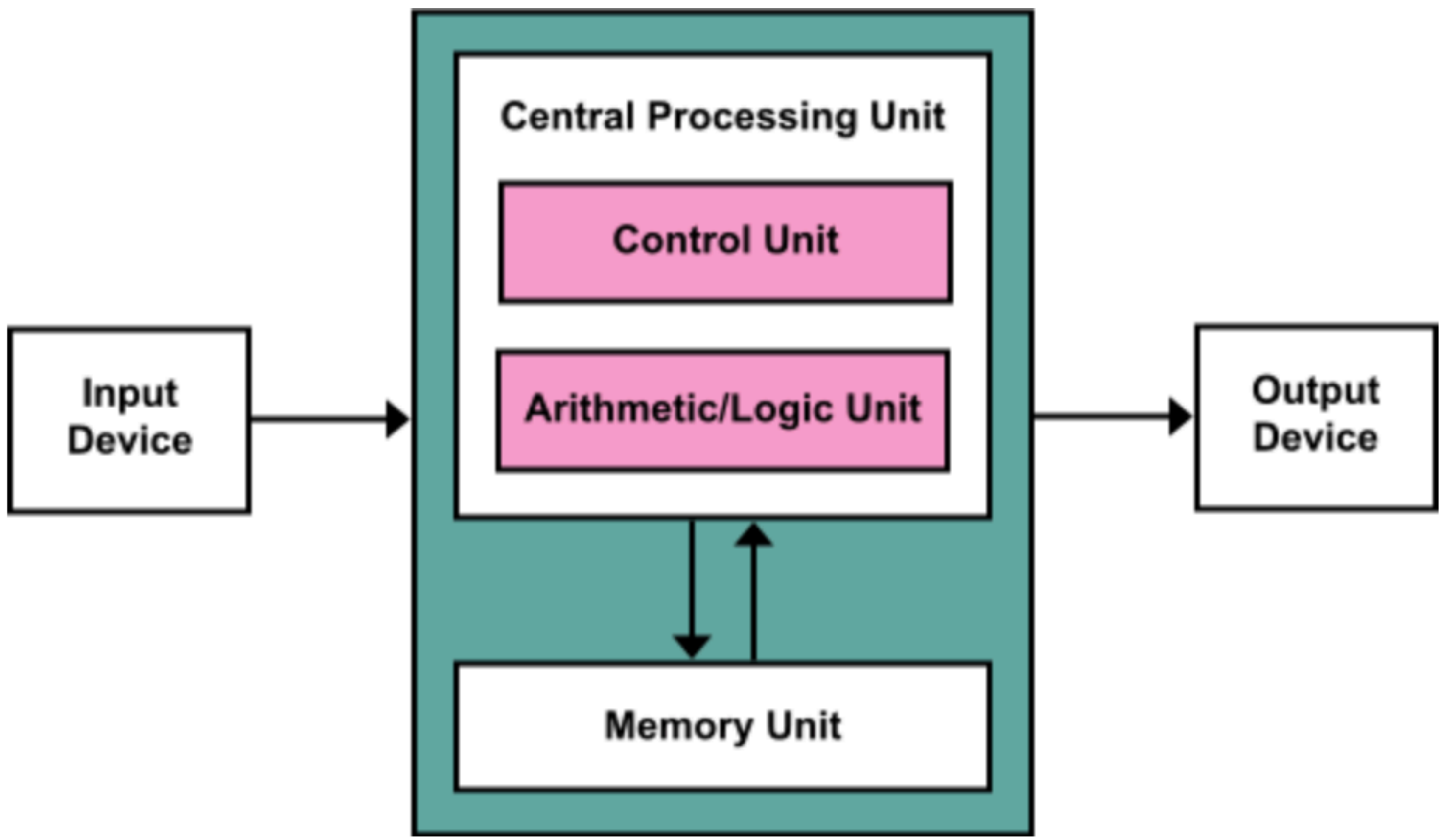
Machines

- There is a similar concept for machines.

➡ ????

- Specifically, most computers follow a fairly standard "model" of computation

von Neumann Machines

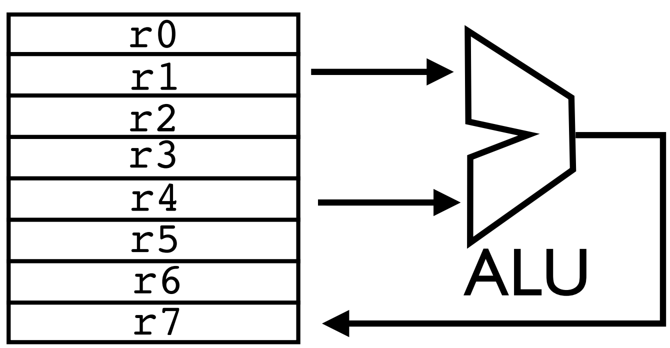


Arithmetic/Logic

- CPUs have instructions that perform <u>single</u> arithmetic operations

```
add, sub, mul, div, and, or, xor, not, eq, lt, ...
```

- These operations are applied to values, typically supplied from "registers" on the CPU



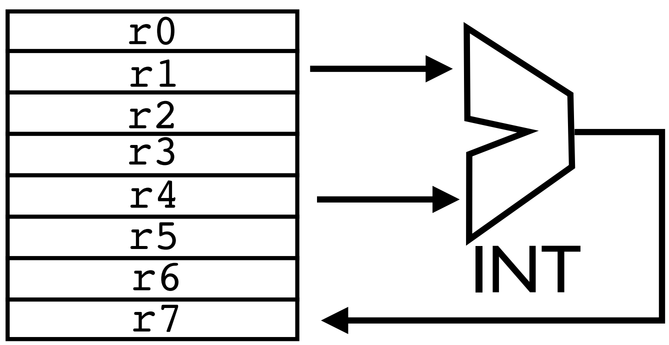

```
add r1, r4, r7
```

Data Types

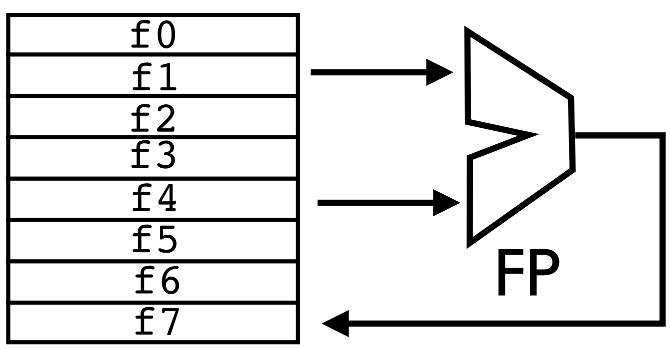
- There are two main datatypes (of varying sizes)

```
Integers (i8, i16, i32, i64, i128, etc.)
Floats (f32, f64, f128, etc.)
```

- Often handled by different ALUs & instructions



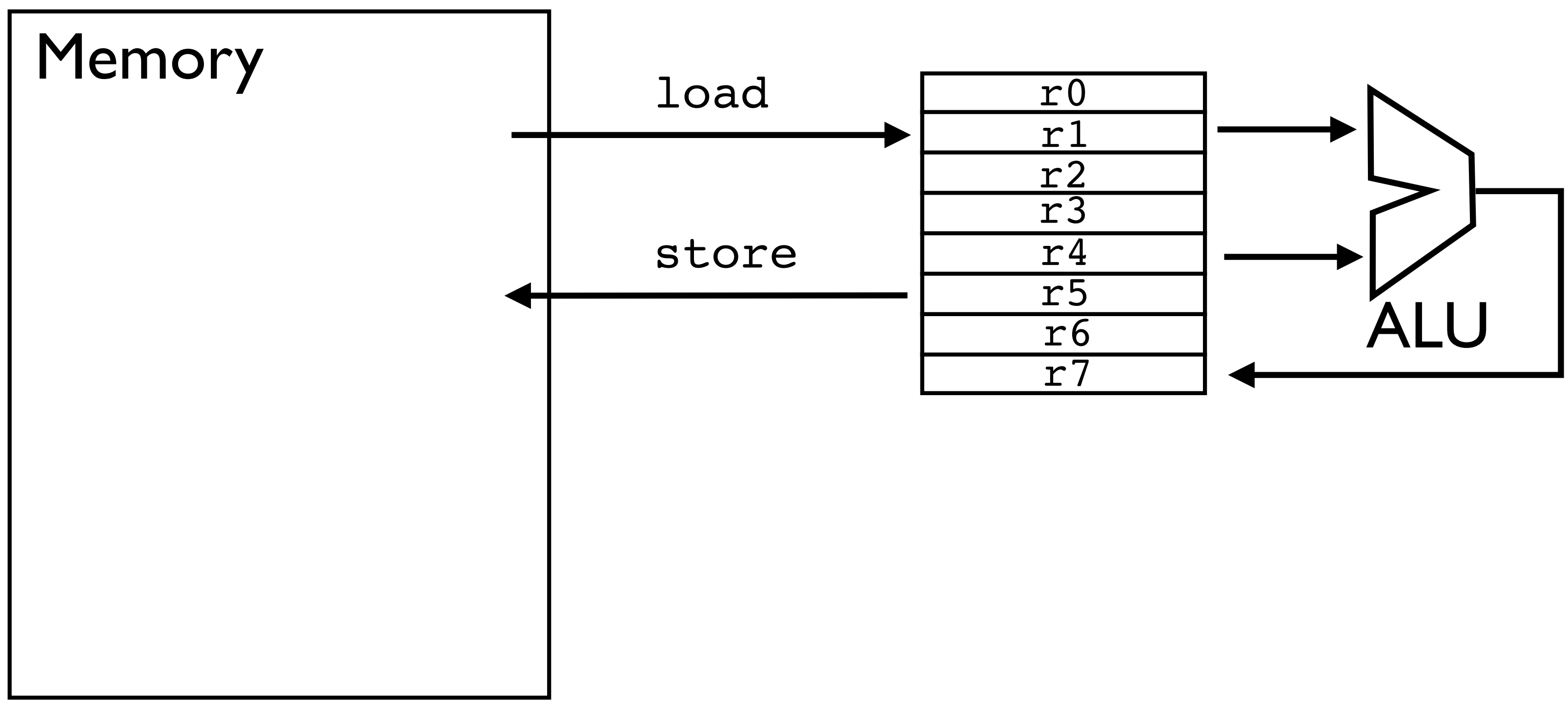

```
i32.add r1, r4, r7
```




```
f64.add f1, f4, f7
```

Memory

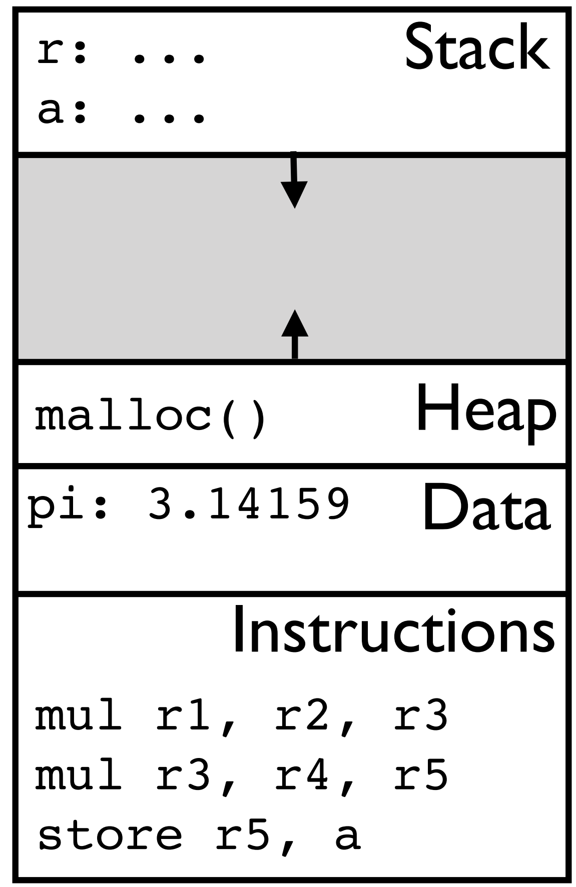
- Computers have memory.
- Two primary operations: load/store



Memory

- Memory is used for both instructions and data
- Often segmented or managed in regions

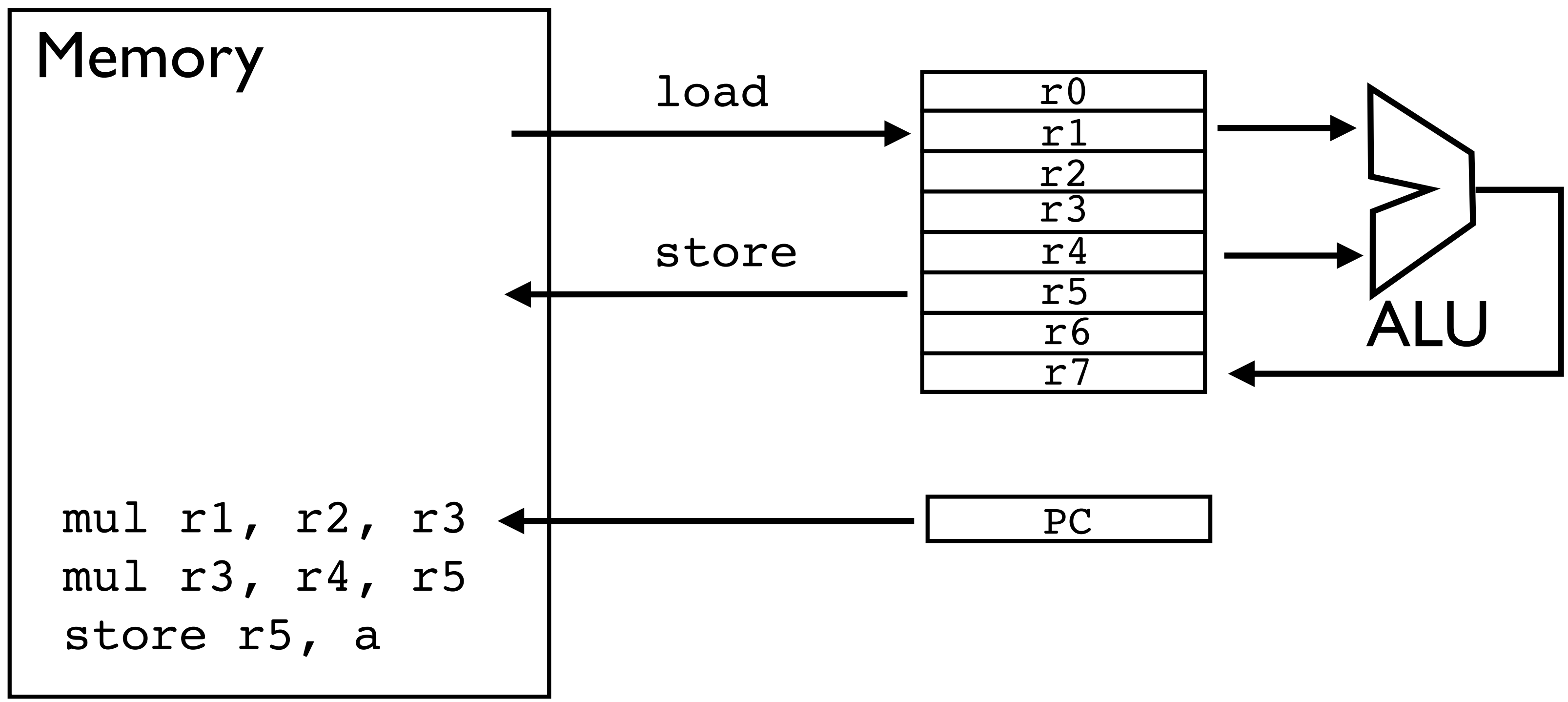
```
const pi = 3.14159;

func area(r float) float {
    var a = pi*r*r;
    return a;
}
```



- Related to "declarations"
- A declaration specifies where a value lives.

Control Flow

- Computers execute stored programs
- Instructions execute in sequence.



- Program counter (PC) points to instruction

Control Flow

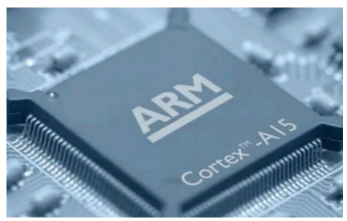
- Certain instructions alter the PC

```
jump            # Goto
bz, bnz         # Conditional branch (zero?, not-zero?)
call            # Call a subroutine
ret             # Return from subroutine
```

- Control flow is typically very low-level
- Often not more than "goto" statements
- No higher-level abstractions (e.g., loops)

A Moment of Reflection

- All CPUs are variations of these concepts



- Only the low-level details vary. For example, number of registers, variety of instructions, etc.

- Question: Do you write a compiler for a single model of a specific CPU? <u>No.</u>

Abstract Machines

- Compilers often target an "abstract machine"
- A generic "CPU"
- With a standard set of basic "instructions"

Intermediate Representation

- The abstract machine is programmed using "intermediate representation" or IR Code
- It's like a generic machine code
  - Mimics architecture of actual CPUs
  - Easy to translate to actual machine code
  - Related to "bytecode" used by interpreters

Big Picture

Source

```
print 2 + 3 * 4;
```

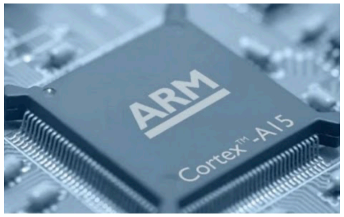
AST

```
PrintStatement(
    BinOp('+',
          Integer(2),
          BinOp('*',
                Integer(3),
                Integer(4)
               )
         )
)
```

IR

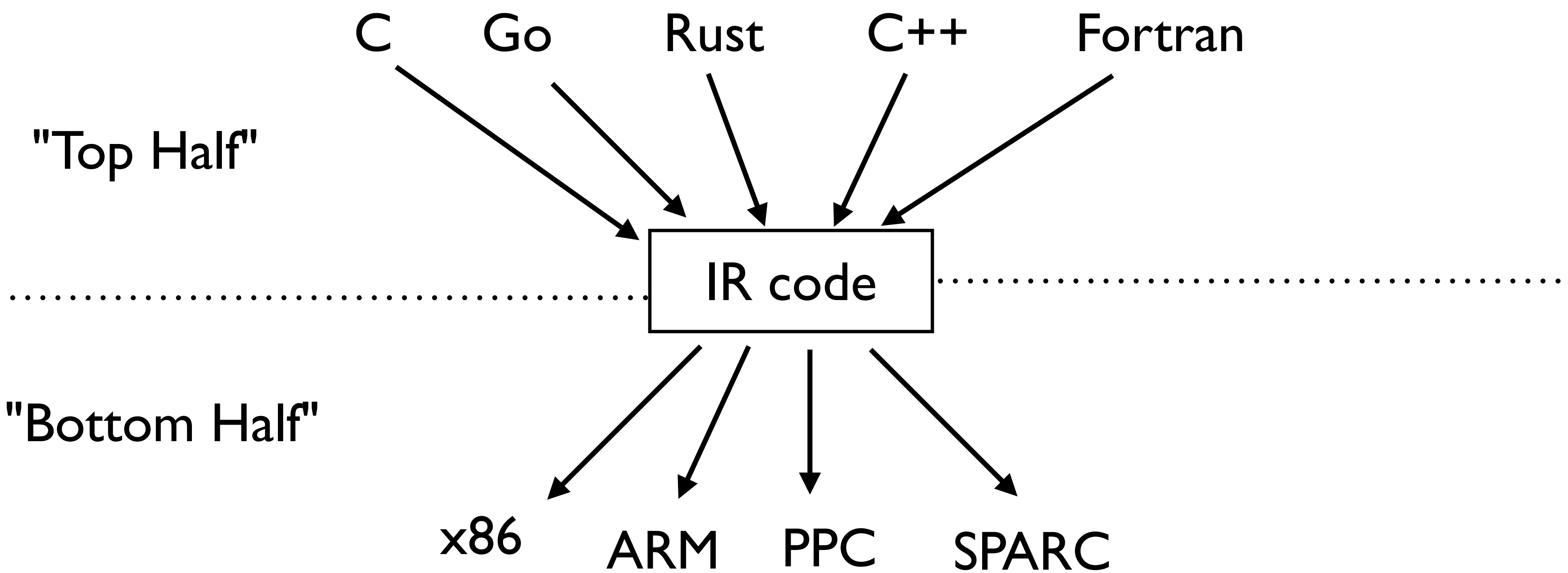
```
('i32.const', 2, 'r1'),
('i32.const', 3, 'r2'),
('i32.const', 4, 'r3'),
('i32.mul','r2','r3','r4')
('i32.add','r1','r4','r5')
...
```

translate



Metal

Compiler Design



IR Code

- What does IR code look like?
- It embodies a few essential concepts
  - A Model of Computation
  - Control Flow
  - Memory
  - Modules

Stack Machines

- Values are pushed on a stack
- Operations are carried out on the stack

```
2 + 3 * (10 - 2) + 5
```

```
i32.push 2     [2]
i32.push 3     [2, 3]
i32.push 10    [2, 3, 10]
i32.push 2     [2, 3, 10, 2]
i32.sub        [2, 3, 8]
i32.mul        [2, 24]
i32.add        [26]
i32.push 5     [26, 5]
i32.add        [31]
```

- No temporary registers. Just a stack.

Stack Machines

- Stack machines are quite simple
- Extremely common in practice
    - Python
    - Java JVM
    - .NET CIL
    - WebAssembly
- Most interpreters are stack-based

Register Machines

- Values are put into "registers"
- A register is a named storage location

```
2 + 3 * (10 - 2) + 5
```

```
r1 = 2
r2 = 3
r3 = 10
r4 = 2
r5 = r3 - r4      ; 8
r6 = r2 * r5      ; 24
r7 = r1 + r6      ; 26
r8 = 5
r9 = r7 + r8      ; 31
```

- All operations involve registers

Three-Address Code

- A common register-based IR code

```
r1 = 2                          ('i32.const', 2, 'r1')
r2 = 3                          ('i32.const', 3, 'r2')
r3 = 10                         ('i32.const', 10, 'r3')
r4 = 2                          ('i32.const', 2, 'r4')
r5 = r3 - r4                    ('i32.sub', 'r3', 'r4', 'r5')
r6 = r2 * r5                    ('i32.mul', 'r2', 'r5', 'r6')
r7 = r1 + r6                    ('i32.add', 'r1', 'r6', 'r7')
r8 = 5                          ('i32.const', 5, 'r8')
r9 = r7 + r8                    ('i32.add', 'r7', 'r8', 'r9')
```

- Encode instructions as tuples.
  - (op, src1, src2, target)

SSA Code

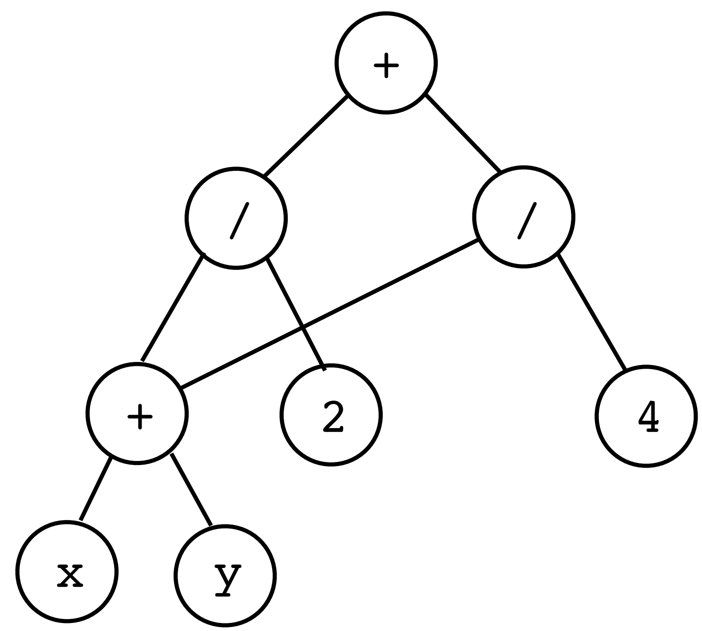
- Single Static Assignment
- A restriction of 3-address code
  - Infinite registers
  - Registers are immutable
- Also: Versioned variables

SSA vs. Stack

- SSA is a more powerful IR than a stack machine
- Enables certain optimizations

```
(x + y)/2 + (x + y)/4

r1 = load x
r2 = load y
r3 = r1 + r2    # (x + y)
r4 = r3 / 2
r5 = r3 / 4
r6 = r4 + r5
```

reuse r3



- Example: Expression trees -> Expression DAGs

Modeling Variables

- Programming languages have variables

```
const pi = 3.14159;

func area(radius float) float {
   var a = pi * radius * radius;
   return a;
}
```

- Variables appear in different contexts
  - Globals (outside of any function)
  - Locals (inside a function)
  - Parameters (inside a function)

Modeling Variables

- In IR, variables are often managed via tables

```
const pi = 3.14159;

func area(radius float) float {
   var a = pi * radius * radius;
   return a;
}
```

globals

0:	('pi', 'f64')
	...

locals

0:	('radius', 'f64')
1:	('a', 'f64')
	...

- Globals are shared by all
- Locals are per-function
- Think about scoping rules - it mirrors that

Modeling Variables

- In IR, variables are often managed via tables

```
const pi = 3.14159;

func area(radius float) float {
   var a = pi * radius * radius;
   return a;
}
```

globals

0:	('pi', 'f64')
	...

locals

0:	('radius', 'f64')
1:	('a', 'f64')
	...

- Instructions:

```
('load_global', slot)
('store_global', slot)
('load_local', slot)
('store_global', slot)
```

Modeling Variables

- In IR, variables are often managed via tables

```
const pi = 3.14159;

func area(radius float) float {
   var a = pi * radius * radius;
   return a;
}
```

globals

0:	('pi', 'f64')
	...

locals

0:	('radius', 'f64')
1:	('a', 'f64')
	...

- Example:

```
('load_global', 0, 'r1')      # "pi"
('load_local', 0, 'r2')       # "radius"
('f64.mul', 'r1', 'r2', 'r3')
('f64.mul', 'r3', 'r2', 'r4')
('store_local', 'r4', 1)      # "a"
```

Modeling Structures

- Structures also modeled via tables

```
struct Complex {
    real float;
    imag float;
}
```

Complex

0:	('real', 'f64')
1:	('imag', 'f64')

- Might require a more general memory access operation

```
('f64.load', base, offset, target)   # target=base[offset]
('f64.store', source, base, offset)  # base[offset]=source
```

- May also introduce concepts related to pointers

Modeling Control Flow

- Programming languages have control-flow

```
if a < b {
   statements
} else {
   statements
}

while a < b {
   statements
}
```

- Introduces branching to the underlying code

Basic Blocks

- Consecutive statements often appear in groups

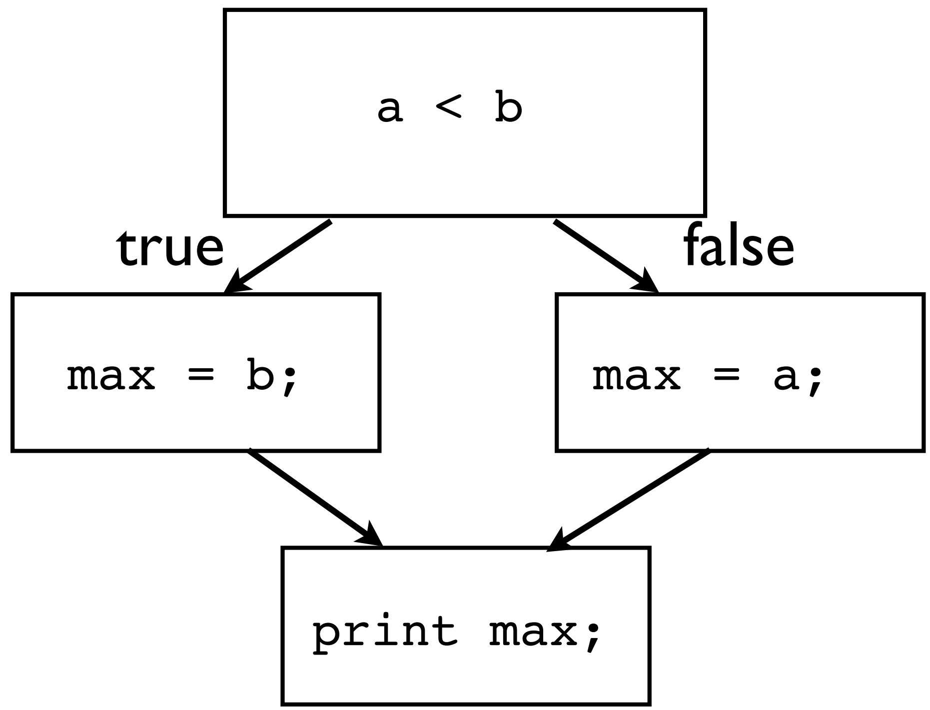
```
var a int = 2;
var b int = 3;
var c int = a + b;
print(2*c);
...
```

- A sequence of statements with <u>no change</u> in control-flow is known as a "basic block"

Control-Flow

- Control flow statements break code into basic blocks connected in a graph

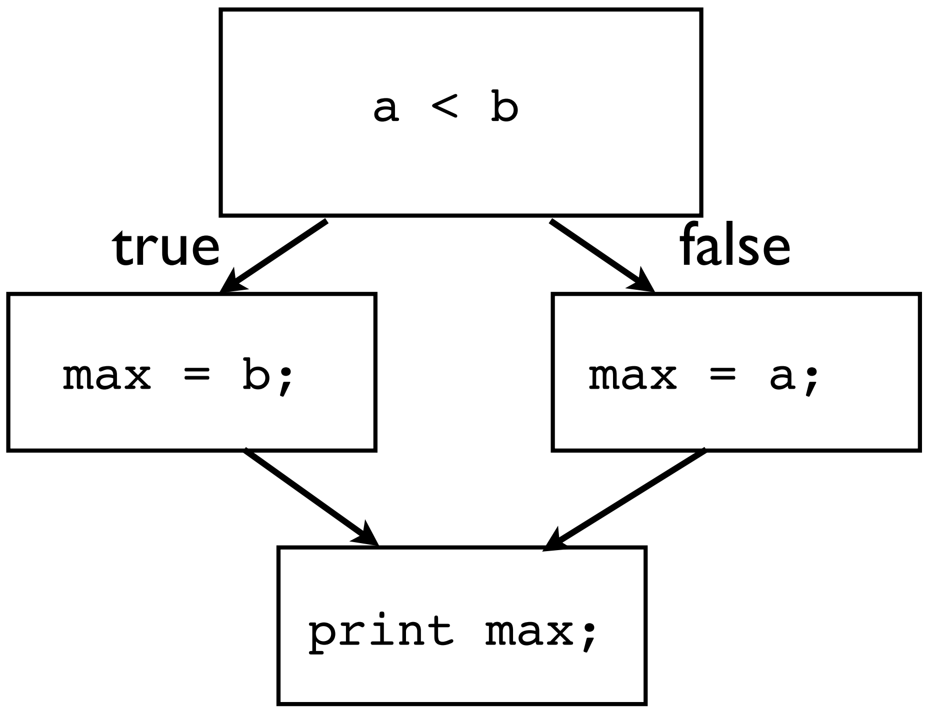
```
var a int = 2;
var b int = 3;
var max int;

if a < b {
   max = b;
} else {
   max = a;
}

print max;
```



- Control flow graph

Problem

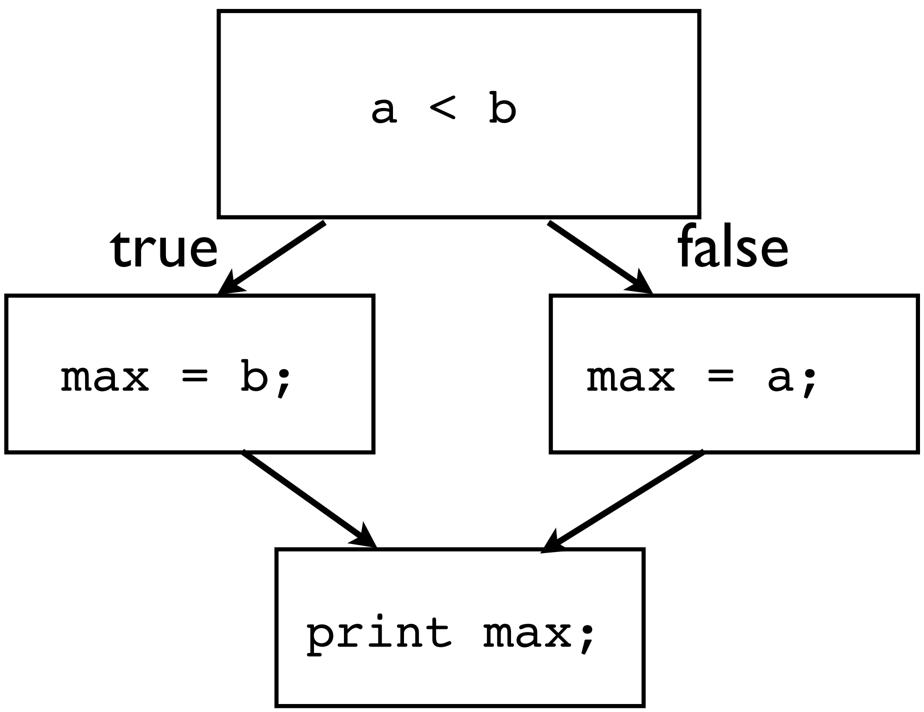
- How do you encode the control-flow graph into intermediate code?



- How is control-flow expressed?

One Approach: Gotos

- Label each block and emit jump/gotos




```
b1: test = a < b
    if (test) goto b2;
    goto b3;

b2: max = b;
    goto b4;

b3: max = a;
    goto b4;

b4: print max;
```

IR Example

```
if a < b {
   statements1
} else {
   statements2
}
statements3
```

Generated Labels

```
true_label = 'b2'
false_label = 'b3'
merge_label = 'b4'
```

current block

```
...
('load_local', 0, 'r1')
('load_local', 1, 'r2')
('i32.lt', 'r1', 'r2', 'r3')
('br_if', 'r3', 'b2', 'b3')

('label', 'b2')
... statements1 ...
('goto', 'b3')

('label', 'b3')
... statements2 ...
('goto', 'b4')

('label', 'b4')
... statements3 ...
```

Control-Flow Analysis

- There are many common programming errors related to control-flow issues
- Often a control-flow check is performed
- In addition to type checking.
- Will illustrate some common scenarios.

Inconsistent Return

- Missing return statement on one branch

```
func f(x int) int {
    if x > 0 {
        return x + 10;
    }
}

print f(-2);    // ?????????
```

- Must check that all valid control-flow paths leads to a proper function return

Inconsistent Return

- It's tricky...

```
func f(x int) int {
    if x > 0 {
        return x + 10;
    } else {
        return x - 10;
    }
}

print f(-2);    // Good
```

- In this example, there's no return at the end, but both branches of conditional return

Dead Code

- There might be statements that never execute

```
while n > 0 {
    if n == 5 {
        break;
        print "Done!";     // <<<< Never executes
    }
    n = n - 1;
}
```

- Should it result in a compiler warning?

Uninitialized Variable

- What is the value?

```
var z int;
print z;
```

- Or this...

```
var z int;
if x > 0 {
    z = 10*x;    // Only initialized on one branch
}
print z;
```

Unused Variable

- What about this?

```
var x = 42;
var z = x + 10;   // z never reference ever again
...
<END>
```

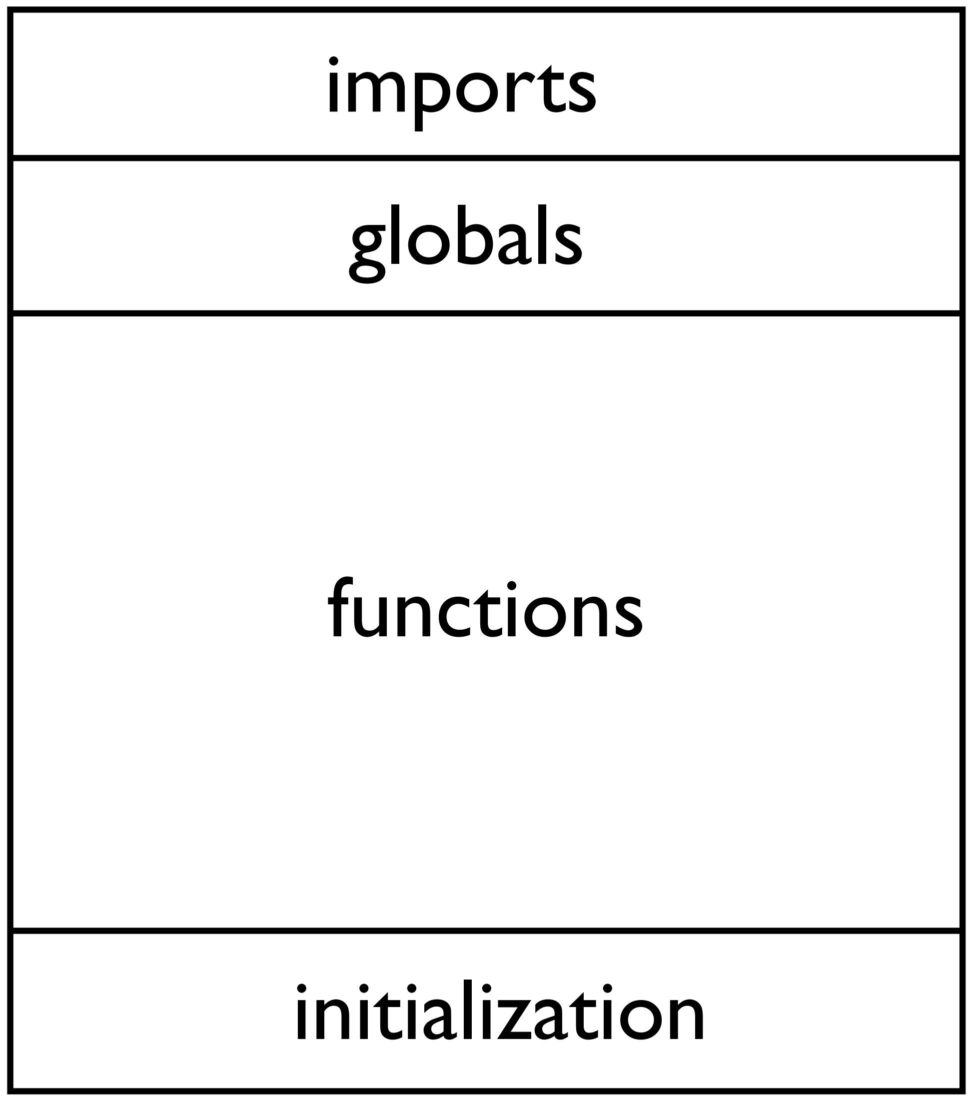
- Does the compiler see the lack of use?
- Note: Such problems often the domain of linters/code checkers.

Packaging of IR Code

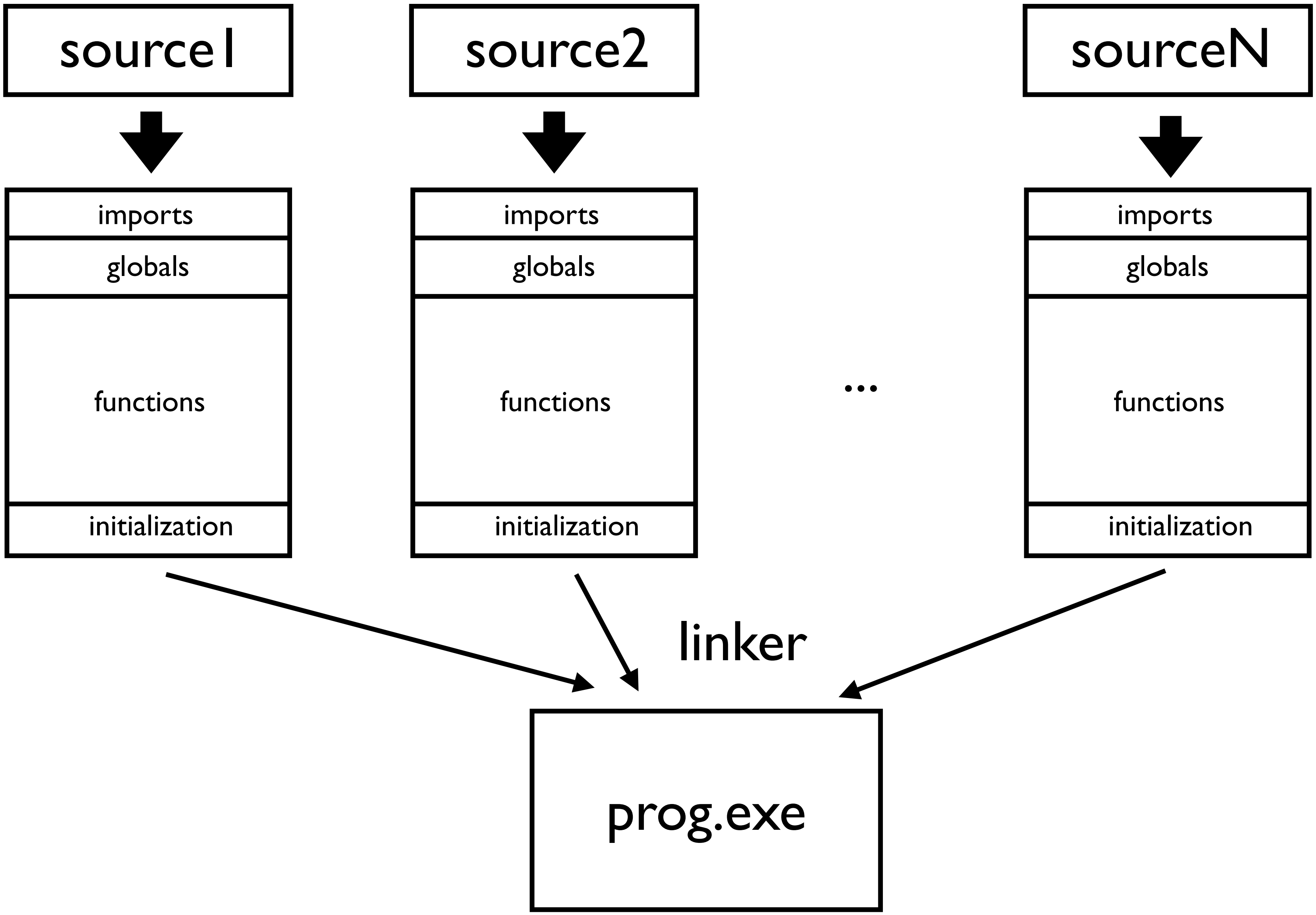
- The final product of IR is a "code module"
- Think Python modules
- A module is an object that contains
  - Global variables/definitions
  - Functions
  - Imports/Exports
  - Initialization

Packaging of IR Code

IRModule



Separate Compilation



Project

- Turn Wabbit into IR code
  - See `wabbit/ircode.py`
- Challenge: Can you also write an IR simulator?