

# Abstract Machine - 裸机运行时环境

余子濠 陈璐

# 提纲

- ▶ 程序的运行 - 从入门到放弃
- ▶ AM - 裸机运行时环境
- ▶ 基于AM的教学生态
- ▶ AM代码导读

# 程序和交叉编译

- ▶ 大家一定很好奇: 怎么编一个程序在自己设计的CPU上运行?

- ▶ 直接用gcc编译, 生成的是x86的二进制

- gcc hello.c -o hello
- file hello
- objdump -d hello

```
// hello.c
#include <stdio.h>
int main() {
    printf("Hello, RISC-V!\n");
    printf("%d\n", 1234);
    return 0;
}
```

- ▶ 为了生成riscv64的二进制, 我们需要交叉编译

- apt-get install g++-riscv64-linux-gnu (一键安装, 不用自己编译riscv工具链了)
- riscv64-linux-gnu-gcc hello.c -o hello-riscv64
- file hello-riscv64
- riscv64-linux-gnu-objdump -d hello-riscv64

# 可以运行了吗?

- ▶ hello-riscv64能在我们的CPU上运行吗?

- printf/puts这些函数在哪里定义的?
- printf@plt是什么?

- ▶ 动态链接: 在程序运行的时候进行链接

- 符号解析 + 重定位

- ▶ 支持动态链接需要

- 动态库(如libc.so.6)
- 以及动态链接器(libdl.so.2, 也是一个动态库)
- 还需要支持动态链接的加载器(ld-linux-riscv64-lp64d.so.1)
  - ▶ 可以执行的动态库, 很神奇!
- 这些文件都在/usr/riscv64-linux-gnu/lib/目录下

- ▶ 只把hello-riscv64放到我们的CPU上, 肯定不能成功运行

```
// hello.c
#include <stdio.h>
int main() {
    printf("Hello, RISC-V!\n");
    printf("%d\n", 1234);
    return 0;
}
```

# 要不打个包试试?

- ▶ 能不能把动态链接需要的东西和程序打包编译成一个文件?
- ▶ 静态链接可以帮助我们
  - riscv64-linux-gnu-gcc **-static** hello.c -o hello-riscv64-static
  - file hello-riscv64-static
  - riscv64-linux-gnu-objdump -d hello-riscv64-static
- ▶ 能在我们的CPU上跑了吗?
  - 有一些特殊的ecall指令(系统调用)
    - ▶ 系统调用 = 请求操作系统提供服务
  - 但我们的CPU上并没有运行操作系统啊
- ▶ 把hello-riscv64-static放到我们的CPU上, 还是不能成功运行

# 程序的运行 - 从入门到放弃

- ▶ 我们发现连一个最简单的hello程序都无法在我们的CPU上运行
- ▶ 多次失败的尝试告诉我们: 不是所有程序都能随随便便运行的
- ▶ 程序的运行需要运行时环境的支持
  - 动态库, 动态链接器, 支持动态链接的加载器, 操作系统的服务, 系统文件...
  - 太复杂了, 放弃吧
- ▶ 明明白白地放弃 - 来看看运行hello程序需要什么样的支持
  - strace - 追踪程序执行过程中的系统调用(和信号)
  - strace ./hello-x86
- ▶ 不想放弃? 那就要重新审视程序和计算机之间的关系了

# 提纲

- ▶ 程序的运行 - 从入门到放弃
- ▶ **AM - 裸机运行时环境**
- ▶ 基于AM的教学生态
- ▶ AM代码导读

# 程序和计算机

- ▶ 目前我们的CPU比较弱, 只能执行(部分)RV64I的指令
- ▶ 要运行一个功能强大的操作系统, 感觉并不现实
  
- ▶ 计算机越强大, 就能跑越复杂的程序
  - 计算指令 - 纯计算任务
  - 输入输出 - 交互式任务
  - 中断异常 - 批处理系统
  - 虚存管理 - 分时多任务
  
- ▶ 计算机也是按照这个顺序发展的!
  - 程序也在协同演进



# AM(Abstract Machine) - 计算机抽象模型

## ▶ 根据计算机发展史定义出计算机的功能描述

- 图灵机(1936)
- 冯诺依曼机(1945)
- 操作系统GM-NAA I/O(1956)
- CTSS(1961)

## ▶ 抽象出相应计算机上程序的运行需求 运行在

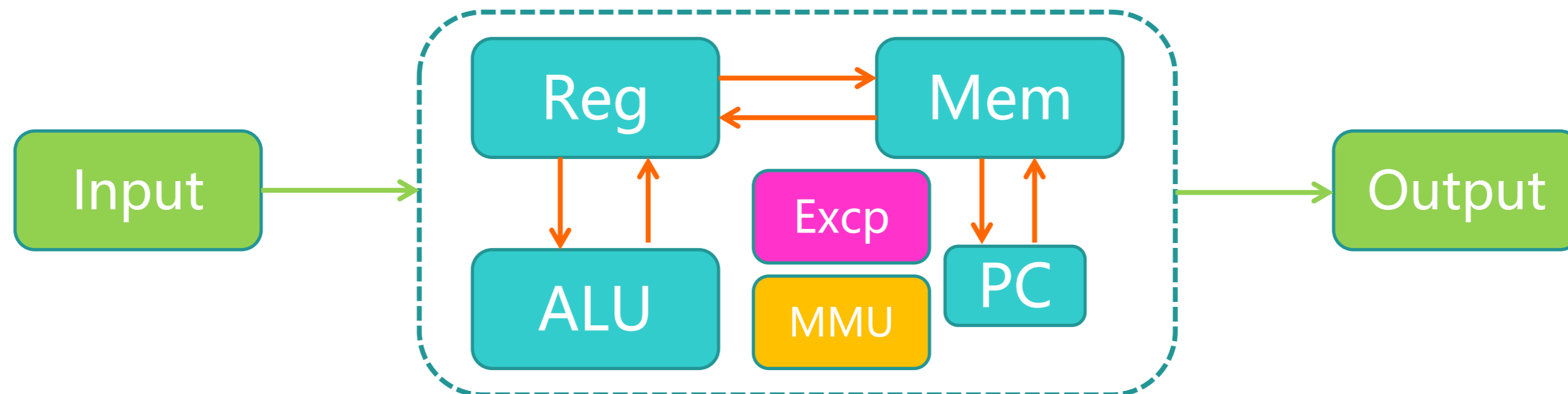
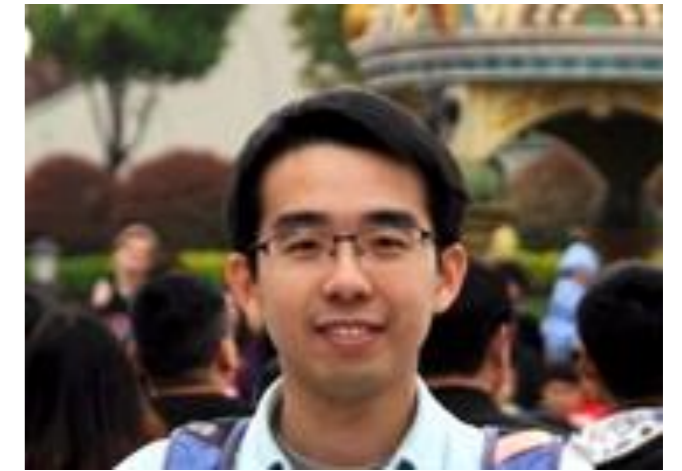
- 纯计算任务(算法) - 计算
- 交互式任务 - 输入输出
- 批处理系统 - 上下文管理
- 分时多任务 - 虚存管理

抽象计算机API

支撑

# AM(Abstract Machine) - 计算机抽象模型

- ▶ AM提出者 - 蒋炎岩老师, 南京大学
- ▶  $AM = TRM + IOE + CTE + VME + MPE$ 
  - TRM(TuRing Machine) - 图灵机
  - IOE(I/O Extension) - 输入输出扩展
  - CTE(ConText Extension) - 上下文扩展 (一生一芯只要求做到这里)
  - VME(Virtual Memory Extension) - 保护扩展
  - MPE(Multi-Processor Extension) - 多处理器扩展



# AM对认识计算机的意义

- ▶ AM = 按照计算机发展史将计算机功能抽象地模块化的裸机运行时环境
  - 计算机发展史 – 从需求的角度更容易理解为什么需要这些模块
    - ▶ 程序要(高效地)计算 -> (支持指令集的)图灵机 [TRM]
    - ▶ 程序要输入输出 -> 冯诺依曼机 [IOE]
    - ▶ 想依次运行多个程序 -> 批处理系统 [CTE]
    - ▶ 想同时运行多个程序 -> 分时多任务 [VME]
  - 裸机运行时环境 – 软硬协同地揭示程序与计算机的关系
    - ▶ (CPU)实现硬件功能 -> (AM)提供运行时环境 -> (APP层)运行程序
    - ▶ (在CPU中)实现更强大的硬件功能 -> (在AM中)提供更丰富的运行时环境 -> (在APP层)运行更复杂的程序
  - 抽象 – 支持多种架构
    - ▶ x86-nemu, riscv32-sodor, mips32-qemu, riscv64-mycpu, Linux native
    - ▶ 在AM上开发的应用(包括OS)可以无缝迁移到各种架构
    - ▶ 打通系统方向各课程实验的关键

# 回到TRM - 最简单的计算机

- ▶ 如果程序只是想计算, 计算机需要提供什么呢?
  - 可以计算的部件 - 运算器
  - 可以驱动计算机进行计算的部件 - 控制器
  - 可以放置程序的部件 - 存储器
  - 指示当前程序执行进度的计数器 - PC(Program Counter)
  - 自动执行程序机制
- ▶ 单周期处理器就可以满足程序的需求!
- ▶ 在程序看来, 它需要什么样的运行时环境?
  - 可以用来自由计算的内存区间 - 堆区
  - 程序“入口” - main()函数
  - “退出”程序的方式 - 停机

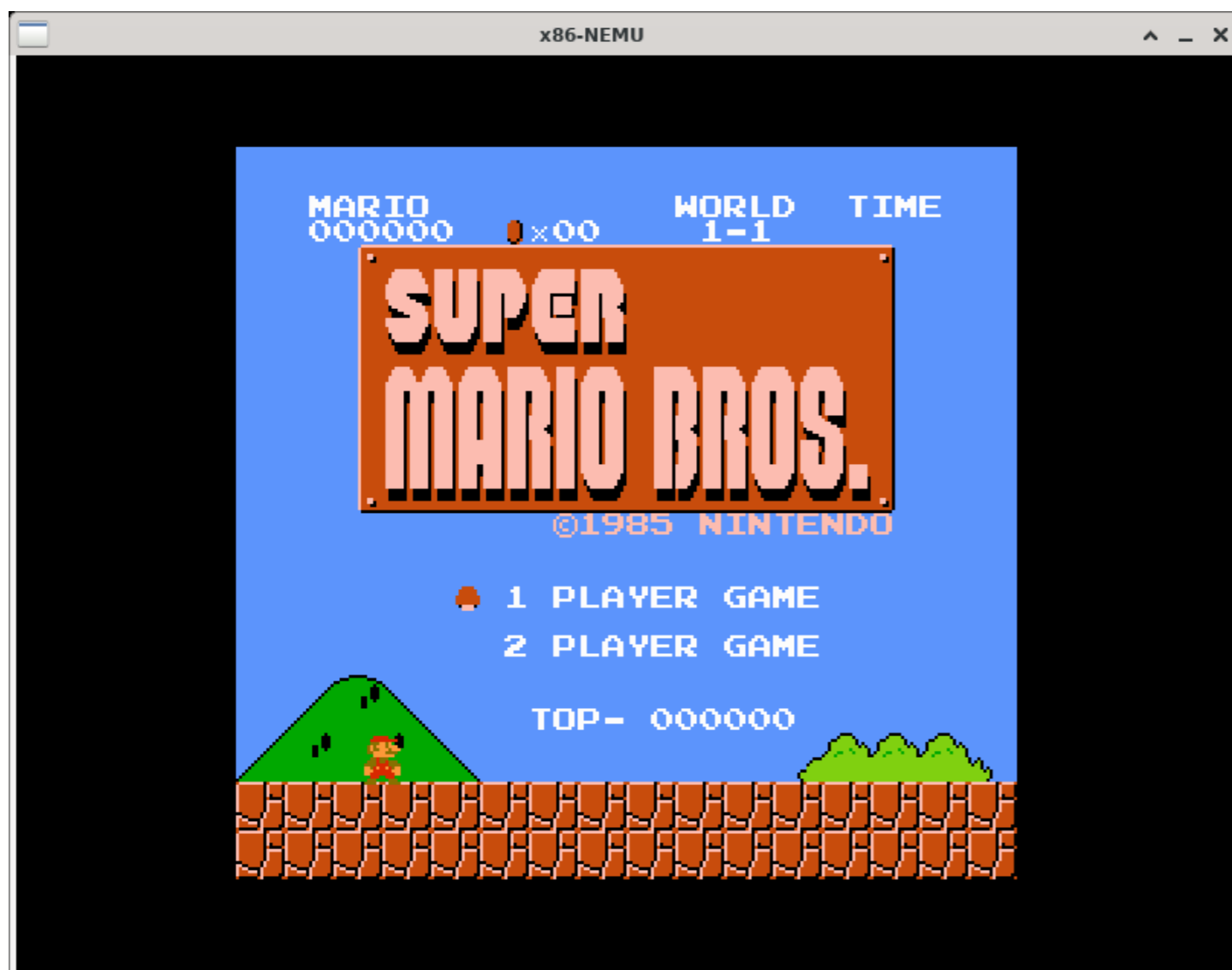
```
while (1) {  
    取出PC指向的指令;  
    指令译码;  
    指令执行;  
    更新PC;  
}
```

# TRM的运行时环境

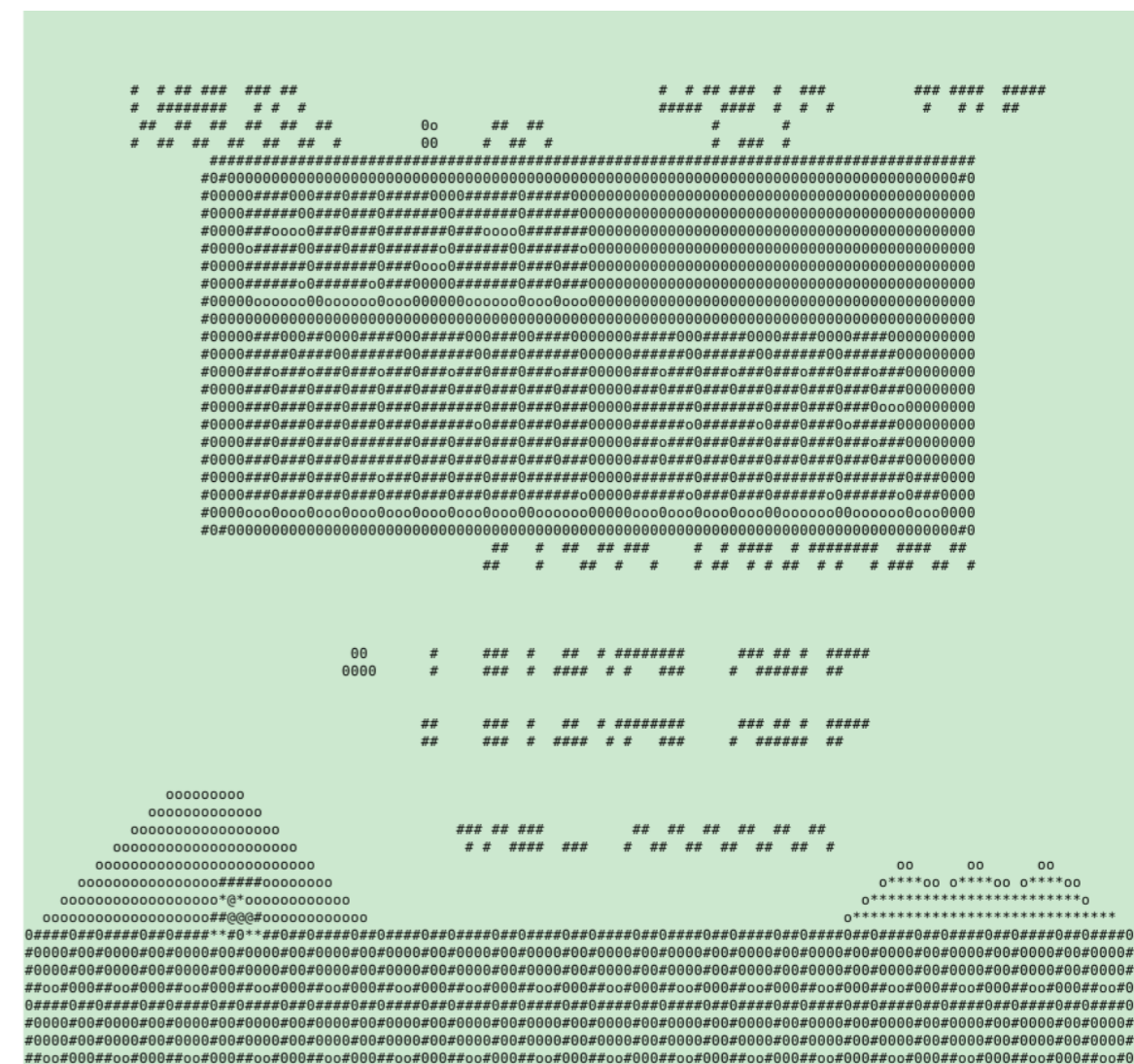
- ▶ 堆区 - `Area heap; // [heap.start, heap.end)`
- ▶ `main()`函数 - 执行`main()`函数前的初始化代码(如设置栈顶指针)
- ▶ 停机 - `halt()`
  
- ▶ 其实还有一个: 打印字符 - `putch()`
  - 对现代计算机来说属于I/O
  - 但也可以理解为图灵机往纸带的特殊部分进行写入
  
- ▶ 可以驱动计算机进行计算的命令 - 指令(集)
  - 指令由编译器生成, 运行时环境不必关心
- ▶ 程序的构建方式 - 编译命令, 链接脚本
  - 指定程序真正的入口地址 `_start`, 程序各个节(section)的链接顺序

# IOE - 输入输出

- ▶ 程序想输入输出, 运行时环境需要提供什么?
  - 输入函数ioe\_read()和输出函数ioe\_write()
  - 还有一些约定的抽象设备: 时钟, 键盘, 2D GPU[, 串口, 声卡, 磁盘, 网卡]



在AM上运行的超级玛丽



一生一芯低配版超级玛丽(无键盘和GPU) 14

# CTE - 上下文管理

## ▶ 想进行中断/异常处理, 运行时环境需要提供什么?

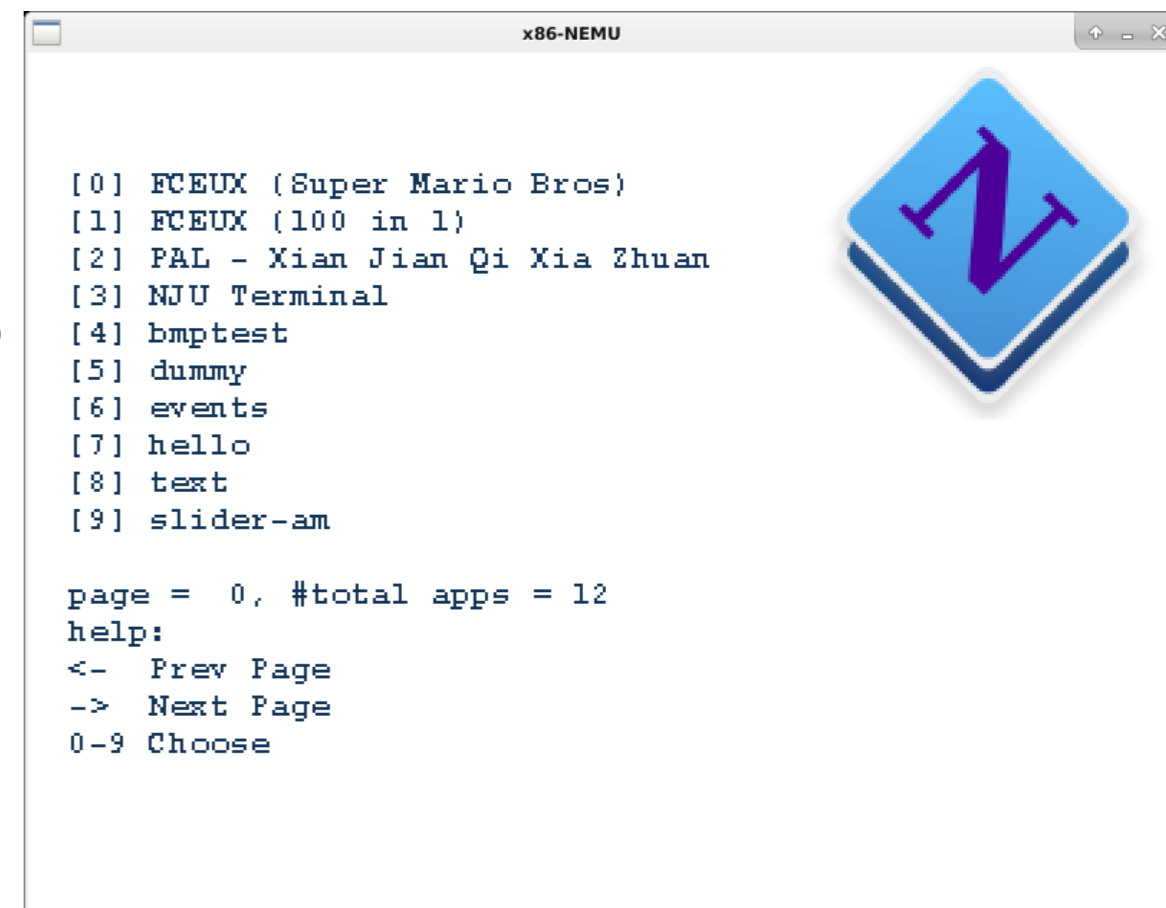
- 上下文保存/恢复
- 事件处理回调函数
- kcontext() 创建内核上下文
- yield() 自陷操作
- ienabled()/iset() 中断查询/设置

## ▶ 有了这些API, 我们就可以运行批处理系统了

- 仙剑都可以跑

## ▶ 甚至是分时多线程操作系统(例如RT-Thread)

- RT-Thread并非基于AM, 但它对计算机有类似需求



```
x86-NEMU

[0] FCEUX (Super Mario Bros)
[1] FCEUX (100 in 1)
[2] PAL - Xian Jian Qi Xia Zhuan
[3] NJU Terminal
[4] bmptest
[5] dummy
[6] events
[7] hello
[8] text
[9] slider-am

page = 0, #total apps = 12
help:
<- Prev Page
-> Next Page
0-9 Choose
```



# VME和MPE - 虚存管理和多处理器

## ▶ 迈向现代计算机系统

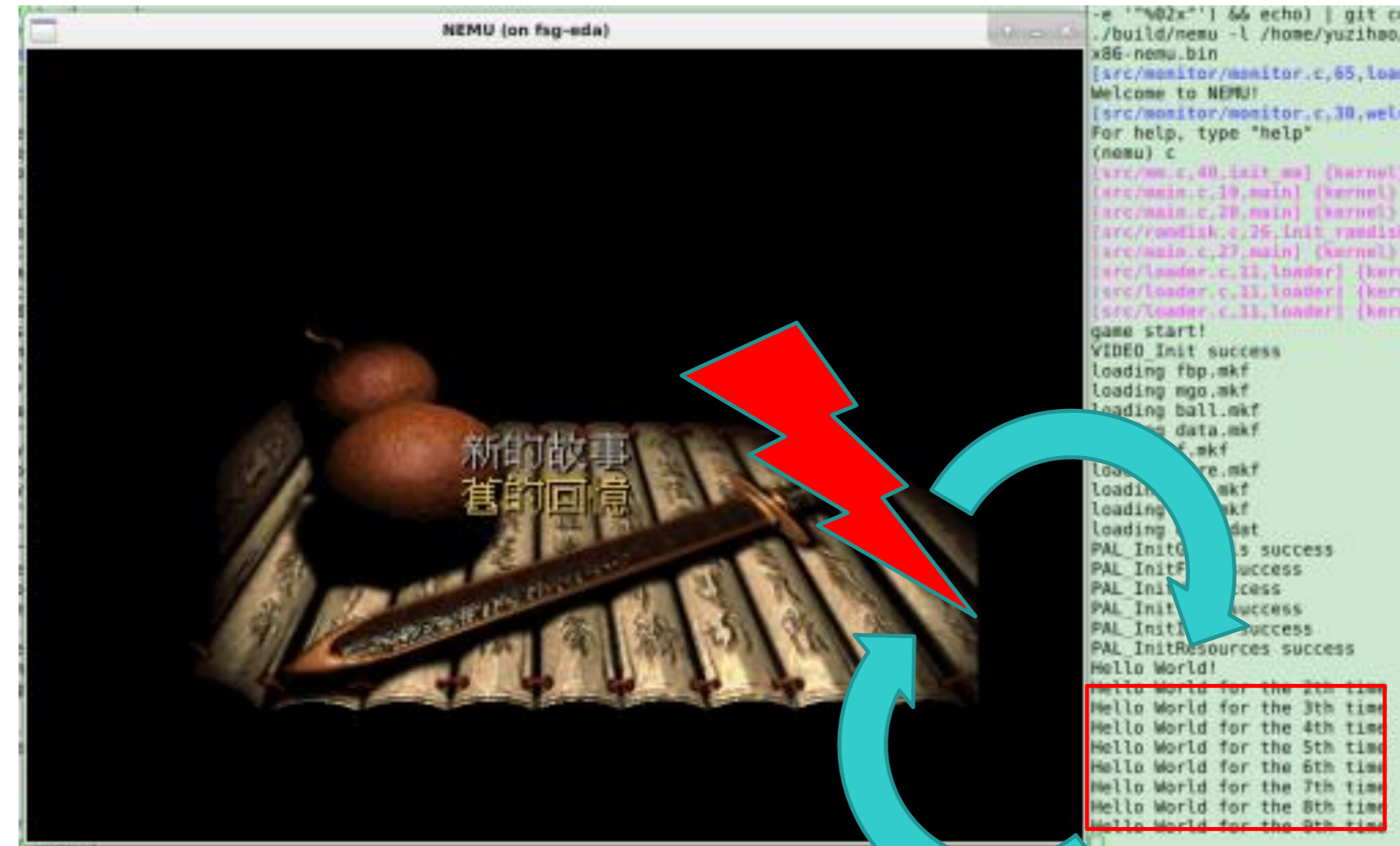
## ▶ 虚存管理

- `protect()` 创建虚拟地址空间
- `map()` 添加va到pa的映射关系
- `ucontext()` 创建用户上下文

## ▶ 多处理器

- `cpu_count()` 返回处理器数量
- `cpu_current()` 返回当前处理器编号
- `atomic_xchg()` 共享内存的原子交换

## ▶ 一生一芯不作要求, 但很酷!





# 提纲

- ▶ 程序的运行 - 从入门到放弃
- ▶ AM - 裸机运行时环境
- ▶ **基于AM的教学生态**
- ▶ AM代码导读

# 花絮 - AM的由来

- ▶ 把系统方向的课程实验串起来, 是各大高校计算机系统教学梦想之一
- ▶ 南京大学也经历了这个过程
  - 2008年: 第一版(流传至今的)组成原理实验, 由06级学长设计
  - 2010年: 第一版(流传至今的)操作系统实验, 由07级学长设计(年轻时的蒋炎岩老师)
  - 2012年: 第一版(流传至今的)编译原理实验, 由08级学长设计
  - 2014年: 第一版(流传至今的)系统基础实验, 由10级学长设计
    - ▶ 你可能听过南京大学PA的传说
  - 2017年1月的一次讨论: 感觉该有的都有了, 要不玩个大的?
    - ▶ 自己写个CPU, 上面跑自己的OS, 上面跑自己编译器编译出来的应用程序
    - ▶ 应用程序可以是NEMU模拟器, 上面又可以跑自己的OS, .....(递归了)
- ▶ 听上去很酷, 但最大的挑战是: 有bug怎么调试?

# 花絮 - AM的由来

- ▶ 2017年3月10日的计算机系统综合实验课上, 蒋炎岩老师首次介绍AM
  - 解决调试问题的核心思想: 通过AM运行时环境, 把ISA/架构和OS解耦

## AM: ISA → OS之间的桥梁

- ISA可以变, 测试程序和OS不用变
  - 程序可以直接在Linux上的AM模拟器运行
  - 为每个体系结构编写一个AM运行库和对应的loader程序
- 程序可以根据AM的支持提供不同服务
  - 打字游戏可以在任何AM上运行, 但不同AM实现方式不同
  - OS在无ASM时是批处理系统; 在无PM时不提供存储保护, 但维持大部分功能

通过AM把ISA/架构和OS解耦

## Turing Machine (TRM)

```
void _start();

void _putc(char ch);
void _panic(int code);

typedef struct _Area {
    void *start, *end;
} _Area;
extern _Area _heap;
```

## IO Machine (IOM)

```
ulong _uptime();

int _peek_key(); // 0x0000d0kk

typedef u32 Pixel; // 0x??rrggbb
typedef struct Screen {
    int width, height;
} Screen;
extern Screen _scr;
void _draw_p(int x, int y, Pixel p);
void _draw_f(Pixel *p);
void _draw_sync();
```

d be implemented using a bus.

## Asynchronous Machine (ASYM)

```
// _RegSet is architectural-dependent
void _listen(_RegSet* (*1)(int ex, _RegSet *regs));

void _make(_Area *stack, void *entry, int nargs, ...);

void _trap();

void _ienable();
void _idisable();
int _istatus();
```

## 更多可以支持的功能

- Protected Machine (PTM)
  - `_protect()/_destroy()` → 保护/释放地址空间
  - 提供缺页处理机制
- Symmetric Multicore Machine (MCM)
  - `_cpu()` → 返回per-cpu的local storage
  - `_atomic_xchg()` → 原子操作

3月17日AM第一版API

# Project-N(南京大学系列实验)组件



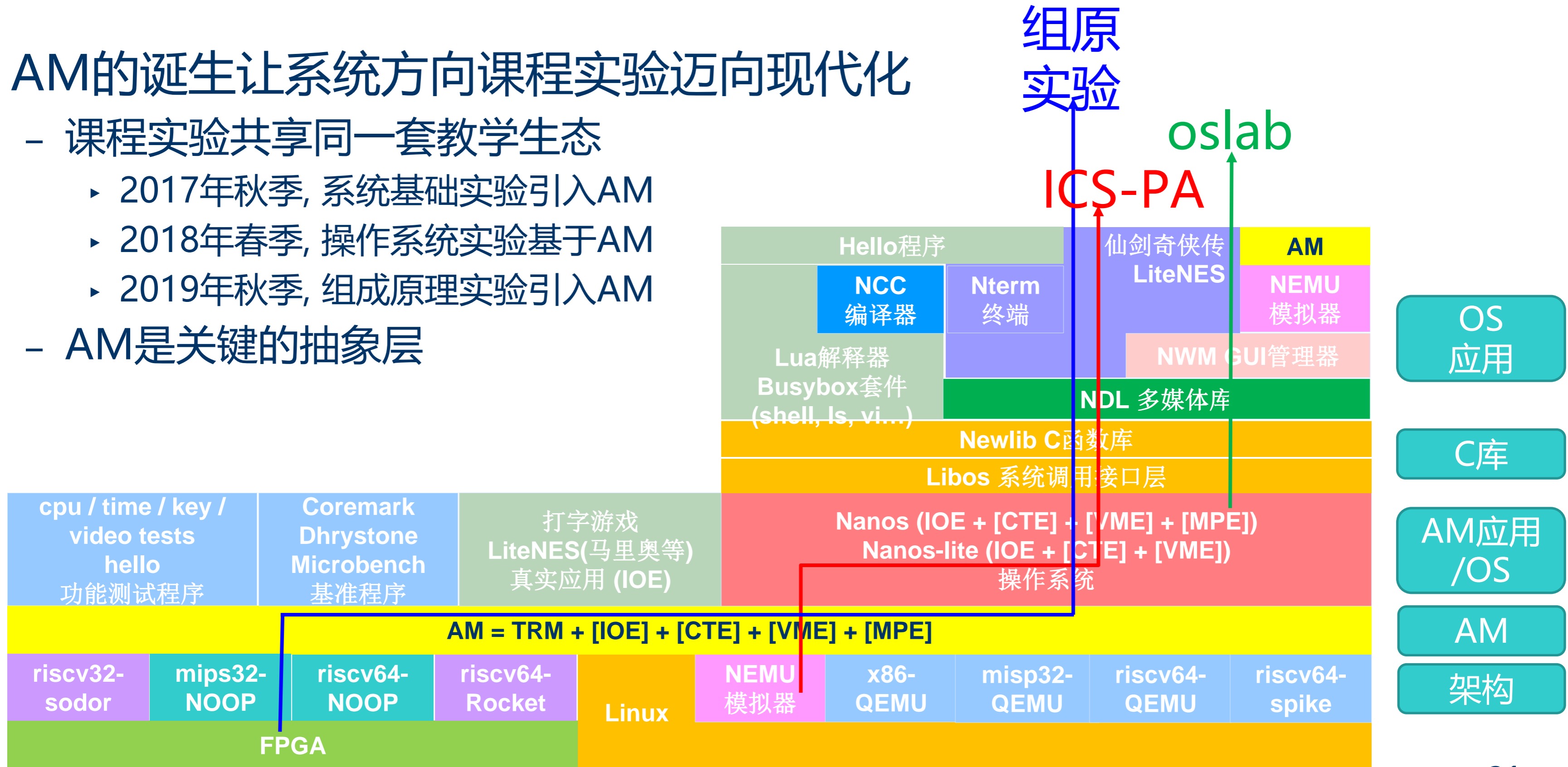
- ▶ 和蒋炎岩老师共同维护

Project-N组件	说明
Navy-apps	应用程序集
NCC	NJU C Compiler, C编译器
Newlib	嵌入式C库(从官方版本移植)
Nanos/Nanos-lite	NanJU OS, 操作系统/简化版
Nexus-AM (新名称Abstract-Machine)	抽象计算机, 裸机运行时环境
NEMU	NJU EMUlator, 全系统模拟器
NPC	NJU Personal Computer, SoC
NOOP	NJU Out-of-Order Processor, 处理器

# 基于AM的Project-N

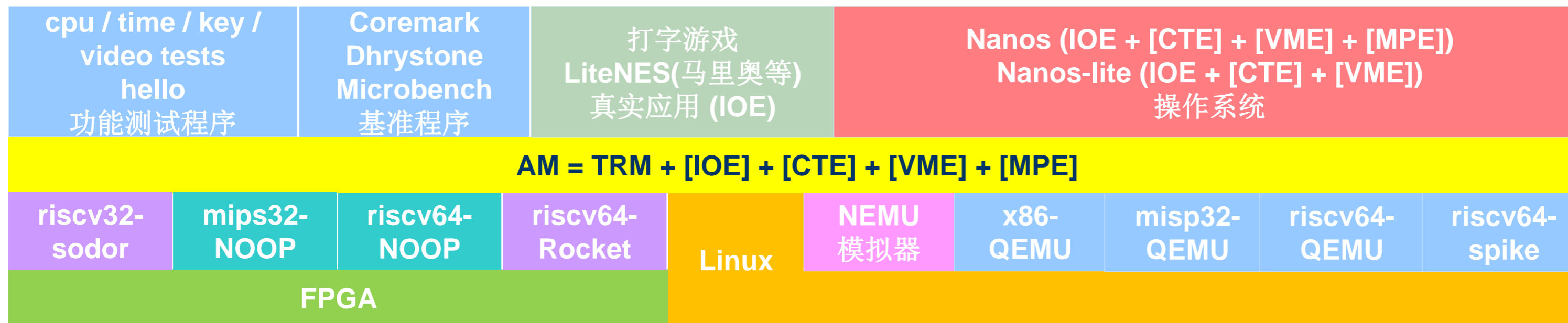
## AM的诞生让系统方向课程实验迈向现代化

- 课程实验共享同一套教学生态
  - 2017年秋季, 系统基础实验引入AM
  - 2018年春季, 操作系统实验基于AM
  - 2019年秋季, 组成原理实验引入AM
- AM是关键抽象层



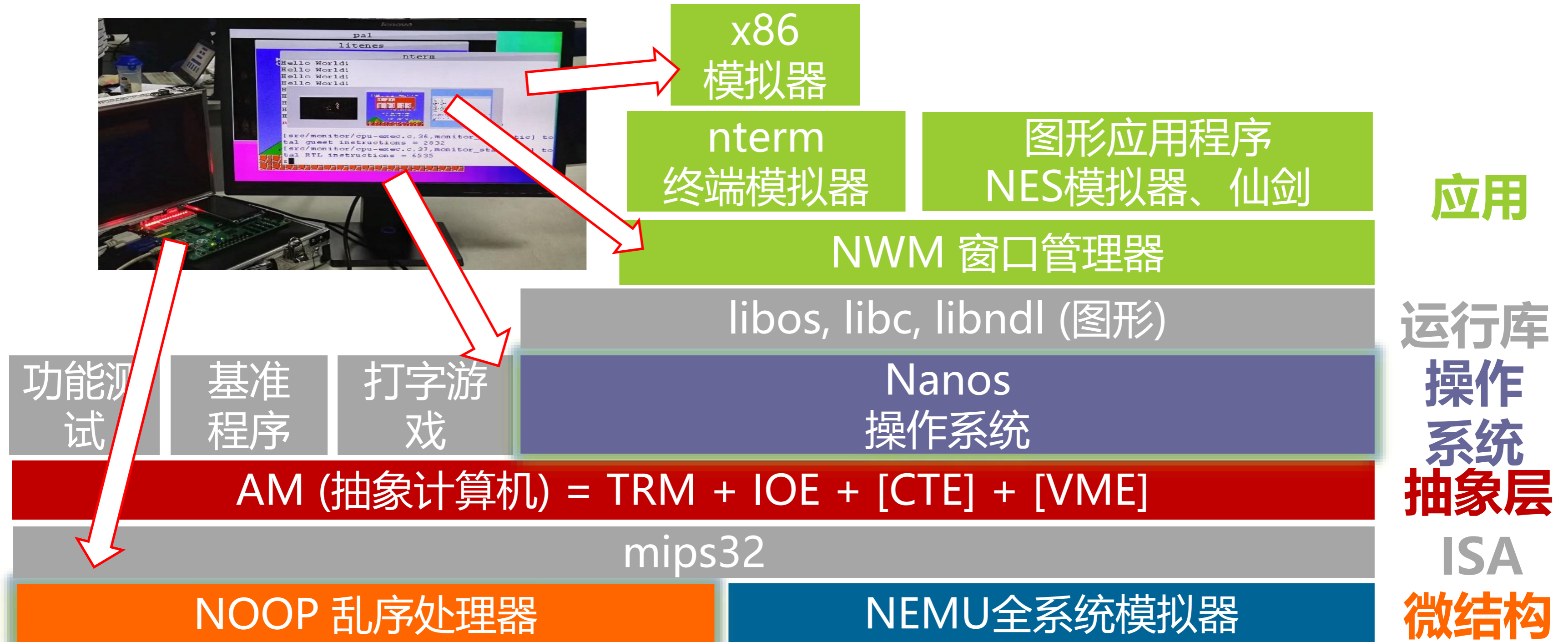
# AM生态的好处

- ▶ 应用移植到AM, 即可运行在各种架构
  - 基于AM开发的OS甚至能在Linux native运行
  - 先在Linux native上调试应用程序, 然后再运行到目标架构
- ▶ 架构支持AM的API, 即可运行各种AM应用
  - 首期一生一芯开发果壳初期, 就有丰富的程序可以测试
    - ▶ 中期可运行超级玛丽和仙剑等复杂应用



# 2018年龙芯杯南京大学作品 (系统综合实验)

- 展示全自制计算机系统生态(基准程序, libc, NES模拟器和仙剑除外)



# Project-N中OS上的运行库和应用程序

库	说明
libc	Newlib嵌入式C库, 支持POSIX标准
libos	系统调用接口
compiler-rt	为32位ISA提供64位除法的支持
libfixedptc	定点数计算, 替代范围不大的浮点数
libndl	NJU DirectMedia Layer, 提供时钟, 按键, 绘图, 声音的底层抽象
libbmp	BMP图片解析
libbdf	BDF字体解析
libminiSDL	提供部分兼容SDL库的API, 基于NDL
libvorbis	OGG音频解码
libam	用Navy运行时环境实现的AM API
libSDL_ttf	Truetype字体解析
libSDL_image	JPG, PNG, BMP, GIF图片解码
libSDL_mixer	多通道混声

应用程序	说明
NSlider	NJU幻灯片播放器
menu	启动菜单, 可选择其他应用来运行
NTerm	NJU终端, 包含简易内建Shell, 也支持外部Shell
bird	Flappy Bird小游戏
PAL	仙剑奇侠传, 支持音效
am-kernels	AM应用
FCEUX	红白机模拟器
oslab0	学生在OS课上编写的裸机游戏集合
NPlayer	NJU音频播放器
lua	LUA脚本解释器
busybox	Busybox套件, 包含vi, shell等工具
ONScripter	NScripter脚本解释器, 可运行Clannad等游戏
NWM	NJU窗口管理器



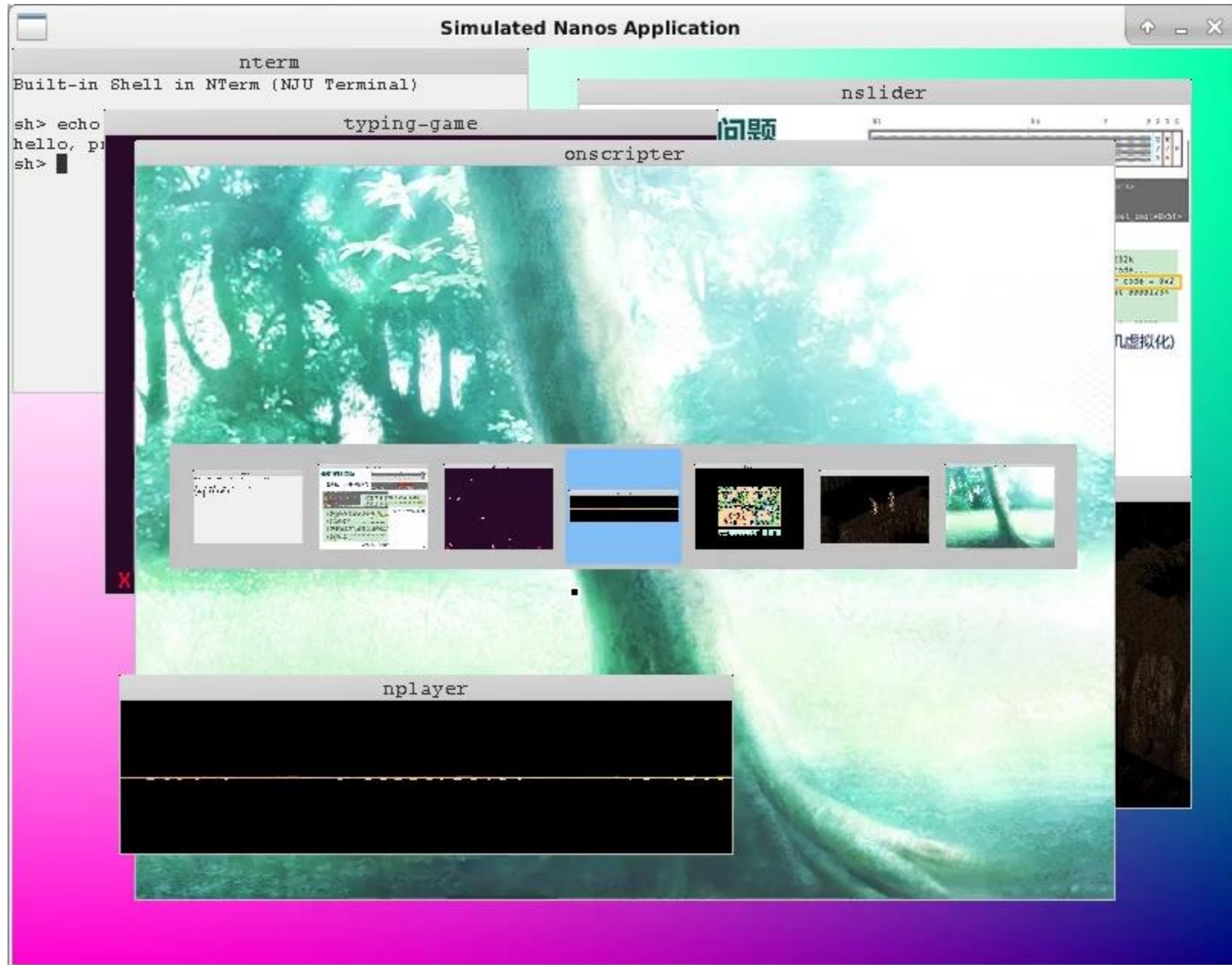
# 运行现代游戏

- ▶ 无需VME即可运行
- ▶ 运行一个批处理系统来提供系统调用的支持就可以了



# 运行自制GUI操作系统

- ▶ 在自制窗口管理器的管理下, 可分时运行
  - 自制终端
  - 自制幻灯片播放器
  - 自制打字游戏
  - 自制音频播放器
  - 上古游戏红白机超级玛丽
  - 近代游戏仙剑奇侠传
  - 现代游戏ONS版Clannad
- ▶ 期待有一天能运行在自己设计的芯片上



# 提纲

- ▶ 程序的运行 - 从入门到放弃
- ▶ AM - 裸机运行时环境
- ▶ 基于AM的教学生态
- ▶ **AM代码导读 (请陈璐主持)**

# 实现AM是一项软硬协同的工作

- ▶ 在硬件中实现机制, 然后在AM中使用该机制实现相应API
- ▶ 实现TRM的halt()
  - 在CPU中实现一条特殊的自定义指令
    - ▶ 若在仿真环境中执行该指令, 则仿真终止
  - 在halt()中通过内联汇编编写该指令
- ▶ 实现TRM的putch()
  - 在CPU中实现一条特殊的自定义指令
    - ▶ 若在仿真环境中执行该指令, 则通过\$write(verilog)或printf(chisel)输出某寄存器中的字符
    - ▶ 这样就能以最小代价支持程序输出了, 将来接入串口后再修改
  - 在putch()中通过内联汇编编写该指令
    - ▶ 思考: 应该怎么写?
- ▶ IOE和CTE会在将来的报告中介绍

# 在自己的CPU上运行AM程序

- ▶ riscv64-mycpu的构建脚本已经帮助大家生成bin文件了
  - 在相应的build目录下
- ▶ klib中的库函数并没有完整实现(因为是学生的编程作业)
  - 对一生一芯来说是选做
  - 但还是建议大家尝试实现一下, 可以跑更多的AM程序来测试自己的CPU
    - ▶ 例如了解一下printf是如何实现的(只需要实现%d和%s即可)
  - 如果实在不想自己实现, 网上肯定能找到参考代码
- ▶ cpu-tests
  - 其中的string和hello-str需要实现相应的库函数才能正确运行
- ▶ riscv-tests
  - 我们把rv64i的测试移植到AM了, 运行方式和cpu-tests类似
  - <https://github.com/NJU-ProjectN/riscv-tests>

# 在自己的CPU上运行AM程序

- ▶ 实现putch()后, 可以额外运行
  - am-kernels/kernels/hello
- ▶ 实现klib的printf()后, 可以额外运行am-kernels/benchmarks/目录下的基准测试
  - 包括coremark, dhrystone和microbench
  - 不过由于未实现时钟, 所以计时结果为0
- ▶ 有了这么多测试程序, 够大家做一段时间了 😊

# 小结

- ▶ 程序的运行 - 从入门到放弃
  - 程序的运行需要运行时环境的支持
- ▶ AM = 按照计算机发展史将计算机功能抽象地模块化的裸机运行时环境
- ▶ AM作为运行时环境, 将架构和应用进行解耦
  - 应用移植到AM, 即可运行在各种架构
    - ▶ 有兴趣的同学可以尝试把xv6移植到AM
  - 架构支持AM的API, 即可运行各种AM应用
- ▶ 理解AM = 理解程序如何生成并在计算机上运行
- ▶ 实现AM = 软硬协同地理解计算机系统

# 参考资料

## ▶ AM repo

- <https://github.com/NJU-ProjectN/abstract-machine>

## ▶ AM概述

- 计算机系统基础课程视角 - <https://nju-projectn.github.io/ics-pa-gitbook/ics2020/2.3.html>
- 操作系统课程视角 - [http://jyywiki.cn/OS/AbstractMachine/AM\\_Design](http://jyywiki.cn/OS/AbstractMachine/AM_Design)

## ▶ AM接口规范

- [http://jyywiki.cn/OS/AbstractMachine/AM\\_Spec](http://jyywiki.cn/OS/AbstractMachine/AM_Spec)

## ▶ AM选讲(习题课)

- 习题课ppt - <http://jyywiki.cn/ICS/2020/slides/9.slides#/>
- B站录播 - <https://www.bilibili.com/video/BV1qa4y1j7xk?p=8>