implementing virtual machines in Go & C

Eleanor McHugh

@feyeleanor

http://github.com/feyeleanor

I made my own machine
Yes, we're building steam
I hate the same routine

— Building Steam, Abney Park

software is where
machine meets thought

and as programmers we think a lot about thinking

whilst thinking very little about machines

we style ourselves
philosophers & wizards

when machines need us
to be engineers & analysts

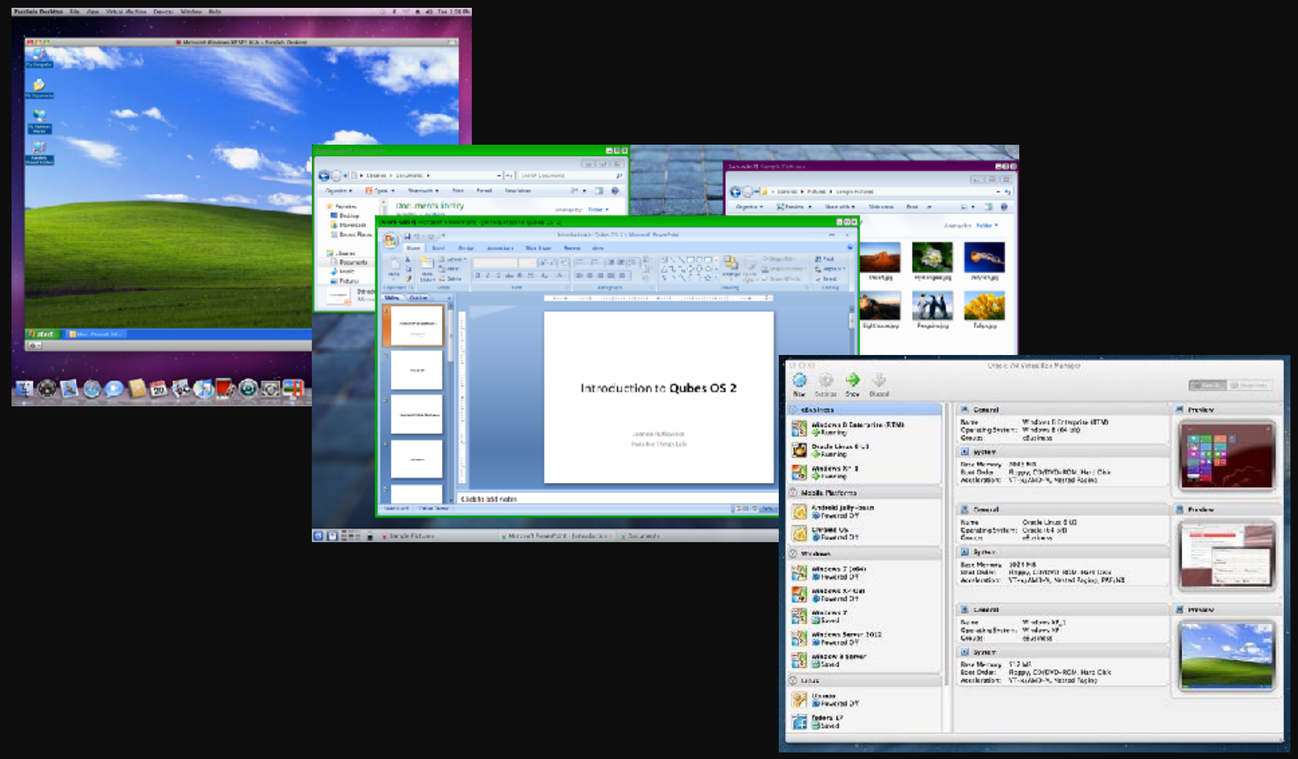
so this is a talk about machine philosophy

framed in a language we humans can understand

so let's learn to love our Turing machines

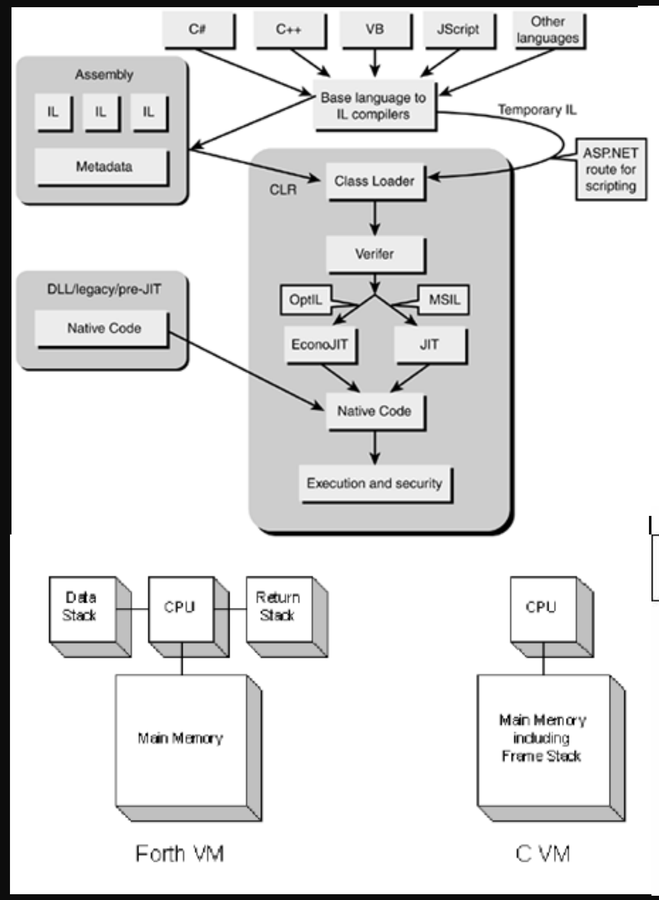
by building other Turing machines' in their image

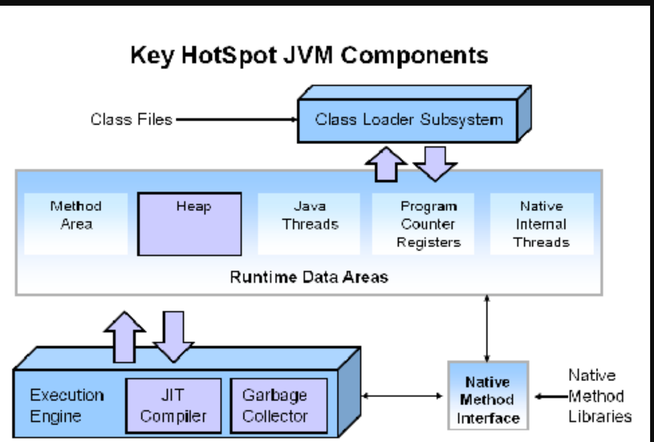
system virtualisation

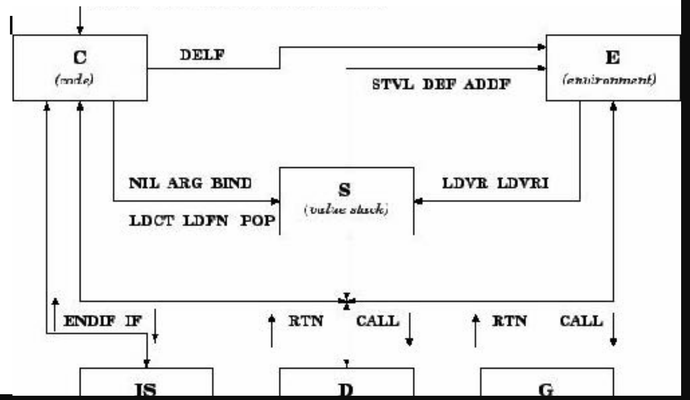


hardware emulation

program execution





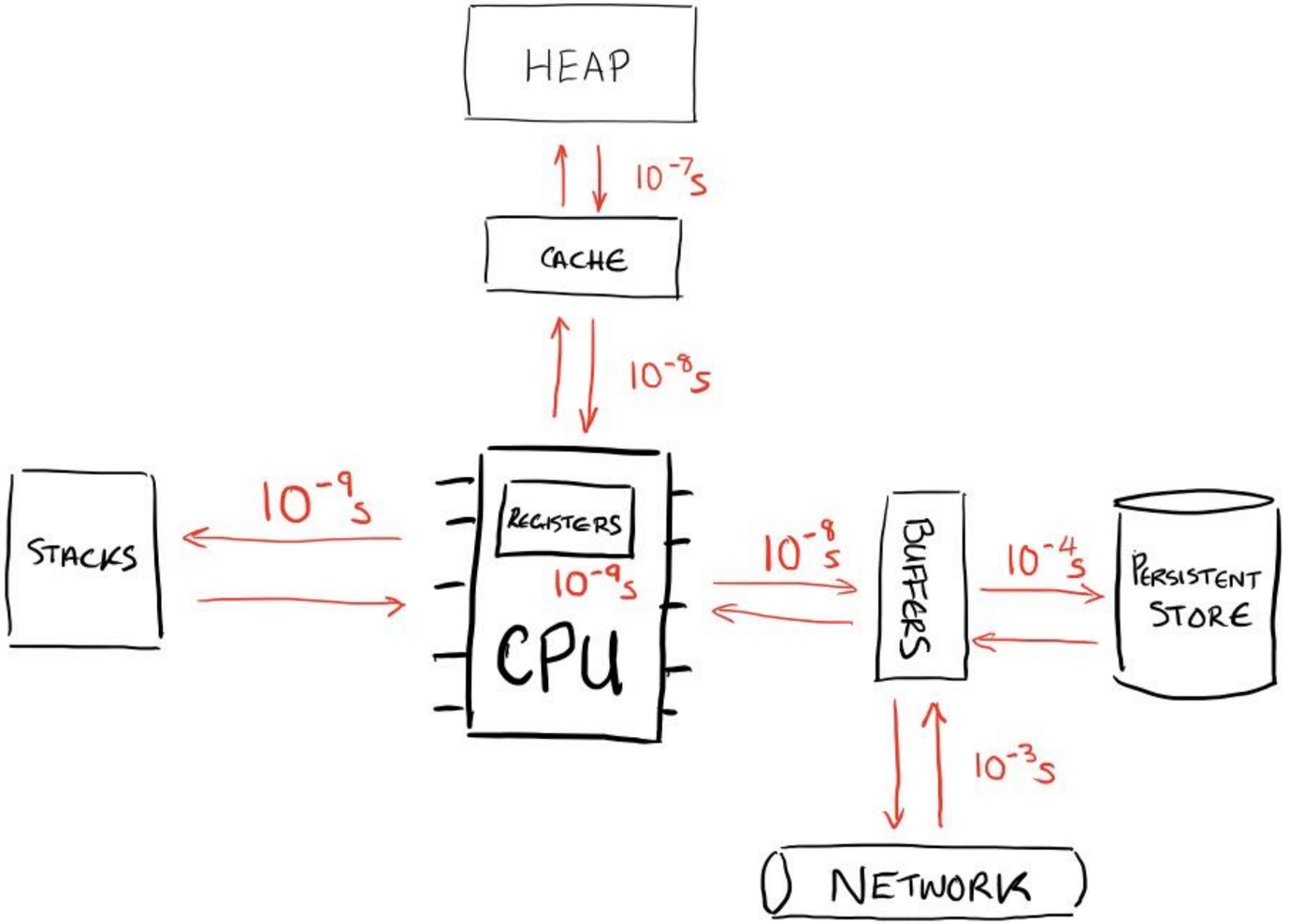


timely

stateful

conversational

discrete

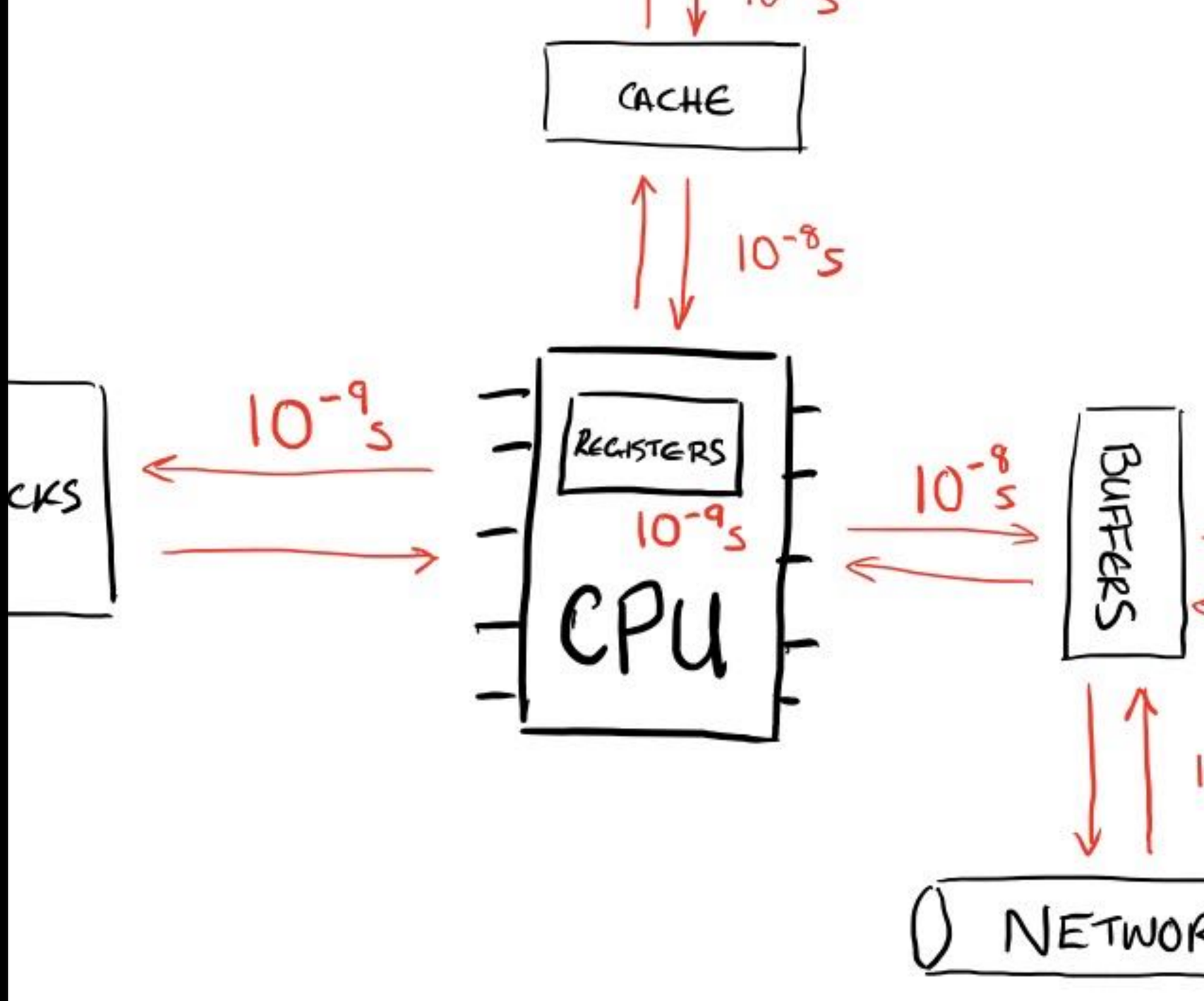


despatch loops

fetch instruction

decode

execute



c: switch

bytes or tokens

portable

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define READ_OPCODE *PC++

typedef enum {
  PUSH = 0,
  ADD,
  PRINT,
  EXIT
} opcodes;

int program [] = {
  (int)PUSH, 13,
  (int)PUSH, 28,
  (int)ADD,
  PRINT,
  EXIT,
};

STACK *S;
```

```
void interpret(int *PC) {
  int l, r;
  while (1) {
    switch(READ_OPCODE) {
    case PUSH:
      S = push(S, READ_OPCODE);
      break;
    case ADD:
      S = pop(S, &l);
      S = pop(S, &r);
      S = push(S, l + r);
      break;
    case PRINT:
      printf("%d + %d = %d\n, l, r, S->data);
      break;
    case EXIT:
      return;
    }
  }
}

int main() {
  interpret(program);
}
```

go: switch constants

```
package main
import "fmt"

func main() {
  interpret([]interface{}{
    PUSH, 13,
    PUSH, 28,
    ADD,
    PRINT,
    EXIT,
  })
}

type stack struct {
  data int
  tail *stack
}

func (s *stack) Push(v int) (r *stack) {
  r = &stack{data: v, tail: s}
  return
}

func (s *stack) Pop() (v int, r *stack) {
  return s.data, s.tail
}

type OPCODE int
```

```
const (
  PUSH = OPCODE(iota)
  ADD
  PRINT
  EXIT
)

func interpret(p []interface{}) {
  var l, r int
  S := new(stack)
  for PC := 0; ; PC++ {
    if op, ok := p[PC].(OPCODE); ok {
      switch op {
      case PUSH:
        PC++
        S = S.Push(p[PC].(int))
      case ADD:
        l, S = S.Pop()
        r, S = S.Pop()
        S = S.Push(l + r)
      case PRINT:
        fmt.Printf("%v + %v = %v\n", l, r, S.data)
      case EXIT:
        return
      }
    } else {
      return
    }
  }
}
```

c: direct call

pointer to function

multi-byte

portable

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define READ_OPCODE *PC++
typedef void (*opcode)();

STACK *S;
opcode *PC;

void op_add_and_print() {
  int l, r;
  S = pop(S, &l);
  S = pop(S, &r);
  S = push(S, l + r);
  printf("%d + %d = %d\n", l, r, S->data);
}

void op_exit() {
  exit(0);
}
```

```
opcode program [] = {
  op_push, (opcode)(long)13,
  op_push, (opcode)(long)28,
  op_add_and_print,
  op_exit
};

int main() {
  PC = program;
  while (1) {
    (READ_OPCODE)();
  }
}
```

go: direct call

```
package main
import "fmt"
import "os"

func main() {
  p := new(Interpreter)
  p.m = []interface{}{
    p.Push, 13,
    p.Push, 28,
    p.Add, p.Print, p.Exit,
  }
  p.Run()
}

type stack struct {
  data int
  tail *stack
}

func (s *stack) Push(v int) (r *stack) {
  return &stack{data: v, tail: s}
}

func (s *stack) Pop() (v int, r *stack) {
  return s.data, s.tail
}

type Interpreter struct {
  S       *stack
  I, r, PC int
  m       []interface{}
}
```

```
func (i *Interpreter) read_program() (r interface{}) {
  return i.m[i.PC]
}

func (i *Interpreter) Run() {
  for {
    i.read_program().(func())()
    i.PC++
  }
}

func (i *Interpreter) Push() {
  i.PC++
  i.S = i.S.Push(i.read_program().(int))
}

func (i *Interpreter) Add() {
  i.I, i.S = i.S.Pop()
  i.r, i.S = i.S.Pop()
  i.S = i.S.Push(i.I + i.r)
}

func (i *Interpreter) Print() {
  fmt.Printf("%v + %v = %v\n", i.I, i.r, i.S.data)
}

func (i *Interpreter) Exit() {
  os.Exit(0)
}
```

go: direct call

raw closures

```
package main

import "fmt"
import "os"

type stack struct {
  data int
  tail *stack
}

func (s *stack) Push(v int) (r *stack) {
  return &stack{data: v, tail: s}
}

func (s *stack) Pop() (v int, r *stack) {
  return s.data, s.tail
}

type Primitive func(*Interpreter)

type Interpreter struct {
  S        *stack
  l, r, PC int
  m        []Primitive
}

func (i *Interpreter) read_program() Primitive {
  return i.m[i.PC]
}
```

```
func (i *Interpreter) Run() {
  for {
    i.read_program()(i)
    i.PC++
  }
}

func main() {
  p := &Interpreter{
  m: []Primitive{
    func(i *Interpreter) { i.S = i.S.Push(13) },
    func(i *Interpreter) { i.S = i.S.Push(28) },
    func(i *Interpreter) {
      i.l, i.S = i.S.Pop()
      i.r, i.S = i.S.Pop()
      i.S = i.S.Push(i.l + i.r)
    },
    func(i *Interpreter) {
      fmt.Printf("%v + %v = %v\n", i.l, i.r, i.S.data)
    },
    func (i *Interpreter) { os.Exit(0) },
    },
  }
  p.Run()
}
```

go: direct call

raw closures

```
func main() {
  p := &Interpreter{
    m: []Primitive{
      func(i *Interpreter) {
        i.S = i.S.Push(13)
        i.PC++
        i.read_program()(i)
        i.PC++
        i.read_program()(i)
      },
      func(i *Interpreter) {
        i.S = i.S.Push(28)
      },
      func(i *Interpreter) {
        i.l, i.S = i.S.Pop()
        i.r, i.S = i.S.Pop()
        i.S = i.S.Push(i.l + i.r)
      },
      func(i *Interpreter) {
        fmt.Printf("%v + %v = %v\n", i.l, i.r, i.S.data)
      },
      func (i *Interpreter) {
        os.Exit(0)
      },
    },
  }
  p.Run()
}
```

go: direct call

raw closures

```
type Primitive func()

func (i *Interpreter) Run() {
  for {
    i.read_program()()
    i.PC++
  }
}
```

```
func main() {
  i := new(Interpreter)

  npush := func(vals ...int) Primitive {
    return func() {
      for _, v := range vals {
        i.S = i.S.Push(v)
      }
    }
  }

  i.m = []Primitive{
    npush(13, 28),
    func() {
      i.I, i.S = i.S.Pop()
      i.r, i.S = i.S.Pop()
      i.S = i.S.Push(i.I + i.r)
    },
    func() {
      fmt.Printf("%v + %v = %v\n", i.I, i.r, i.S.data)
    },
    func() { os.Exit(0) },
  }
  i.Run()
}
```

go: direct call

raw closures

jit assembly

```
package main

import "fmt"
import "strings"
import "os"

type stack struct {
	data int
	tail *stack
}

func (s *stack) Push(v int) (r *stack) {
	return &stack{data: v, tail: s}
}

func (s *stack) Pop() (v int, r *stack) {
	return s.data, s.tail
}

func (s *stack) String() string {
	r := []string{}
	for i := s; i != nil; i = i.tail {
		r = append(r, fmt.Sprint(i.data))
	}
	return "[" + strings.Join(r, ", ") + "]"
}
```

go: direct call

raw closures

jit assembly

```
type Label string
type labels map[Label] int

type VM struct {
  S          *stack
  I, r, PC   int
  m          []interface{}
  labels
}

func (v *VM) read_program() interface{} {
  return v.m[v.PC]
}

func (v *VM) String() string {
  return fmt.Sprintf("@pc[%v] => #{%v}, %v",
  v.PC, v.m[v.PC], v.S)
}

func (v *VM) Load(program ...interface{}) {
  v.labels = make(labels)
  v.PC = -1
  for _, token := range program {
    v.assemble(token)
  }
}
```

```
func (v *VM) assemble(token interface{}) {
  switch t := token.(type) {
  case Label:
    if i, ok := v.labels[t]; ok {
    v.m = append(v.m, i)
  } else {
    v.labels[t] = v.PC
  }
  default:
    v.m = append(v.m, token)
    v.PC++
  }
}

func (v *VM) Run() {
  v.PC = -1
  for {
    v.PC++
    v.read_program().(func())()
  }
}
```

go: direct call

raw closures

jit assembly

```
type Interpreter struct { VM }

func (i *Interpreter) Push() {
  i.PC++
  i.S = i.S.Push(i.read_program().(int))
}

func (i *Interpreter) Add() {
  i.l, i.S = i.S.Pop()
  i.r, i.S = i.S.Pop()
  i.S = i.S.Push(i.l + i.r)
}

func (i *Interpreter) JumpIfNotZero() {
  i.PC++
  if i.S.data != 0 {
    i.PC = i.m[i.PC].(int)
  }
}
```

```
func main() {
  i := new(Interpreter)
    print_state := func() {
    fmt.Println(i)
  }
  skip := func() { i.PC++ }
  i.Load(
    i.Push, 13,
    Label("decrement"),
    func() {
      i.S = i.S.Push(-1)
    },
    i.Add,
    print_state,
    skip,
    print_state,
    i.JumpIfNotZero, Label("decrement"),
    print_state,
    func() {
      os.Exit(0)
    },
  )
  i.Run()
}
```

c: indirect thread

local jumps

gcc/clang specific

indirect loading

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

typedef enum {
  PUSH = 0, ADD, PRINT, EXIT
} opcodes;

void interpret(int *program) {
  static void *opcodes [] = {
    &&op_push,
    &&op_add,
    &&op_print,
    &&op_exit
  };

  int l, r;
  STACK *S;
  int *PC = program;
  goto *opcodes[*PC++];

op_push:
  S = push(S, *PC++);
  goto *opcodes[*PC++];

op_add:
  S = pop(S, &l);
  S = pop(S, &r);
  S = push(S, l + r);
  goto *opcodes[*PC++];

op_print:
  printf("%d + %d = %d\n", l, r, S->data);
  goto *opcodes[*PC++];

op_exit:
  return;
}

int main() {
  int program [] = {
    PUSH, 13,
    PUSH, 28,
    ADD,
    PRINT,
    EXIT
  };
  interpret(program);
}
```

c: direct thread

jit local jumps

gcc/clang specific

direct loading

```
void interpret(int *PC, int words) {
  static void *dispatch_table[] = {
    &&op_push,
    &&op_add,
    &&op_print,
    &&op_exit
  };

  STACK *S;
  int l, r;
  void **program = compile(PC, words, dispatch_table);
  if (program == NULL)
    exit(1);
  goto **program++;

op_push:
  S = push(S, (int)(long)*program++);
  goto **program++;

op_add:
  S = pop(S, &l);
  S = pop(S, &r);
  S = push(S, l + r);
  goto **program++;

op_print:
  printf("%d + %d = %d\n", l, r, S->data);
  goto **program++;

op_exit:
  return;
}

int main() {
  int program[] = {
    PUSH, 13,
    PUSH, 28,
    ADD,
    PRINT,
    EXIT
  };
  interpret(program, 7);
}
```

c: direct thread

```
#define INTERPRETER(body, ...) \
  DISPATCHER(__VA_ARGS__); \
  void **p = compile(PC, d); \
  if (p == NULL) \
    exit(1); \
  EXECUTE_OPCODE \
  body

#define DISPATCHER(...) \
  static void *d[] = { __VA_ARGS__ }

#define READ_OPCODE \
  *p++

#define EXECUTE_OPCODE \
  goto *READ_OPCODE;

#define PRIMITIVE(name, body) \
  name: body; \
  EXECUTE_OPCODE
```

```
void interpret(int *PC) {
  STACK *S;
  int l, r;
  INTERPRETER(
    PRIMITIVE(push,
      S = push(S, (int)(long)READ_OPCODE)
    )
    PRIMITIVE(add,
      S = pop(S, &l);
      S = pop(S, &r);
      S = push(S, l + r)
    )
    PRIMITIVE(print,
      printf("%d + %d = %d\n", l, r, S->data)
    )
    PRIMITIVE(exit,
      return
    ),
    &&push, &&add, &&print, &&exit
    )
}
```

c: direct thread

jit local jumps

gcc/clang specific

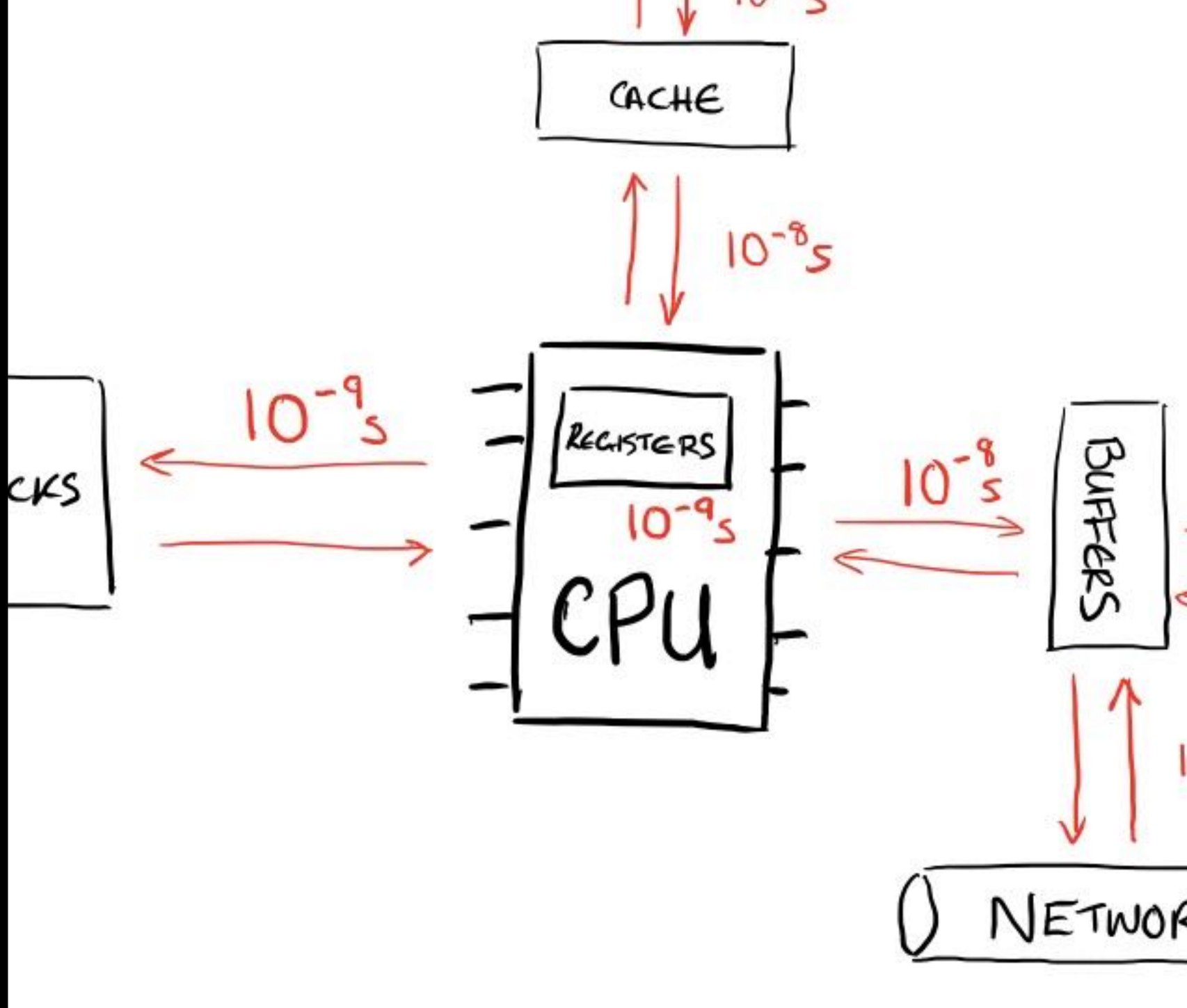
direct loading

```
void **compile(int *PC, int words, void *dispatch_table[]) {
  static void *compiler [] = {
    &&comp_push,
    &&comp_add,
    &&comp_print,
    &&comp_exit
  };

  if (words < 1)
    return NULL;
  void **program = malloc(sizeof(void *) * words);
  void **cp = program;
  goto *compiler[*PC++];

comp_push:
  *cp++ = dispatch_table[PUSH];
  *cp++ = (void *)(long)*PC++;
  words -= 2;
  if (words == 0)
    return program;
  goto *compiler[*PC++];

comp_add:
  *cp++ = dispatch_table[ADD];
  words--;
  if (words == 0)
    return program;
  goto *compiler[*PC++];

comp_print:
  *cp++ = dispatch_table[PRINT];
  words--;
  if (words == 0)
    return program;
  goto *compiler[*PC++];

comp_exit:
  *cp++ = dispatch_table[EXIT];
  words--;
  if (words == 0)
    return program;
  goto *compiler[*PC++];
}
```

registers

operands

local caching



vm harness

dispatch

program

program counter

```
package main

import "fmt"

type Label string
type labels map[Label] int

type VM struct {
  PC        int
  m         []interface{}
  labels
}

func (v *VM) read_program() interface{} {
  return v.m[v.PC]
}

func (v *VM) String() string {
  return fmt.Sprintf("@pc[%v] => #{%v}, %v", v.PC, v.m[v.PC])
}

func (v *VM) Load(program ...interface{}) *VM {
  v.labels = make(labels)
  v.PC = -1
  for _, token := range program {
    v.assemble(token)
  }
  return v
}
```

```
func (v *VM) assemble(token interface{}) {
  switch t := token.(type) {
  case Label:
    if i, ok := v.labels[t]; ok {
    v.m = append(v.m, i)
    } else {
    v.labels[t] = v.PC
    }
  default:
    v.m = append(v.m, token)
    v.PC++
  }
}

func (v *VM) Run() {
  v.PC = -1
  for {
    v.PC++
    v.read_program().(func())()
  }
}
```

stack machine

zero operands

```
package main

import "fmt"
import "os"

type S []int
type StackMachine struct {
  S
  VM
}

func (s *StackMachine) String() string {
  return fmt.Sprintf("%v, %v", s.VM, s.S)
}

func (s *StackMachine) Push() {
  s.PC++
  s.S = append(s.S, s.read_program().(int))
}

func (s *StackMachine) Add() {
  s.S[len(s.S) - 2] += s.S[len(s.S) - 1]
  s.S = s.S[:len(s.S) - 1]
}
```

```
func (s *StackMachine) JumpIfNotZero() {
  s.PC++
  if s.S[len(s.S) - 1] != 0 {
    s.PC = s.m[s.PC].(int)
  }
}

func main() {
  s := new(StackMachine)
  print_state := func() {
    fmt.Println(s)
  }
  skip := func() { s.PC++ }
  s.Load(
    s.Push, 13,
    Label("dec"),
    func() { s.S = append(s.S, -1) },
    s.Add,
    print_state,
    skip,
    print_state,
    s.JumpIfNotZero, Label("dec"),
    print_state,
    func() { os.Exit(0) },
  ).Run()
}
```

accumulator machine

single register

single operand

```
package main

import "fmt"
import "os"

type AccMachine struct {
	A int
	VM
}

func (a *AccMachine) String() string {
	return fmt.Sprintf("%v, A=%v", a.VM, a.A)
}

func (a *AccMachine) Clear() {
	a.A = 0
}

func (a *AccMachine) LoadValue() {
	a.PC++
	a.A = a.read_program().(int)
}

func (a *AccMachine) Add() {
	a.PC++
	a.A += a.read_program().(int)
}
```

```
func (a *AccMachine) JumpIfNotZero() {
	a.PC++
	if a.A != 0 {
		a.PC = a.m[a.PC].(int)
	}
}

func main() {
	a := new(AccMachine)
	print_state := func() {
		fmt.Println(a)
	}
	skip := func() { a.PC++ }
	a.Load(
		a.Clear,
		a.LoadValue, 13,
		Label("decrement"),
		a.Add, -1,
		print_state,
		skip,
		print_state,
		a.JumpIfNotZero, Label("decrement"),
		print_state,
		func() { os.Exit(0) },
	).Run()
}
```

register machine

multi-register

multi-operand

```
package main

import "fmt"
import "os"

type RegMachine struct {
  R []int
  VM
}

func (r *RegMachine) String() string {
  return fmt.Sprintf("%v, R=%v", r.VM, r.R)
}

func (r *RegMachine) Clear() {
  r.R = make([]int, 2, 2)
}

func (r *RegMachine) read_value() int {
  r.PC++
  return r.read_program().(int)
}

func (r *RegMachine) LoadValue() {
  r.R[r.read_value()] = r.read_value()
}

func (r *RegMachine) Add() {
  i := r.read_value()
  j := r.read_value()
  r.R[i] += r.R[j]
}
```

```
func (r *RegMachine) JumpIfNotZero() {
  if r.R[r.read_value()] != 0 {
    r.PC = r.read_value()
  } else {
    r.PC++
  }
}

func main() {
  r := new(RegMachine)
  print_state := func() {
    fmt.Println(r)
  }
  skip := func() { r.PC++ }
  r.Load(
    r.Clear,
    r.LoadValue, 0, 13,
    r.LoadValue, 1, -1,
    Label("decrement"),
    r.Add, 0, 1,
    print_state,
    skip,
    print_state,
    r.JumpIfNotZero, 0, Label("decrement"),
    print_state,
    func() { os.Exit(0) },
  ).Run()
}
```

vector machine

matrix machine

hypercube

graph processor

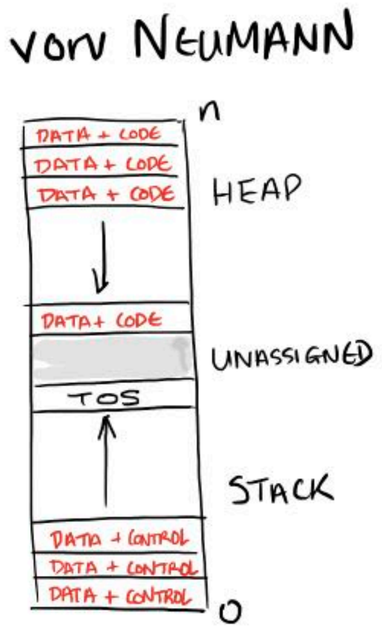
any datatype can be a register

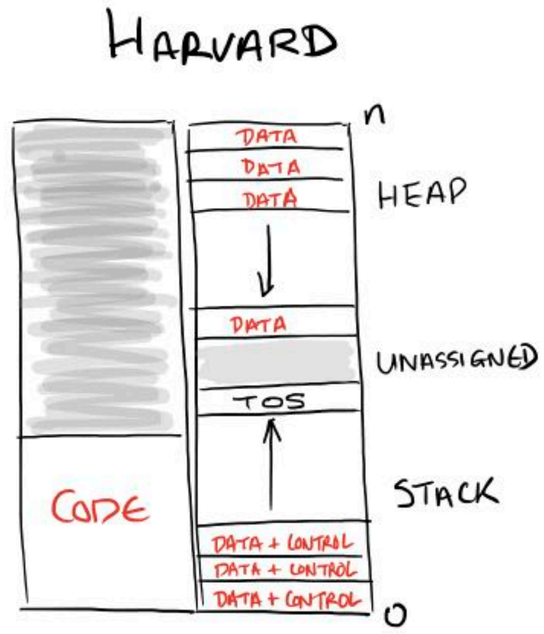
memory model

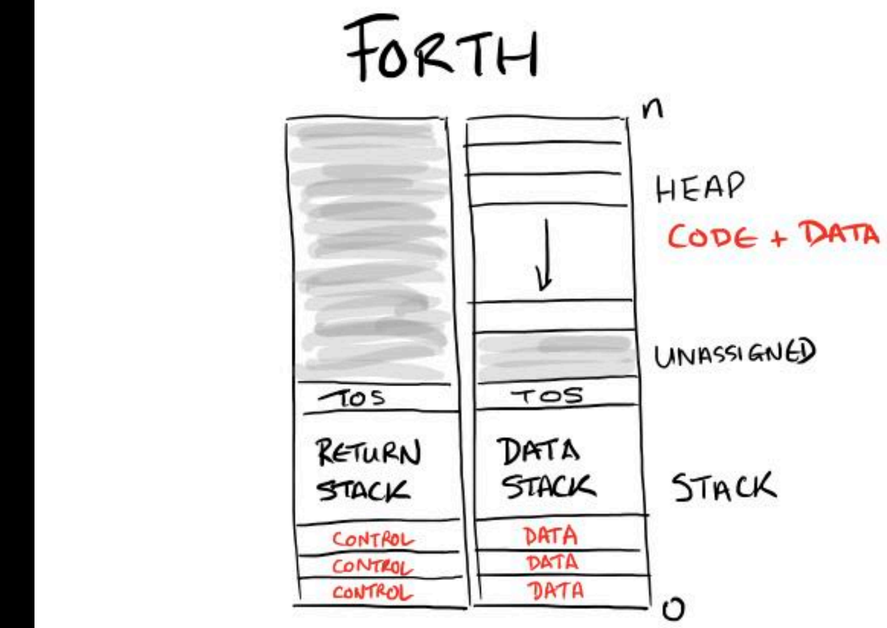
instructions

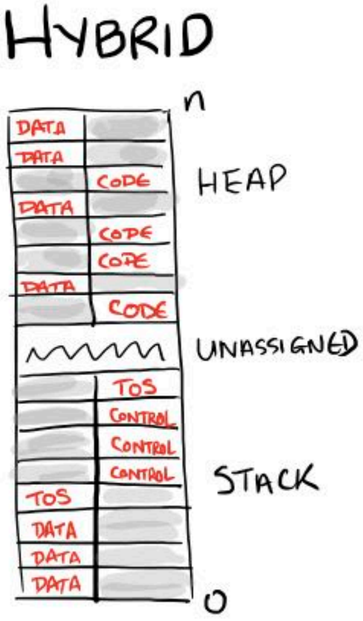
computation

state







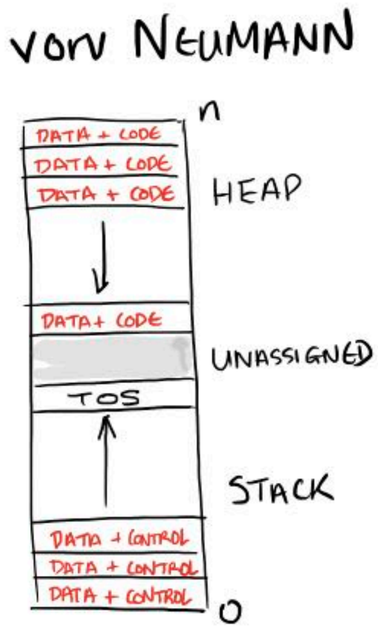


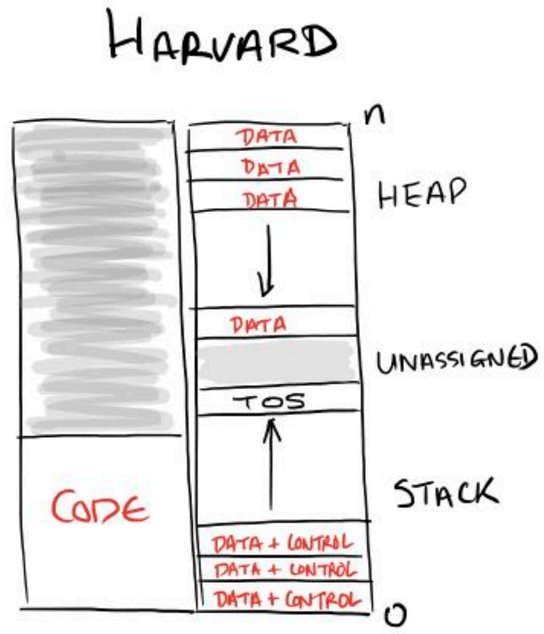
memory model

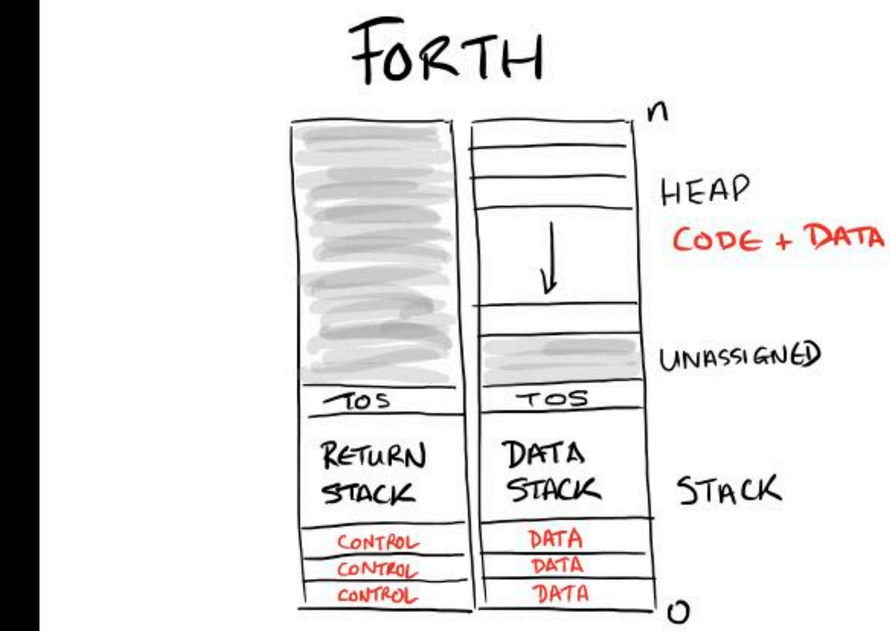
opcodes

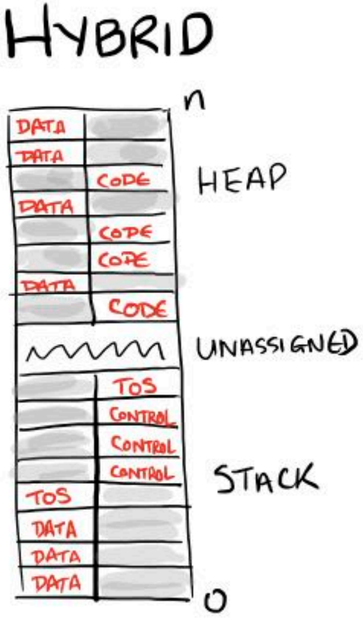
stack

heap







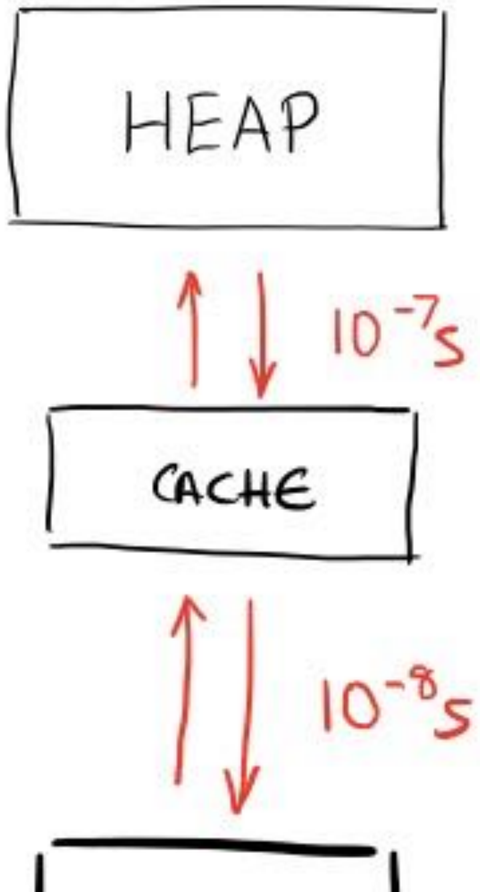


heaps

word-aligned

contiguous

byte-addressable



go: slice heap

```
package main
import r "reflect"
import "unsafe"

type Memory []uintptr

var _BYTE_SLICE = r.TypeOf([]byte(nil))
var _MEMORY = r.TypeOf(Memory{})
var _MEMORY_BYTES = int(_MEMORY.Elem().Size())

func (m Memory) newHeader() (h r.SliceHeader) {
  h = *(*r.SliceHeader)(unsafe.Pointer(&m))
  h.Len = len(m) * _MEMORY_BYTES
  h.Cap = cap(m) * _MEMORY_BYTES
  return
}

func (m *Memory) Bytes() (b []byte) {
  h := m.newHeader()
  return *(*[]byte)(unsafe.Pointer(&h))
}

func (m *Memory) Serialise() (b []byte) {
  h := m.newHeader()
  b = make([]byte, h.Len)
  copy(b, *(*[]byte)(unsafe.Pointer(&h)))
  return
}
```

```
func (m *Memory) Overwrite(i interface{}) {
  switch i := i.(type) {
  case Memory:
    copy(*m, i)
  case []byte:
    h := m.newHeader()
    b := *(*[]byte)(unsafe.Pointer(&h))
    copy(b, i)
  }
}

func main() {
  m := make(Memory, 2)
  b := m.Bytes()
  s := m.Serialise()
  fmt.Println("m (cells) =", len(m), "of", cap(m), ":", m)
  fmt.Println("b (bytes) =", len(b), "of", cap(b), ":", b)
  fmt.Println("s (bytes) =", len(s), "of", cap(s), ":", s)

  m.Overwrite(Memory{3, 5})
  fmt.Println("m (cells) =", len(m), "of", cap(m), ":", m)
  fmt.Println("b (bytes) =", len(b), "of", cap(b), ":", b)
  fmt.Println("s (bytes) =", len(s), "of", cap(s), ":", s)

  s = m.Serialise()
  m.Overwrite([]byte{8, 7, 6, 5, 4, 3, 2, 1})
  fmt.Println("m (cells) =", len(m), "of", cap(m), ":", m)
  fmt.Println("b (bytes) =", len(b), "of", cap(b), ":", b)
  fmt.Println("s (bytes) =", len(s), "of", cap(s), ":", s)
}
```

ruby: c heap

```
require "fiddle"

class Fiddle::Pointer
  NIL = Pointer.new(0)
  SIZE = Fixnum::SIZE

  PACKING_PATTERN = case SIZE
  when 2 then "S"
  when 4 then "L"
  when 8 then "Q"
  end + "!"

  def write(value)
    str = Fiddle::format(value)
    pad = Fiddle::padding(str)
    l = pad + str.length
    raise BufferOverflow.new(self, l) if l > size
    self[0, l] = str + 0.chr * pad
    self + l
  end

  def to_bin
    [self].pack(PACKING_PATTERN)
  end
end
```

access DLLs

call C functions

Fiddle

A libffi wrapper for Ruby.

Description¶ (#module-Fiddle-label-Description) ↑ (#top)

Fiddle (Fiddle.html) is an extension to translate a foreign function interface (FFI) with ruby.

It wraps libffi (http://sourceware.org/libffi/), a popular C library which provides a portable interface that allows code written in one language to call code written in another language.

Example¶ (#module-Fiddle-label-Example) ↑ (#top)

Here we will use Fiddle::Function (Fiddle/Function.html) to wrap floor(3) from libm (http://linux.die.net/man/3/floor)

```
require 'fiddle'

libm = Fiddle.dlopen('/lib/libm.so.6')

floor = Fiddle::Function.new(
  libm['floor'],
  [Fiddle::TYPE_DOUBLE],
  Fiddle::TYPE_DOUBLE
)
```

MRI stdlib

C pointers

malloc

not portable

Fiddle::Pointer

Fiddle::Pointer (Pointer.html) is a class to handle C pointers

Public Class Methods

Fiddle::Pointer[val] => cptr
to_ptr(val) => cptr

Get the underlying pointer for ruby object `val` and return it as a Fiddle::Pointer (Pointer.html) object.

Fiddle::Pointer.malloc(size, freefunc = nil) => fiddle pointer instance

Allocate `size` bytes of memory and associate it with an optional `freefunc` that will be called when the pointer is garbage collected.

`freefunc` must be an address pointing to a function or an instance of Fiddle::Function (Function.html)

Fiddle::Pointer.new(address) => fiddle_cptr
new(address, size) => fiddle_cptr
new(address, size, freefunc) => fiddle_cptr

Create a new pointer to `address` with an optional `size` and `freefunc`.

`freefunc` will be called when the instance is garbage collected.

ruby: c heap

```
def Fiddle::Pointer.format(value)
  value.respond_to?(:to_bin) ? value.to_bin : Marshal.dump(value)
end
```

ruby: c heap

```
require "fiddle"

class Fiddle::Pointer
  NIL = Pointer.new(0)
  SIZE = Fixnum::SIZE

  PACKING_PATTERN = case SIZE
  when 2 then "S"
  when 4 then "L"
  when 8 then "Q"
  end + "!"

  def write(value)
    str = Fiddle::format(value)
    pad = Fiddle::padding(str)
    l = pad + str.length
    raise BufferOverflow.new(self, l) if l > size
    self[0, l] = str + 0.chr * pad
    self + l
  end

  def to_bin
    [self].pack(PACKING_PATTERN)
  end
end
```

ruby: c heap

```
class Fixnum
	SIZE = 1.size

	PACKING_PATTERN = case SIZE
	when 2 then "s"
	when 4 then "l"
	when 8 then "q"
	end + "!"

	def to_bin
		[self].pack(PACKING_PATTERN)
	end

	def self.read_bin(pointer)
		pointer[0, SIZE].unpack(PACKING_PATTERN).first
	end
end
```

ruby: c heap

```
m = Fiddle::Pointer.malloc 64
begin
  m.write(0.chr * 59)
      m.write(0.chr * 60)
      m.write(0.chr * 61)
rescue Fiddle::BufferOverflow => e
      p e.message
end
```

```
"Buffer overflow: 72 bytes at #<Fiddle::Pointer:0x007f8849052160
ptr=0x007f8849051da0 size=64 free=0x0000000000000>"
```

ruby: c heap

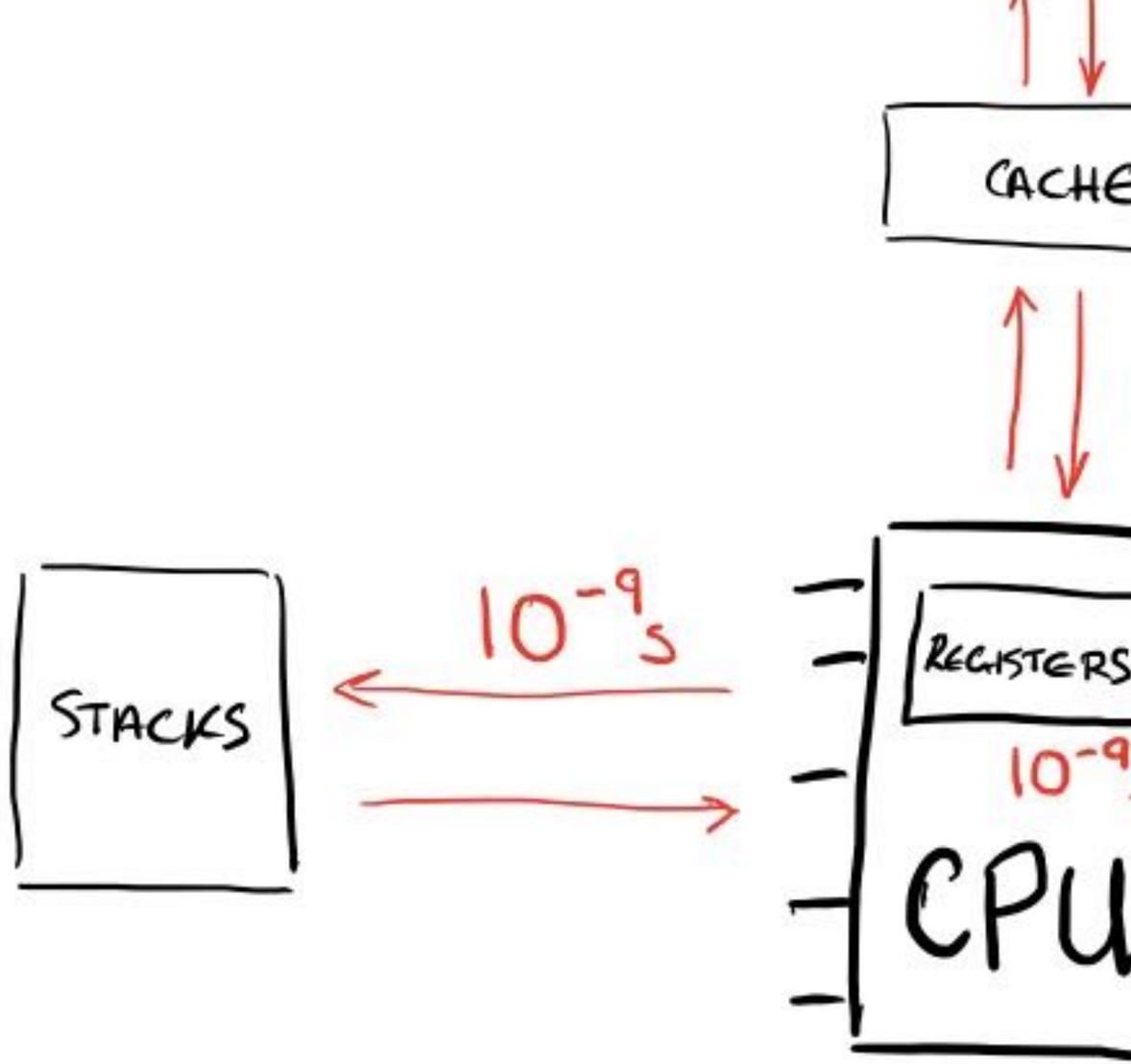
```
s = "Hello, Terrible Memory Bank!"
i = 4193
f = 17.00091

m.write(i)
puts m.read                                  => 4193
q = m.write(-i)
puts m.read                                  => -4193

q.write(s)
puts q.read(String)                          => Hello, Terrible Memory Bank!

r = q.write(s[0, s.length - 1])
puts q.read(String)                          => Hello, Terrible Memory Bank

t = r.write(f)
puts r.read(Float)                           => 17.00091

t.write(-f)
puts t.read(Float)                           => -17.00091
```

stacks

sequential

bounded depth

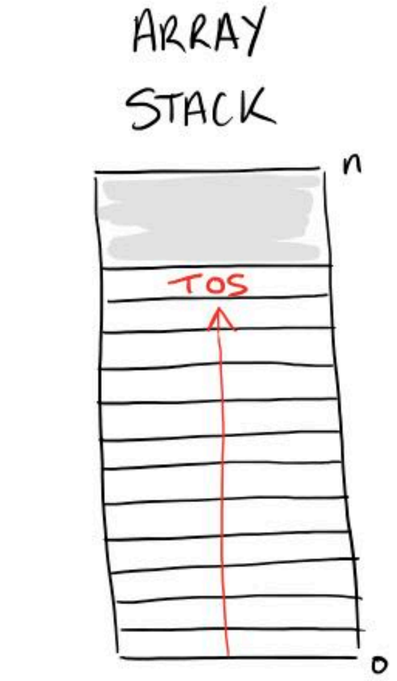
push & pop

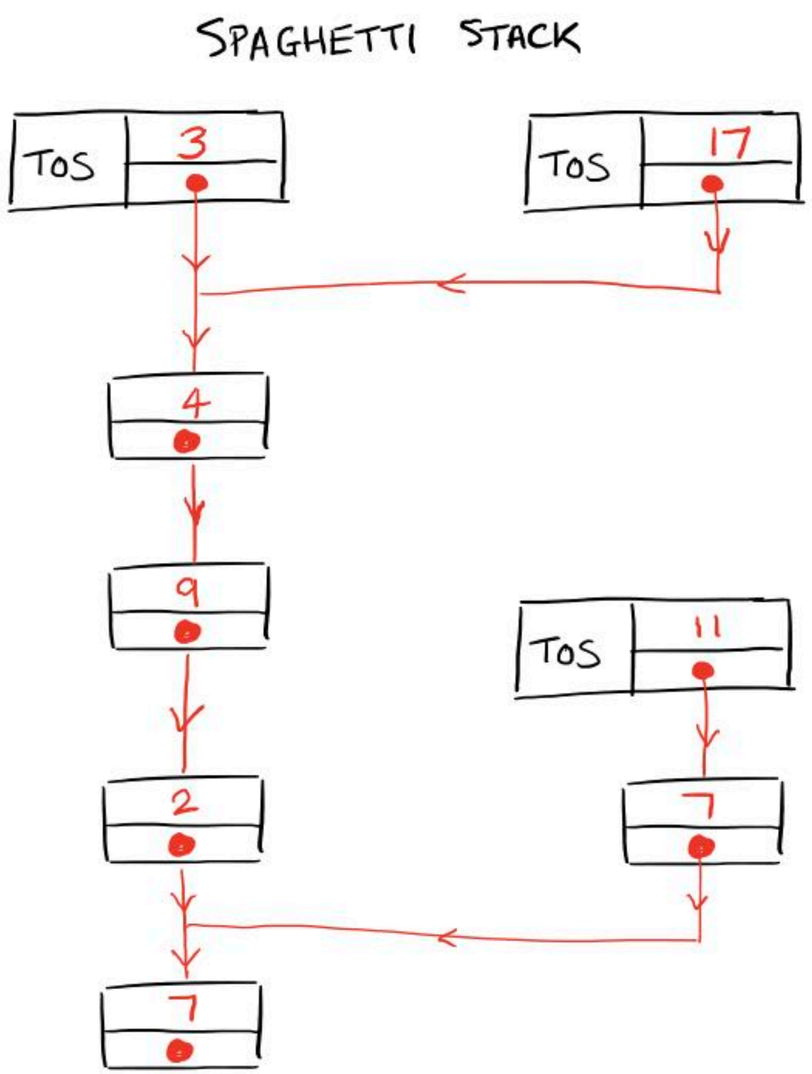


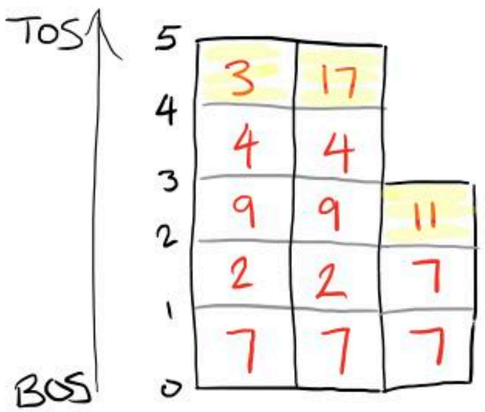
sequential

push data on

pop data off

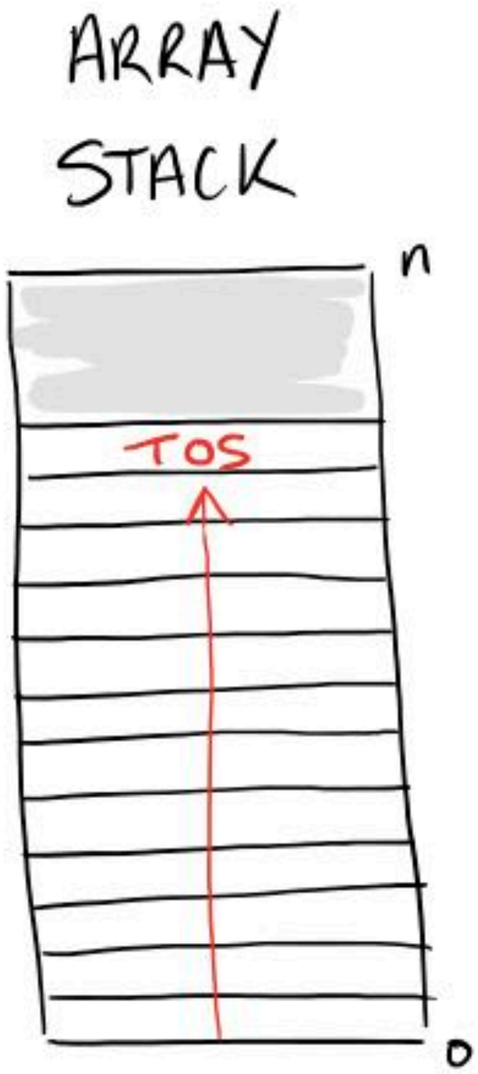


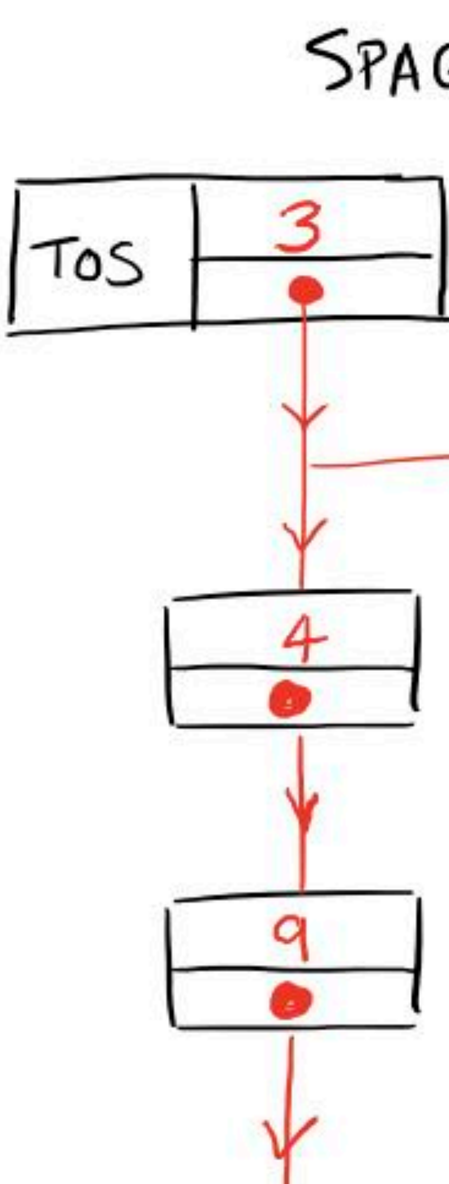




contiguous

bounded depth





c: array stack

fixed size

malloc to create

realloc to resize

```
#include <stdlib.h>
#define STACK_MAX 100

typedef enum {
  STACK_OK = 0,
  STACK_OVERFLOW,
  STACK_UNDERFLOW
} STACK_STATUS;

typedef struct stack STACK;
struct stack {
  int data[STACK_MAX];
  int size;
};

STACK *NewStack() {
  STACK *s;
  s = malloc(sizeof(STACK));
  s->size = 0;
  return s;
}

int depth(STACK *s) {
  return s->size;
}
```

```
STACK_STATUS push(STACK *s, int data) {
  if (s->size < STACK_MAX) {
    s->data[s->size++] = data;
    return STACK_OK;
  }
  return STACK_OVERFLOW;
}

STACK_STATUS pop(STACK *s, int *r) {
  if (s->size > 0) {
    *r = s->data[s->size - 1];
    s->size--;
    return STACK_OK;
  }
  return STACK_UNDERFLOW;
}
```

go: slice stack

```
package main

import "fmt"

type stack []int

func (s *stack) Push(data int) {
  (*s) = append((*s), data)
}

func (s *stack) Pop() (r int) {
  sp := len(*s) - 1
  r = (*s)[sp]
  *s = (*s)[:sp]
  return
}

func (s stack) Depth() int {
  return len(s)
}
```

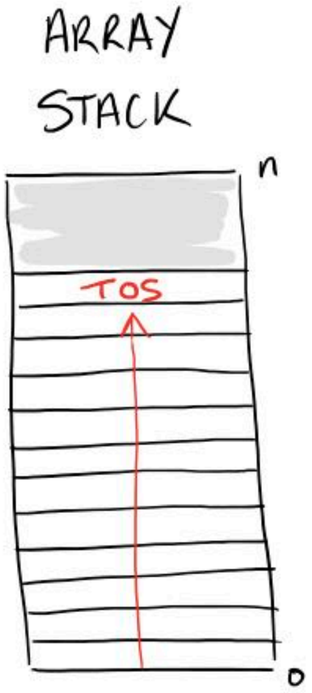
```
func main() {
  s := new(stack)
  s.Push(1)
  s.Push(3)
  fmt.Printf("depth = %d\n", s.Depth())
  l := s.Pop()
  r := s.Pop()
  fmt.Printf("%d + %d = %d\n", l, r, l+r)
  fmt.Printf("depth = %d\n", s.Depth())
}
```

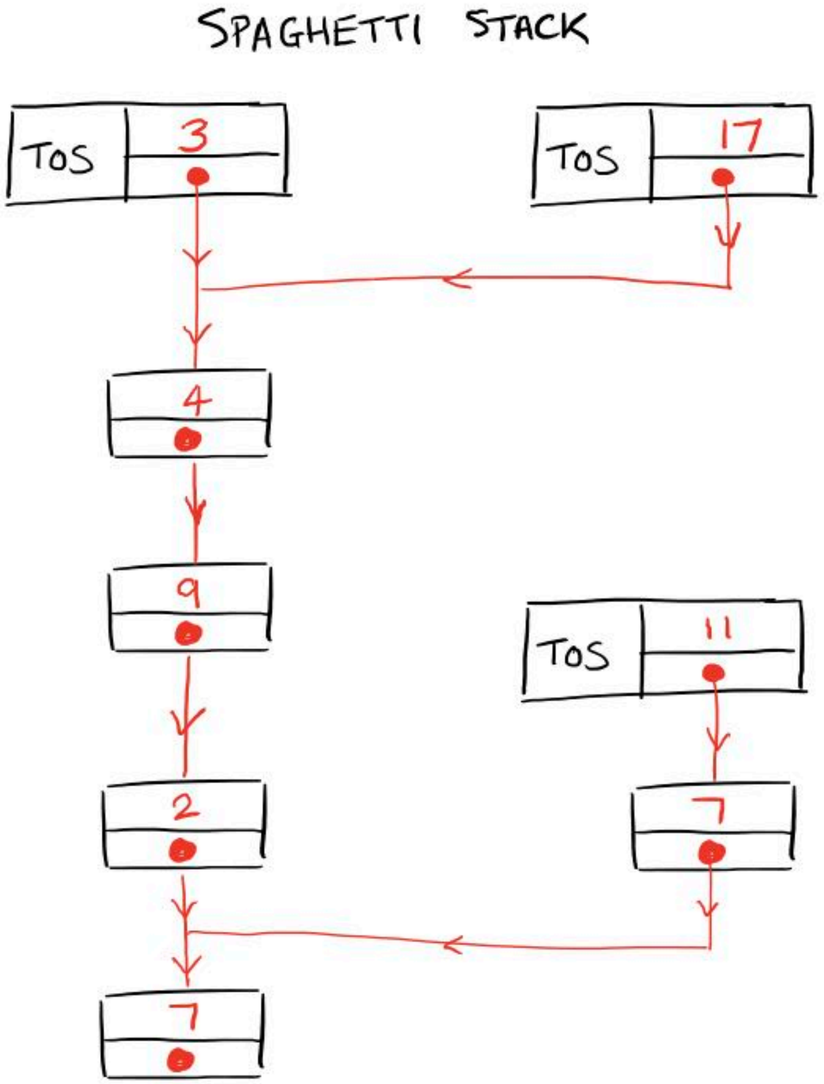
singly linked list

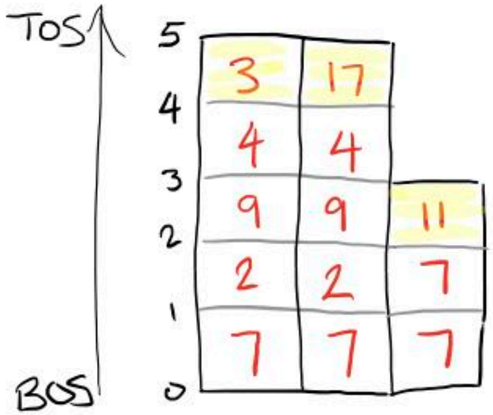
functional

shared elements

immutability







c: cactus stack

nil is empty

grows on push

manual GC

```c
#include <stdlib.h>

typedef struct stack STACK;
struct stack {
  int data;
  STACK *next;
};

STACK *push(STACK *s, int data) {
  STACK *r = malloc(sizeof(STACK));
  r->data = data;
  r->next = s;
  return r;
}

STACK *pop(STACK *s, int *r) {
  if (s == NULL)
    exit(1);
  *r = s->data;
  return s->next;
}
```

```c
int depth(STACK *s) {
  int r = 0;
  for (STACK *t = s; t != NULL; t = t->next) {
    r++;
  }
  return r;
}

void gc(STACK **old, int items) {
  STACK *t;
  for (; items > 0 && *old != NULL; items--) {
    t = *old;
    *old = (*old)->next;
    free(t);
  }
}
```

c: cactus stack

nil is empty

grows on push

manual GC

```
#include <stdio.h>
#include <cactus_stack.h>

int sum(STACK *tos) {
  int a = 0;
  for (int p = 0; tos != NULL;) {
    tos = pop(tos, &p);
    a += p;
  }
  return a;
}

void print_sum(STACK *s) {
  printf("%d items: sum = %d\n", depth(s), sum(s));
}

int main() {
  STACK *s1 = push(NULL, 7);
  STACK *s2 = push(push(s1, 7), 11);
  s1 = push(push(push(s1, 2), 9), 4);

  STACK *s3 = push(s1, 17);
  s1 = push(s1, 3);
  print_sum(s1);
  print_sum(s2);
  print_sum(s3);
}
```

go: cactus stack

nil is empty

grows on push

automatic GC

```
package main

import "fmt"

type stack struct {
  data int
  tail *stack
}

func (s stack) Push(v int) (r stack) {
  r = stack{data: v, tail: &s}
  return
}

func (s stack) Pop() (v int, r stack) {
  return s.data, *s.tail
}

func (s stack) Depth() (r int) {
  for t := s.tail; t != nil; t = t.tail {
    r++
  }
  return
}
```

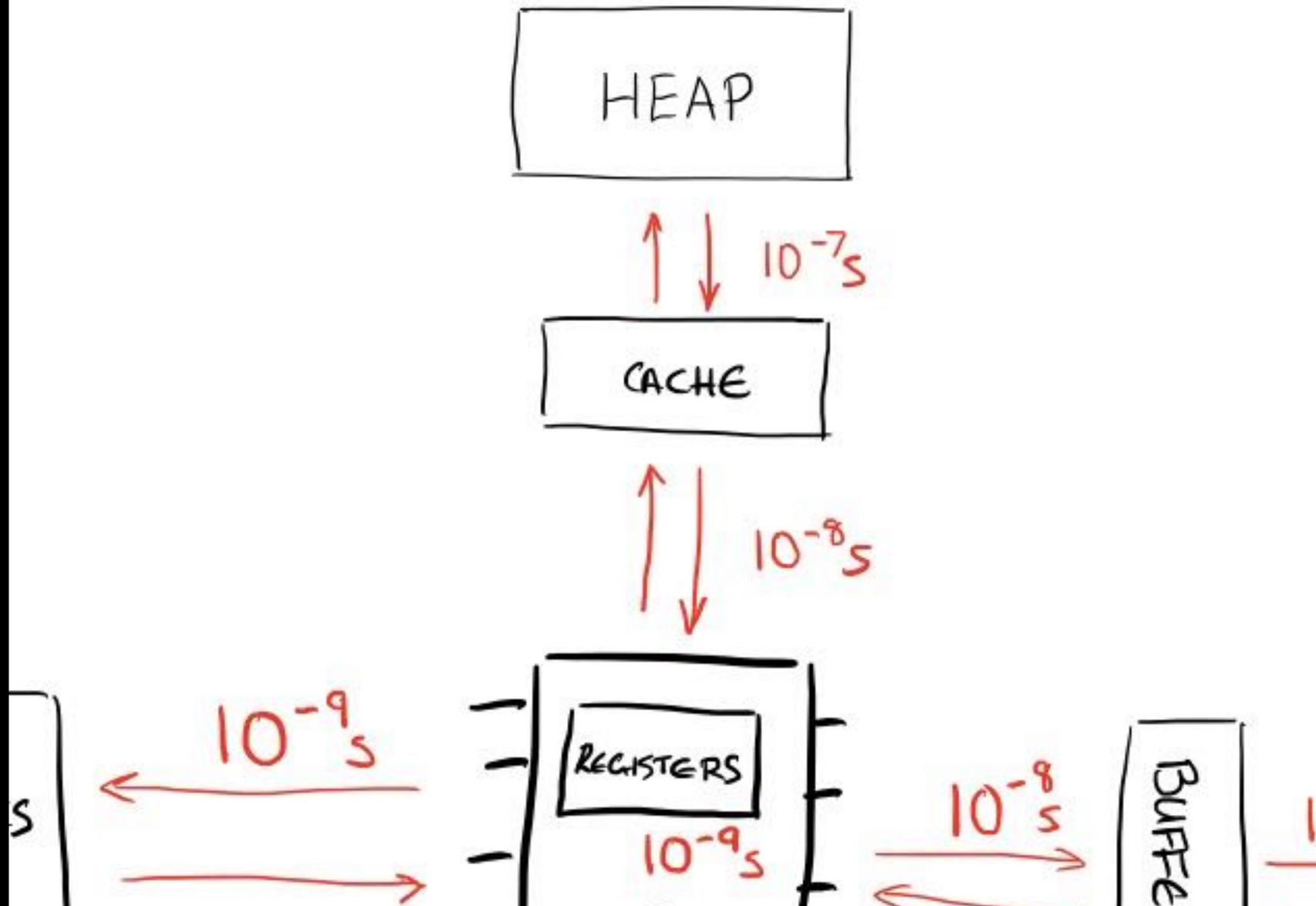
```
func (s *stack) PrintSum() {
  fmt.Printf("sum(%v): %v\n", s.Depth(), s.Sum())
}

func (s stack) Sum() (r int) {
  for t, n := s, 0; t.tail != nil; r += n {
    n, t = t.Pop()
  }
  return
}

func main() {
  s1 := new(stack).Push(7)
  s2 := s1.Push(7).Push(11)
  s1 = s1.Push(2).Push(9).Push(4)
  s3 := s1.Push(17)
  s1 = s1.Push(3)

  s1.PrintSum()
  s2.PrintSum()
  s3.PrintSum()
}
```

caches

word-aligned

discontiguous

label-addressable



c: hash map

array

associative arrays

search

```
#include <limits.h>

struct map {
  int size;
  assoc_array_t **chains;
};
typedef struct map map_t;

map_t *map_new(int size) {
  map_t *m = malloc(sizeof(map_t));
  m->chains = malloc(sizeof(assoc_array_t*) * size);
  for (int i = 0; i < size; i++) {
    m->chains[i] = NULL;
  }
  m->size = size;
  return m;
}

int map_chain(map_t *m, char *k) {
  unsigned long int b;
  for (int i = strlen(k) - 1; b < ULONG_MAX && i > 0; i--) {
    b = b << 8;
    b += k[i];
  }
  return b % m->size;
}
```

```
char *map_get(map_t *m, char *k) {
  search_t *s = search_find(m->chains[map_chain(m, k)], k);
  if (s != NULL) {
    return s->value;
  }
  return NULL;
}

void map_set(map_t *m, char *k, char *v) {
  int b = map_chain(m, k);
  assoc_array_t *a = m->chains[b];
  search_t *s = search_find(a, k);
  if (s->value != NULL) {
    s->cursor->value = strdup(v);
  } else {
    assoc_array_t *n = assoc_array_new(k, v);
    if (s->cursor == a) {
      n->next = s->cursor;
      m->chains[b] = n;
    } else if (s->cursor == NULL) {
      s->memo->next = n;
    } else  {
      n->next = s->cursor;
      s->memo->next = n;
    }
  }
  free(s);
}
```

c: hash map

array

associative arrays

search

```
#include <stdlib.h>
#include <string.h>

struct assoc_array {
  char *key;
  void *value;
  struct assoc_array *next;
};
typedef struct assoc_array assoc_array_t;

assoc_array_t *assoc_array_new(char *k, char *v) {
  assoc_array_t *a = malloc(sizeof(assoc_array_t));
  a->key = strdup(k);
  a->value = strdup(v);
  a->next = NULL;
  return a;
}

char *assoc_array_get_if(assoc_array_t *a, char *k) {
  char *r = NULL;
  if (a != NULL && strcmp(a->key, k) == 0) {
    r = strdup(a->value);
  }
  return r;
}
```

c: hash map

array

associative arrays

search

```
struct search {
  char *term, *value;
  assoc_array_t *cursor, *memo;
};
typedef struct search search_t;

search_t *search_new(assoc_array_t *a, char *k) {
  search_t *s = malloc(sizeof(search_t));
  s->term = k;
  s->value = NULL;
  s->cursor = a;
  s->memo = NULL;
  return s;
}

void search_step(search_t *s) {
  s->value = assoc_array_get_if(s->cursor, s->term);
}

int searching(search_t *s) {
  return s->value == NULL && s->cursor != NULL;
}

search_t *search_find(assoc_array_t *a, char *k) {
  search_t *s = search_new(a, k);
  for (search_step(s); searching(s); search_step(s)) {
    s->memo = s->cursor;
    s->cursor = s->cursor->next;
  }
  return s;
}
```

c: hash map

array

associative arrays

search

```
#include <stdio.h>
#include “map.h

int main( int argc, char **argv ) {
  map_t *m = map_new(1024);
  map_set(m, "apple", "rosy");                    rosy
  printf("%s\n", map_get(m, "apple"));
  map_set(m, "blueberry", "sweet");
  printf("%s\n", map_get(m, "blueberry"));        sweet
  map_set(m, "cherry", "pie");
  printf("%s\n", map_get(m, "cherry"));           pie
  map_set(m, "cherry", "tart");
  printf("%s\n", map_get(m, "cherry"));           tart
  printf("%s\n", map_get(m, “tart"));             (null)
}
```

go: assoc array

searcher

```
package main

type AssocArray struct {
  Key string
  Value interface{}
  Next *AssocArray
};

func (a *AssocArray) GetIf(k string) (r interface{}) {
  if a != nil && a.Key == k {
    r = a.Value
  }
  return
}
```

```
type Search struct {
  Term string
  Value interface{}
  Cursor, Memo *AssocArray
};

func (s *Search) Step() *Search {
  s.Value = s.Cursor.GetIf(s.Term)
  return s
}

func (s *Search) Searching() bool {
  return s.Value == nil && s.Cursor != nil
}

func Find(a *AssocArray, k string) (s *Search) {
  s = &Search{ Term: k, Cursor: a }
  for s.Step(); s.Searching(); s.Step() {
    s.Memo = s.Cursor
    s.Cursor = s.Cursor.Next
  }
  return
}
```

go: assoc array

searcher

```
package main

type AssocArray struct {
  Key string
  Value interface{}
  Next *AssocArray
};

func (a *AssocArray) GetIf(k string) (r interface{}) {
  if a != nil && a.Key == k {
    r = a.Value
  }
  return
}
```

```
type Search struct {
  Term string
  Value interface{}
  Cursor, Memo *AssocArray
};

func (s *Search) Step() *Search {
  s.Value = s.Cursor.GetIf(s.Term)
  return s
}

func (s *Search) Searching() bool {
  return s.Value == nil && s.Cursor != nil
}

func Find(a *AssocArray, k string) (s *Search) {
  s = &Search{ Term: k, Cursor: a }
  for s.Step(); s.Searching(); s.Step() {
    s.Memo = s.Cursor
    s.Cursor = s.Cursor.Next
  }
  return
}
```

ruby: hash map

associative array

searcher

slice of arrays

```
package main

type Map []*AssocArray

func (m Map) Chain(k string) int {
  var c uint
  for i := len(k) - 1; i > 0; i-- {
    c = c << 8
    c += (uint)(k[i])
  }
  return int(c) % len(m)
}

func (m Map) Get(k string) (r interface{}) {
  if s := Find(m[m.Chain(k)], k); s != nil {
    r = s.Value
  }
  return
}
```

```
func (m Map) Set(k string, v interface{}) {
  c := m.Chain(k)
  a := m[c]
  s := Find(a, k)
  if s.Value != nil {
    s.Cursor.Value = v
  } else {
    n := &AssocArray{ Key: k, Value: v }
    switch {
    case s.Cursor == a:
      n.Next = s.Cursor
      m[c] = n
    case s.Cursor == nil:
      s.Memo.Next = n
    default:
      n.Next = s.Cursor
      s.Memo.Next = n
    }
  }
}
```

ruby: map

hashmap v. native

```
package main
import "fmt"

func main() {
  hashmap():
  nativemap();
}

func hashmap() {
  m := make(Map, 1024)
  m.Set("apple", "rosy")
  fmt.Printf("%v\n", m.Get("apple"))

  m.Set("blueberry", "sweet")
  fmt.Printf("%v\n", m.Get("blueberry"))

  m.Set("cherry", "pie")
  fmt.Printf("%v\n", m.Get("cherry"))

  m.Set("cherry", "tart")
  fmt.Printf("%v\n", m.Get("cherry"))

  fmt.Printf("%v\n", m.Get("tart"))
}
```

```
func nativemap() {
  m := make(map[string] interface{})
  m["apple"] = "rosy"
  fmt.Printf("%v\n", m["apple"])

  m["blueberry"] = "sweet"
  fmt.Printf("%v\n", m["blueberry"])

  m["cherry"] = "pie"
  fmt.Printf("%v\n", m["cherry"])

  m["cherry"] = "tart"
  fmt.Printf("%v\n", m["cherry"])

  fmt.Printf("%v\n", m["tart"])
}
```

cgo

inline c

Go & cgo: integrating existing C code with Go

Andreas Krennmair <ak@synflood.at>

Golang User Group Berlin

Twitter: @der_ak

assembly

Go & Assembly

Caleb Doxsey *Feb 5, 2013 at 10:28PM*

One of my favorite parts about Go is its unwavering focus on utility. Sometimes we place so much emphasis on language design that we forget all the other things programming involves. For example:

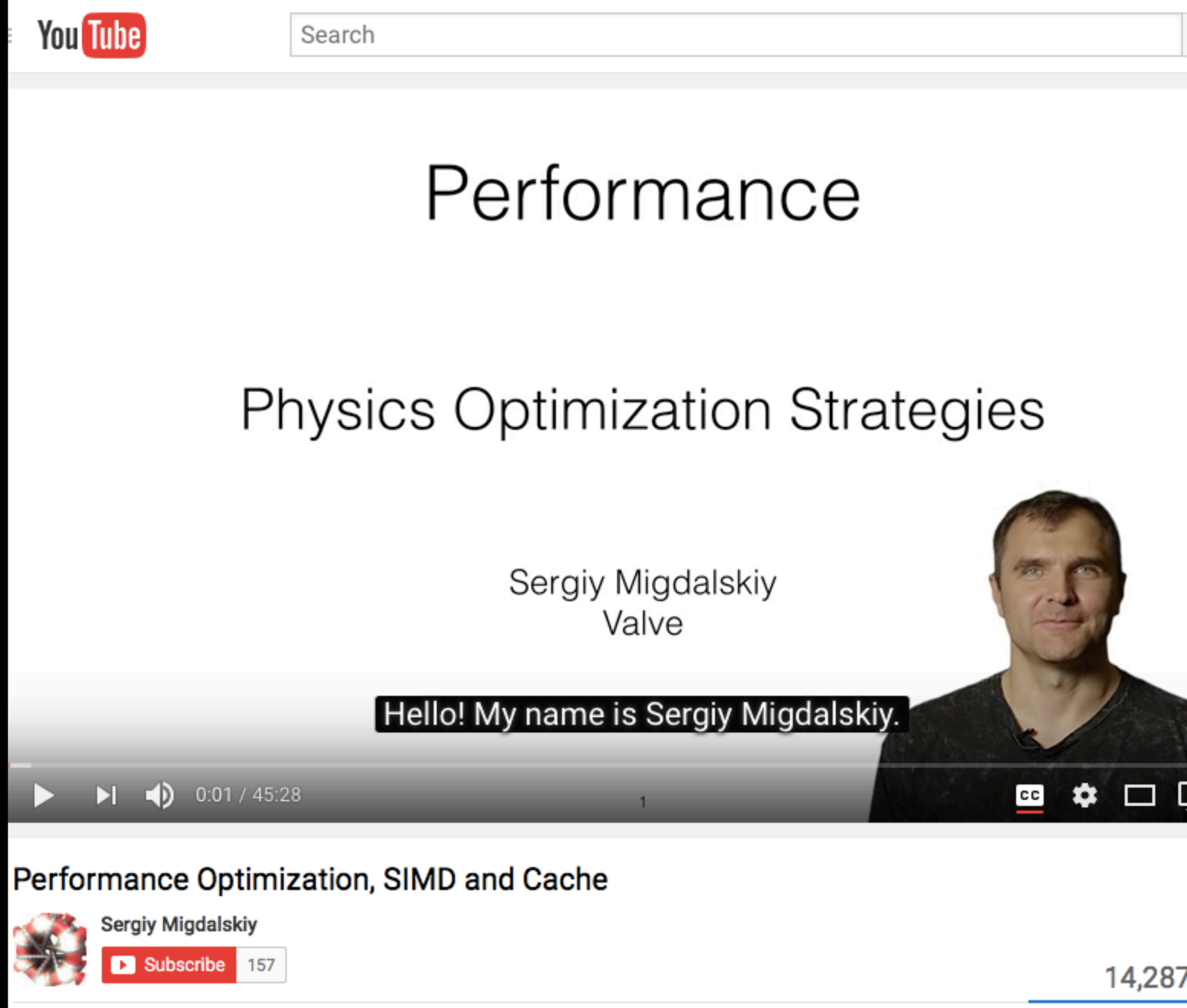
- Go's compiler is fast
- Go comes with a robust standard library
- Go works on a multitude of platforms
- Go comes with a complete set of documentation available from the command line / a local web server / the internet
- All Go code is statically compiled so deployment is trivial
- The entirety of the Go source code is available for perusal in an easy format online (like this)
- Go has a well defined (and documented) grammar for parsing. (unlike C++ or Ruby)
- Go comes with a package management tool. `go get X` (for example `go get code.google.com/p/go.net/websocket`)
- Like all languages Go has a set of style guidelines, some enforced by the compiler

SIMD

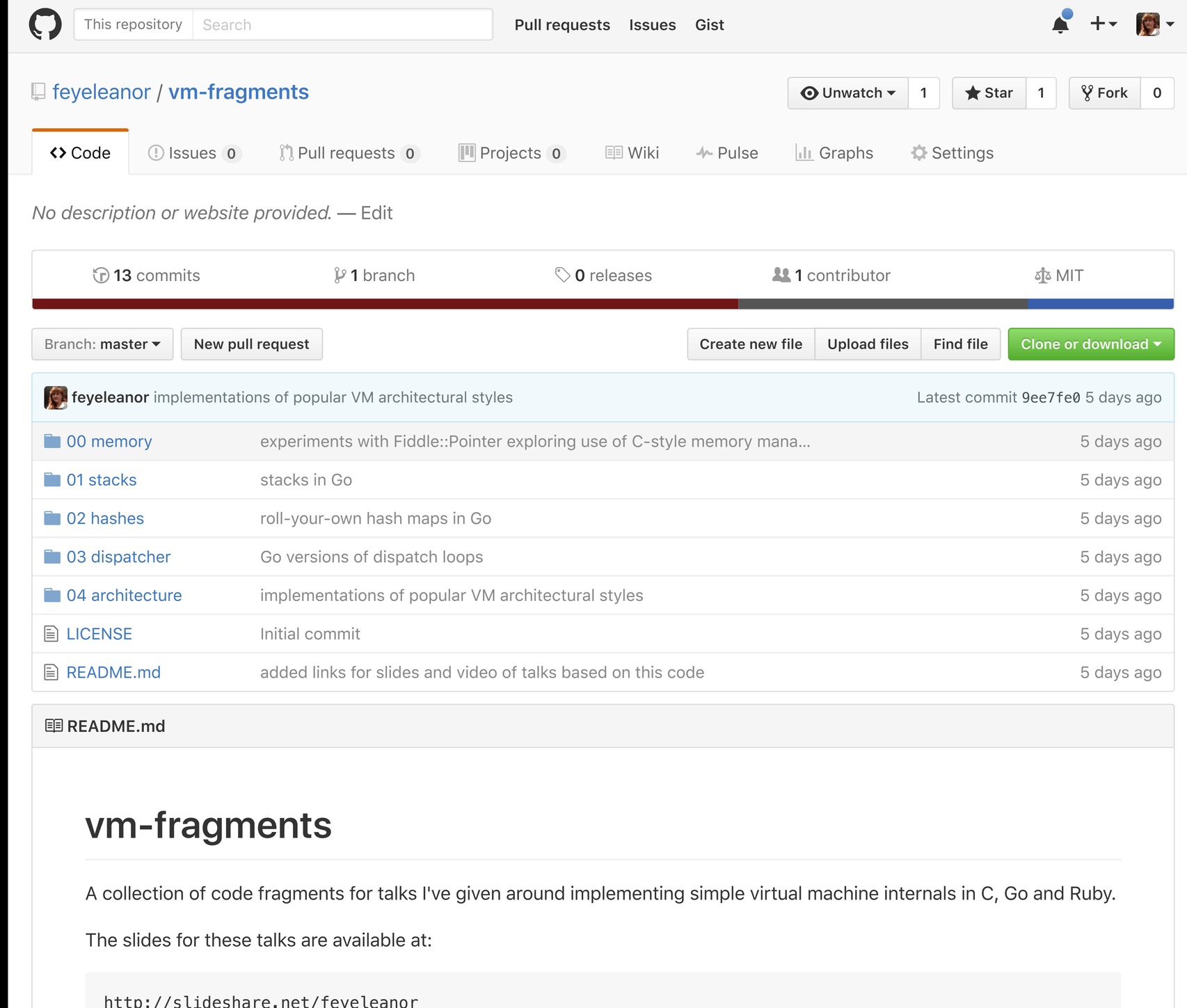
branch prediction

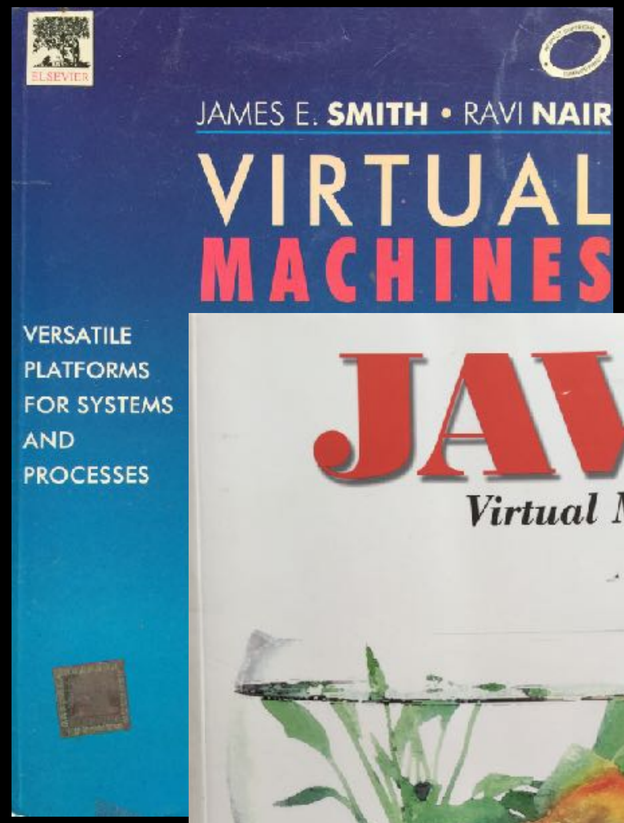
cache misses

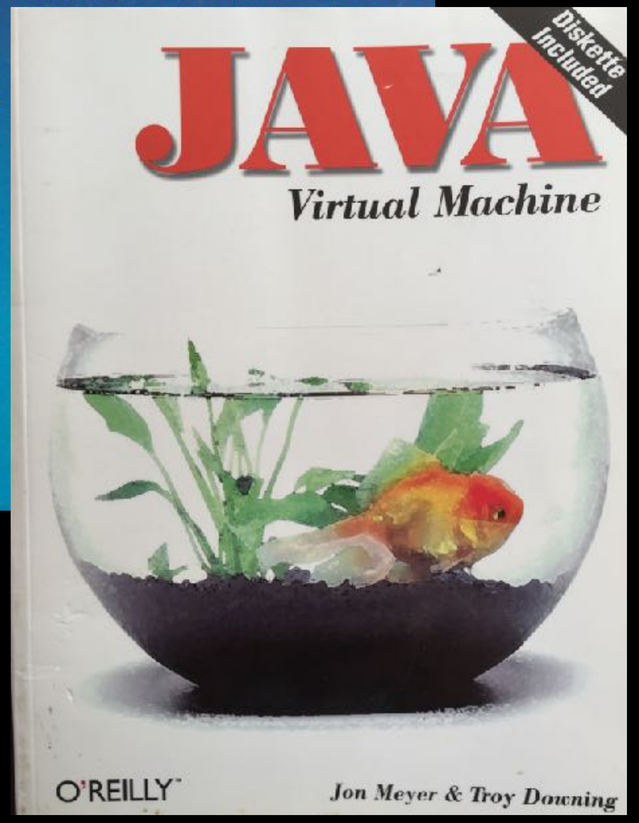
memory latency

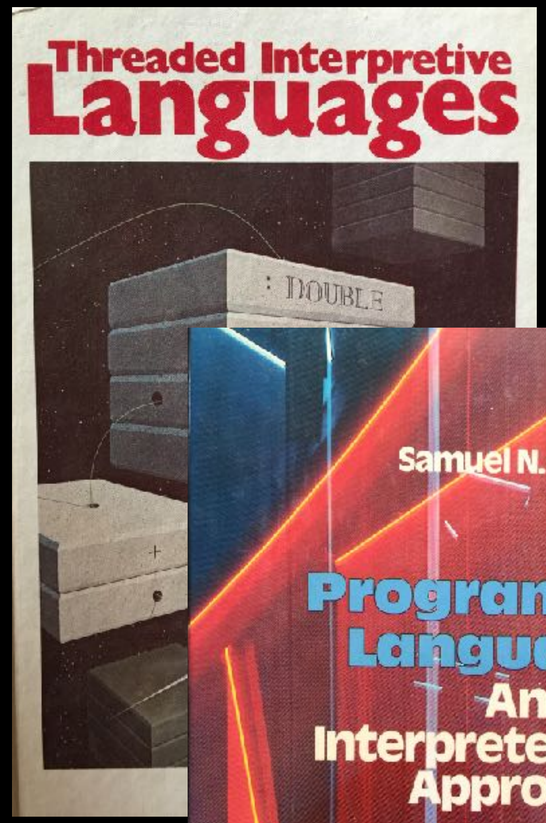


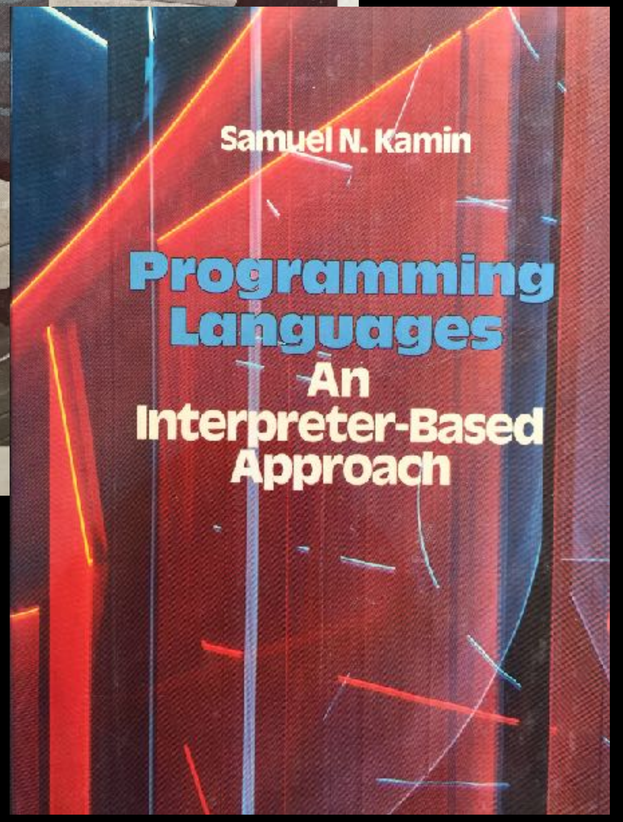
source code


This repository
Search
Pull requests
Issues
Gist
feyeleanor / vm-fragments
Unwatch 1
Star 1
Fork 0
Code
Issues 0
Pull requests 0
Projects 0
Wiki
Pulse
Graphs
Settings
No description or website provided. — Edit
13 commits
1 branch
0 releases
1 contributor
MIT
Branch: master
New pull request
Create new file
Upload files
Find file
Clone or download
feyeleanor implementations of popular VM architectural styles
Latest commit 9ee7fe0 5 days ago
00 memory
experiments with Fiddle::Pointer exploring use of C-style memory mana...
5 days ago
01 stacks
stacks in Go
5 days ago
02 hashes
roll-your-own hash maps in Go
5 days ago
03 dispatcher
Go versions of dispatch loops
5 days ago
04 architecture
implementations of popular VM architectural styles
5 days ago
LICENSE
Initial commit
5 days ago
README.md
added links for slides and video of talks based on this code
5 days ago
README.md
vm-fragments
A collection of code fragments for talks I've given around implementing simple virtual machine internals in C, Go and Ruby.
The slides for these talks are available at:
http://slideshare.net/feyeleanor


JAMES E. SMITH • RAVI NAIR
VIRTUAL MACHINES
VERSATILE PLATFORMS FOR SYSTEMS AND PROCESSES


JAVA
Virtual Machine
Diskette Included
O'REILLY
Jon Meyer & Troy Downing


Threaded Interpretive Languages
: DOUBLE


Samuel N. Kamin
Programming Languages
An Interpreter-Based Approach