如何用Go模拟CPU

×

蒙卓

华为 – 2012实验室
工程师

成为盘古？

让这个世界里面的人（程序）无法察觉
这个世界是创造出来的

目录

- 计算机的演化历史 – 硬件计算到冯诺伊曼架构
- 构建虚拟世界 – MOS 6502
- 控制单元（control unit）
- 运算逻辑单元（arithmetic logic unit）
- 6502汇编器与链接器
- 未来目标

1970年程序员
CPU 80KHz 单核
内存 64KB 手编磁芯

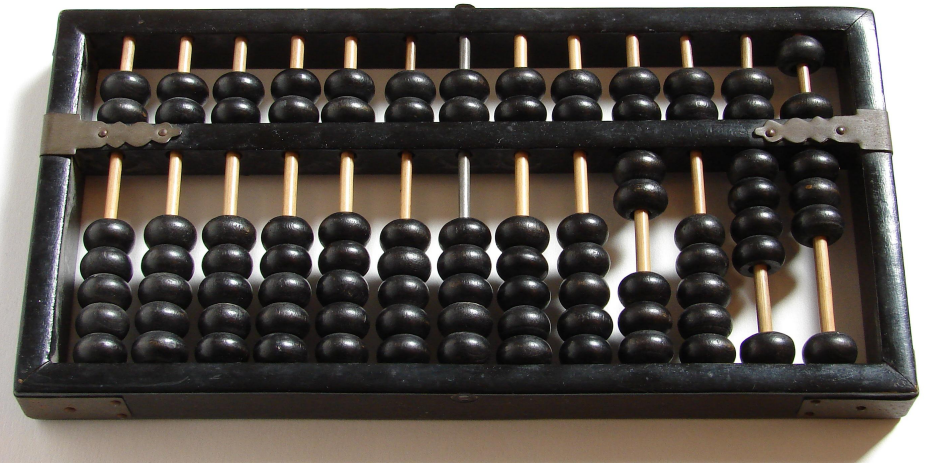
老娘把你送上月球

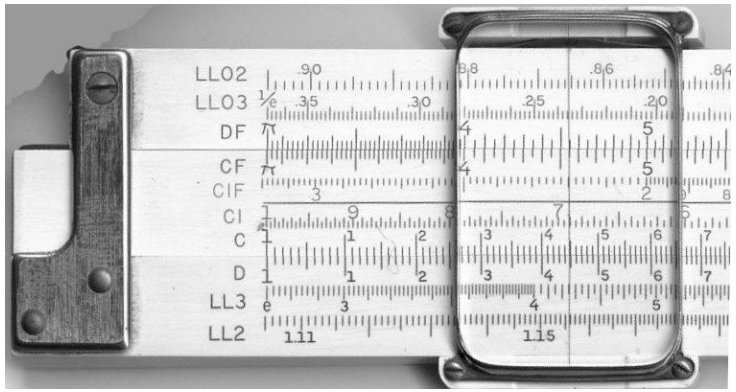
2021年程序员
CPU 2,400,000KHz 4核
内存 8,000,000KB DDR3

呜呜
App内存不足
外卖下不了单

计算机的演化历史

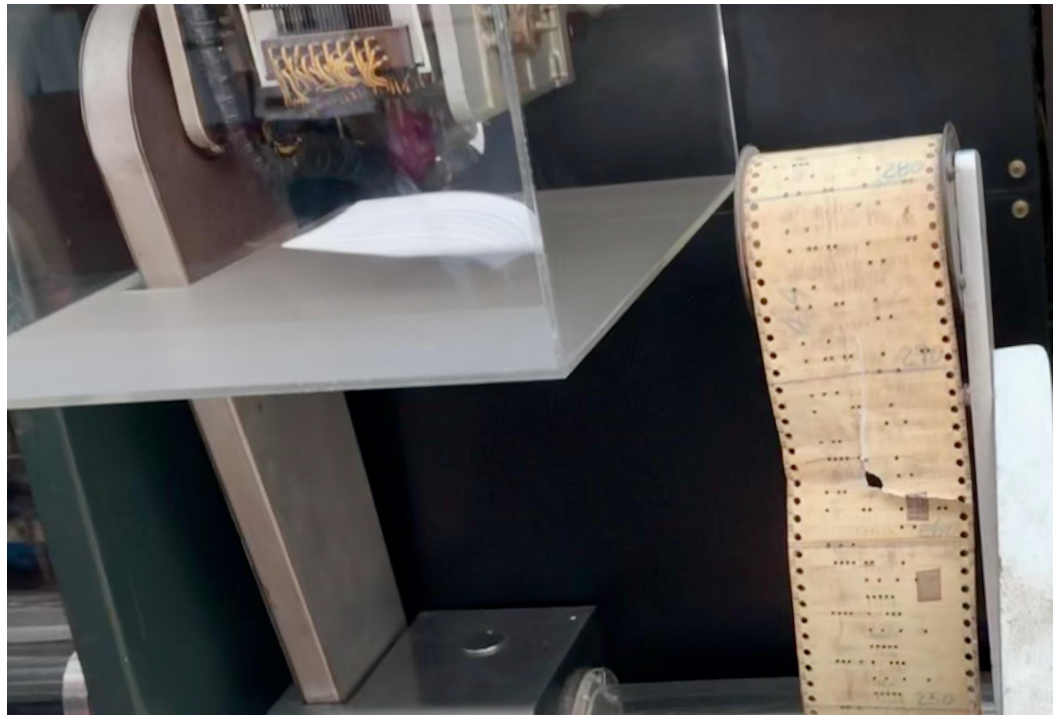
- 一部偷懒的历史
- 硬件“计算机”时代
  - 不擅长计算和记忆的人使用工具帮助计算：算盘，计算尺，手摇计算器
  - 硬件计算机改进支持的算法，需要变更或重新发明整个硬件，比如让算盘支持对数





计算机的演化历史

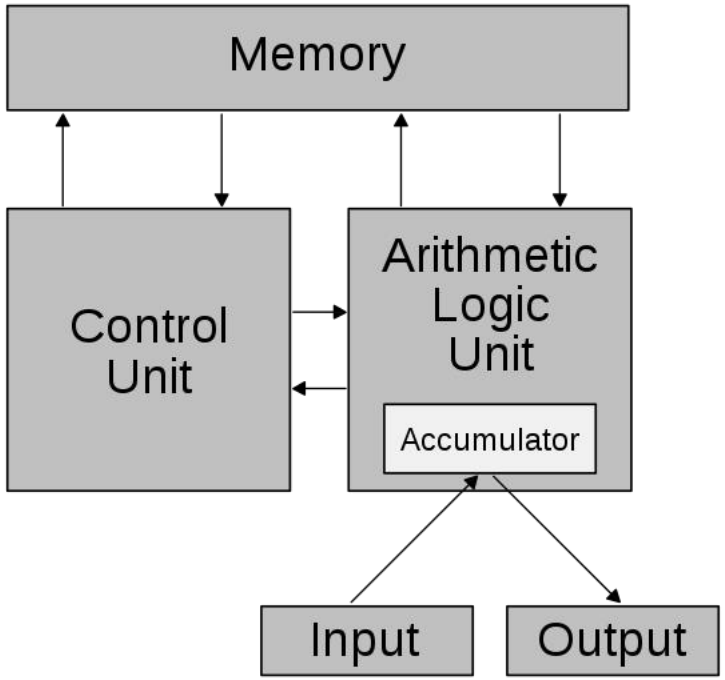
- 改硬件太麻烦了，还慢
  - 继电器计算机
    - 1937年贝尔实验室：model k
    - 计算一次复数速度30—40s，手摇计算机15分钟
    - **从织布机来源的灵感，可使用纸片打孔的方式编写程序（patch的来源）**
  - 真空管计算机
    - 1946年 ENIAC，由于减少了继电器的机械装置速度更快，但寿命短。
    - **因为真空管会发光，吸引飞蛾（bug的来源）**
  - 集成电路计算机
    - 1947年 贝尔实验室：晶体管诞生，现代计算机开始得以应用



本人摄于哈佛计算机学院

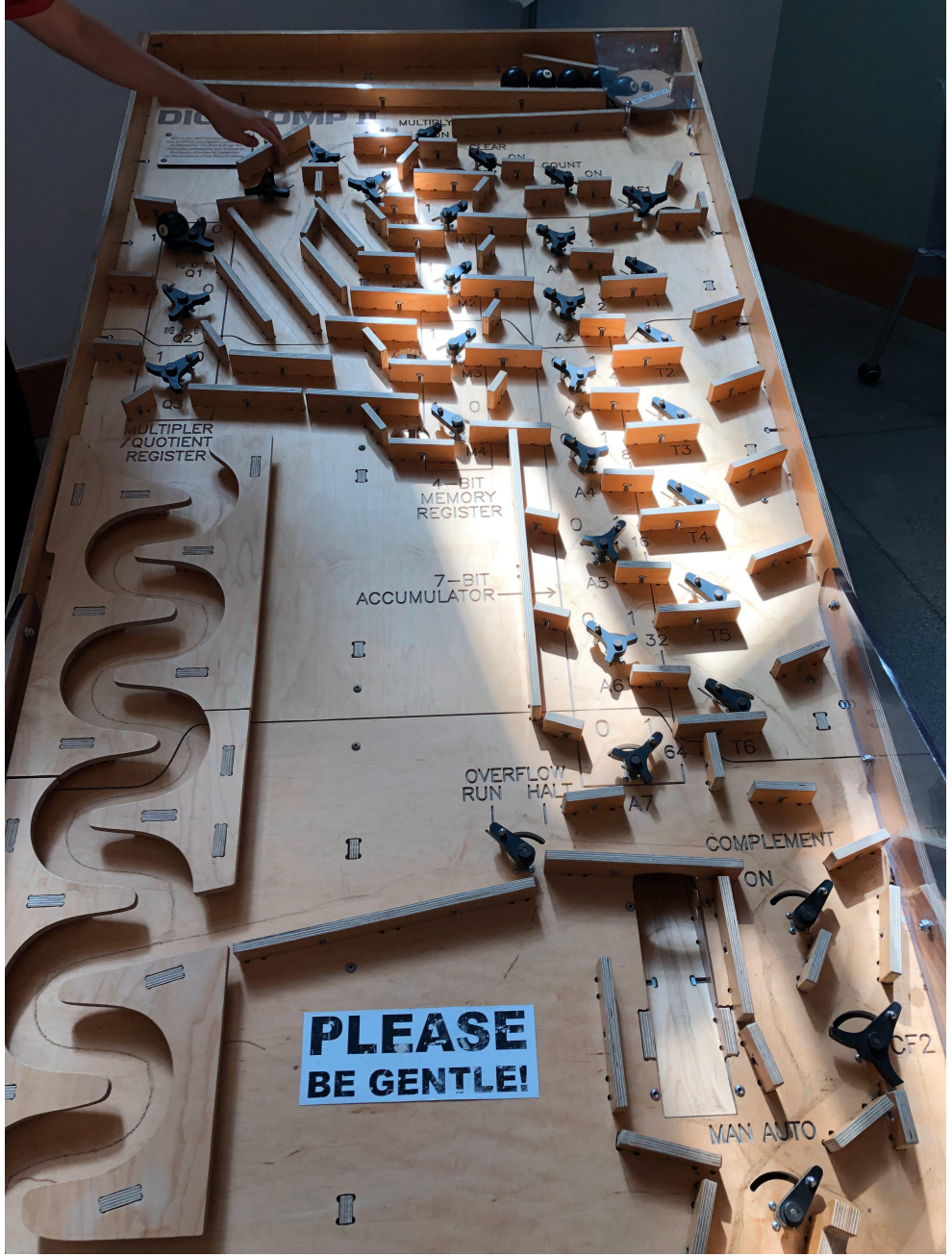
计算机的演化历史

- 因为改纸带比较麻烦
- 冯诺伊曼架构
  - 又称存储程序型计算机
  - 可在运行时改变指令
  - 指令控制指令和数据



计算机的演化历史

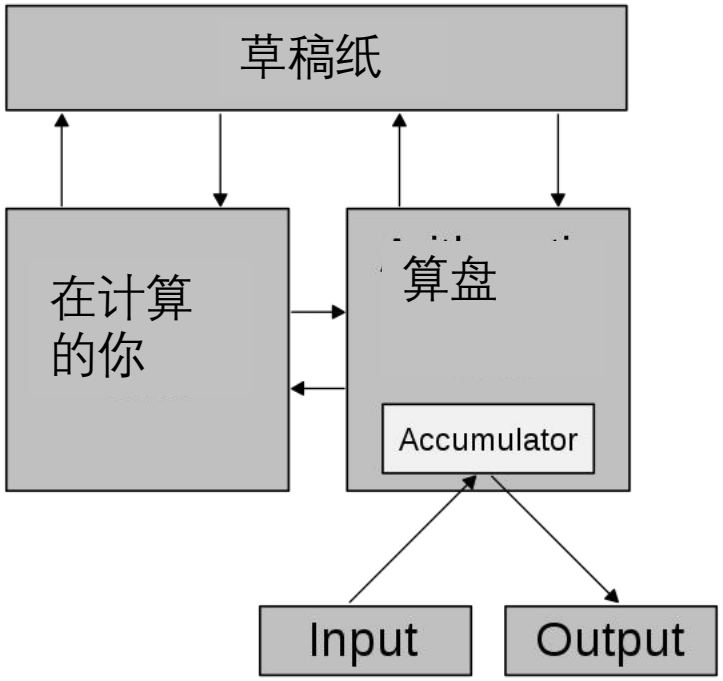
- 因为改纸带比较麻烦
- 冯诺伊曼架构
  - 又称存储程序型计算机
  - 可在运行时改变指令
  - 指令控制指令和数据
  - 用啥实现他老人家可没说

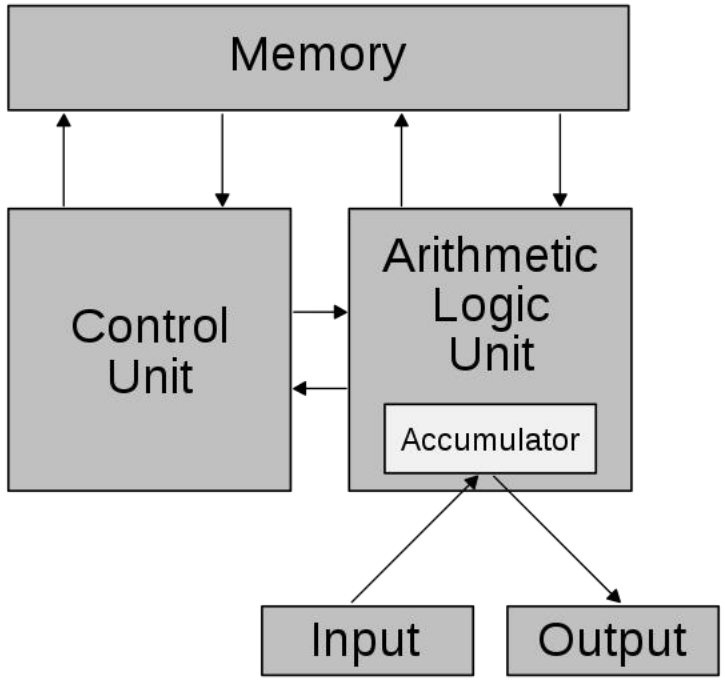


本人摄于MIT CSAI Lab

计算机的演化历史

- 冯诺伊曼架构





计算机的演化历史

- 小结
  - 所有发明都是有基础的
  - 任何的方便都是有代价的：抽象层概念
  - 抽象意味着更慢
- 为啥现在程序员好像更弱了？
- 因为我们处在最好也是最坏的时代
  - 抽象多且环环嵌套
  - 硬件过于复杂
  - 软件基于操作系统等复杂概念
  - 真的快且便宜

Go模拟CPU

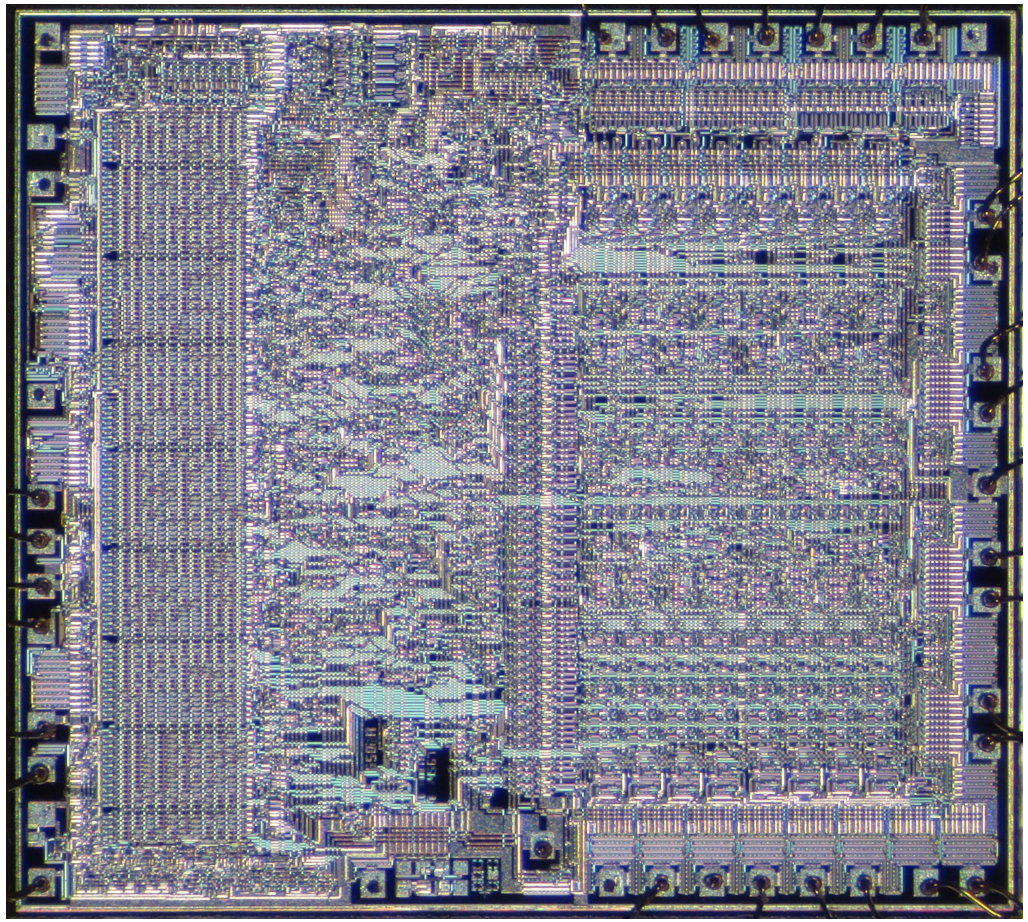
- 如何用Go实现冯诺伊曼架构CPU？
- 简单：一个循环+一个大数组

模拟目标 – MOS 6502

- 诞生于1975年
- MOS 6502应用范围广
- 资料多且易获得
- 简单、容易实现的现代CPU

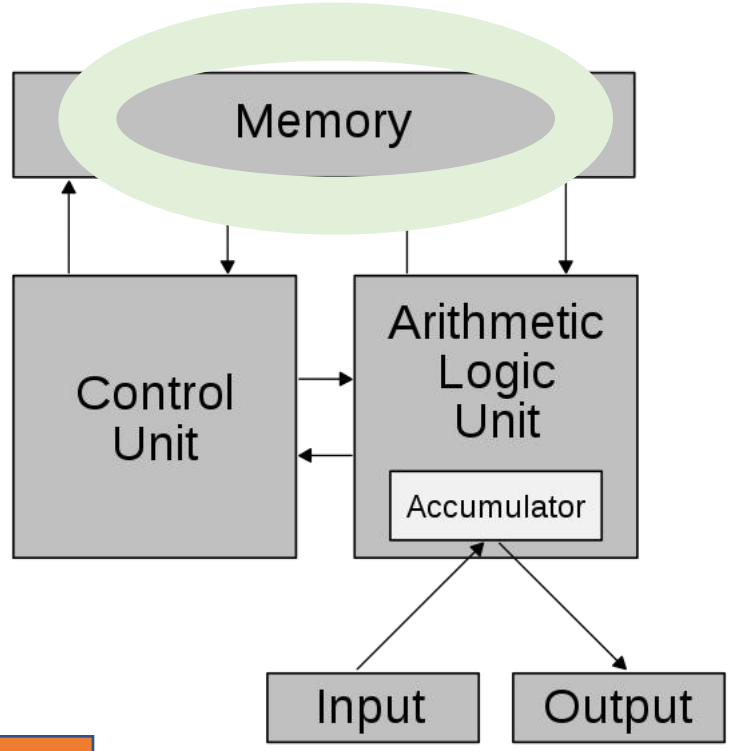
MOS 6502简介

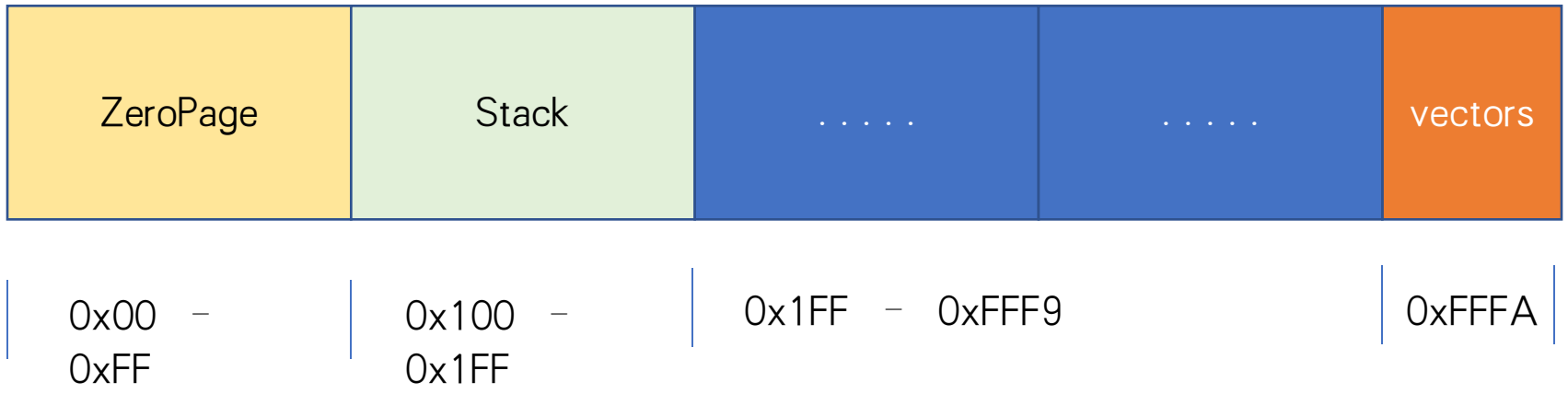
- 8位，变长ISA（CISC）
- 中断（NMI，IRQ）
- 寄存器
  - 1个累加寄存器（Accumulator）
  - 2个地址索引寄存器（X，Y）
  - 1个状态寄存器（PS）
  - 1个16位程序指针寄存器（PC）
  - 1个栈寄存器（SP）



Go模拟内存

- 内存空间 [65536]Byte
- 每个块是一个page （256Byte）





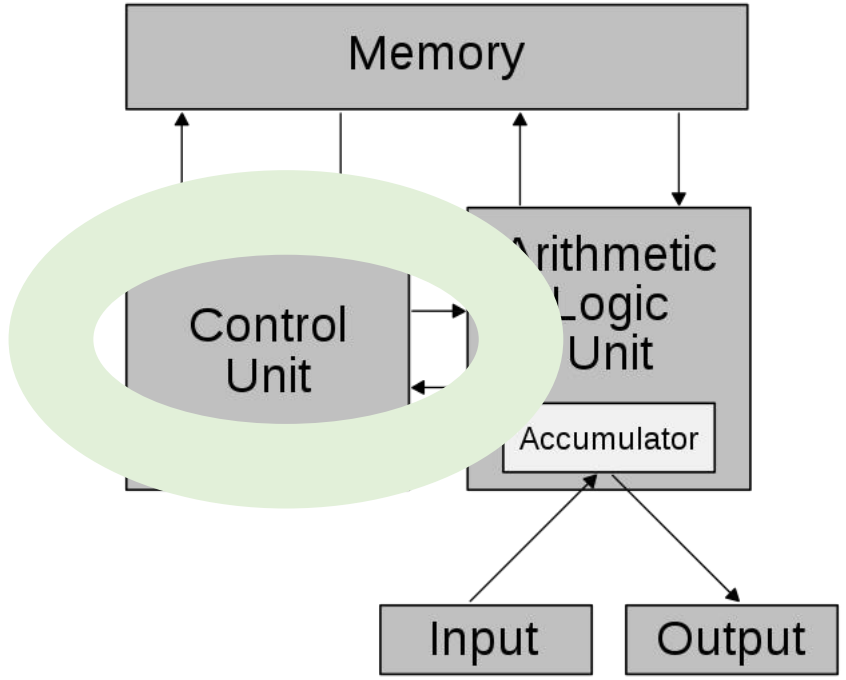
Go模拟6502控制单元

- 读取当前指令：16位PC寄存器
- 执行指令
  - 指令译码器（读出来的指令是什么）
  - 指令执行器（按指令执行）
  - 6502支持NOP指令（啥都不做）

```go
type CPU struct {
  PC uint16
}

func (c *CPU) Run() {
  for {
      op := mem[c.PC]
      ins := c.Decode(op)
      c.Execute(ins)
  }
}

func (c *CPU) Decode(op uint8) (ins Instrction) {
  if op == 0xEA {
    return NOP
  }
}

func (c *CPU) Execute(ins Instruction) {
  switch ins {
    case NOP:
        c.PC += 1
  }
}
```

Go模拟6502控制单元

- 指令结构（instruction）
  - 操作码（valid opcode）
  - 寻址模式 （address mode）
    - Implied
    - Accumulator
    - Immediate
    - Absolute
    - Zeropage，X，Y
    - Indirected，X，Y
  - 指令长度（instruction length）
  - 指令周期（cycle）



- NOP： 啥都不做指令
  - 操作码：0xEA
  - 寻址模式：Implied（ 默认）
  - 指令长度：1
  - 指令周期：1

Go模拟6502控制单元

- 不过6502支持的指令：151个

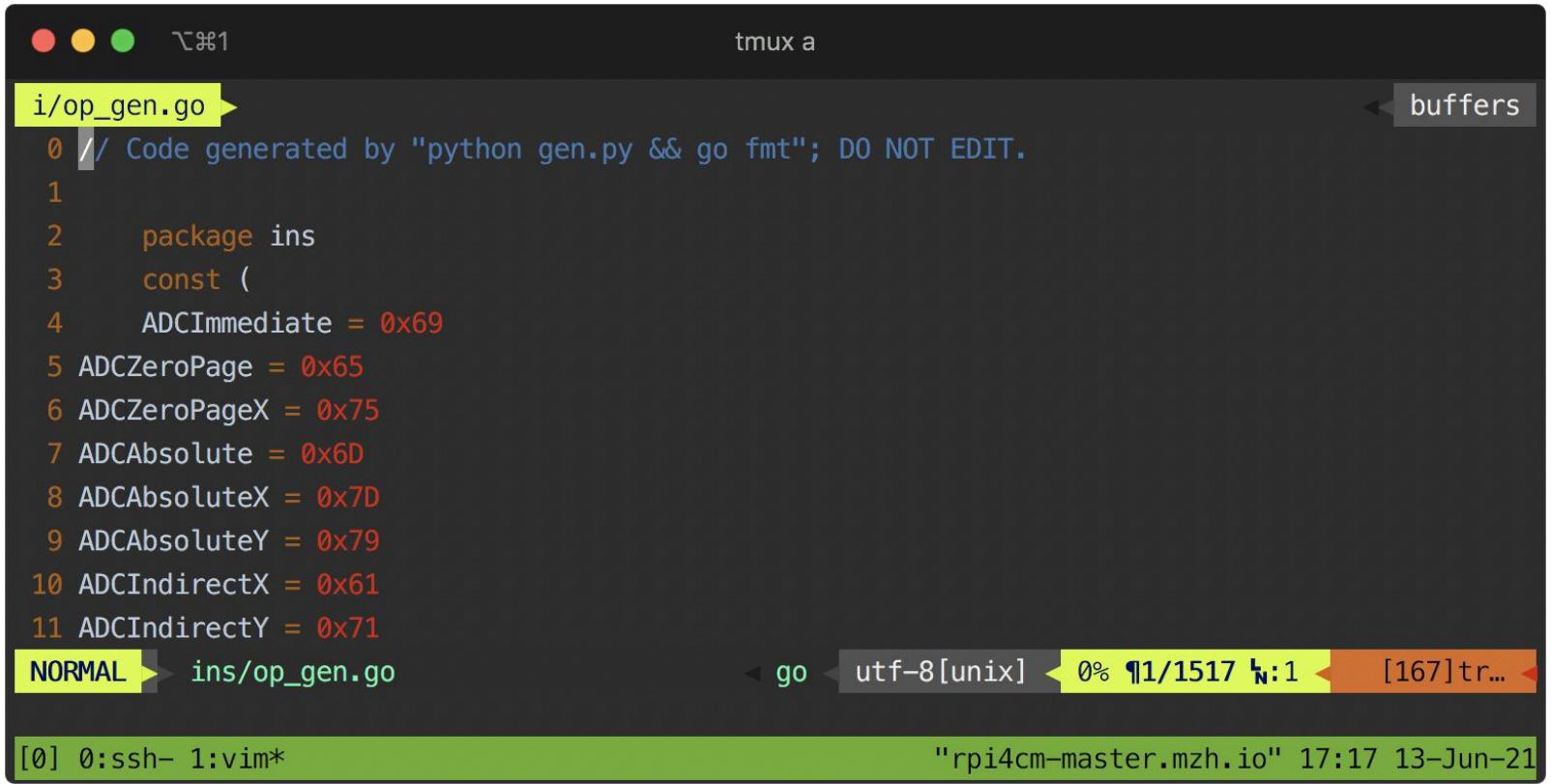
6502 Instruction Set

TOC: Description / Instructions in Detail / "Illegal" Opcodes / Jump Vectors and Stack Operations / Instruction Layout / 65xx-Family

HI	*LO-NIBBLE*															
	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-	BRK impl	ORA X,ind				ORA zpg	ASL zpg		PHP impl	ORA #	ASL A			ORA abs	ASL abs	
1-	BPL rel	ORA ind,Y				ORA zpg,X	ASL zpg,X		CLC impl	ORA abs,Y				ORA abs,X	ASL abs,X	
2-	JSR abs	AND X,ind			BIT zpg	AND zpg	ROL zpg		PLP impl	AND #	ROL A		BIT abs	AND abs	ROL abs	
3-	BMI rel	AND ind,Y				AND zpg,X	ROL zpg,X		SEC impl	AND abs,Y				AND abs,X	ROL abs,X	
4-	RTI impl	EOR X,ind				EOR zpg	LSR zpg		PHA impl	EOR #	LSR A		JMP abs	EOR abs	LSR abs	
5-	BVC rel	EOR ind,Y				EOR zpg,X	LSR zpg,X		CLI impl	EOR abs,Y				EOR abs,X	LSR abs,X	
6-	RTS impl	ADC X,ind				ADC zpg	ROR zpg		PLA impl	ADC #	ROR A		JMP ind	ADC abs	ROR abs	
7-	BVS rel	ADC ind,Y				ADC zpg,X	ROR zpg,X		SEI impl	ADC abs,Y				ADC abs,X	ROR abs,X	
8-		STA X,ind			STY zpg	STA zpg	STX zpg		DEY impl		TXA impl		STY abs	STA abs	STX abs	
9-	BCC rel	STA ind,Y			STY zpg,X	STA zpg,X	STX zpg,Y		TYA impl	STA abs,Y	TXS impl			STA abs,X		
A-	LDY #	LDA X,ind	LDX #		LDY zpg	LDA zpg	LDX zpg		TAY impl	LDA #	TAX impl		LDY abs	LDA abs	LDX abs	
B-	BCS rel	LDA ind,Y			LDY zpg,X	LDA zpg,X	LDX zpg,Y		CLV impl	LDA abs,Y	TSX impl		LDY abs,X	LDA abs,X	LDX abs,Y	
C-	CPY #	CMP X,ind			CPY zpg	CMP zpg	DEC zpg		INY impl	CMP #	DEX impl		CPY abs	CMP abs	DEC abs	
D-	BNE rel	CMP ind,Y				CMP zpg,X	DEC zpg,X		CLD impl	CMP abs,Y				CMP abs,X	DEC abs,X	
E-	CPX #	SBC X,ind			CPX zpg	SBC zpg	INC zpg		INX impl	SBC #	NOP impl		CPX abs	SBC abs	INC abs	
F-	BEQ rel	SBC ind,Y				SBC zpg,X	INC zpg,X		SED impl	SBC abs,Y				SBC abs,X	INC abs,X	

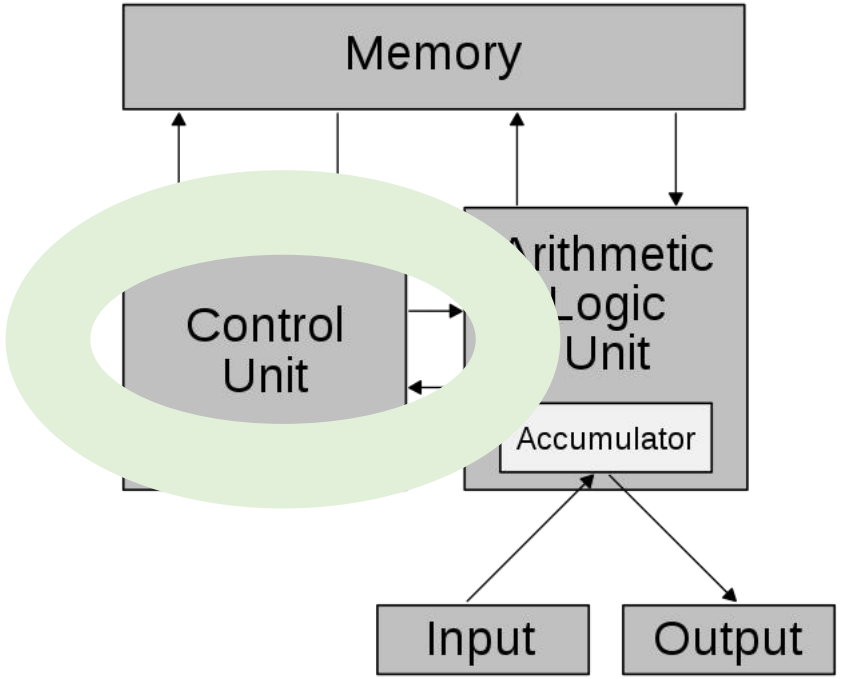
Go模拟6502控制单元

- 不过6502支持的指令：151个
  - html解析器生成对应的指令
    https://www.masswerk.at/6502/6502_instruction_set.html



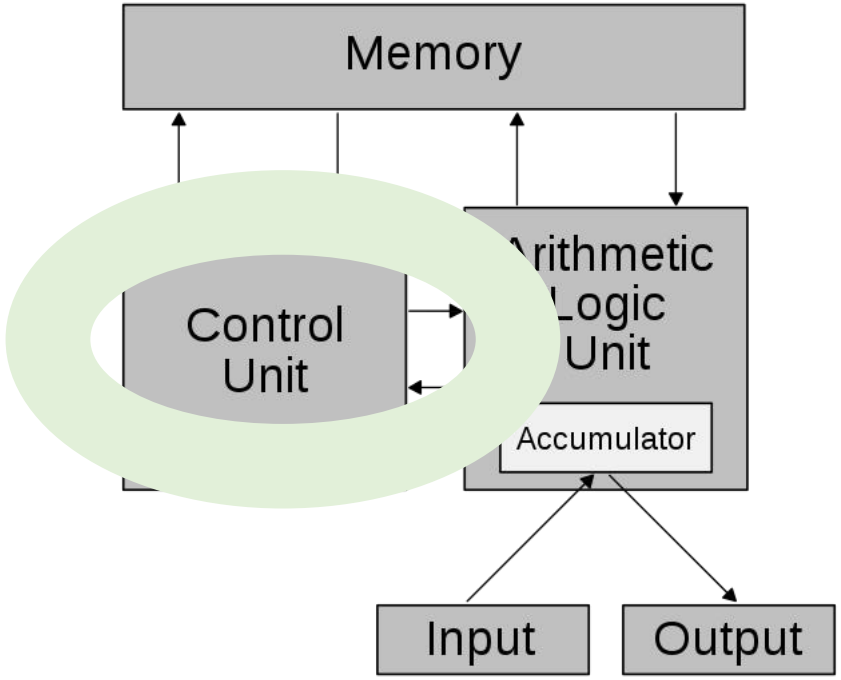
Go模拟6502控制单元

- 指令执行过程：
  - 读取指令码（opcode）
  - 如果有运算数（operand），则读取
  - 根据逻辑执行对应指令
  - 让PC根据字长移动，或跳转（Branch）

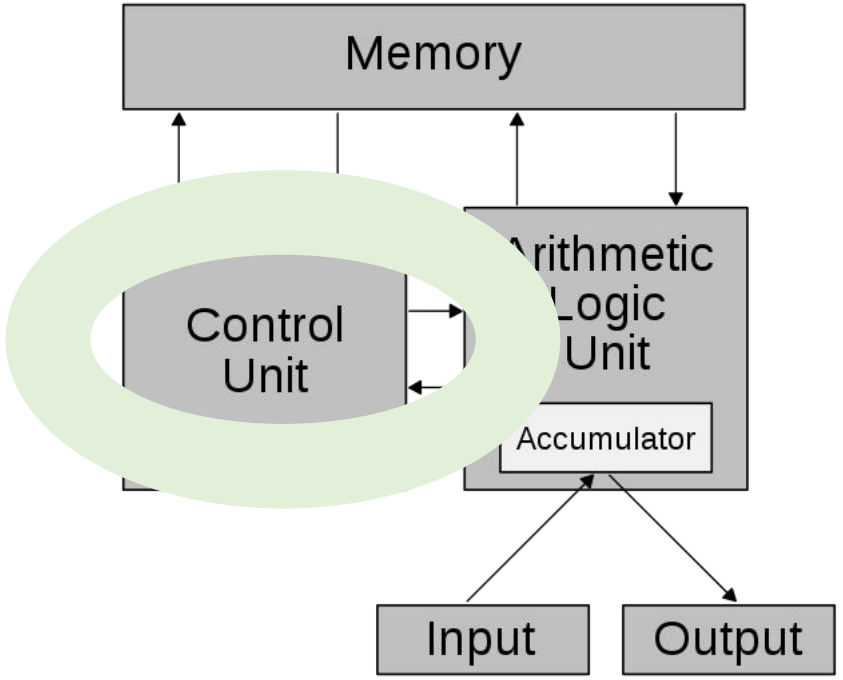


Go模拟6502控制单元

- 指令执行过程：
  - 读取指令码（opcode）
  - 如果有运算数（operand），则读取
  - 根据逻辑执行对应指令
  - 让PC根据字长移动，或跳转
- SP与栈空间、ZeroPage
  - SP跟常见的栈指针一样，压栈SP则减掉对应字长
  - 栈最大0xFF，咦……stackoverflow？
  - ZeroPage因为电路原因，寻址速度比一般指令快



Go模拟6502控制单元



- 指令执行过程：
  - 读取指令码（opcode）
  - 如果有运算数（operand），则读取
  - 根据逻辑执行对应指令
  - 让PC根据字长移动，或跳转
- SP与栈空间、ZeroPage
  - SP跟常见的栈指针一样，压栈SP则减掉对应字长
  - 栈最大0xFF，咦……stackoverflow？
  - ZeroPage因为电路原因，寻址速度比一般指令快
- 中断与向量
  - IRQ ＝ 可忽略的中断 ＝ 水平触发 ＝ 读取0xFFFC里的绝对地址
  - NMI ＝ 不可忽略的中断 ＝ 上沿触发 ＝ 读取0xFFFE里的绝对地址
  - Reset ＝ 重置 ＝ 读取0xFFFA 里的地址

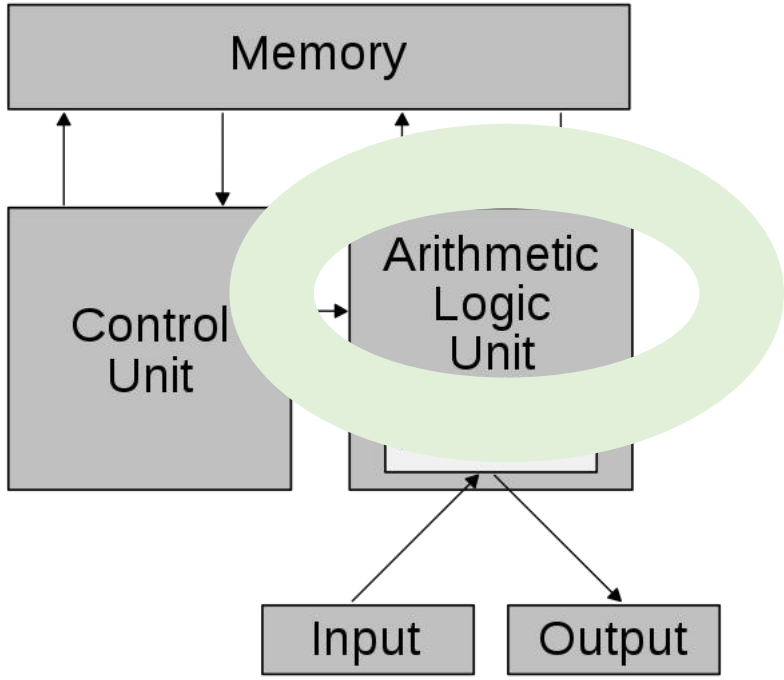
Go模拟6502控制单元

```
for {
        select {
        case <-c.NMI:
                c.pushWordToStack(c.PC)
                c.pushByteToStack(c.PS | FlagUnused)
                c.PC = c.Mem.ReadWord(c.nmiVec)
                c.PS |= FlagIRQDisable
        case <-c.Reset:
                c.ResetF(m)
        default:
        }

        if c.IRQ != 0 && c.PS&FlagIRQDisable == 0 {
                c.pushWordToStack(c.PC)
                c.pushByteToStack(c.PS | FlagUnused)
                c.PC = c.Mem.ReadWord(c.irqVec)
                c.PS |= FlagIRQDisable
        }
```

总共100行左右

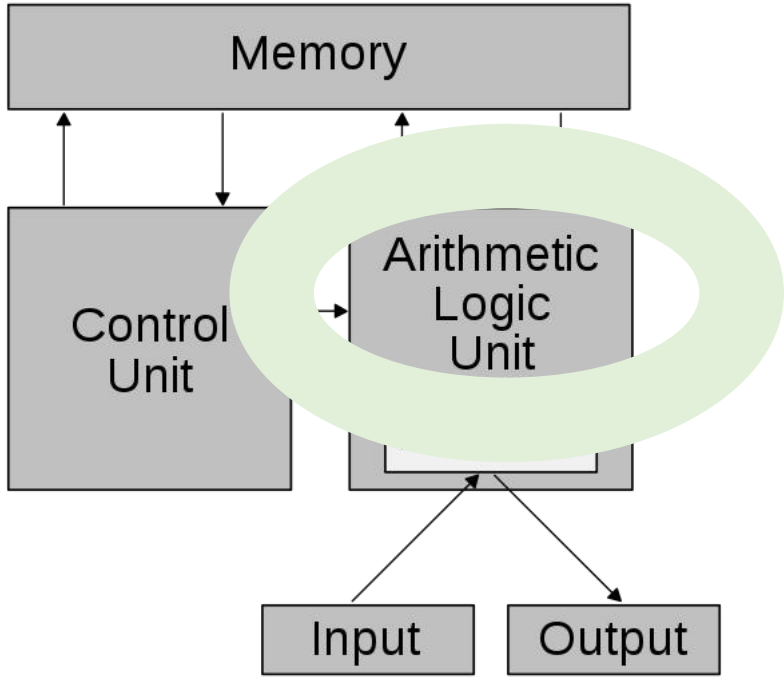
Go模拟6502计算单元

- 指令类型：
  - Uint8 加减运算
  - Uint8 位（布尔）运算
- 影响PS（运行状态寄存器）



Go模拟6502计算单元

- 指令类型：
  - Uint8 加减运算
  - Uint8 位（布尔）运算
- 影响PS（运行状态寄存器）



- ADC： 带进位加法
- A ＝ A+运算数（operand）+Carry
- 如果溢出则PS |= FlagCarry

Go模拟6502计算单元

- 指令类型：
  - Uint8 加减运算
  - Uint8 位（布尔）运算
- 影响PS（运行状态寄存器）
- 加法指令
  - 也就30行
  - 还兼容减法（省电路又省钱）

```
// Arithmetic
case ins.ADC, ins.SBC:
        oper, cycles = c.getOper(i.Mode)
        if i.Name == ins.SBC {
                oper = ^oper
        }

        if c.PS&FlagDecimalMode != 0 {
                err = fmt.Errorf("no decimal mode")
                return
        }

        sameSigned := (c.AC^oper)&FlagNegtive == 0
        sum := uint16(c.AC)
        sum += uint16(oper)
        if c.PS&FlagCarry != 0 {
                sum += 1
        }
        c.AC = uint8(sum & 0xff)
        c.setZN(c.AC)
        if sum > 0xff {
                c.PS |= FlagCarry
        } else {
                c.PS &= ^FlagCarry
        }

        if sameSigned && ((c.AC^oper)&FlagNegtive != 0) {
                c.PS |= FlagOverflow
        } else {
                c.PS &= ^FlagOverflow
        }
```

Go模拟6502

- 小结
  - 冯诺伊曼架构 约等于 有限状态机
  - 用Go实现全部合法指令并测试通过也就1000行左右

Go模拟6502

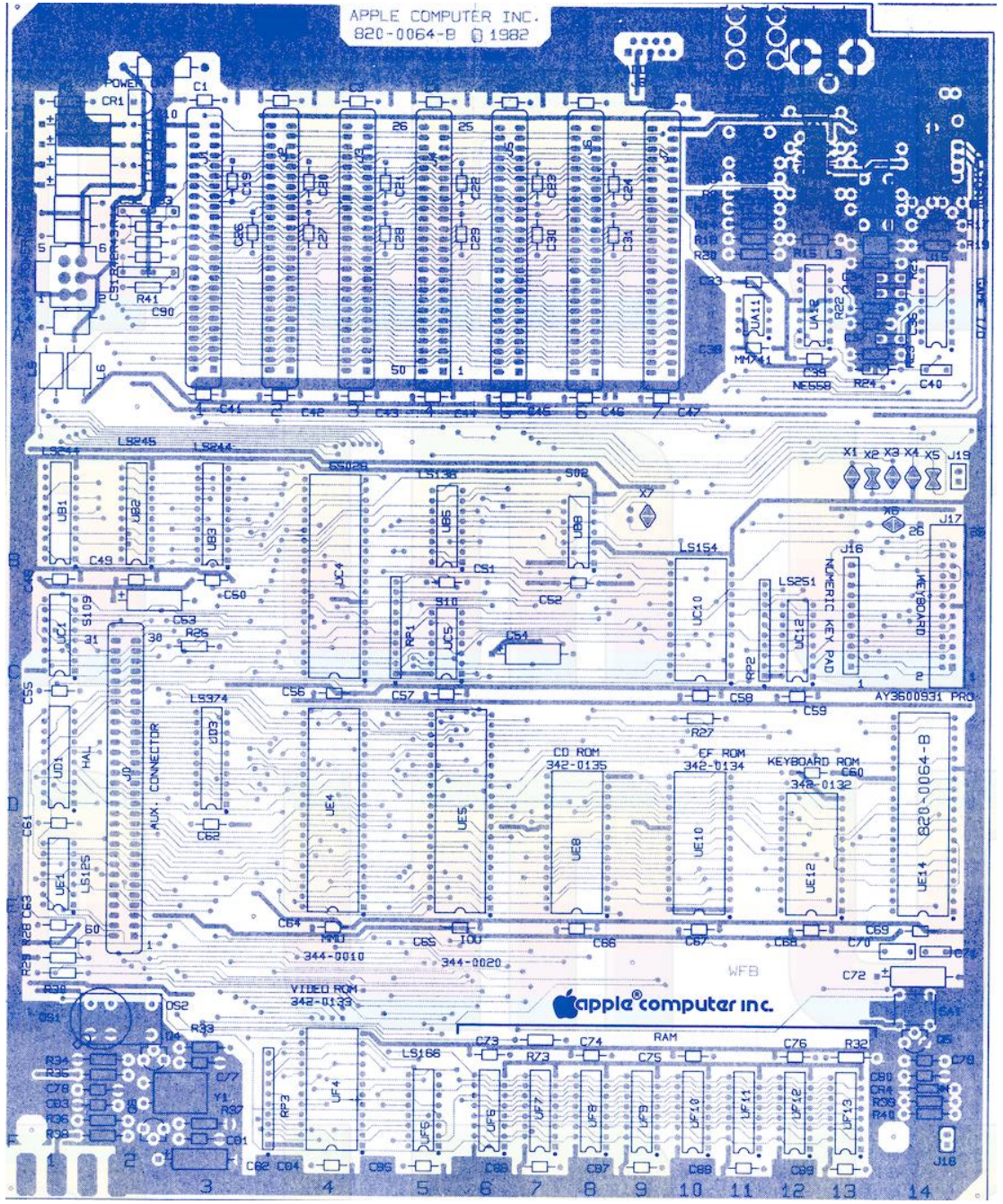
- 小结
  - 冯诺伊曼架构 约等于 有限状态机
  - 用Go实现全部合法指令并测试通过也就1000行左右

说了这么多CPU
那电脑咋工作的？

Go模拟Apple II

- Apple II 规格
  - CPU （MOS 6502）
  - 内存（64KiB）
  - 显示（LoRes 40x24）
  - 显示（280×192像素）
  - 输入（内置键盘）
  - 存储（磁带/5.25英寸）

- 早期内存非常昂贵
  - 4KiB = 5,543 USD（2020）
  - 64KiB = 11,266 USD（2020）
  - 这么看现在的厨子是不是超良心



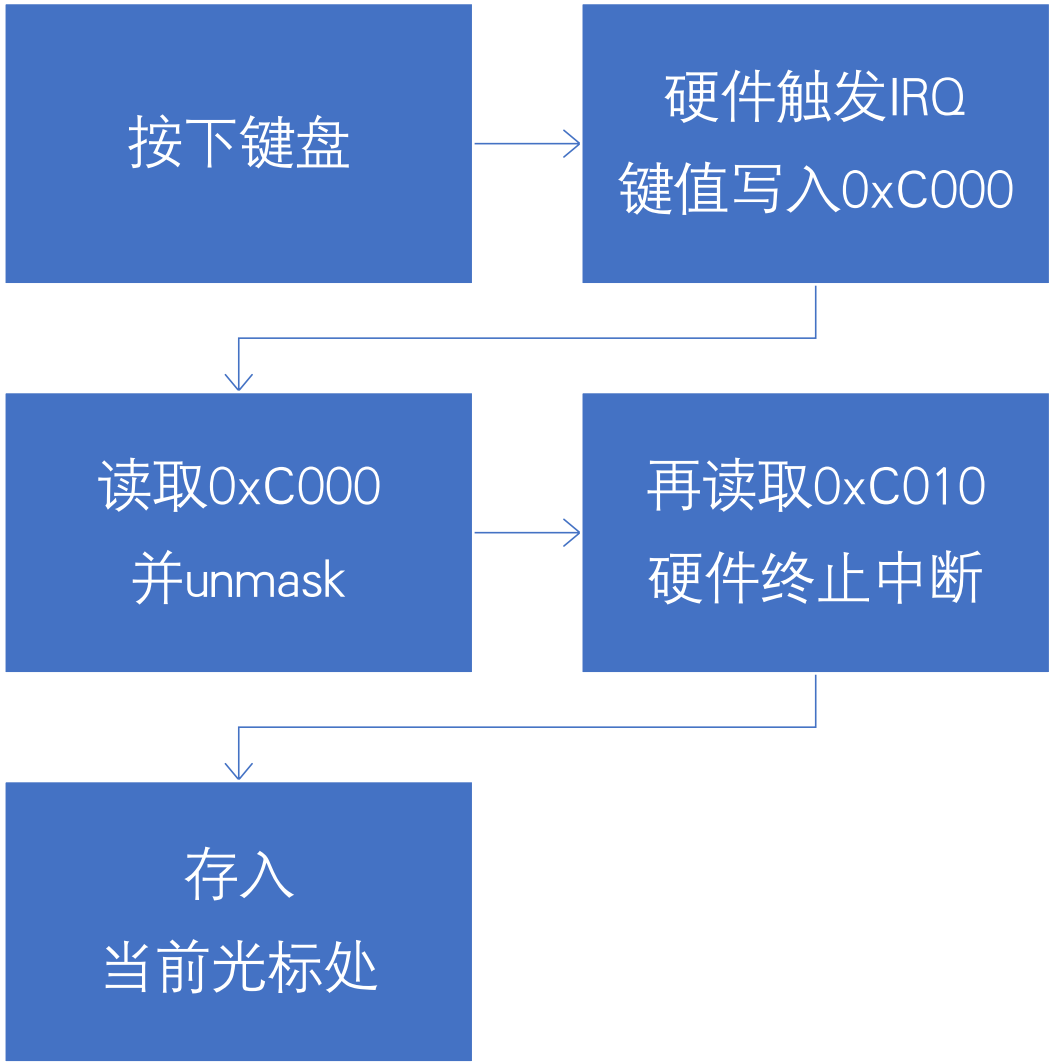
Go模拟Apple II

- 模拟Apple II 规格
  - CPU （MOS 6502）
  - 内存（64KiB）
  - 显示（LoRes 40x24）
  - 输入（内置键盘）
  - ~~存储（磁带/5.25英寸）~~
- 早期内存非常昂贵
  - 4KiB = 5,543 USD（2020）
  - 64KiB = 11,266 USD（2020）
  - 这么看现在的厨子是不是超良心
  - 真果粉应该搞一套Apple II

Go模拟Apple II

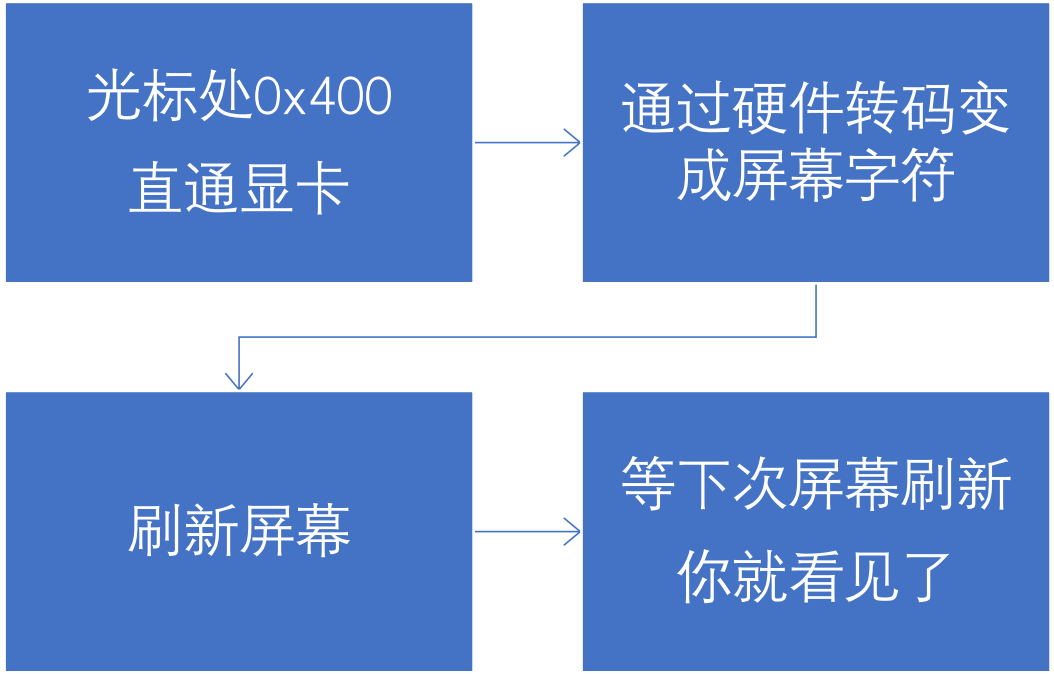
- 又双叒叕是一个大循环
  - Reset -> 启动检查程序
  - 检查完成把电脑规格写入内存指定区域
  - 一直循环monitor 函数
  - 直到有NMI/IRQ中断
- 还是不懂……
  - 就来个键盘敲下，到显示的全流程？

Go模拟Apple II

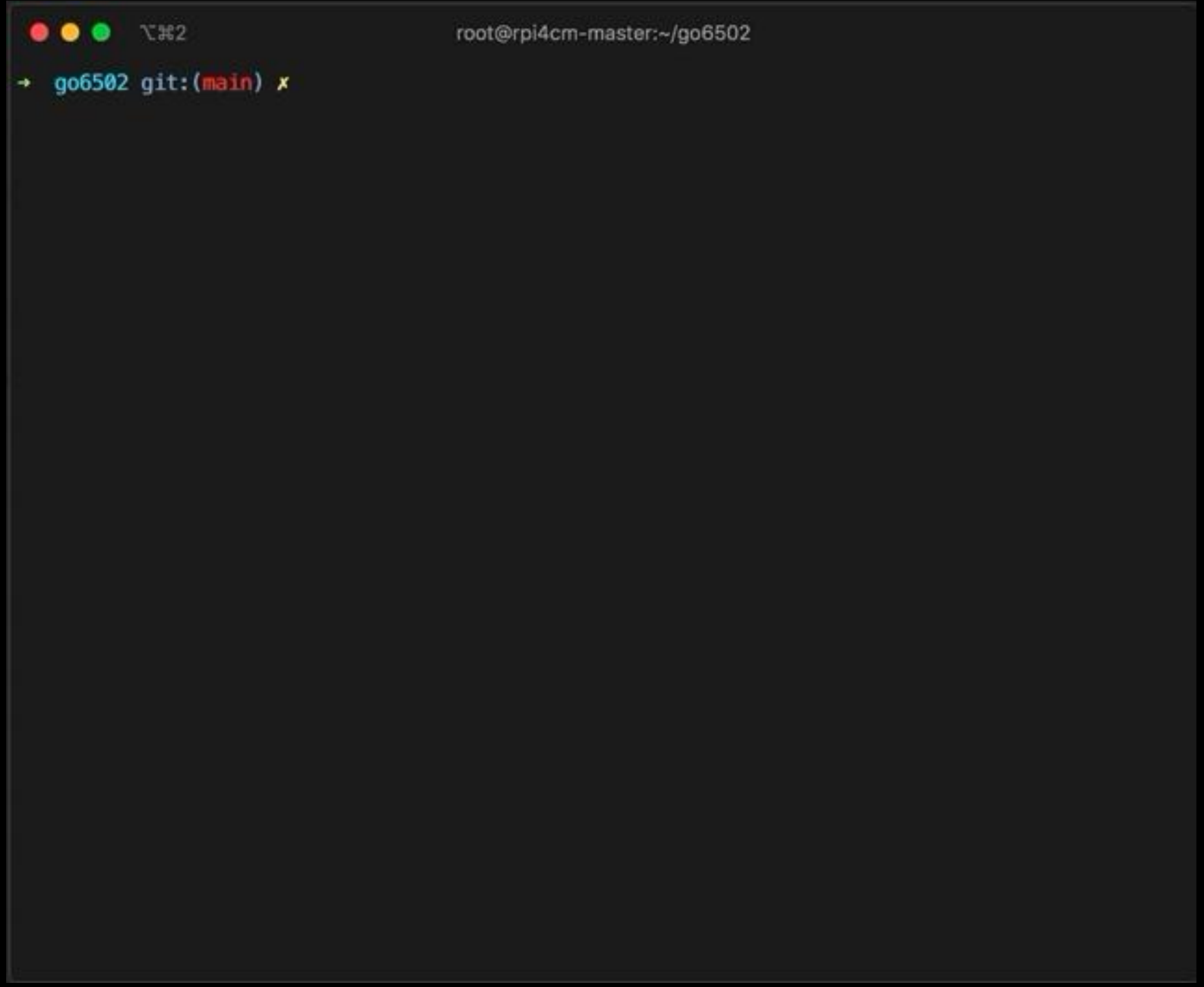



```
KBD = $C000
KBDSTRB = $C010
ASCBS  = $08 | %10000000

        ORG _TEXT
KEYIN   LDA   #$20
        LDY   CH
        STA   (BASL),Y   ;REPLACE FLASHING SCREEN
        LDA   KBD        ;GET KEYCODE
        CMP   #ASCBS     ;BACK SPACE? (CNTRL-H)
        BEQ   >1
        INC   CH
        INC   CH
^1      DEC   CH
        AND   #$7f
        STA   (BASL),Y
        BIT   KBDSTRB    ;CLR KEY STROBE
        RTS
```

Go模拟Apple II




```go
func (a *AppleII) WriteByte(pc uint16, b uint8) {

        if pc >= 0x400 && pc <= 0x7ff {
                // must be printable
                if cell := a.pixelMap[pc]; cell != nil {
                        cell.SetText(string(b))
                }
        }

        if a.log != nil {
                a.log.Printf("WB 0x%04X -> %x", pc, b)
        }
        a.Mem.WriteByte(pc, b)
}
```


⌥⌘2
root@rpi4cm-master:~/go6502
➜ go6502 git:(main) ✗

Go模拟Apple II

- ZHUOS = Zhuo's Hardly Usable Operating System for fun
- 小结
  - 早期电脑都是直接读写内存/硬件
  - 操作系统仅仅是帮助处理IO
    - 进程？不存在的
    - 用户？不存在的
    - 虚拟内存？不存在的
    - 都是用汇编编写程序

Go模拟Apple II

- ZHUOS = Zhuo's Hardly Usable Operating System for fun
- 小结
  - 早期电脑都是直接读写内存/硬件
  - 操作系统仅仅是帮助处
    - 进程？不存在的
    - 用户？不存在的
    - 虚拟内存？不存在的

你说汇编？

可GCC 不支持6502啊

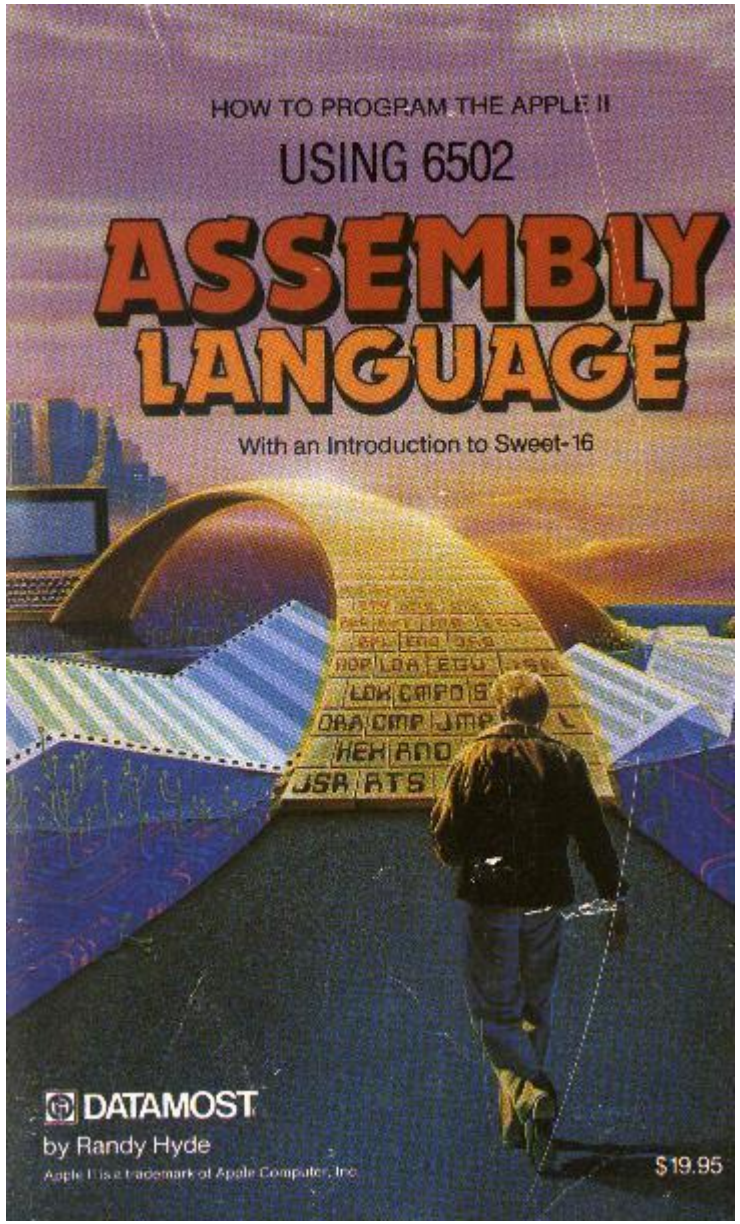
Go 编写汇编器

- 破除迷信
- Rob Pike：The assembler is just doing text processing

https://www.youtube.com/watch?v=KINIAgRpkDA

Go 编写汇编器

- 如何开始？
  - http://www.appleoldies.ca/anix/
- 先读读前人的工作
- 然后用Go实现！



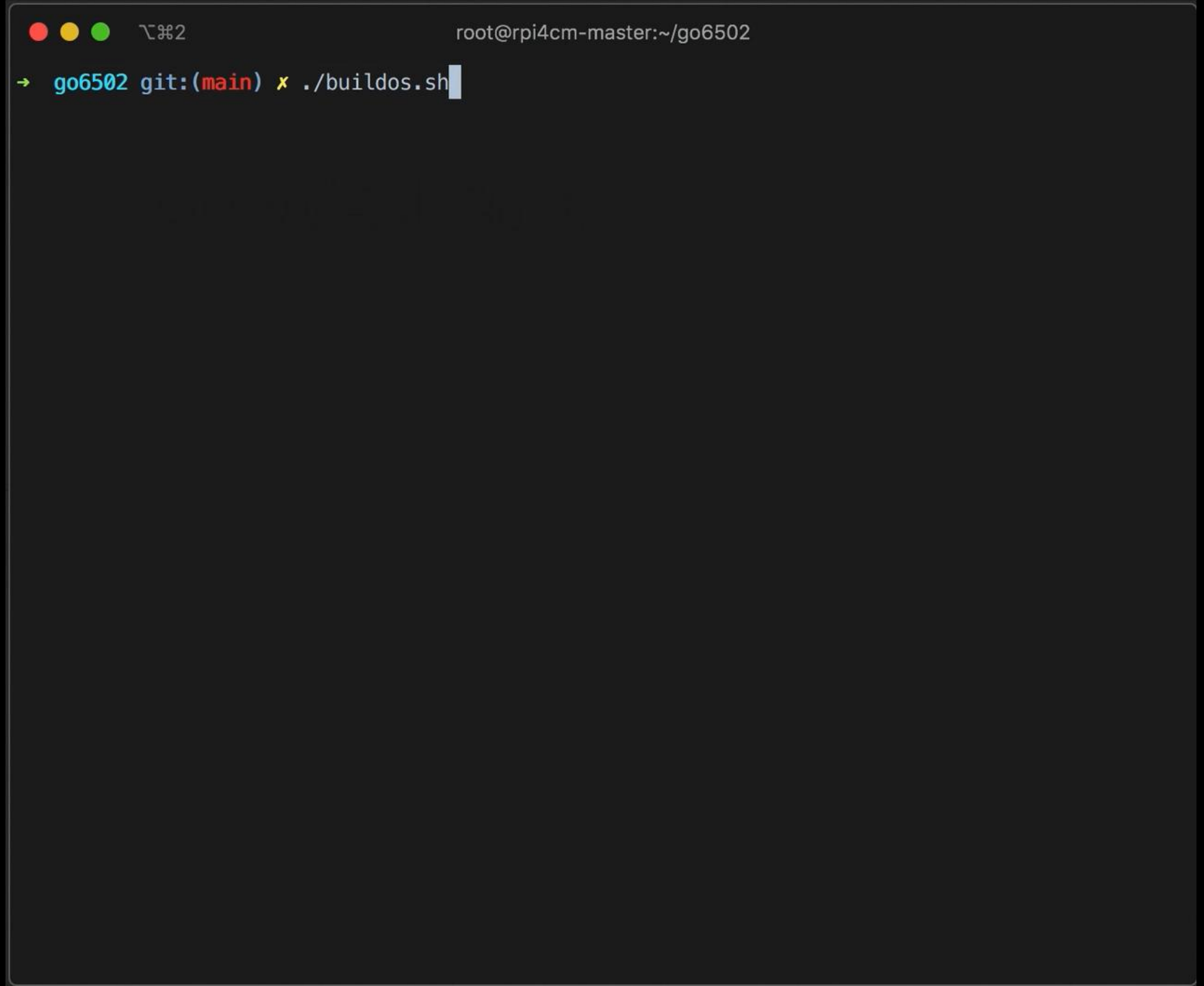
Go 编写汇编器等工具

- 用Go编写的汇编器编写ZHUOS
  - Tokenizer ＋ Parser（可以参加比赛）
  - 解析语句
  - 几乎不用的就直接不实现
  - 大概2000行左右就输出obj（json）
- 实现链接器
  - 最头疼的是指令地址确定
- 想debug方便还得
  - 实现objdump （检查指令和地址）
  - 实现nm（named map，检查链接）

```
KBD = $C000
KBDSTRB = $C010
ASCBS  = $08 | %10000000

        ORG _TEXT
KEYIN   LDA   #$20
        LDY   CH
        STA   (BASL),Y   ;REPLACE FLASHING SCREEN
        LDA   KBD        ;GET KEYCODE
        CMP   #ASCBS     ;BACK SPACE? (CNTRL-H)
        BEQ   >1
        INC   CH
        INC   CH
^1      DEC   CH
        AND   #$7f
        STA   (BASL),Y
        BIT   KBDSTRB    ;CLR KEY STROBE
        RTS
```

Go 编写汇编器等工具

```
#!/bin/bash

set -ue

rm -rf zhuos/src/*.out

for f in zhuos/src/*.s
do
        if [ "$f" != 'zhuos/src/symbols.s' ]; then
            echo $f
            go run cmd/asm/main.go -i $f
        fi
done

go run cmd/link/main.go z.dat zhuos/src/*.out
```


⌥⌘2
root@rpi4cm-master:~/go6502
➜ go6502 git:(main) ✗ ./buildos.sh

未来计划

- 让Go能跑在6502上（10%）
  - 已经能输出部分语句
  - 语法上除了直接操作指针之外跟C没区别
  - Runtime/map/atomic/channel/goroutine没戏
- 说白了，the compiler is just doing text processing

Thanks and Happy Hacking