CS110 Midterm I Review

Software Foundations & Digital Systems

Chaofan Li

lichf2025@shanghaitech.edu.cn

上海科技大学
ShanghaiTech University

Contents

Introduction

What to Cover Today?

Software Side

- Information representation
- Floating point numbers
- C review and memory management
- Micro Pitfall
- Compiler, assembler, linker, loader(CALL)
- Assembly(RISC-V ISA)

Hardware Side

- Combinational logic
- Sequential logic and timing
- FSM basics and typical exam problems

How to Use This Review

- 目标不是重新上完所有内容，而是把知识点串成一张图
- 复习时优先关注: definition, mechanism, typical examples, common mistakes
- 软件部分常考“表示方式”和“程序行为”
- 硬件部分常考“看图分析”“时序计算”“FSM 构造”
- 如果一个概念说不清“为什么”，通常也做不好题

Great Ideas

Great Ideas in Computer Systems

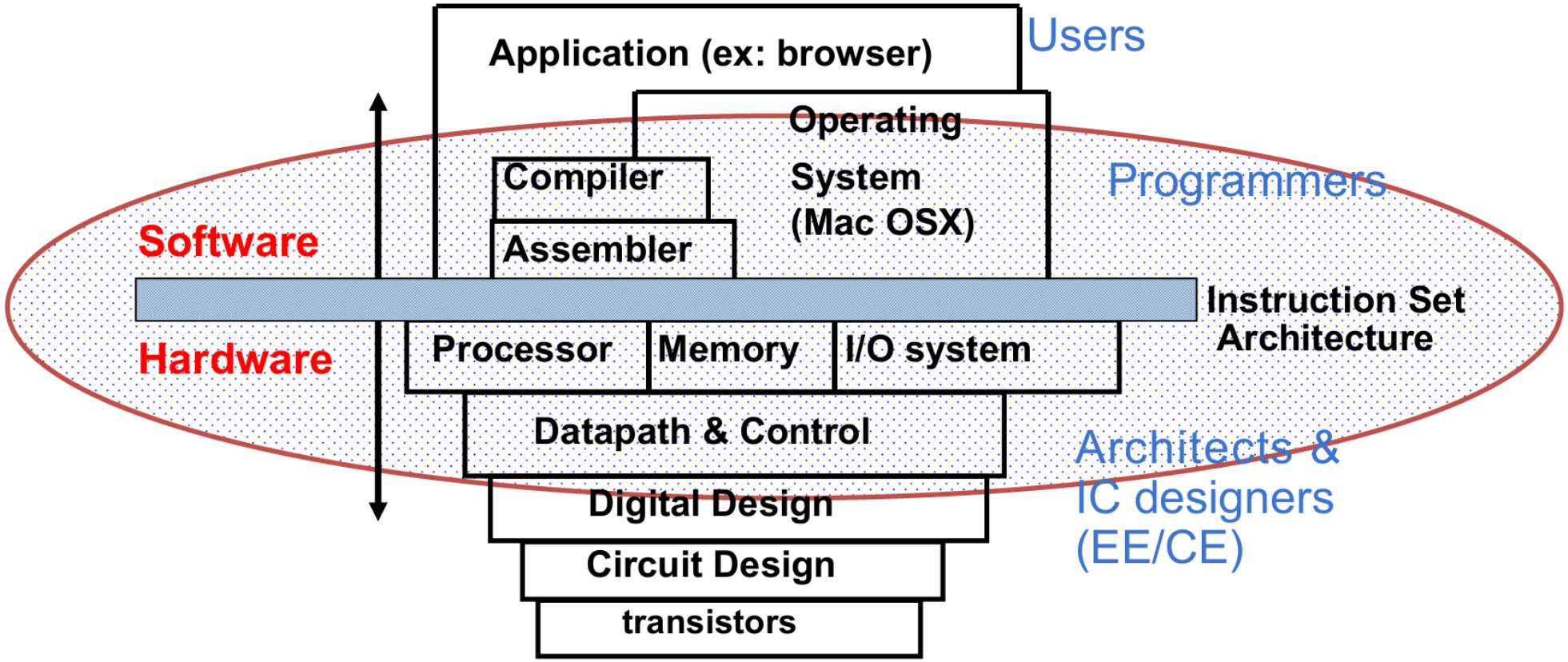
1. **Abstraction**: 用层次隐藏细节
2. **Moore's Law**: 通过技术趋势推动设计
3. **Make the common case fast**: 优化最常出现的路径
4. **Principle of locality**: 利用时间局部性和空间局部性
5. **Parallelism**: 同时处理多个工作单元
6. **Performance measurement & improvement**: 先量化，再优化
7. **Dependability via redundancy**: 用冗余提高可靠性

Abstraction Across Layers

- 同一个程序可以从多个层次理解: source code, assembly, machine code, hardware
- **Compiler**: 把高级语言翻译成 assembly / machine-oriented representation
- **Assembler**: 把 assembly 转成 object code
- **Linker**: 把多个 object files 和 libraries 组合起来
- **Loader**: 把 executable 放入内存并开始运行

把这条链看清楚，很多“程序为什么这样运行”的问题就会更自然。

Abstraction



Amdahl's Law

设程序中有比例 f 的部分无法被加速，其余部分可被加速 S 倍，则总体加速比为

$$\text{Speedup} = \frac{1}{f + \frac{1-f}{S}}$$

- 即使并行部分无限快，系统加速也有上限
- 例: 若 20% 不能并行，则

$$\lim_{S \to \infty} \text{Speedup} = \frac{1}{0.2} = 5$$

- 结论: 瓶颈部分决定最终收益

Software Review

Floating Point Representation

对于 normalized floating point number:

$$V = (-1)^S \times 1.M \times 2^{E-Bias}$$

- `S`: sign bit
- `E`: exponent field
- `M`: mantissa / fraction field
- `Bias`: 指数偏置，例如 single precision 中是 `127`
- normalized number 默认有隐含前导 `1`

Norm, Denorm, and Range

- **Normalized**: exponent 既不是全 0，也不是全 1
- **Denormalized**: exponent 全 0，此时没有隐含前导 `1`
- denorm 的意义: 让 very small numbers 不会突然掉到 `0`
- 正数范围相关问题通常围绕以下几类:
  - minimum positive denorm
  - maximum denorm
  - minimum positive norm
  - maximum finite value

Exam Tip

- 不要只背公式，要会判断 exponent 的特殊情况
- `0`, `denorm`, `norm`, `inf`, `NaN` 要能区分

Endian

7. **Number representation.** Let's consider the hexadecimal number 0xFCC48493. How is the number interpreted, if we treat it as. . .

   (a) an array A of 4 signed, 8-bit integers? Please write each number in decimal, assume the machine is little-endian. If the value is unknown, fill in *GARBAGE* (in all caps).

   i. A[0]: _____

   ii. A[1]: _____

   iii. A[2]: _____

   iv. A[3]: _____

Endian

7. **Number representation.** Let's consider the hexadecimal number 0xFCC48493. How is the number interpreted, if we treat it as. . .

   (a) an array A of 4 signed, 8-bit integers? Please write each number in decimal, assume the machine is little-endian. If the value is unknown, fill in *GARBAGE* (in all caps).

   i. A[0]: 93 1001 0011 -109

   ii. A[1]: 84 1000 0100 -124

   iii. A[2]: C4 1100 0100 -60

   iv. A[3]: FC 1111 1100 -4

How about Big Endian?

C basic

(d) The following code is compiled, and an executable file "main.out" is produced. Execute "./main.out Make CS110 great again!" in the terminal. Write down the content that will be printed.

```c
#include <stdio.h>
int main(int argc, const char *argv[]) {
   printf("argc = %d\n", argc);
   for (int ndx = 0; ndx != argc; ++ndx)
      printf("argv[%d] --> %s\n", ndx, argv[ndx]);
   return 0;
}
```

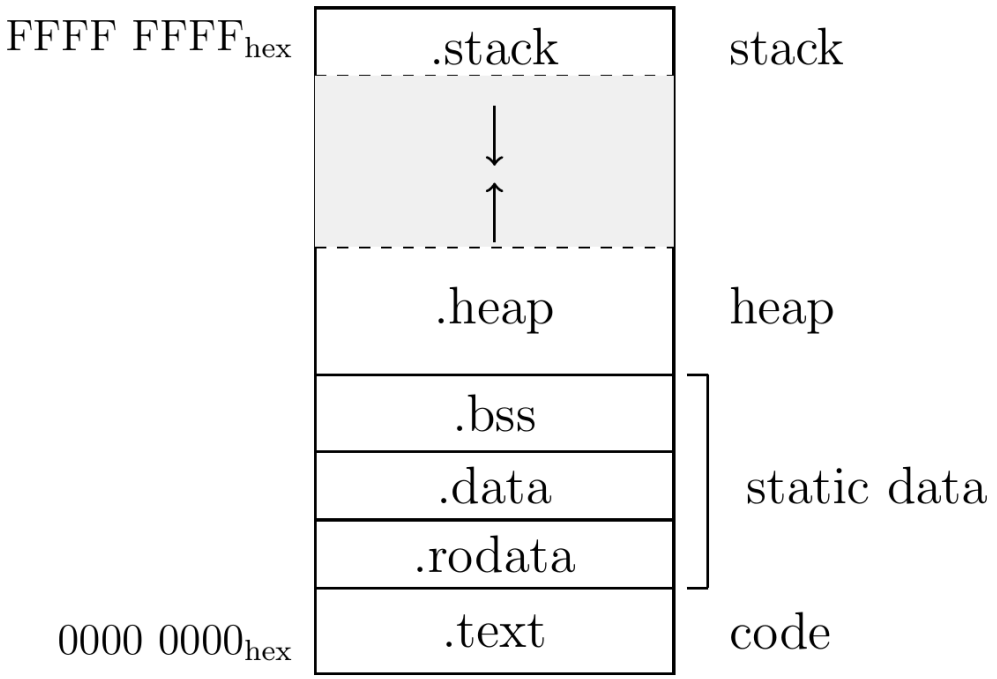
Solution:

```
argc = 5
argv[0] --> ./main.out
argv[1] --> Make
argv[2] --> CS110
argv[3] --> great
argv[4] --> again!
```

1 for `argc = 5`; 1 for `argv[0] --> ./main.out`; 0.5 for the others.

C Memory Management

- 一个进程的内存并不是“混在一起”的
- 这张图先帮我们建立整体布局：**stack**, **heap**, **static data**
- 考试里常考的不是死记位置，而是区分生命周期和管理方式
- 后面几页分别拆开讲每一块区域和对应错误

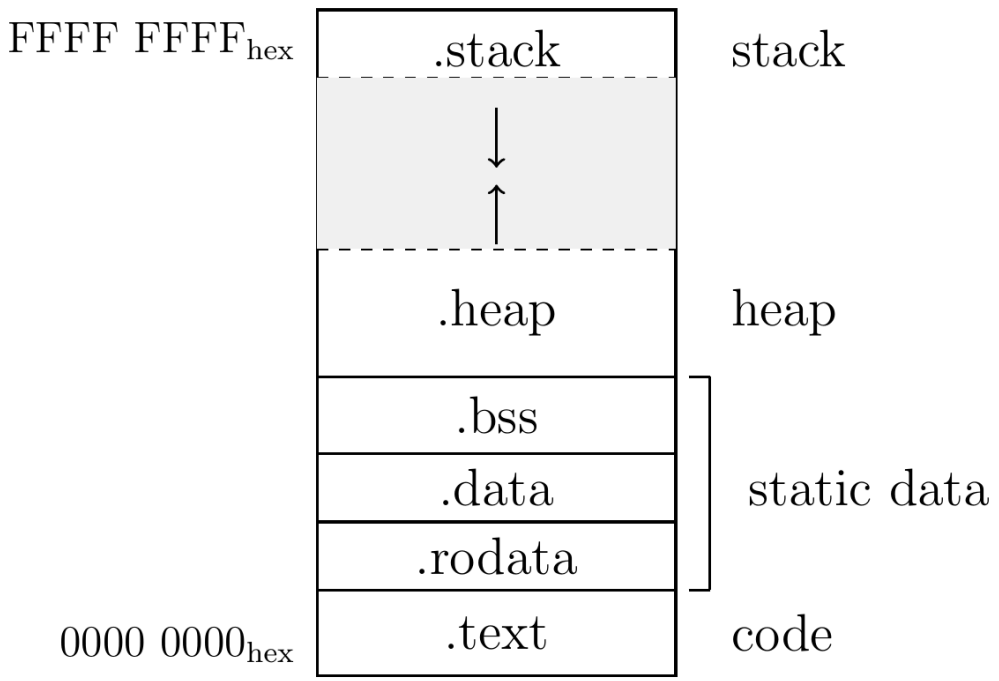


What to Focus On

- 谁负责分配
- 谁负责释放
- 对象能活多久
- 最常见 bug 是什么

C Memory Management: Stack vs Heap

- **Stack**: 函数调用时自动分配和释放，适合局部变量
  - 函数每调用一次，都会建立新的 stack frame
  - 函数返回后，这一层 stack frame 会被回收，所以局部变量不会一直存在
- **Heap**: 动态内存区域，需要显式 `malloc` / `free`
  - heap 上的对象适合跨函数共享
  - 也适合大小在运行时才确定的场景

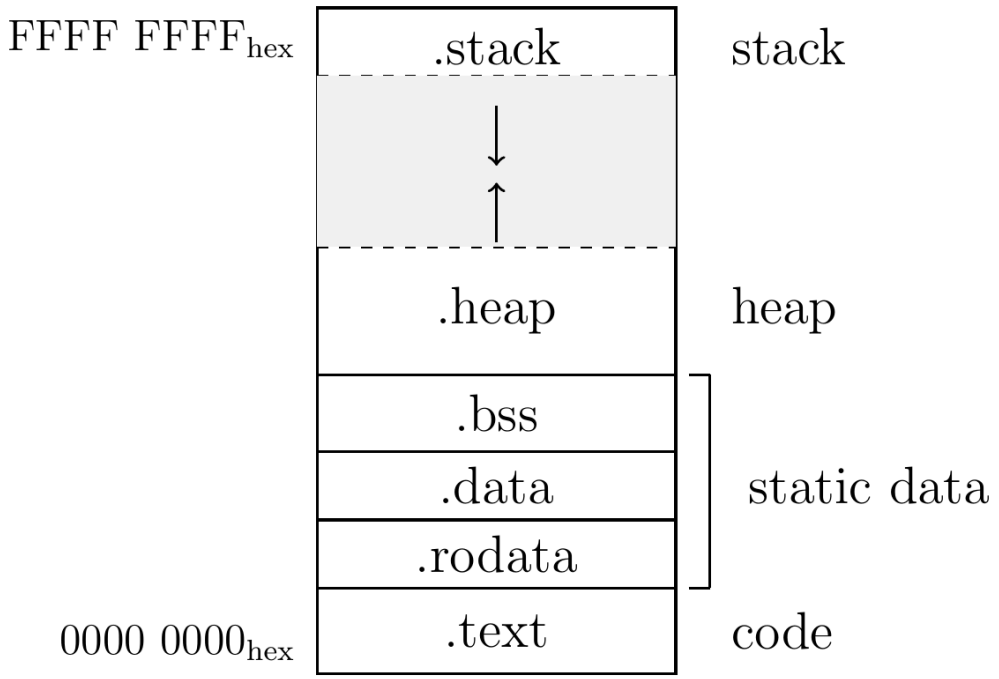


Quick Contrast

- stack: 自动管理，速度快，但容量通常更有限
- heap: 更灵活，但更容易写出内存错误

C Memory Management: Static Data

- **Static Data**: global variables 和 `static` variables 所在区域
  - 它们通常在程序启动时就存在，并持续到程序结束
  - 因此它们和局部变量最大的区别不是“能不能访问”，而是“活多久”
  - 如果一个变量在函数结束后还要保留值，题目常常会用 `static`



Exam Tip

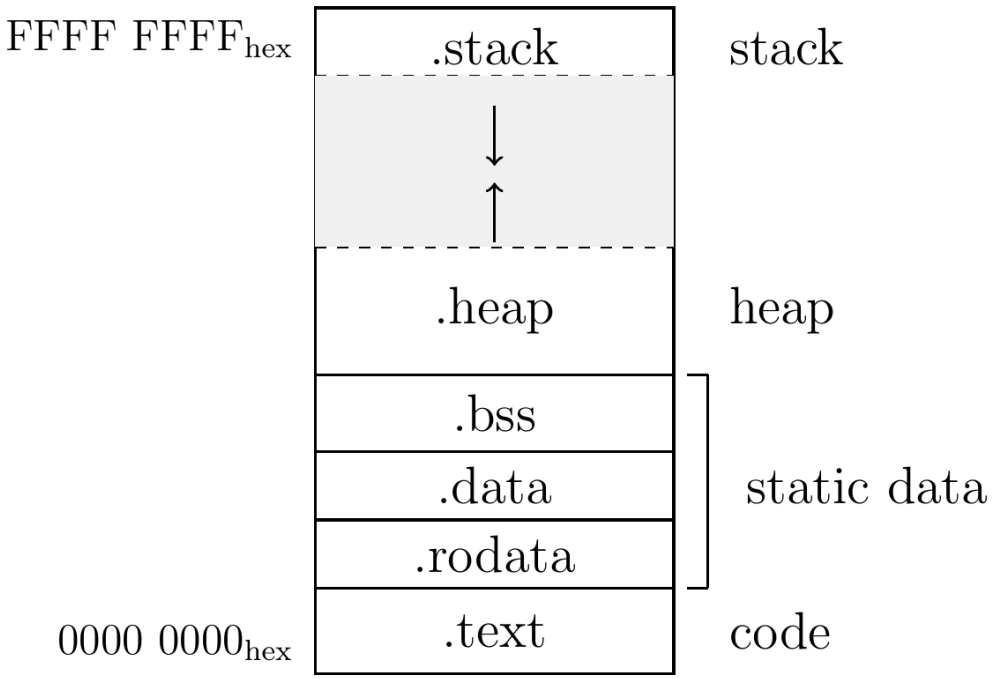
- 看到 global variable，先想到它不在 stack 上
- 看到 `static` local variable，先想到“函数结束后值还在”

C Memory Management: Common Problems

- **Stack overflow**: 递归太深，或者局部数组太大
- **Memory leak**: 分配了 heap memory，但程序路径上忘记 `free`
- **Dangling pointer**: 指针本身还在，但指向对象已经失效
- **Use-after-free**: `free` 之后还继续访问这块内存

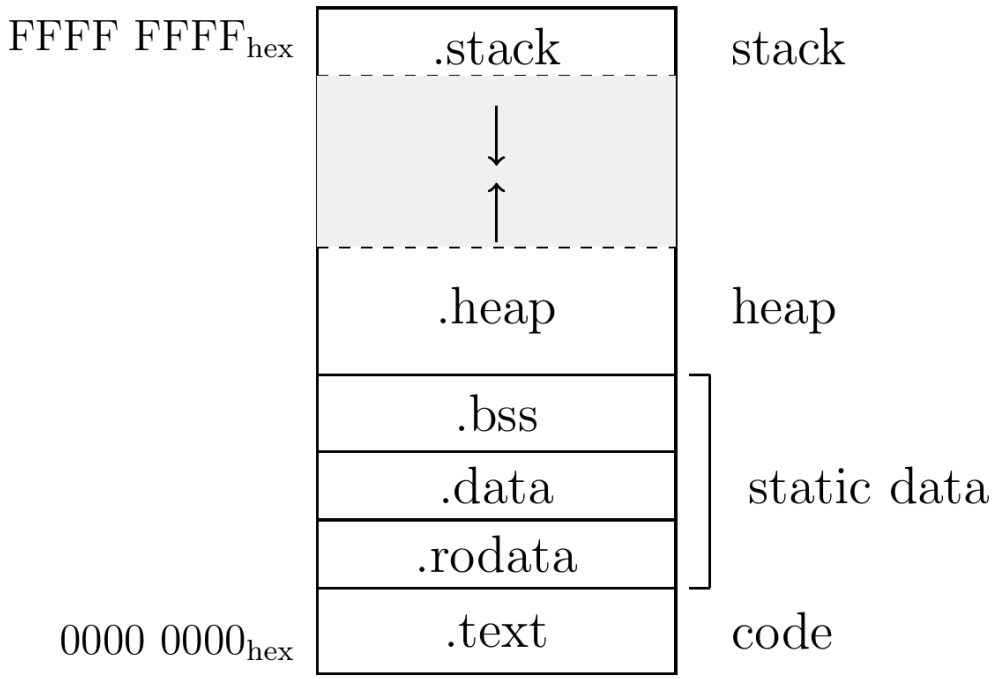
How to Judge

- 先判断对象在哪个区域
- 再判断对象什么时候失效
- 最后判断当前访问是否合法



C Memory Management: Review Checklist

- 局部变量默认先想到 **stack**
- 动态申请的对象默认先想到 **heap**
- global 和 `static` 变量默认先想到 **static data**
- 凡是指针题，都要多问一句：它现在指向的对象还活着吗



Common Exam Traps

- 返回局部变量地址
- 重复 `free`
- 忘记 `free`
- 把“指针还在”误认为“对象还在”

Macro Pitfall: Missing Parentheses

```
#define SUM(a, b) a + b
int x = 4 * SUM(1, 2);
```

展开后:

```
int x = 4 * 1 + 2;
```

- 宏只是文本替换，不懂“优先级”
- 因此参数和整体表达式都应加括号

```
#define SUM(a, b) ((a) + (b))
```

Macro Pitfall: Multi-line Statements

```
#define PRINT(a, b) \
    printf("Value of " #a " is %d\n", a); \
    printf("Value of " #b " is %d\n", b);
```

- 这两种写法都“能展开”，但都不够安全
- `SUM` 的问题是表达式优先级
- `PRINT` 的问题是多条语句没有被包装成一个整体
- 宏本质上只是文本替换，不会自动理解上下文

Macro Pitfall: Why It Breaks Under `if`

```
#define PRINT(a, b) \
    printf("Value of " #a " is %d\n", a); \
    printf("Value of " #b " is %d\n", b);

if (condition)
    PRINT(x, y);
```

展开后:

```
if (condition)
    printf("Value of " "x" " is %d\n", x);
printf("Value of " "y" " is %d\n", y);
```

Macro Pitfall: Multi-line Statements

```
#define PRINT(a, b) do { \
    printf("Value of " #a " is %d\n", a); \
    printf("Value of " #b " is %d\n", b); \
} while (0)
```

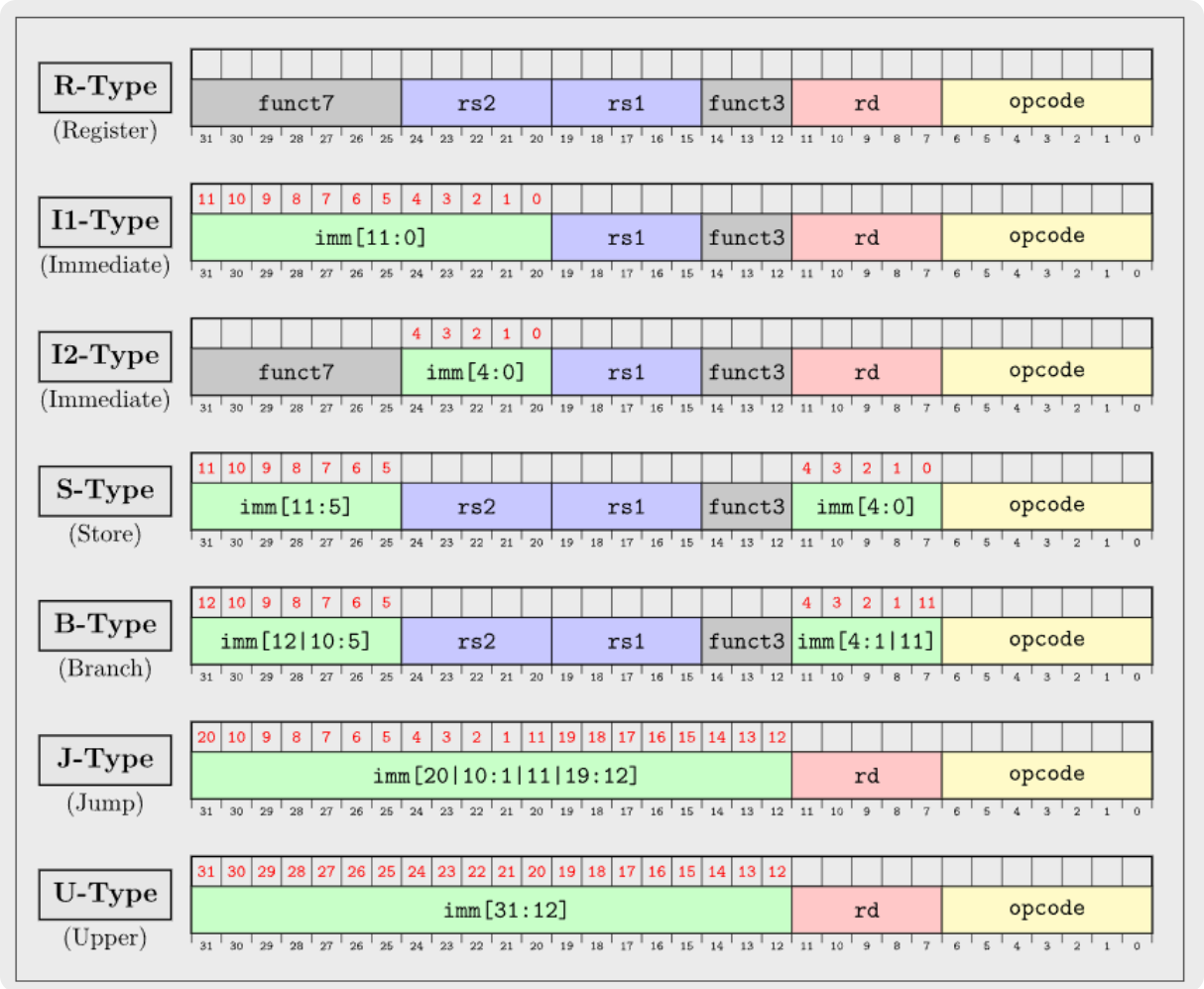
- 多行宏如果不包成单条语句，在 `if` 语句中容易出错
- `do { ... } while (0)` 的作用是让宏像普通语句一样使用
- `#a` 表示 stringification，把 token 转成字符串

Compiler, Assembler, Linker, Loader

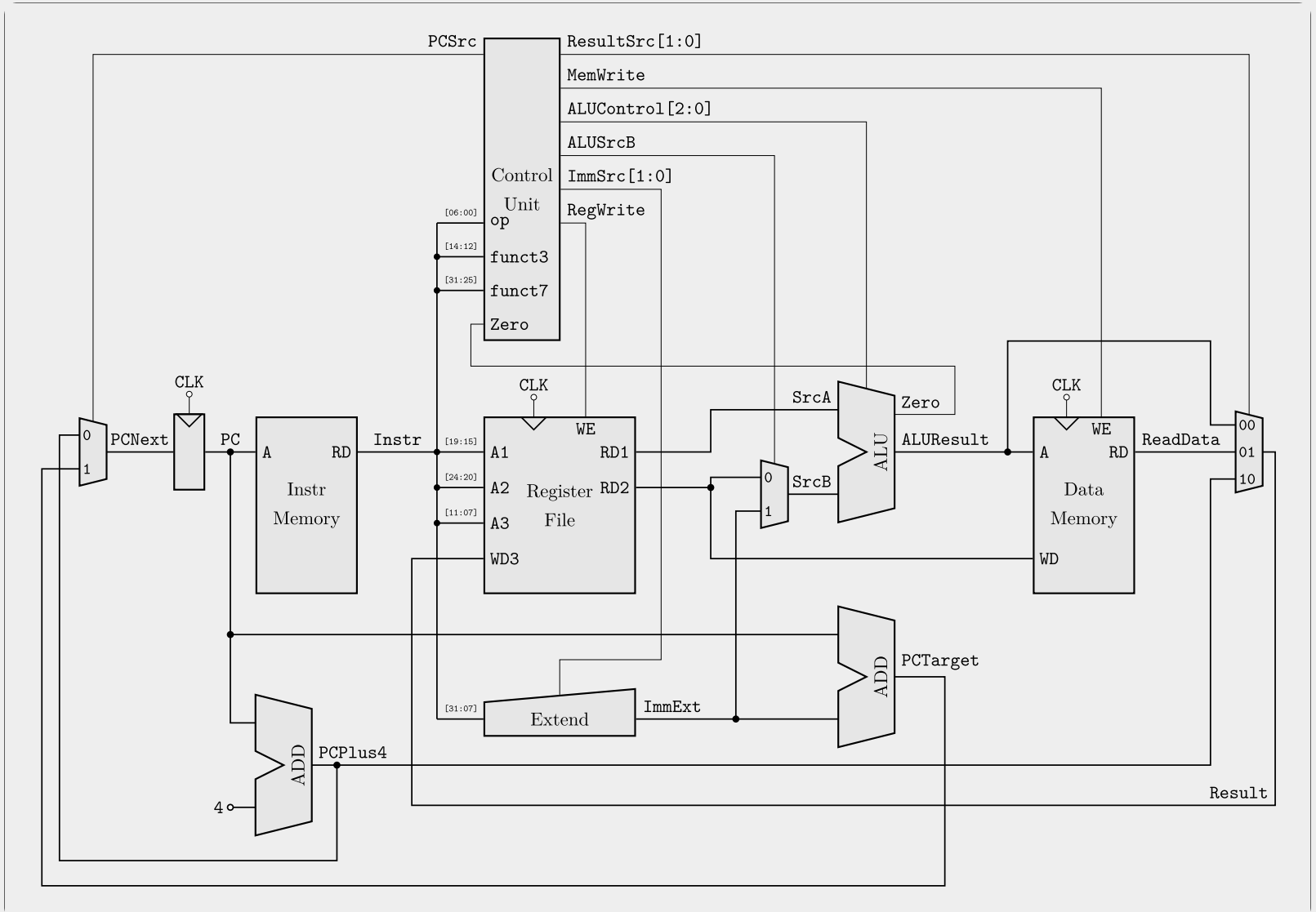
```
C source
  -> compiler
assembly
  -> assembler
object file
  -> linker
executable
  -> loader
process in memory
```

- **Compiler** 关注翻译和优化
- **Assembler** 关注 instruction encoding
- **Linker** 解决 symbol references
- **Loader** 建立进程地址空间并启动程序

Assembly


R-Type
(Register)
funct7 rs2 rs1 funct3 rd opcode
I1-Type
(Immediate)
imm[11:0] rs1 funct3 rd opcode
I2-Type
(Immediate)
funct7 imm[4:0] rs1 funct3 rd opcode
S-Type
(Store)
imm[11:5] rs2 rs1 funct3 imm[4:0] opcode
B-Type
(Branch)
imm[12|10:5] rs2 rs1 funct3 imm[4:1|11] opcode
J-Type
(Jump)
imm[20|10:1|11|19:12] rd opcode
U-Type
(Upper)
imm[31:12] rd opcode

Datapath


PCSrc
ResultSrc[1:0]
MemWrite
ALUControl[2:0]
ALUSrcB
ImmSrc[1:0]
RegWrite
Control Unit
op
funct3
funct7
Zero
[06:00]
[14:12]
[31:25]
CLK
PCNext
PC
Instr
Instr Memory
A
RD
[19:15]
[24:20]
[11:07]
A1
A2
A3
WD3
WE
RD1
RD2
Register File
SrcA
SrcB
ALU
Zero
ALUResult
Data Memory
WD
ReadData
00
01
10
ADD
PCPlus4
4
PCTarget
Extend
[31:07]
ImmExt
Result

Combinational Logic

What is Combinational Logic?

- 输出只取决于“当前输入”
- 没有内部状态，不记忆过去
- 常见例子: adder, mux, decoder, ALU
- 分析方法通常有三种:
  - circuit -> boolean expression
  - boolean expression -> circuit
  - truth table -> simplified logic

Basic Logic Skills

你至少需要会下面这些基本操作:

- 识别常见逻辑门: `AND`, `OR`, `NOT`, `NAND`, `NOR`, `XOR`
- 从电路直接写出 boolean expression
- 从 boolean expression 画出电路结构
- 用 truth table 判断逻辑功能
- 理解“组合逻辑没有状态”

Truth Table

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

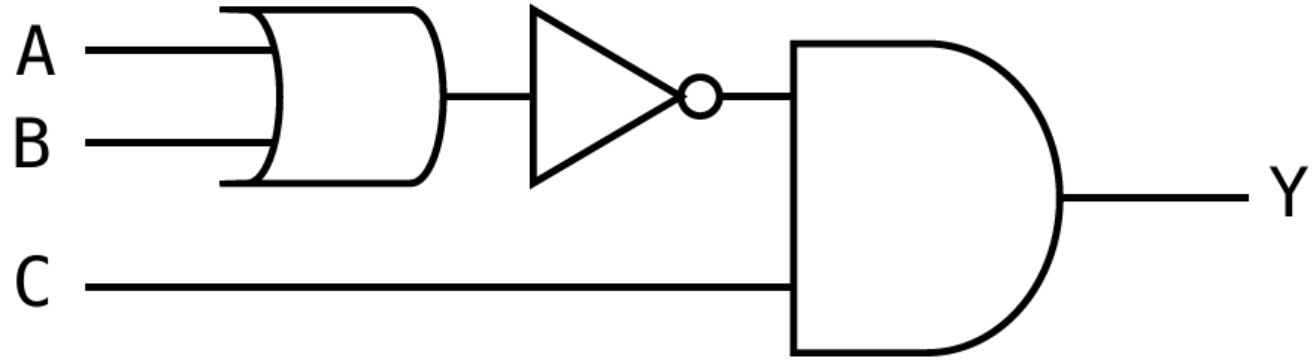
(c) Write down the logic expression that implements the truth table using sum of minterm.

Solution: $Y = \bar{A}\bar{B}C$. This is the only form of the logic expression using minterm. 0 mark for all the other cases.

(d) Build a logic circuit that uses only 2-input **AND**, 2-input **OR** and **NOT** gates implementing the same logic above. Use as less logic gates as possible.

(d) Build a logic circuit that uses only 2-input **AND**, 2-input **OR** and **NOT** gates implementing the same logic above. Use as less logic gates as possible.

Solution: Optimal solution for this is shown below for full mark using the required logic gates. Direct implementation of $Y = \bar{A}\bar{B}C$ get 1 mark for no simplification. You also lose point(s) if not using the required gates. The other cases for 0.



Sequential logic

Sequential logic

What is Sequential Logic?

- 输出不仅取决于当前输入，也取决于过去状态
- 状态通常由 registers / flip-flops 保存
- 状态更新通常由 clock 控制
- 这类系统能“记住”之前发生了什么

Keywords

- state
- register
- clock
- next state
- output

FSM

FSM Basics

- FSM = Finite State Machine
- 它用有限个状态描述“系统随时间如何演化”
- 每次接收输入后，根据当前状态和输入决定:
  - next state
  - output

Moore vs. Mealy

- **Moore 状态机**: 输出只由当前状态决定
- **Mealy 状态机**: 输出由当前状态和当前输入共同决定

直观来说，Moore 的输出是“状态决定一切”，所以输出通常只会在状态更新之后变化；Mealy 的输出会直接受到输入影响，因此输入一变，输出也可能立刻变化。

两者常见区别如下：

- **Moore** 更稳定，输出不容易因为输入瞬时变化而抖动
- **Mealy** 响应更快，因为不一定要等到进入下一个状态才改变输出
- **Moore** 往往需要更多状态
- **Mealy** 往往可以用更少状态实现同样功能

考试里如果让你判断是哪一种，一个很直接的方法是看输出标在什么地方：

- 如果输出写在 **state** 上，通常是 **Moore**
- 如果输出写在 **transition edge** 上，通常是 **Mealy**

Why FSM?

- 很多“序列检测”“协议控制”“控制器设计”问题都适合建模成 FSM

Example 1: Detect `101`

目标: 输入 bit stream，每当最近三位构成 `101` 时输出 `1`

State Meaning

- `S0`: 还没有匹配到有效前缀
- `S1`: 刚看到 `1`
- `S2`: 刚看到 `10`

Transition Idea

- `S0 --1--> S1`
- `S1 --0--> S2`
- `S2 --1-->` 输出 `1`，并回到表示后缀 `1` 的状态

关键不是死背图，而是先定义“每个状态表示已匹配到什么前缀”。

FSM Example Figure

Solution:

state bit1	state bit0	Input	next state bit1	next state bit0	Output
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	0	1	1

先读状态含义，再看边上的输入输出标记。

FSM Example Figure

Solution:

state bit1	state bit0	Input	next state bit1	next state bit0	Output
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	0	1	1

Example 2: Two or More Successive `0`

题意: 当输入中出现连续至少两个 `0` 时输出 `1`

A Natural Design

- `S0`: 最近没有连续 `0`
- `S1`: 最近看到了一个 `0`
- `S2`: 最近已经看到了至少两个连续 `0`

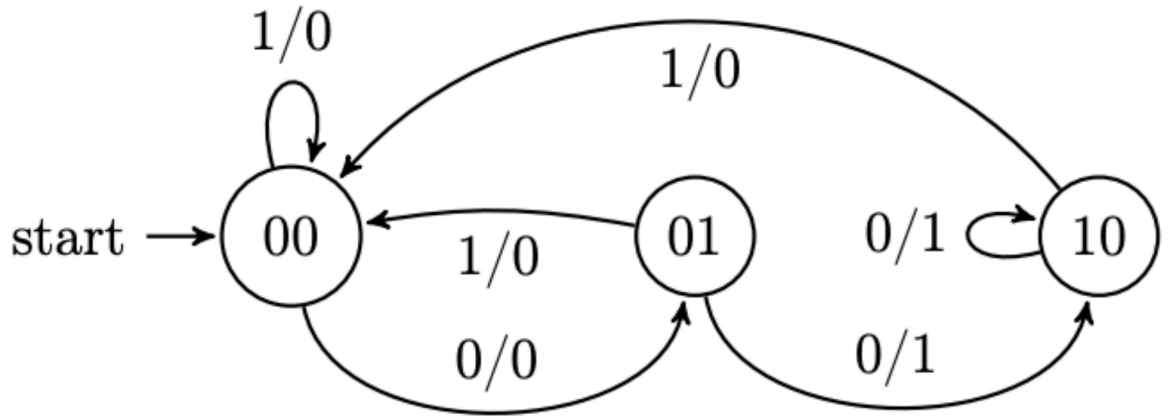
Transition Logic

- `S0 --0--> S1`
- `S1 --0--> S2`, 输出 `1`
- `S2 --0--> S2`, 继续输出 `1`
- 任意状态读到 `1`，回到 `S0`

FSM Solution Figure

Draw a FSM that outputs 1 when it receives two or more successive '0'.

Solution:



连续 `0` 的检测题，关键是状态要准确表示"已经连续看到了几个 0"。

Summary

Summary

- 软件部分先抓主线: abstraction, floating point, memory layout, macros, toolchain
- `RISC-V ISA` 和 `datapath` 是重点: 要能把 instruction format、control signal 和 datapath flow 对上
- 组合逻辑重点: gate, boolean expression, truth table, ALU
- 时序逻辑重点: clock, register
- `FSM` 是 **时序逻辑的一部分**: 要会读 state meaning, 画 transition, 做 sequence detection

Midterm 前建议至少再过一遍:

- 一道 floating point 表示题
- 一道 macro / memory 行为题
- 一道 `RISC-V` / datapath 分析题
- 一道 FSM 构造题

复习时不要只背结论, 要把"题目怎么分析、图怎么读、信号怎么走"练熟。