Compiler Design

Transformations of programs into executables; and: implementing features of object-oriented programming

Sydney Newmark
Independent Study

Contents

1. Assembly conventions

2. Register allocation

3. Modular code generation

4a. Garbage collection

4b. Supporting object-oriented concepts

5. Demo

6. Reflection

1. Assembly conventions

not knowing intricacies of operating system conventions beforehand caused me great pain and probably hours of extra work

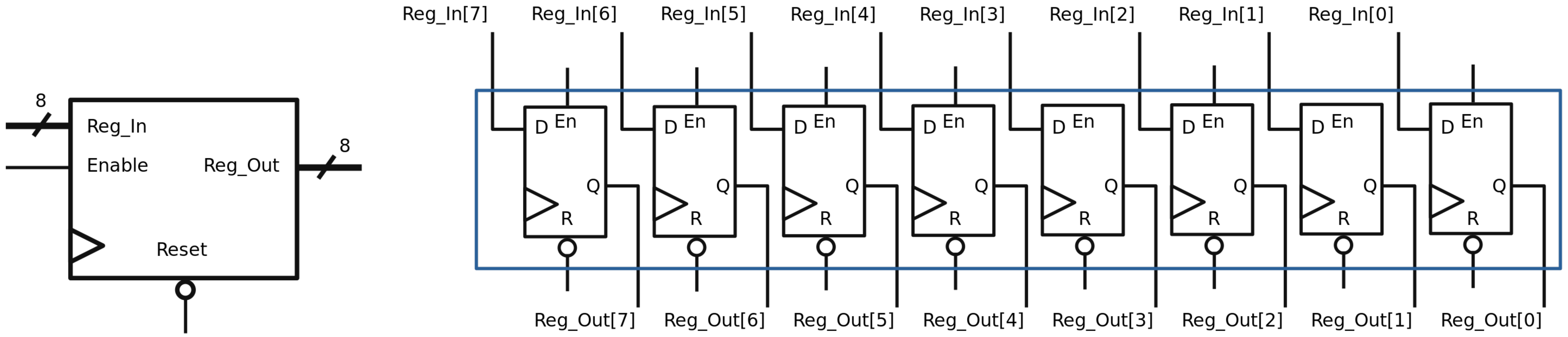
x86_64 and ARM64

A brief introduction

- two widely used processor architectures
- CISC vs. RISC
    - many complex, specific instructions to perform certain tasks in minimal assembly instructions
    - specific complex tasks comprise of multiple foundational instructions
- initially the compiler was written for x86_64 but later switched to ARM

ARM64

Register organization

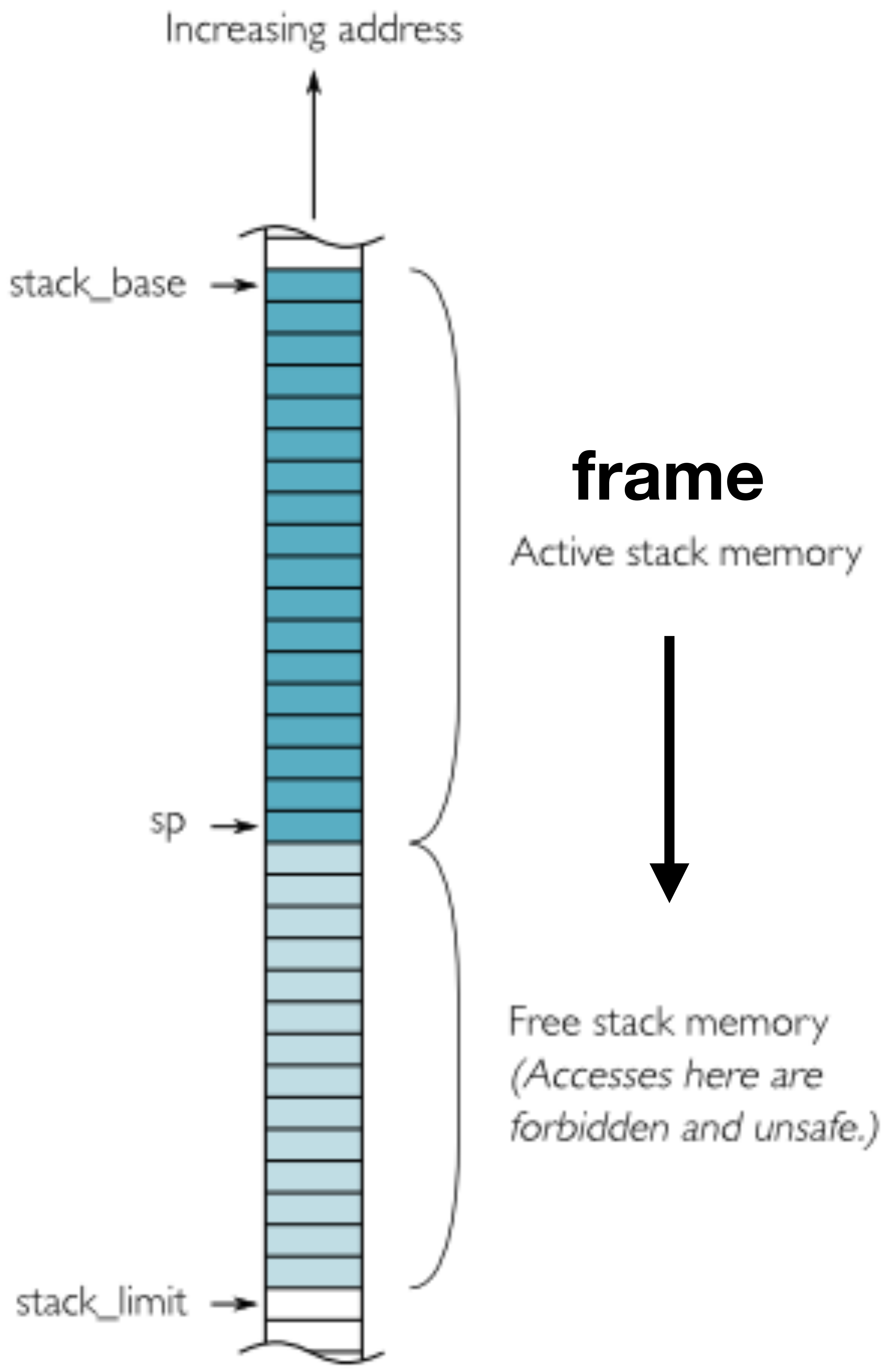


- thirty-one 64-bit general-purpose registers (**x0**, **x1**, …, **x30**)
  - bottom halves accessible with the **w** prefix
  - by convention, **x29** is the frame pointer
  - **x30** is the **link register**
- one dedicated stack pointer (**sp**) or zero register (**xzr**)

x29

What is a frame, anyway?

- **to reserve $N \cdot w$ bytes of memory, where:**
  - N is the amount of “numbers” to store
  - w is the word size of the arch (usually eight bits)
  - “align” (round up) to next multiple of 16
- offsets from **x29** are always positive
- creating frames **decreases** the stack pointer and cleaning up frames **increases** stack pointer



PC and LR

Navigating programs and finding home

- program is laid out in sequential bytes in memory
- **pc** (program counter) controls the currently executing instruction
- the **link register** (**x30** in A64) points to the next instruction after a particular subroutine returns
- LR is more efficient particularly in the case of leaf subroutines

An IBM 701 because I could’t find a useful diagram
(the program counter is in the lower left)

Calling conventions

Callee-save? Caller-save?

- problem: what if two functions want to use the same set of registers?
- callee: the subroutine being called (i.e. some function or method)
- caller: the subroutine calling a function
- **callee-save**: registers to be saved by the callee if used by callee
- **caller-save**: registers to be saved by the caller if used after the callee returns

In x86: r12, r13, r14, r15, rbx, rsp, rbp **In A64:** x19-x28

```
def foo():
  bar()
def bar():
  pass

foo()
```

Parameters

Providing subroutines with data

By convention:

- **x0-x7** are argument registers for “normal” 64-bit integers
- **x0** doubles as the register which contains the **return value**
- vector data and floating-point data use separate conventions

source register

destination register

```
mov x0, T1

bl _println
```

named subroutine

Example

A working function prologue

reserve stack region	`sub sp, sp, #112`
save LR and stack pointer	`stp x29, x30, [sp, #-16]!`
	`add x29, sp, #16`
save callee-save registers	`str x19, [x29, #8]`

store registers in stack frame

Example

A working function epilogue

recover callee-saved registers	`ldr x19, [x29, #8]`
recover LR and stack pointer	`ldp x29, x30, [sp], #16`
release stack region	`add sp, sp, #112`
	`ret`

2. Register allocation

Problem: too many temporaries

T87??

- **temporary**: a placeholder name denoting a variable being stored in a register
- finite number of registers (~30) and potentially hundreds of temporaries
- useful observation: the “lifetime” of most of these temporaries never overlap
- analysis useful to determine lifetimes of temporaries and which lifetimes interfere

```
mov T3, T14
str T2, [T3, #8]
mov T15, #1
str T15, [T3, #16]
mov T16, #2
str T16, [T3, #24]
```

Graphing control flow

to discover where the data flow

- figure out where **uses** and **definitions** of temporaries are
- **backtrace** from each **use** to each **definition** to determine lifetimes
    - each backtrace is the lifetime of a temporary
- when backtraces overlap, temporaries interfere

```
mov T3, T14
str T2, [T3, #8]
mov T15, #1
str T15, [T3, #16]
mov T16, #2
str T16, [T3, #24]
```

Register interference

x19 can’t hold all the ~~world’s~~ program’s ~~data~~ variables

- **T3** cannot use the same register as either **T15** or **T16** because both are in use at the same time
- however, **T15** and **T16** can use the same register because their **live ranges** do not overlap
- also called **liveness analysis**
- alternatively: use **dataflow equations** (complicated)

```
mov T3, T14
str T2, [T3, #8]
mov T15, #1
str T15, [T3, #16]
mov T16, #2
str T16, [T3, #24]
```

Coloring

no, there aren't actually colors

a simple approach:

- iterate through each statement
- for each temporary live at that statement:
  - if no color assigned:
    - determine which registers have been assigned to temporaries which interfere with the current one
    - pick a register that isn't in the above set

```
mov T3, T14
str T2, [T3, #8]
mov T15, #1
str T15, [T3, #16]
mov T16, #2
str T16, [T3, #24]
```

Simplify, Spill, Select

A principled coloring mechanism

- **Simplify**: repeatedly remove (and push onto a stack) nodes of degree **less** than K from a graph G, decreasing the degrees of other nodes and creating more opportunity for simplification

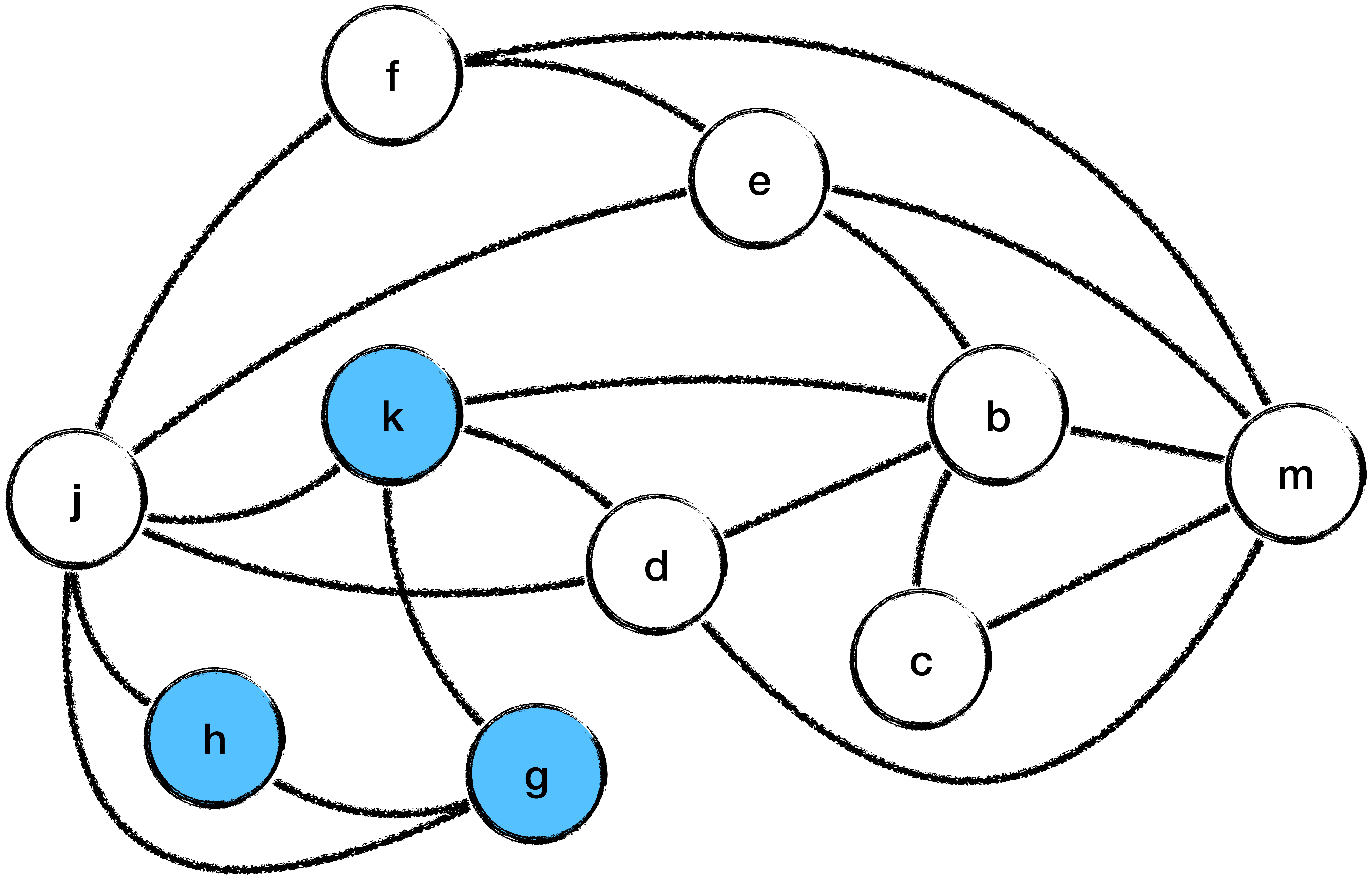
- **Spill**: if graph G containing nodes (instructions) contains only nodes of degree K then **Simplify** fails

    - mark a node for representation in memory rather than a register

- **Select**: assign colors to each node, rebuilding the original graph by adding a node from the top of the stack

Example

Simplify, Spill, Select

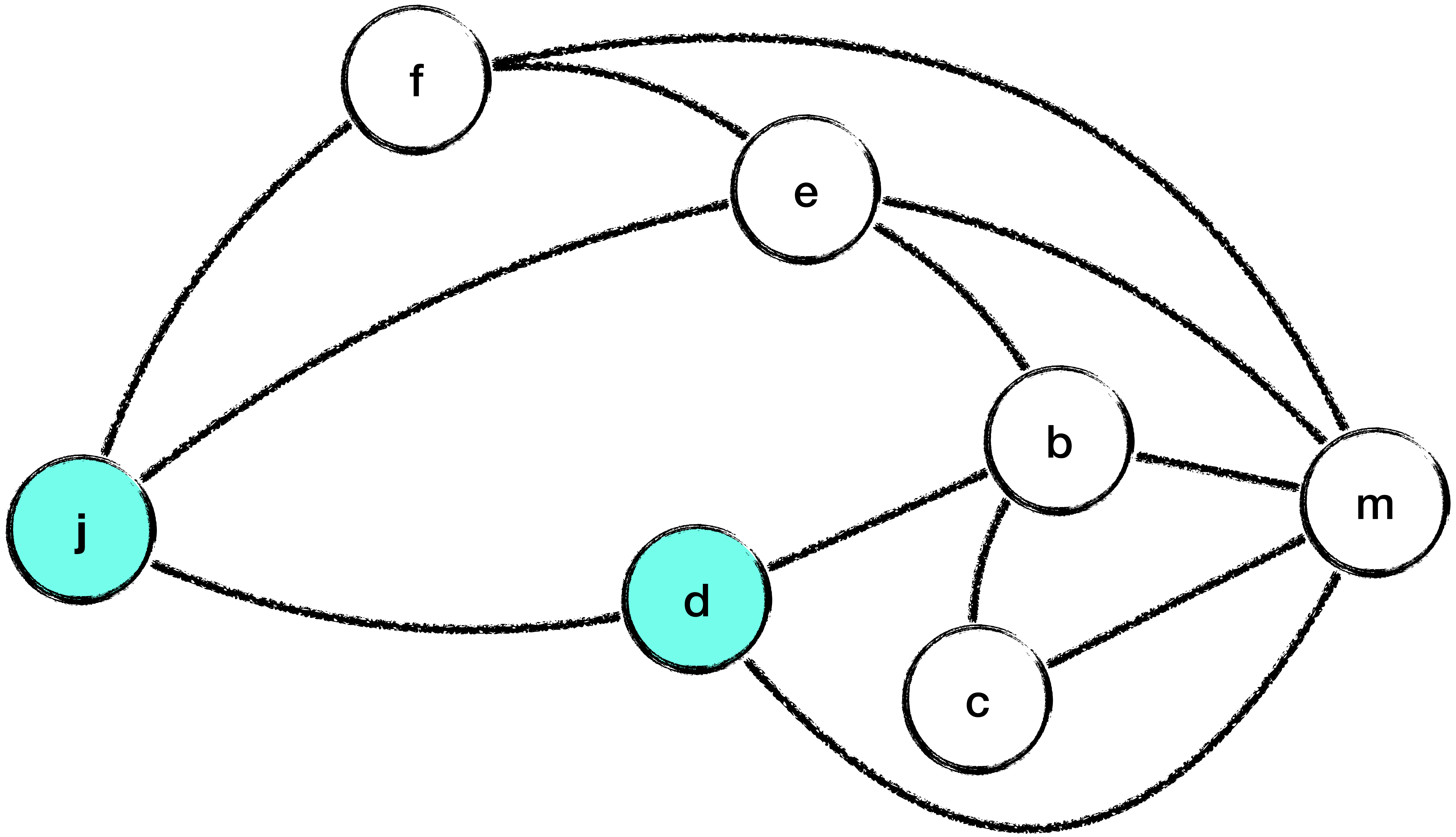


interference graph

g	
h	
k	

Example

Simplify, Spill, Select

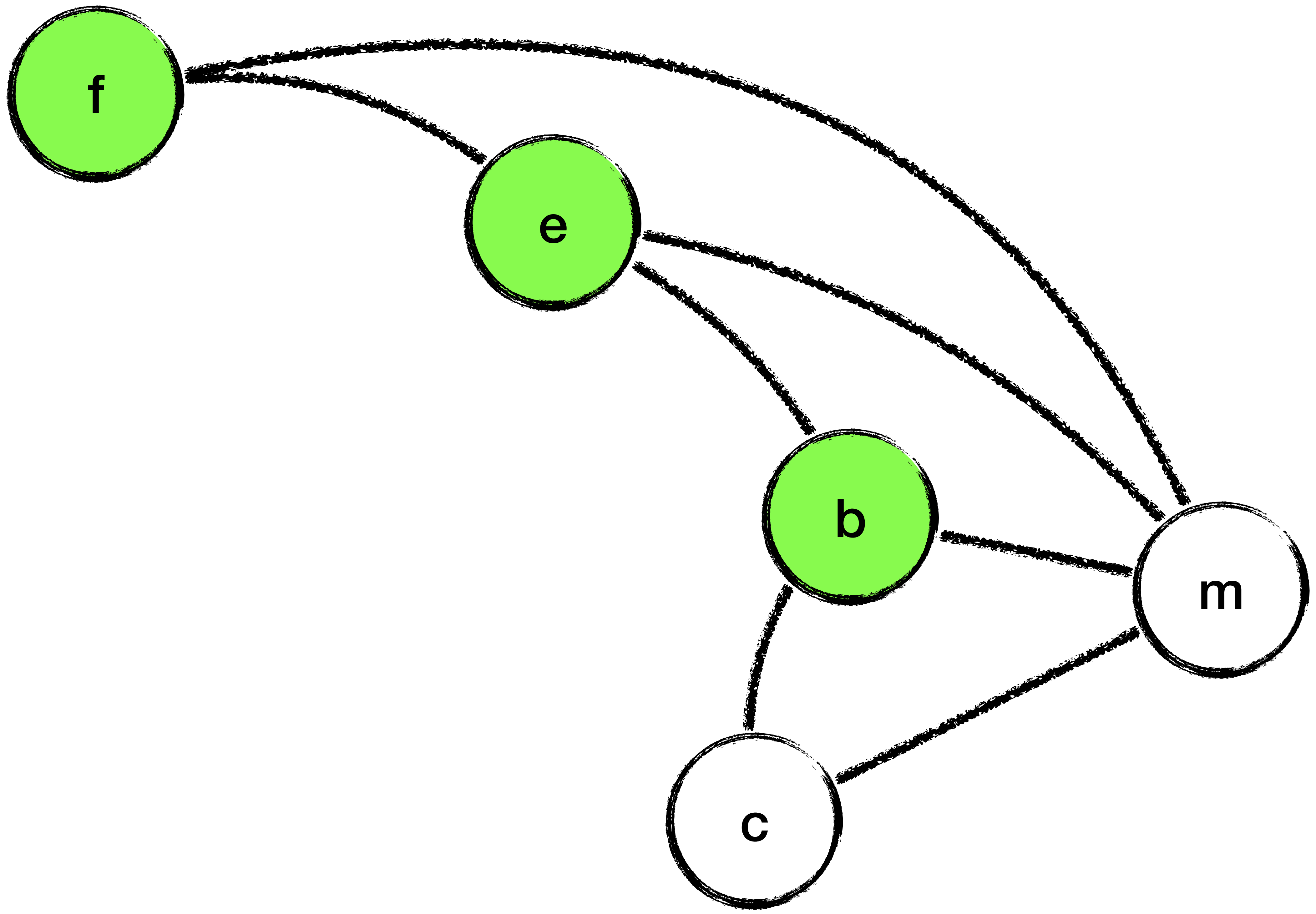


interference graph

g	
h	
k	
d	
j	

Example

Simplify, Spill, Select

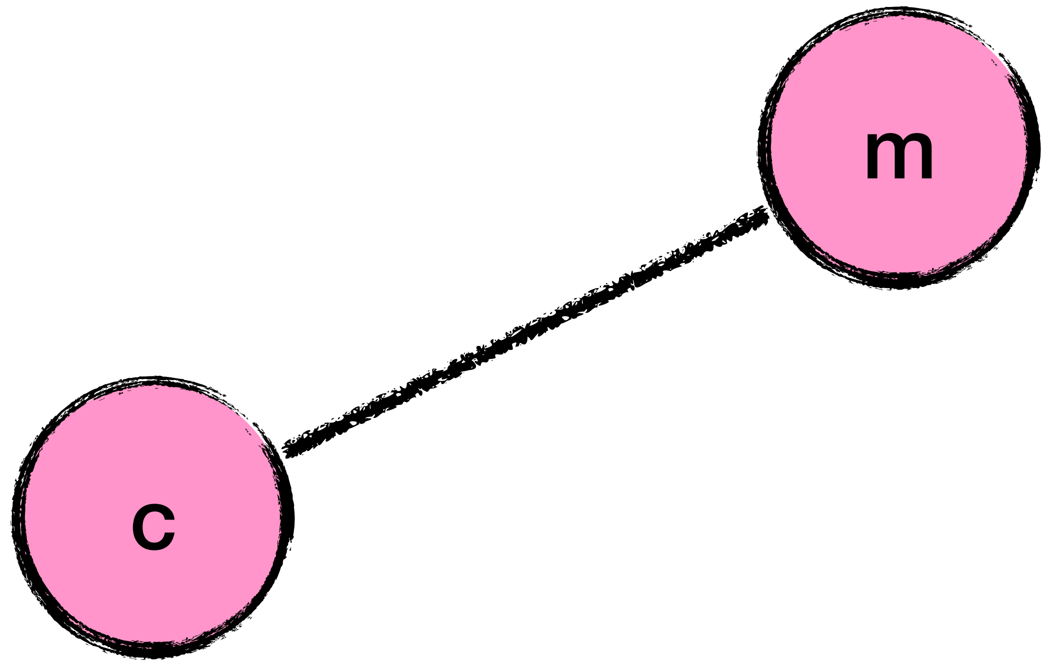


interference graph

g	
h	
k	
d	
j	
e	
f	
b	

Example

Simplify, Spill, Select

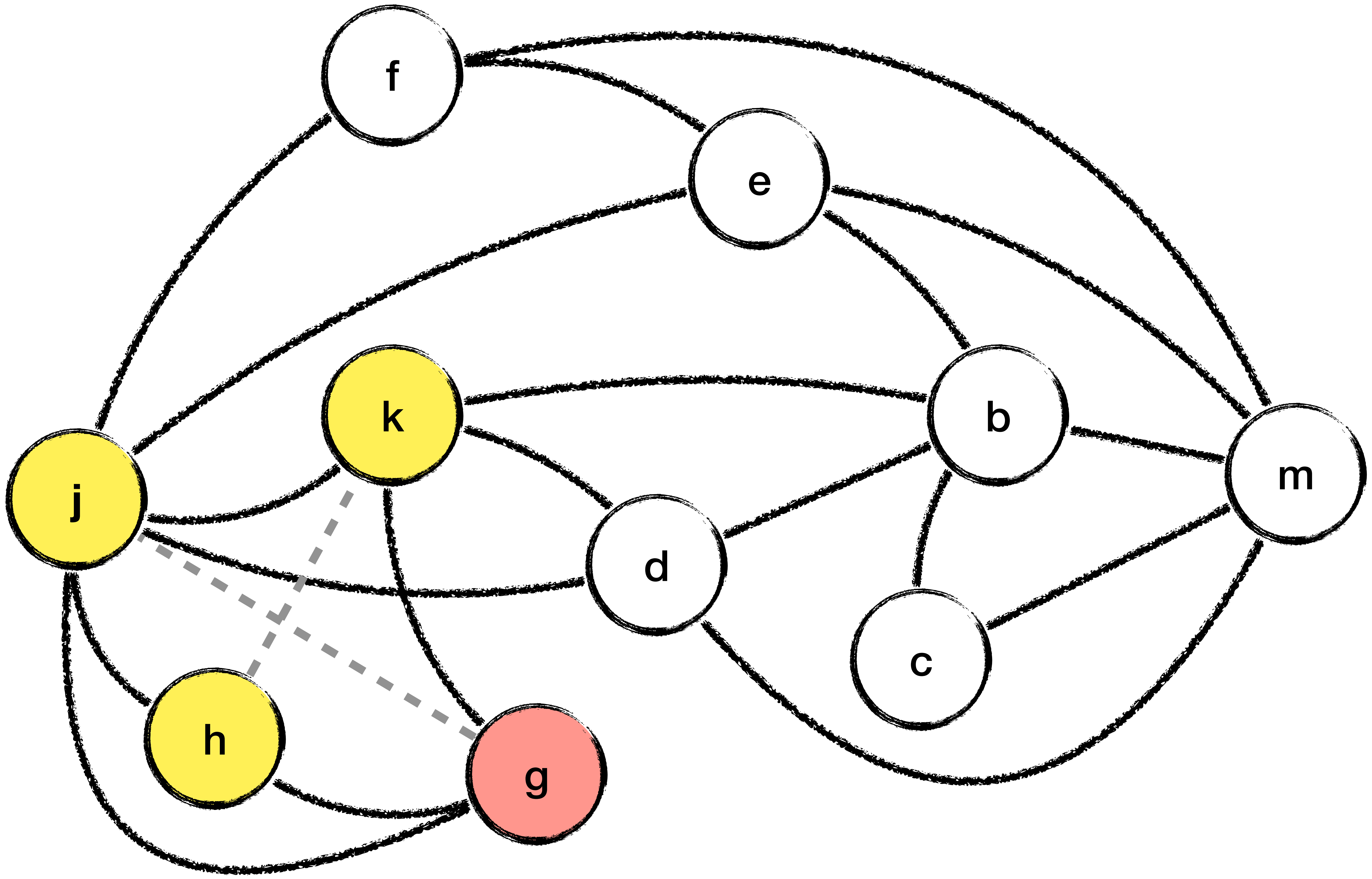


interference graph

g	
h	
k	
d	
j	
e	
f	
b	
c	
m	

Example

Simplify, Spill, Select

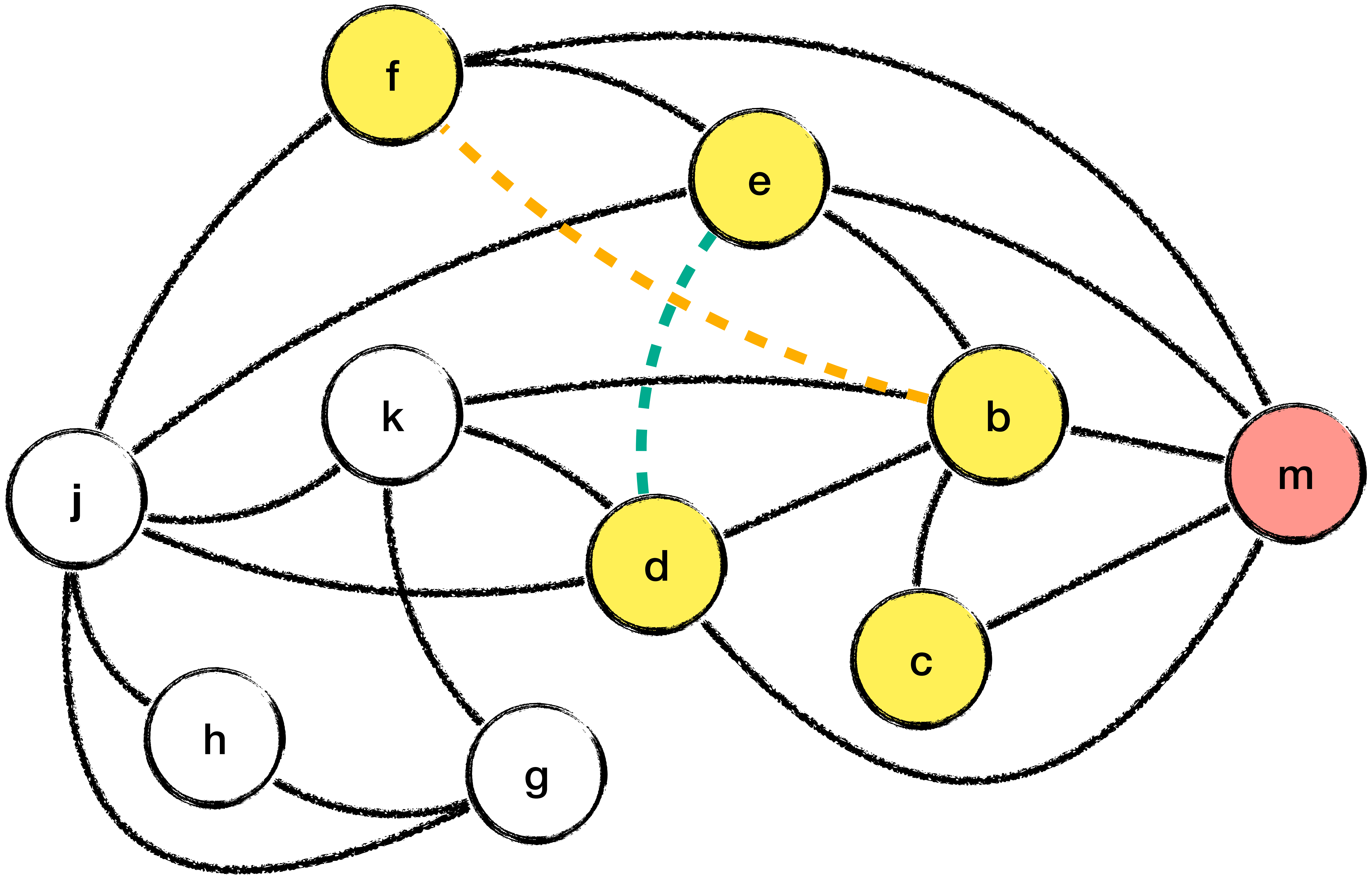


interference graph

g	4
h	2
k	1
d	4
j	3
e	4
f	2
b	2
c	3
m	1

Example

Simplify, Spill, Select



interference graph

g	4
h	2
k	1
d	4
j	3
e	4
f	2
b	2
c	3
m	1

3. Modular code generation

So many architectures

it’s good to be ~~lazy~~ efficient

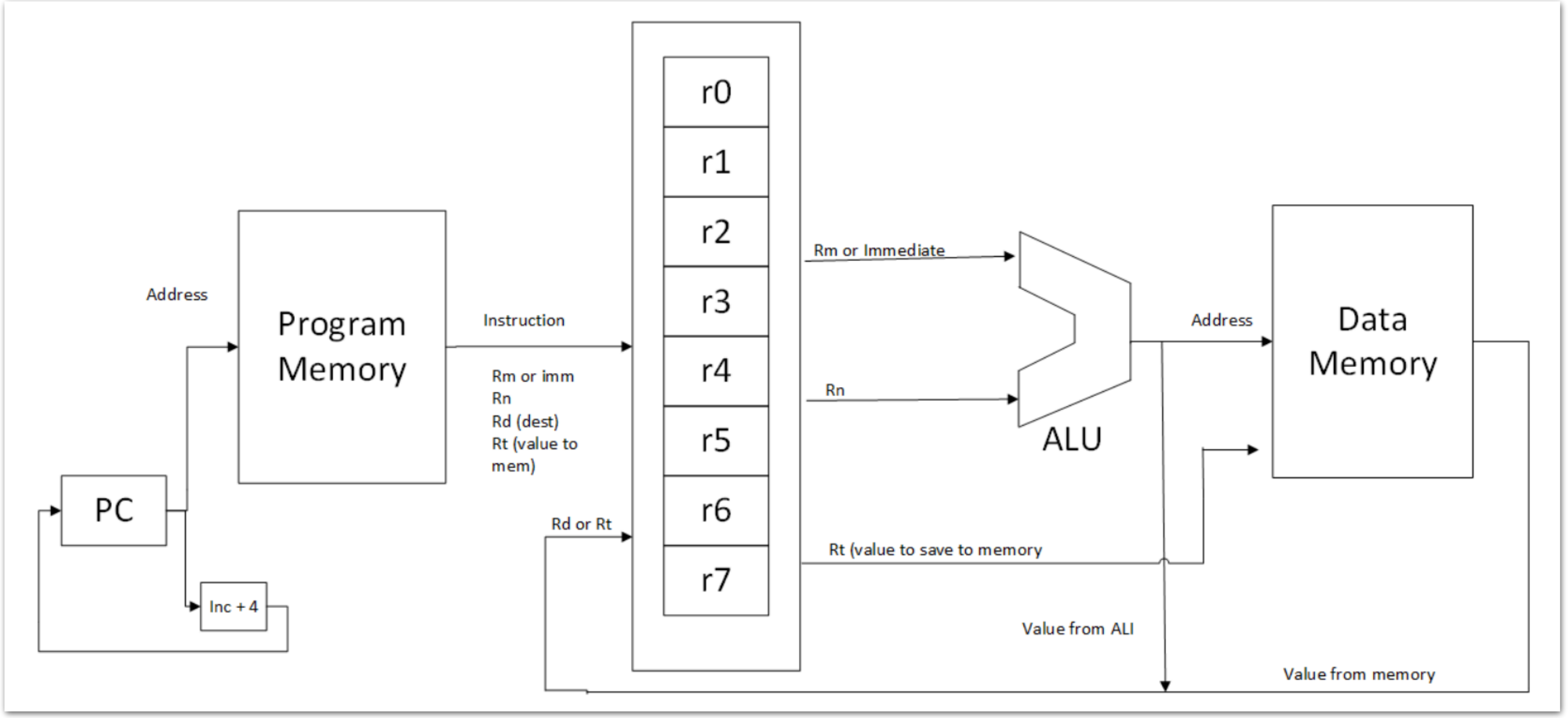
- different architectures have different ways of doing things
- x86 instructions tend to do more than their A64 counterparts
- what instructions are allowed?
- what arguments are allowed for those instructions?
- what are the equivalents from one platform to the other?

? ?

A load/store architecture

ldr and str

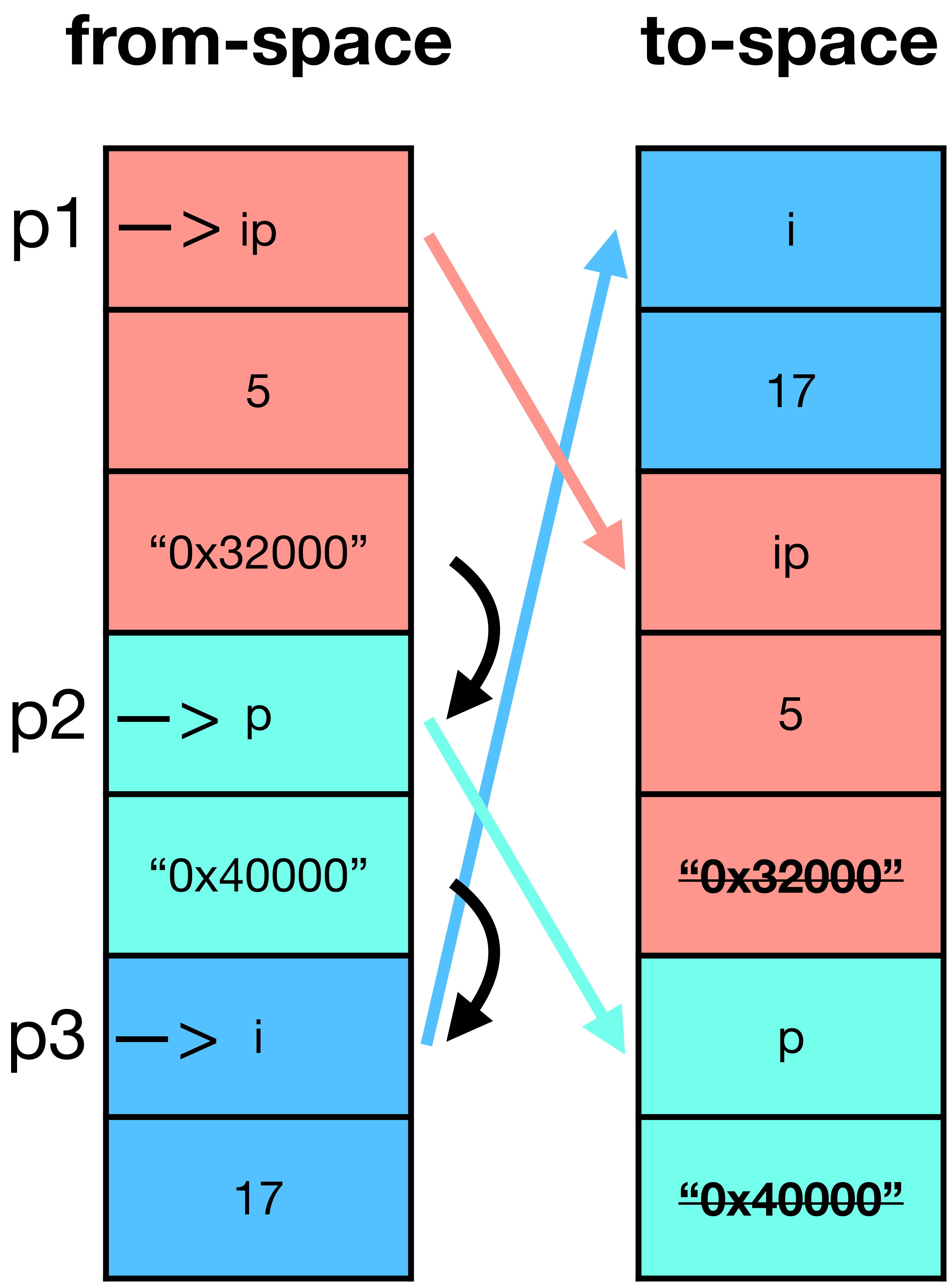
- unlike x86, ARM instructions normally only operate on registers
- data are **loaded** from memory, operated on in registers, and then **stored** to memory
- direct translation from x86 to ARM mnemonics and instruction syntax insufficient
- some processes (accessing/moving around pointers to strings) are handled differently



4a. Garbage collection

Copying collection

- scan the necessary portion of stack to find pointers to valid records
- **breadth first copying**: iterate through each of these pointers and shallowly forward it to to-space
- keep track of fields with pointers to a record, and after all shallow forwards, update these field pointers to point to **to-space**
- poor **locality of reference** alleviated by Cheney's algorithm



order of copy after scanning: p3, p1, p2

Reference counting

- instead of finding what is reachable, instead keep track of how many pointers point to a record; this is called a *reference count* (r)
- when r is zero, put the record on the free list and decrement the reference count of everything that pointed to that record
- can cause memory leaks because cycles cannot be reclaimed
  - if two records mutually point to each other, the reference counts will never be zero even if neither record is accessible
- expensive in terms of instructions required (mitigated somewhat by dataflow analysis)

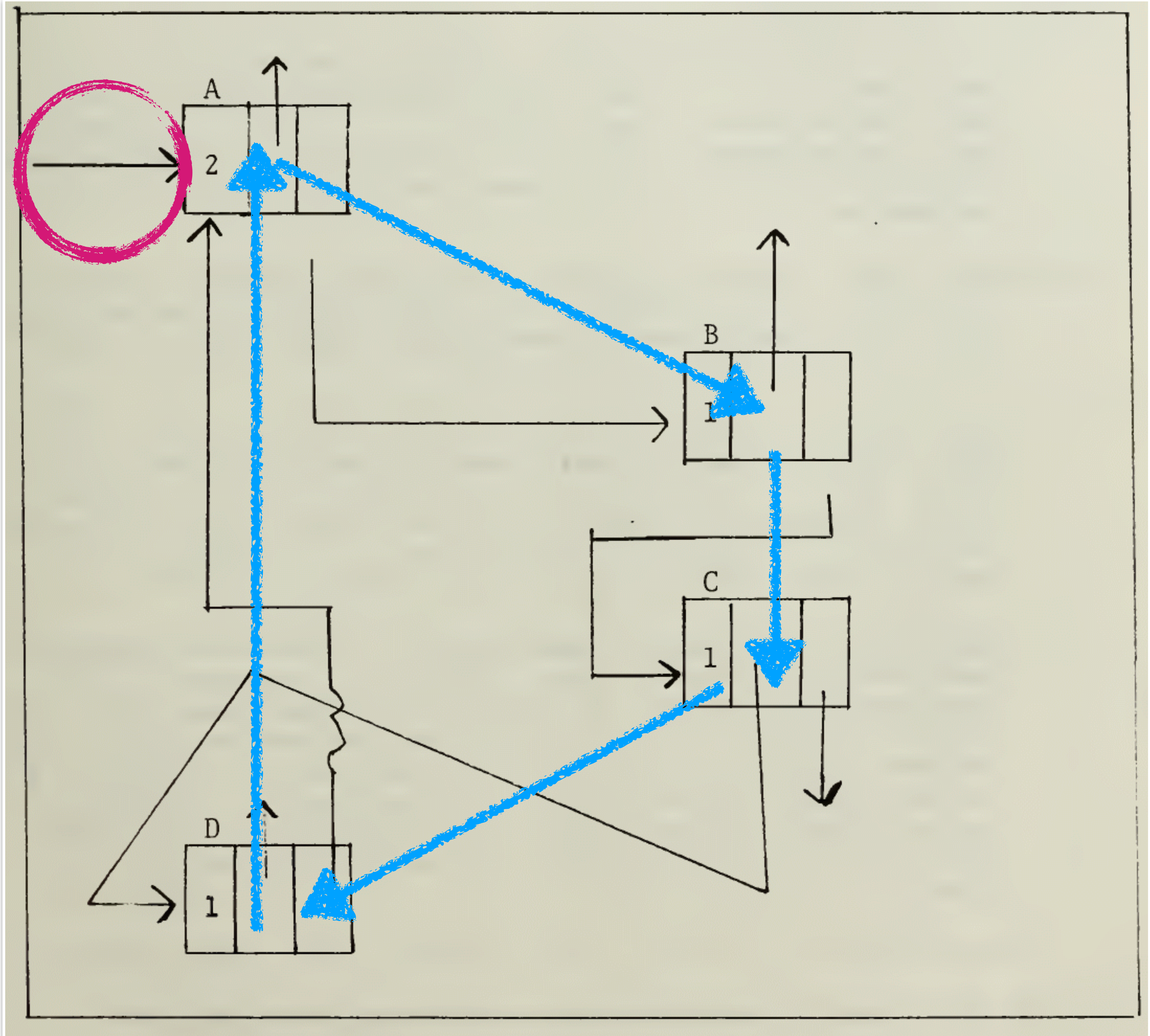
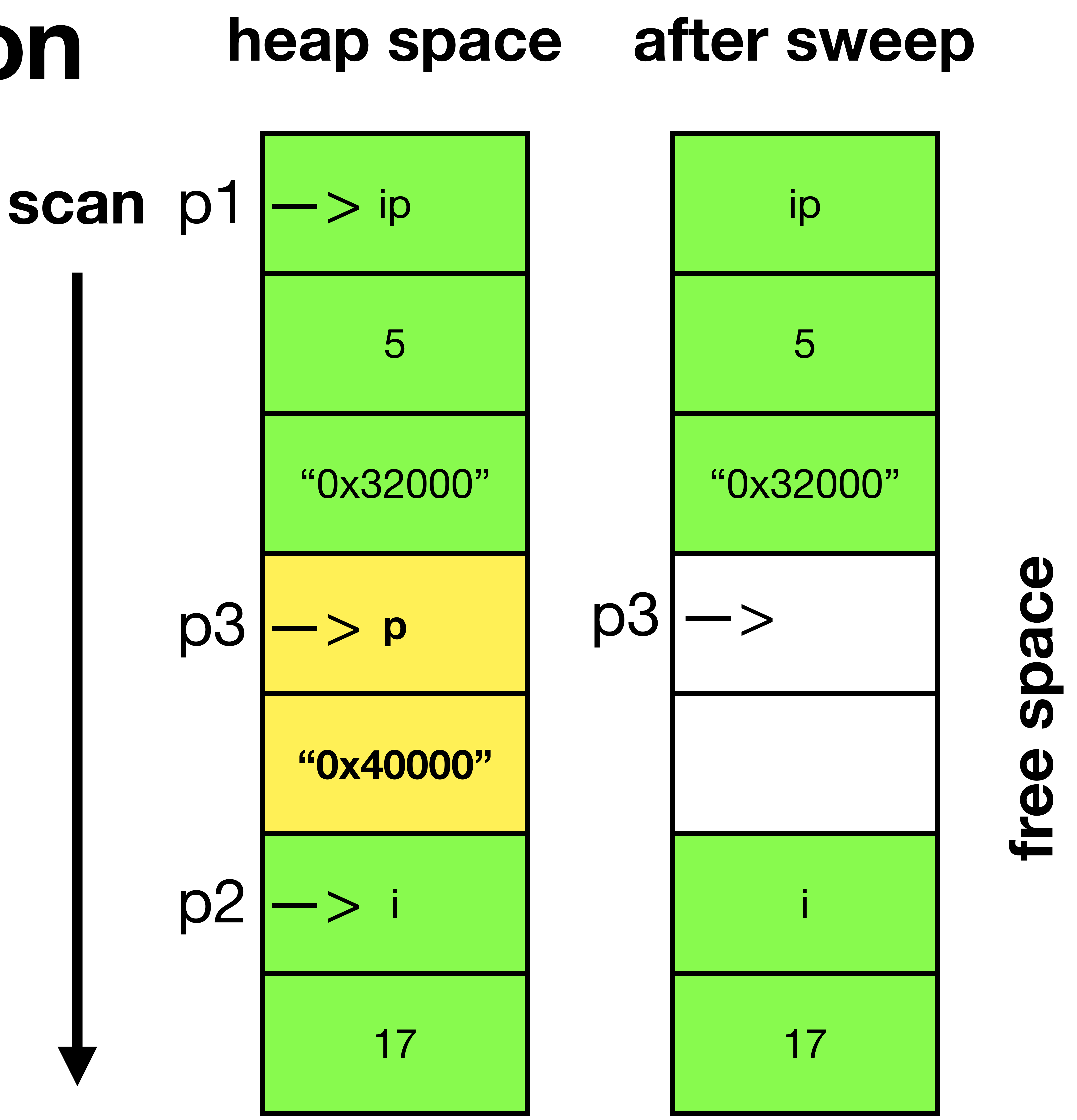


Figure 2.3 Circular List Structure.

cycle shown in **blue**

Mark-and-sweep collection

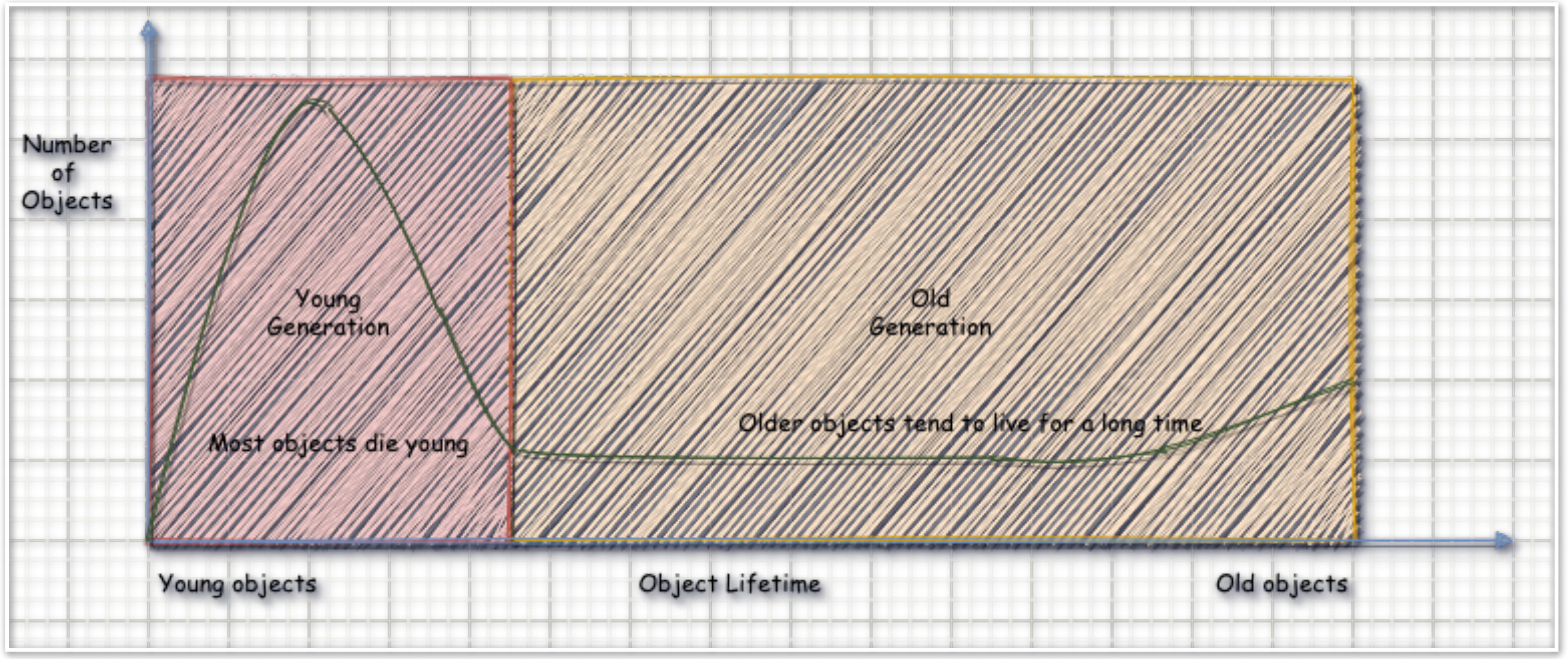
- **mark** phase and **sweep** phase
- **mark phase**: for each root pointer, perform depth-first search to mark all reachable records
- **sweep phase**: scan the entire heap looking for unmarked (garbage) nodes; link these nodes together in a linked list (also known as **freelist**)
  - unmark all previously marked nodes to prepare for next garbage collection
- **time complexity**:
  - DFS takes time proportional to the amount of reachable data
  - sweep phase takes time proportional to the size of the heap



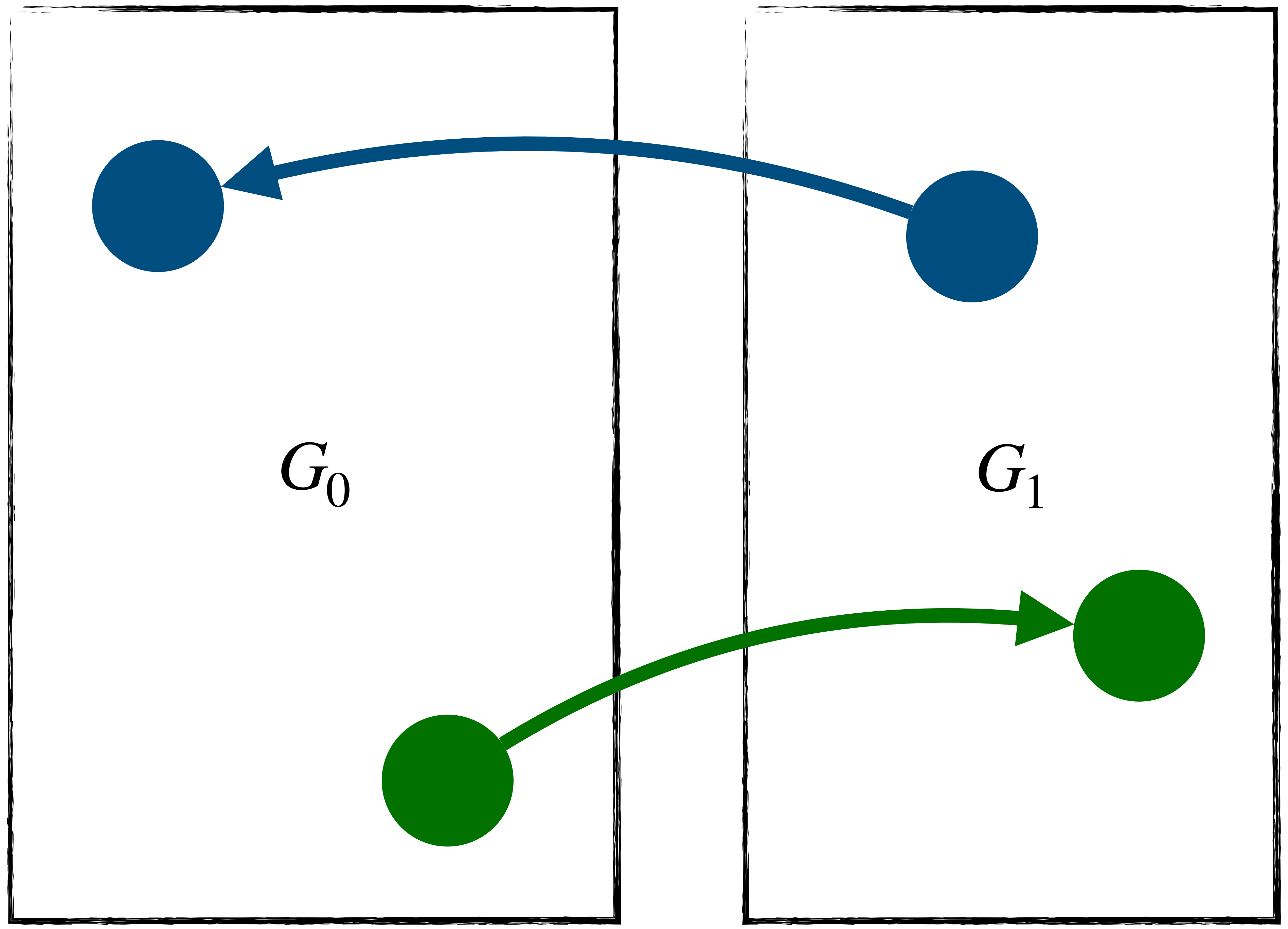
marked reachable nodes shown as **green**

Generational collection

- newly created objects likely to die soon
- objects still reachable after a long period of time will probably survive for many more collections
- concentrate efforts on “young” data
    - known as generations
- G_0 contains youngest and every object in each successive generation is older than any object in the previous generation
- **rare** for **older object to point to younger object**; **common for younger object to point to older object**
    - immediate field initialization (those values are necessarily older than the newly created object)
    - can keep track of this rare occurrence in several ways

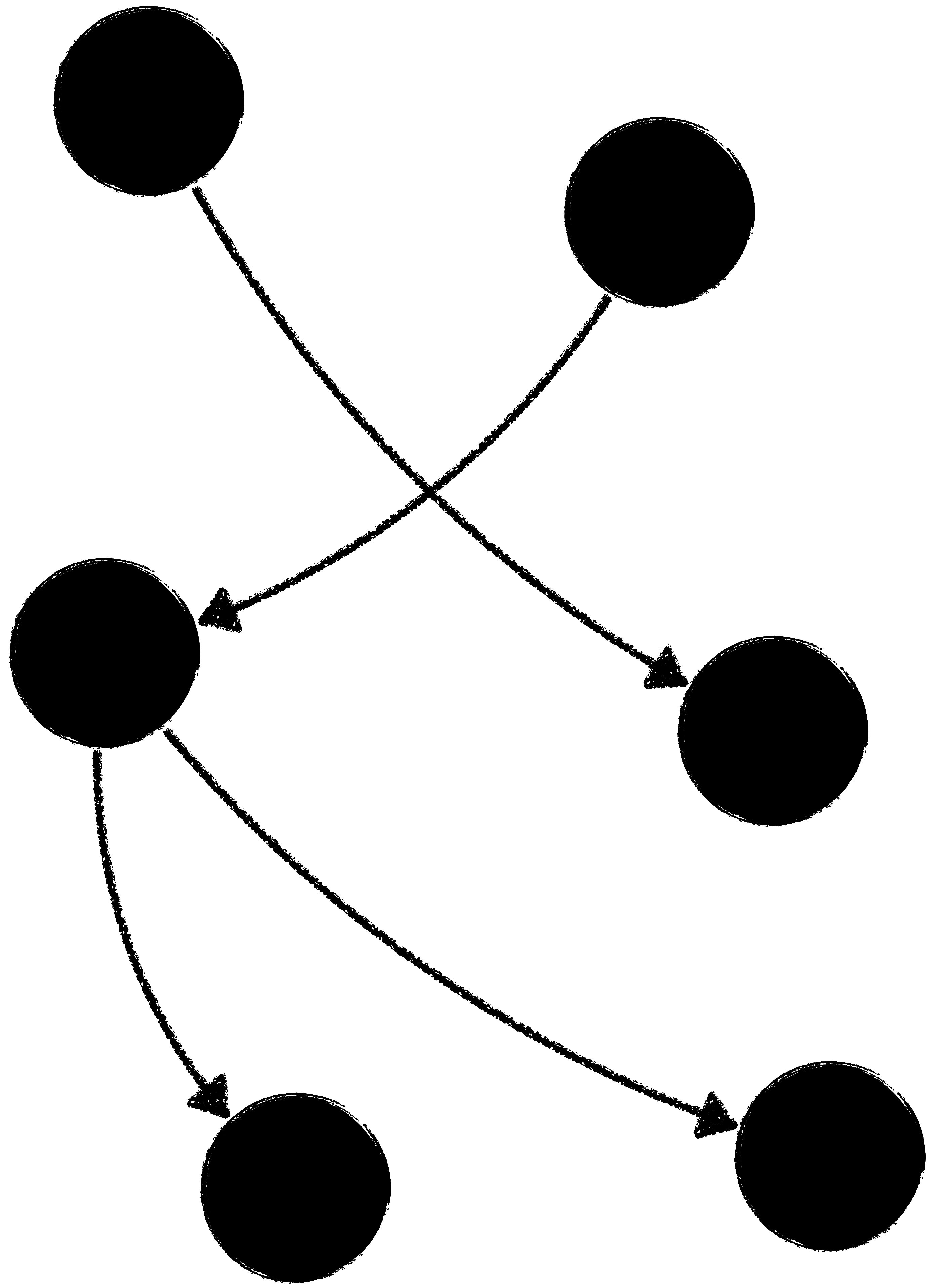


Distribution of object lifetimes



Incremental collection

- three “colors” to denote the status of an object
- nodes unvisited by depth-first or breadth-first search are **white**
- visited (i.e. marked or copied) nodes whose children have not been examined are **gray**
- visited nodes whose children have also been marked are **black**
- when there are no gray objects, all white objects must be garbage
- **two invariants:**
    1. no black object points to a white object
    2. every gray object is in the collector’s data structure (stack/queue) to be scanned
- **write-barrier** and **read-barrier** techniques to preserve invariants
    - <u>example</u>: whenever a program stores a white pointer a into a black object b, it colors b gray



4b. Supporting object-oriented concepts

Class descriptors

Runtime method and field layout

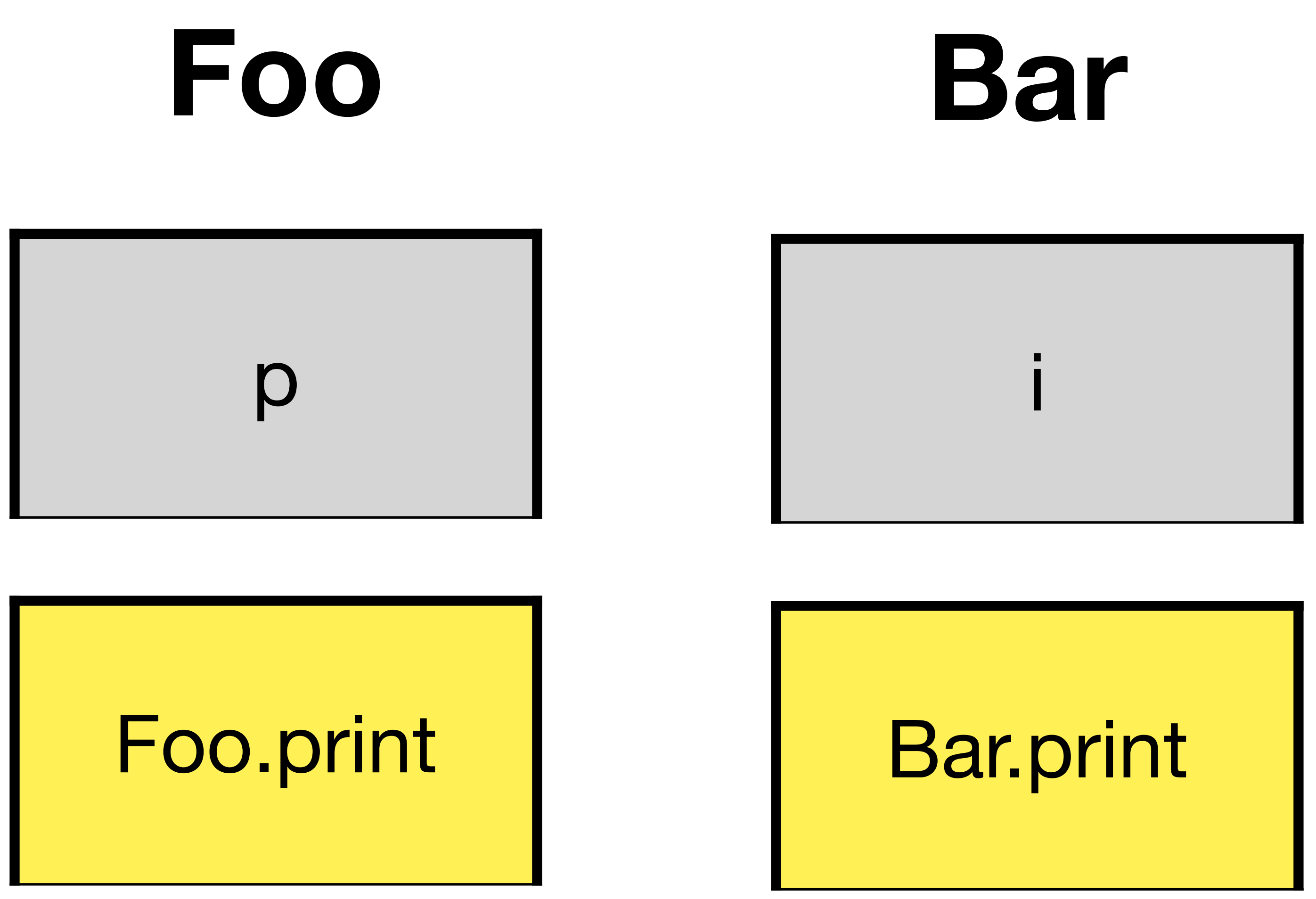
- garbage collection requires the runtime know which fields are pointers and which are non-pointers (integers, floating-point, constant strings)
- **field descriptor**: a string of i (non-pointer) and p (pointer)
    - first char in string corresponds to the type of the first field, and so on
- **method descriptor**: an array where each element corresponds to the address (i.e. assembly subroutine label) which is “active” for a particular class variable

```
class Foo<T: Print>: Print {
    val: T

    fun print(self) {
        self.val.print();
    }
}

class Bar: Print {
    x: int

    fun print(self) {
        println_int(self.x);
    }
}
```

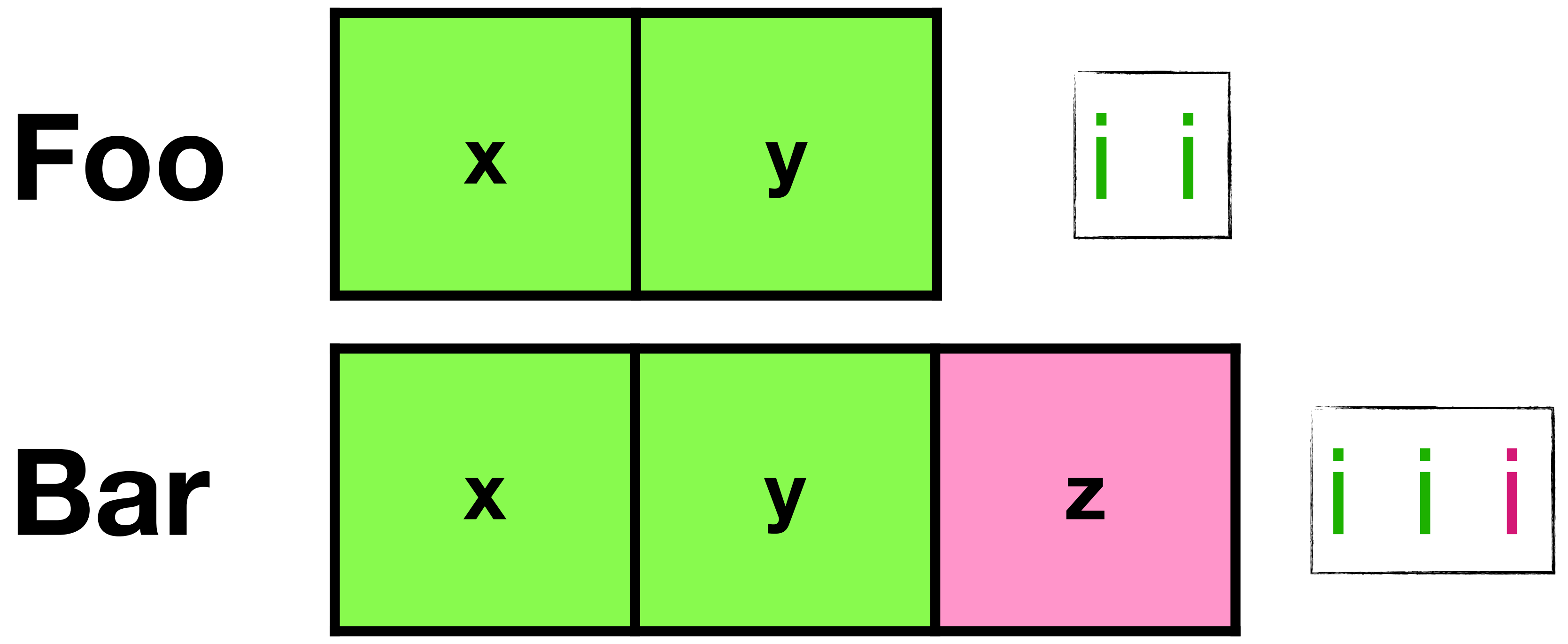


Field Prefixing

(Single) Inheritance and Generics

- all fields of a child class are placed after the fields of the parent class in a record
- casting allowed because accessing any field on a **Foo**-typed variable is valid even if the underlying type is **Bar**
- Generics are very similar to casting
  - i.e. every instance of some type **T** has the same underlying structure

Field Layout




```
class Foo(int x, int y)

class Bar(int z)
extends Foo
```

Method Prefixing

(Single) Inheritance and Generics

- each element in the method descriptor array corresponds to an **address** of an assembly label for the appropriate method

- **dynamic dispatch**: fetch the address from the method descriptor array and jump to that address

```
class Foo(int print(), void
doSomething())

class Bar(int print(), void
doSomething(), void
doSomethingElse()) extends Foo
```

Method Descriptors

Foo	**Foo.print**	**Foo.doSomething**	
Bar	**Bar.print**	**Bar.doSomething**	**Bar.doSomethingElse**

5. Demo

~/D/a/kyanite

```
λ result/bin/kyanite run examples/kyir/fibonacci.kya
```

6. Reflection

Materials / References

- Rust
- LLVM (clang)
- Nix (build & developer environment)
- **formerly:**
    - Zig (cross-compilation)

- Aho, Alfred V, et al. *Compilers: Principles, Techniques, and Tools*. India, Pearson India Education Services, 2015 **(dragon book)**
- Appel, Andrew W, and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge, Uk; New York, Ny, Usa, Cambridge University Press, 2002
- ARM developer documentation @ https://developer.arm.com
- Stack Overflow
- Compiler Explorer @ https://godbolt.org
- elsewhere on the internet

Reflection

- first semester went smoothly except for switching implementation languages halfway through because I wasn't comfortable enough with Kotlin (my original choice — because my primary reference textbook used Java) and I already had a toy Rust compiler from the summer I could modify instead
  - didn't lose much time on this
- lost a lot of time trying to figure out what operating system/assembly conventions were
  - a lot of trial and error to figure out what works and importantly, why does it work
- lots of looking at what existing compilers (gcc, clang) are doing and understanding what and why they emit certain instructions
  - now I can actually read assembly somewhat
- discovered how to use lldb for debugging segmentation faults, etc.
  - in the first half of the semester I was printing instructions to figure out where important data was being unintentionally overwritten
    - this is mostly avoided later because segmentation faults usually coincided with places where I accidentally overwrote data
- name variables properly (lost time trying to figure out what my own garbage collection algorithms were doing); and lots of messy pointer arithmetic made it even more tedious and painful
  - **on a related note:** comment code more
    - my first "large" project and not commenting what I was doing caused some problems later on when I needed to figure out what something was doing
- new computer made it cumbersome to test x86 executables so I switched to ARM in the middle of the second semester

Takeaways

- use familiar languages/tooling
- learn basics before diving into implementation
- look at what other things are doing and understand why
- learn and use a proper debugger to save time
    - segfaults are even more painful in assembly than C/C++
- follow programming conventions (descriptive variable naming, commenting code)
- switching technologies causes time losses

Thanks for listening!