ISCAS 中国科学院软件研究所 Institute of Software Chinese Academy of Sciences | 智能软件研究中心 Intelligent Software Research Center

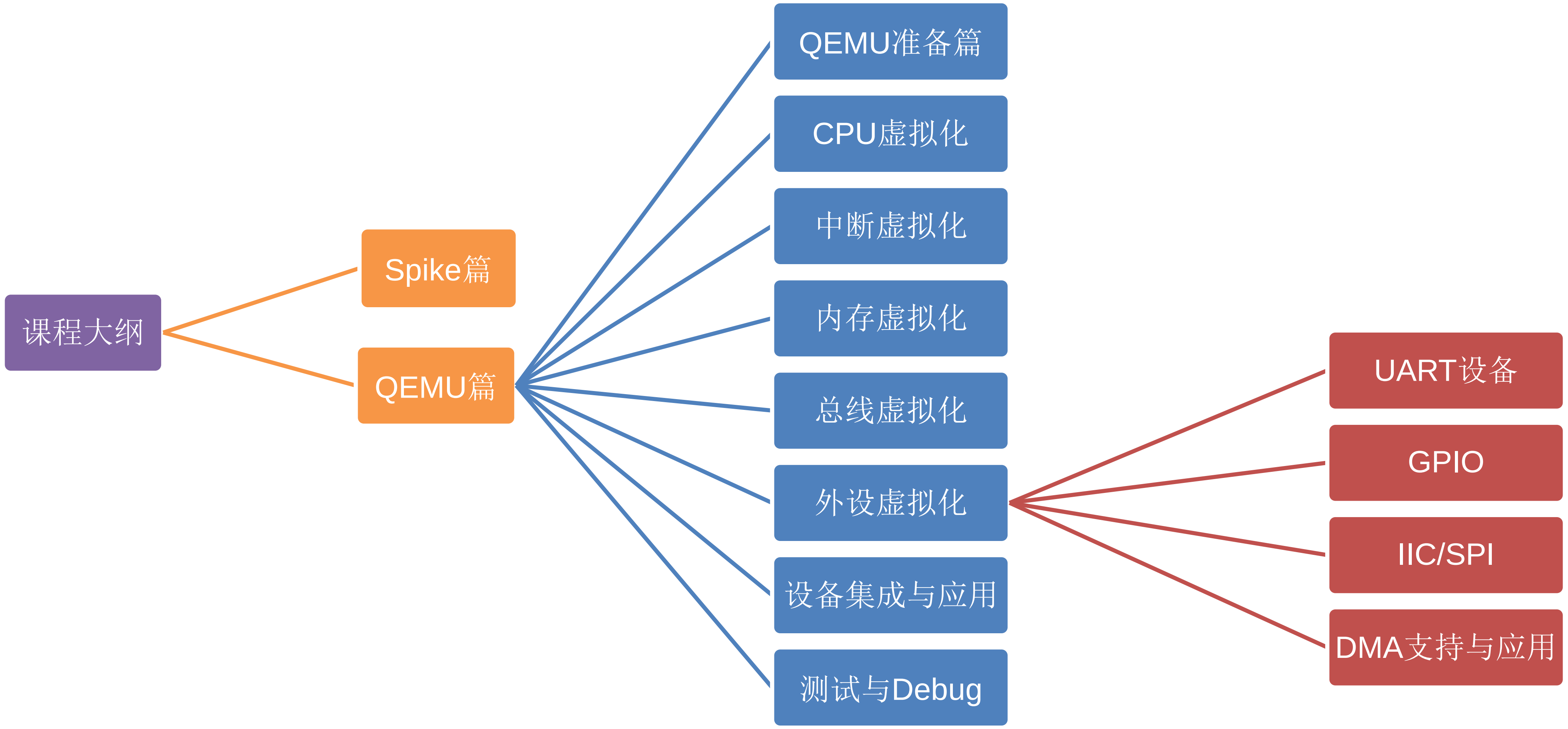
从零开始的RISC-V模拟器开发

第6讲 QEMU篇之基础知识

中国科学院软件研究所
PLCT实验室

王俊强 wangjunqiang@iscas.ac.cn
李威威 liweiwei@iscas.ac.cn
吴伟 wuwei2016@iscas.ac.cn

课程介绍——QEMU篇


课程大纲
Spike篇
QEMU篇
QEMU准备篇
CPU虚拟化
中断虚拟化
内存虚拟化
总线虚拟化
外设虚拟化
设备集成与应用
测试与Debug
UART设备
GPIO
IIC/SPI
DMA支持与应用

QEMU模拟器

基于QEMU实现目标

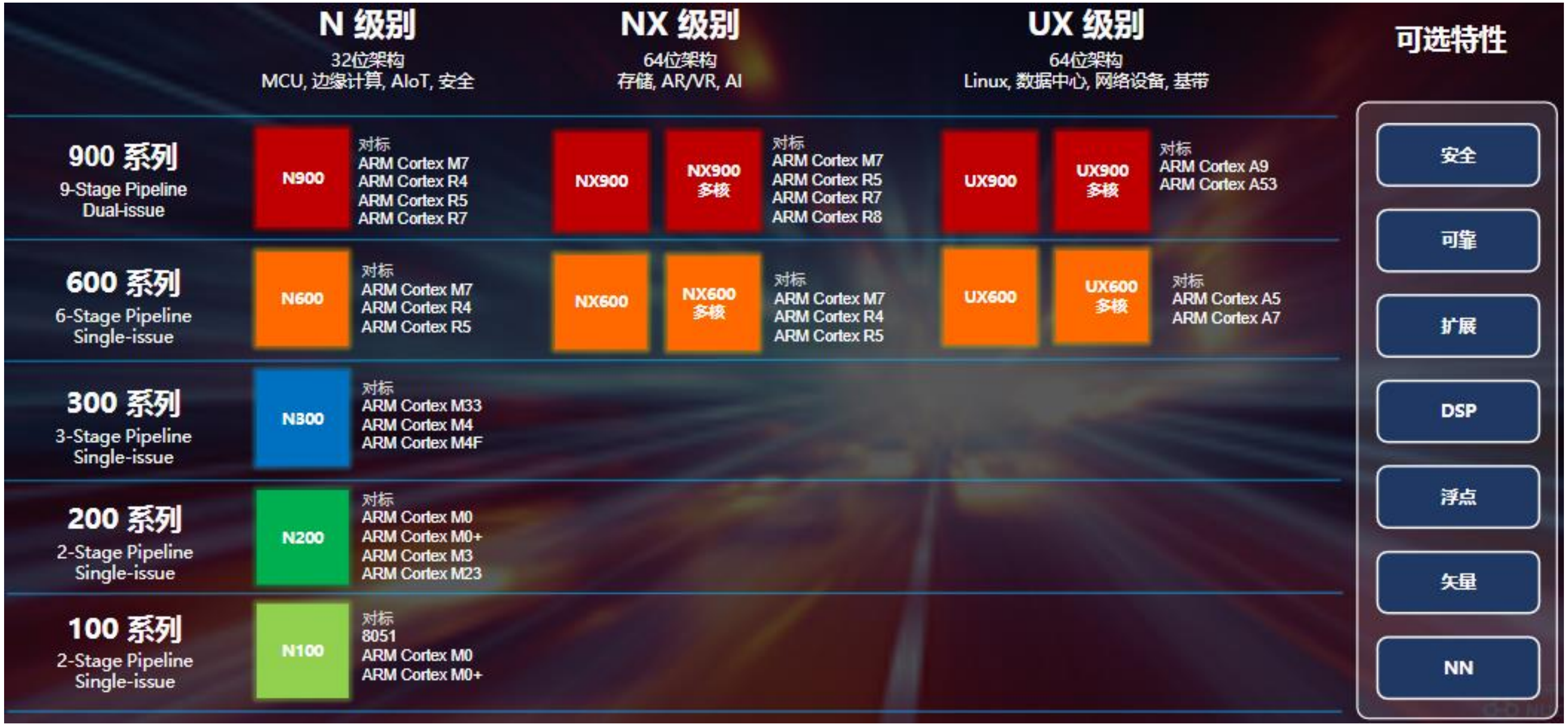
1.实现对芯来HummingBird Evaluation Kit, 200T FPGA Board的模拟
2.支持芯来N600 NX600 UX600 CPU运行baremetal demo, RTOS, Linux
3.介绍整体过程中遇到相关实现的原理

课程点	涉及主要知识点
CPU虚拟化	RISCV CPU实现分析，新RISCV Soc建立，CSR寄存器实现添加，TCG原理，指令添加……
中断虚拟化	QEMU RISC-V IRQ机制，Nuclei Timer 和 Eclic 实现，Sifive Clint 和Plic实现……
内存虚拟化	QEMU 内存原理介绍，用户态/系统态程序加载运行分析……
总线虚拟化	QEMU 总线原理介绍
外设虚拟化	200T上UART实现，GPIO实现与应用， IIC/SPI实现与应用， DMA支持与应用(可能以其他board为例)……
Machine虚拟化	QEMU的整体运行流程分析，MCU/Linux设备组织方式……
QEMU测试	介绍QEMU中主要的测试方法，指令测试方法，设备测试方法……
QEMU Debug	介绍QEMU中DEBUG的实现……

环境介绍

- 硬件环境
  - 目标硬件
- 软件环境
  - 开发环境
  - 测试环境

NUCLEI Processor(Core)



	N 级别 32位架构 MCU, 边缘计算, AIoT, 安全		NX 级别 64位架构 存储, AR/VR, AI			UX 级别 64位架构 Linux, 数据中心, 网络设备, 基带		
900 系列 9-Stage Pipeline Dual-issue	N900	对标 ARM Cortex M7 ARM Cortex R4 ARM Cortex R5 ARM Cortex R7	NX900	NX900 多核	对标 ARM Cortex M7 ARM Cortex R5 ARM Cortex R7 ARM Cortex R8	UX900	UX900 多核	对标 ARM Cortex A9 ARM Cortex A53
600 系列 6-Stage Pipeline Single-issue	N600	对标 ARM Cortex M7 ARM Cortex R4 ARM Cortex R5	NX600	NX600 多核	对标 ARM Cortex M7 ARM Cortex R4 ARM Cortex R5	UX600	UX600 多核	对标 ARM Cortex A5 ARM Cortex A7
300 系列 3-Stage Pipeline Single-issue	N300	对标 ARM Cortex M33 ARM Cortex M4 ARM Cortex M4F						
200 系列 2-Stage Pipeline Single-issue	N200	对标 ARM Cortex M0 ARM Cortex M0+ ARM Cortex M3 ARM Cortex M23						
100 系列 2-Stage Pipeline Single-issue	N100	对标 8051 ARM Cortex M0 ARM Cortex M0+						

可选特性

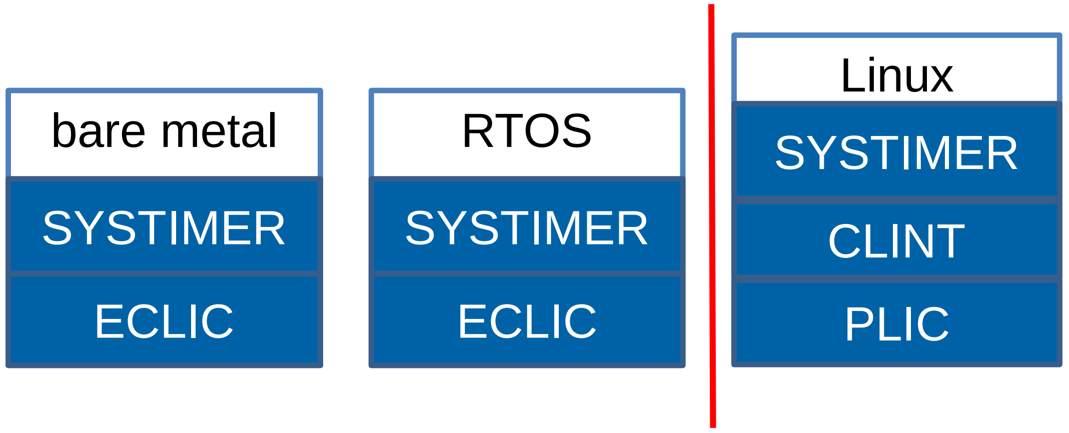
- 安全
- 可靠
- 扩展
- DSP
- 浮点
- 矢量
- NN

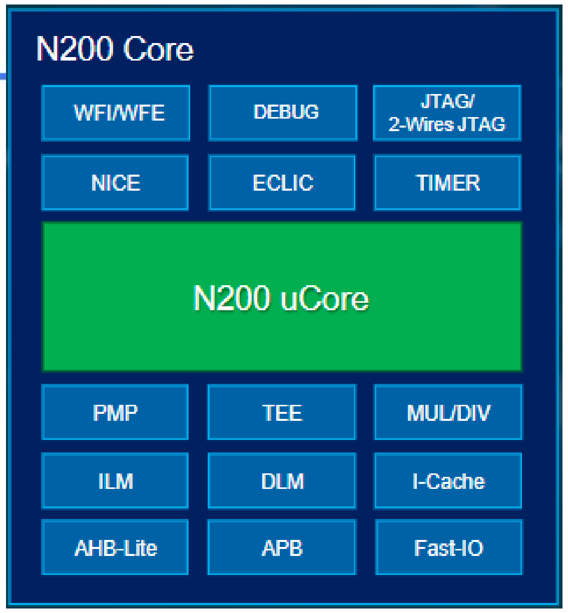
来自: www.nucleisys.com/product.php

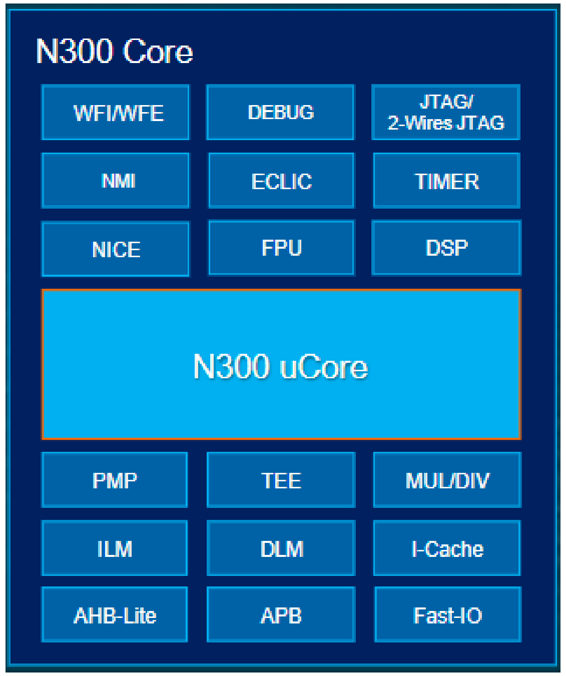
官网: https://www.nucleisys.com/index.php

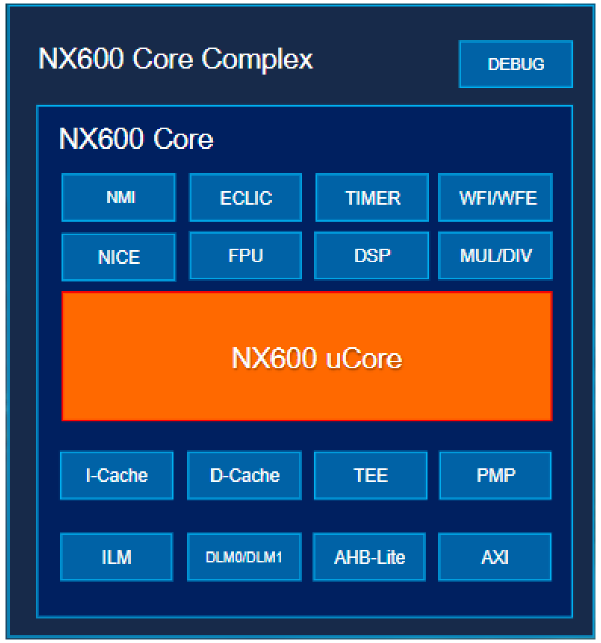
NUCLEI SOC

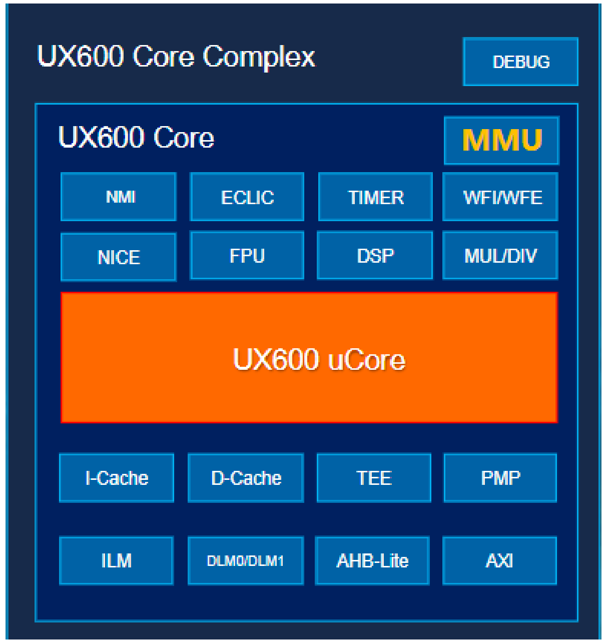
CORE系列	RISC-V支持状态
N200	RV32I/E/M/A/C
N300	RV32I/E/M/A/C/F/D/P
N600	RV32I/M/A/C/F/D/P
NX600	RV64I/M/A/C/F/D/P
UX600(MMU)	RV64I/M/A/C/F/D/P
N900	RV32I/M/A/C/F/D/P/V
NX900	RV64I/M/A/C/F/D/P/V
UX900(MMU)	RV64I/M/A/C/F/D/P/V







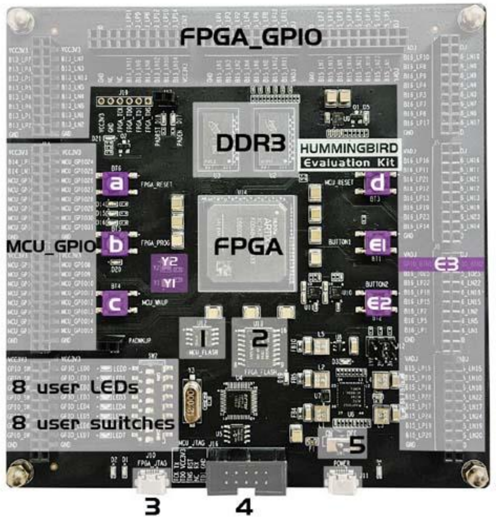




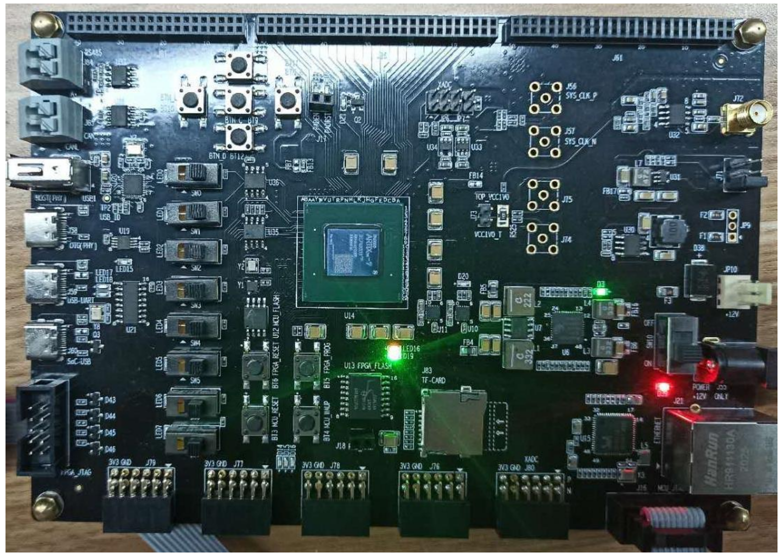
来自: www.nucleisys.com/product.php

NUCLEI Boards(Nuclei FPGA Evaluation Kit)

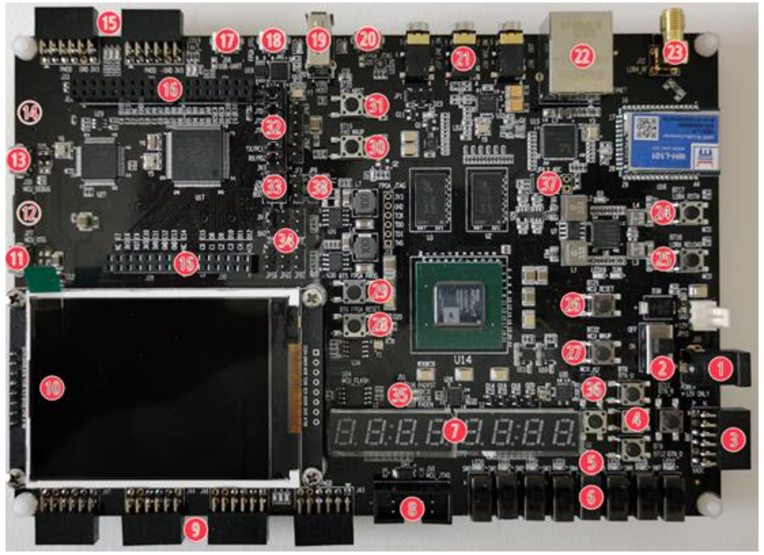
a: FPGA_RESET
b: FPGA_PROG
c: MCU_WKUP
d: MCU_RESET
e1: User button 1
e2: User button 2
e3: User button header
Y1: GCLK
Y2: RTC_CLK
1: MCU_FLASH
2: FPGA_FLASH
3: FPGA_JTAG
4: MCU_JTAG
5: Power switch



100T version
Xilinx XC7A100T FPGA



MCU 200T version
Xilinx XC7A200T FPGA



DDR 200T version
Xilinx XC7A200T FPGA
MCU子系统暂不考虑

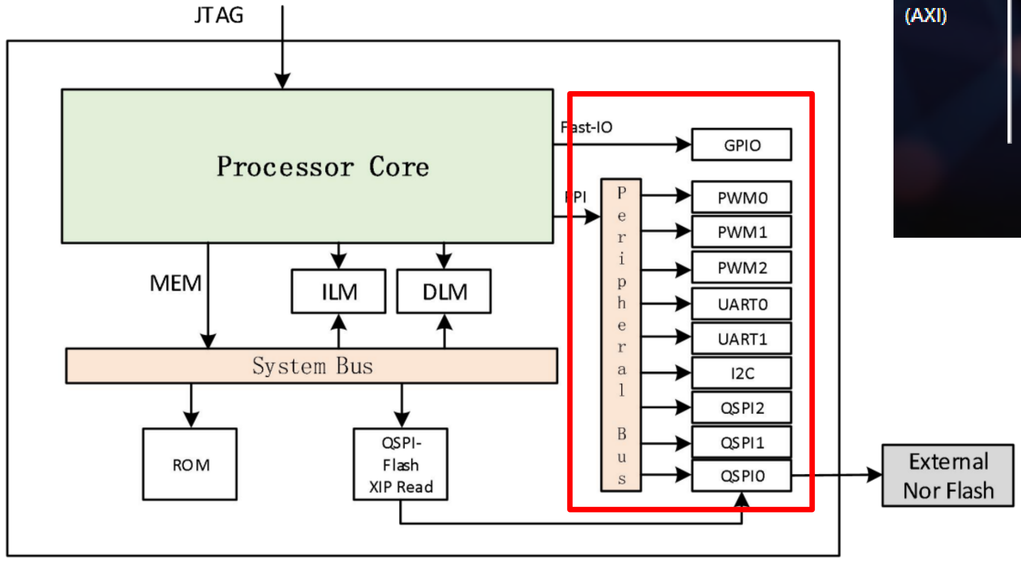
本课程主线

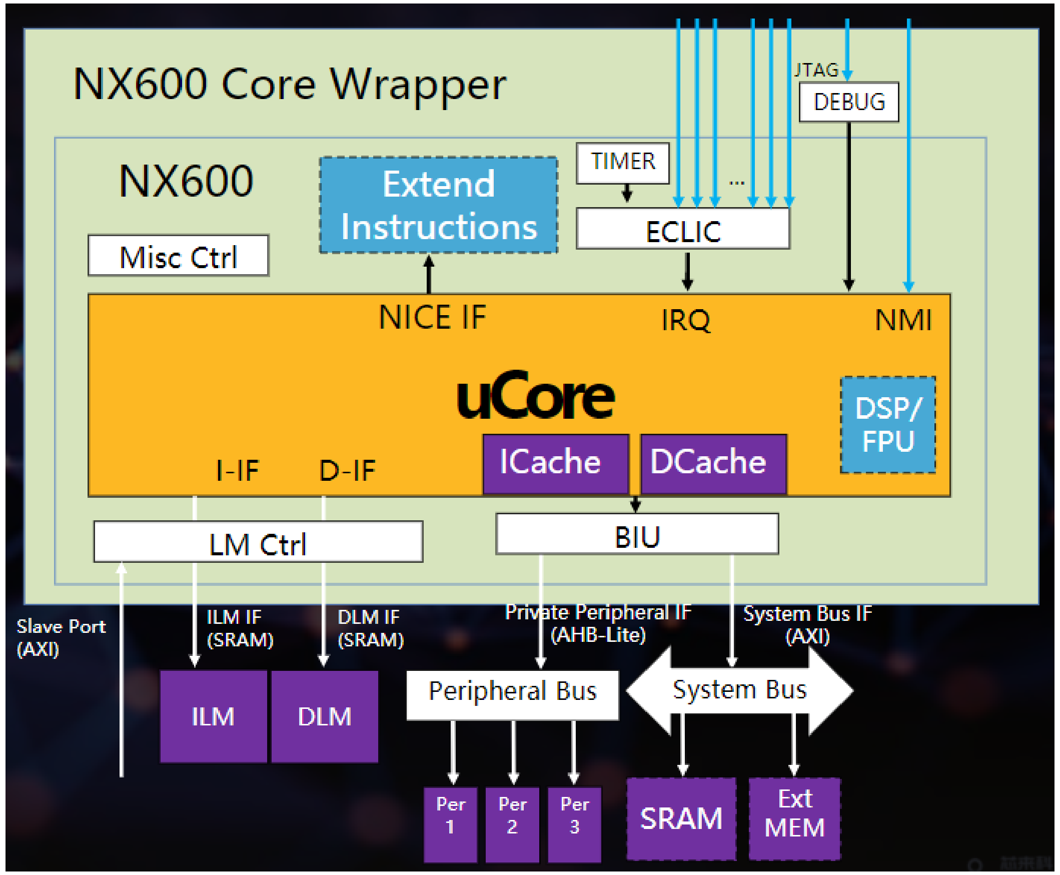
https://nucleisys.com/developboard.php

https://doc.nucleisys.com/nuclei_sdk/design/board/nuclei_fpga_eval.html

主线任务

- CPU模拟
  - CPU Model
  - 指令与CSR
- 内存模拟
- 中断模拟
  - ECLIC/PLIC
- 外设模拟
  - SYSTIMER/CLINT
  - UART
  - GPIO(扩展)
  - SPI/IIC(扩展)
  - DMA(扩展)
- Board模拟



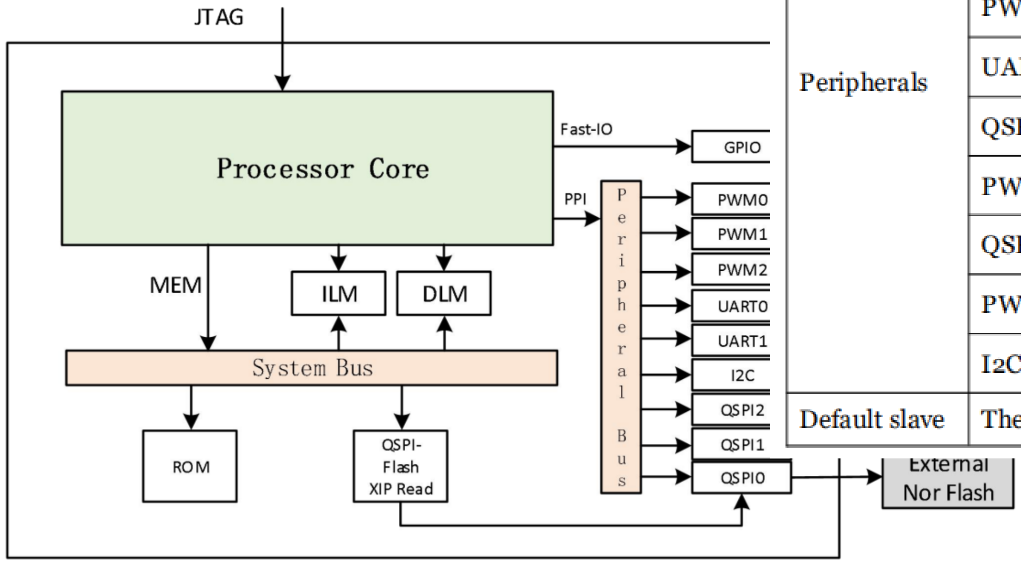


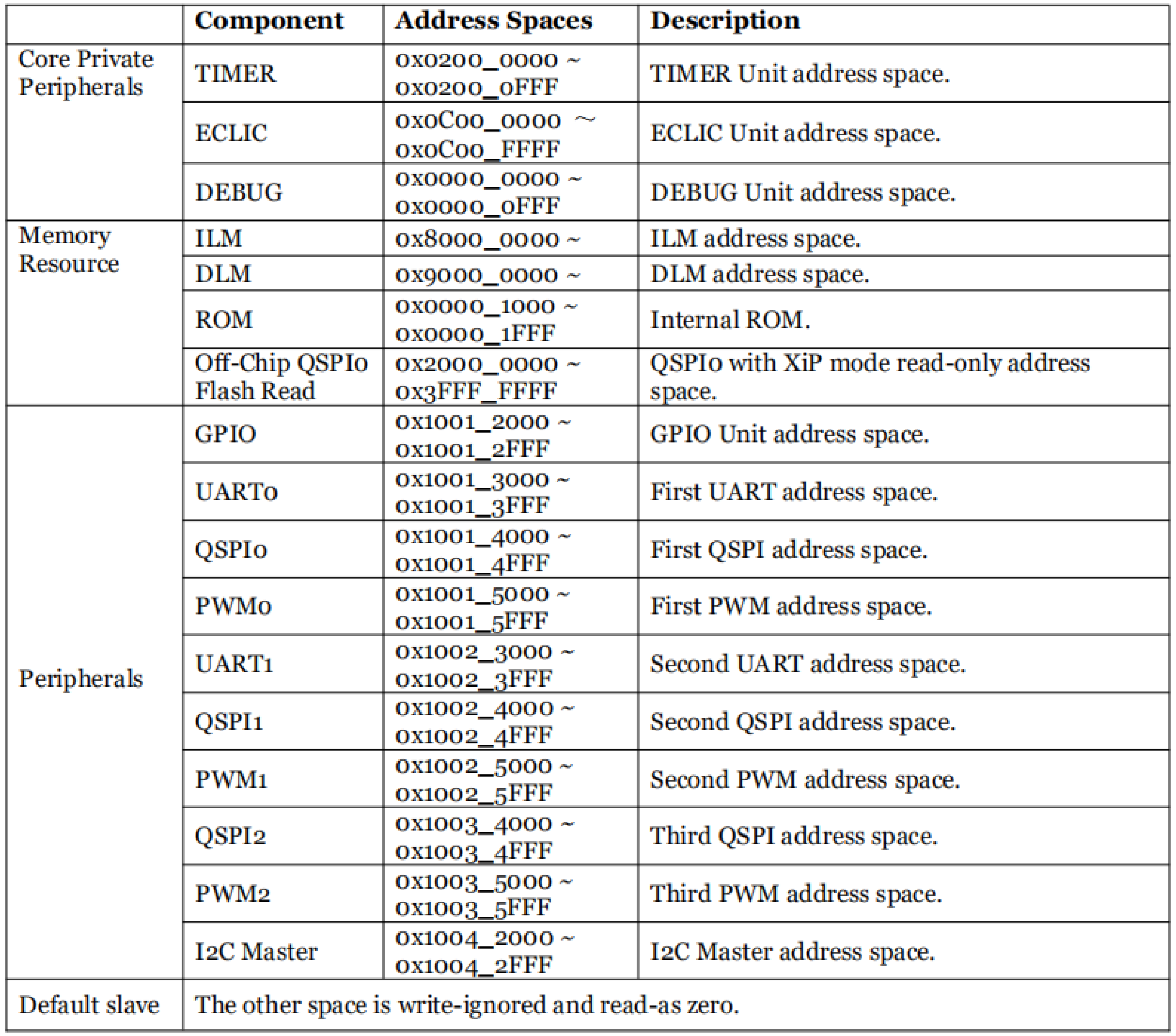
NX600系统框架图示例

https://doc.nucleisys.com/nuclei_sdk/design/soc/demosoc.html

主线任务

- ➢CPU模拟
  CPU Model
  指令与CSR
- ➢内存模拟
- ➢中断模拟
  ECLIC/PLIC
- ➢外设模拟
  SYSTIMER/CLINT
  UART
  GPIO(扩展)
  SPI/IIC(扩展)
  DMA(扩展)
- ➢ Board模拟





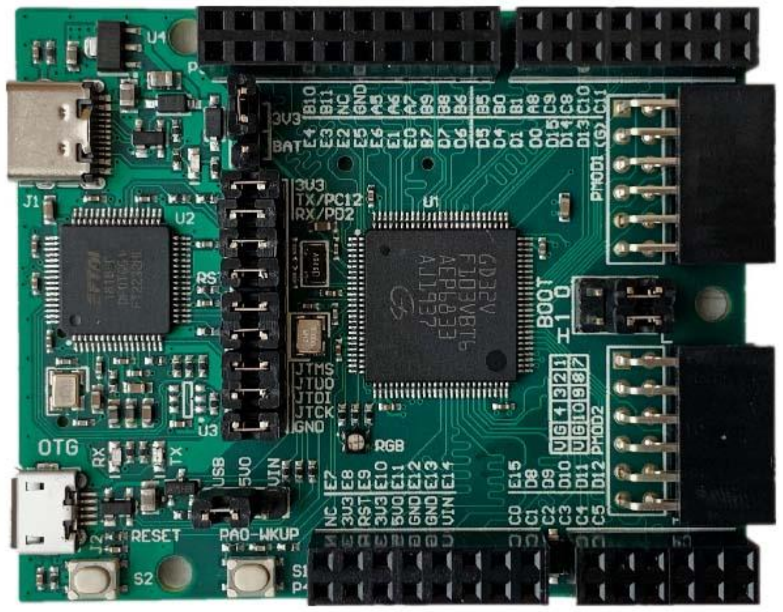
	Component	Address Spaces	Description
Core Private Peripherals	TIMER	0x0200_0000 ~ 0x0200_0FFF	TIMER Unit address space.
	ECLIC	0x0C00_0000 ~ 0x0C00_FFFF	ECLIC Unit address space.
	DEBUG	0x0000_0000 ~ 0x0000_0FFF	DEBUG Unit address space.
Memory Resource	ILM	0x8000_0000 ~	ILM address space.
	DLM	0x9000_0000 ~	DLM address space.
	ROM	0x0000_1000 ~ 0x0000_1FFF	Internal ROM.
	Off-Chip QSPI0 Flash Read	0x2000_0000 ~ 0x3FFF_FFFF	QSPI0 with XiP mode read-only address space.
Peripherals	GPIO	0x1001_2000 ~ 0x1001_2FFF	GPIO Unit address space.
	UART0	0x1001_3000 ~ 0x1001_3FFF	First UART address space.
	QSPI0	0x1001_4000 ~ 0x1001_4FFF	First QSPI address space.
	PWM0	0x1001_5000 ~ 0x1001_5FFF	First PWM address space.
	UART1	0x1002_3000 ~ 0x1002_3FFF	Second UART address space.
	QSPI1	0x1002_4000 ~ 0x1002_4FFF	Second QSPI address space.
	PWM1	0x1002_5000 ~ 0x1002_5FFF	Second PWM address space.
	QSPI2	0x1003_4000 ~ 0x1003_4FFF	Third QSPI address space.
	PWM2	0x1003_5000 ~ 0x1003_5FFF	Third PWM address space.
	I2C Master	0x1004_2000 ~ 0x1004_2FFF	I2C Master address space.
Default slave	The other space is write-ignored and read-as zero.		

Nuclei Demo SoC Memory Map(缺ddr)

https://doc.nucleisys.com/nuclei_sdk/design/soc/demosoc.html

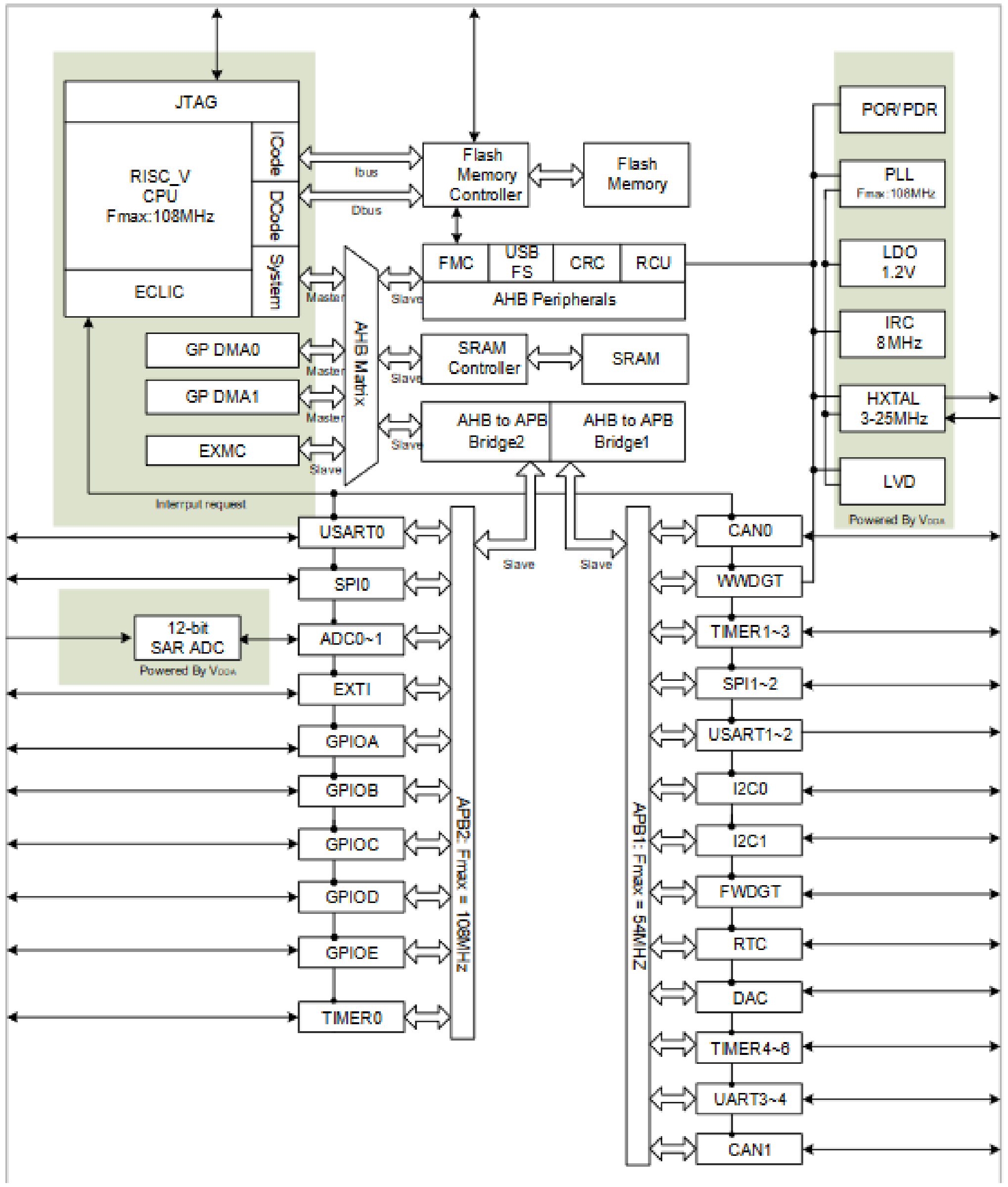
中国科学院软件研究所 Institute of Software Chinese Academy of Sciences　智能软件研究中心 Intelligent Software Research Center

GD32VF103V Boards(RV-STAR Kit)



GD32VF103V RV-STAR Board　N203

本课程支线



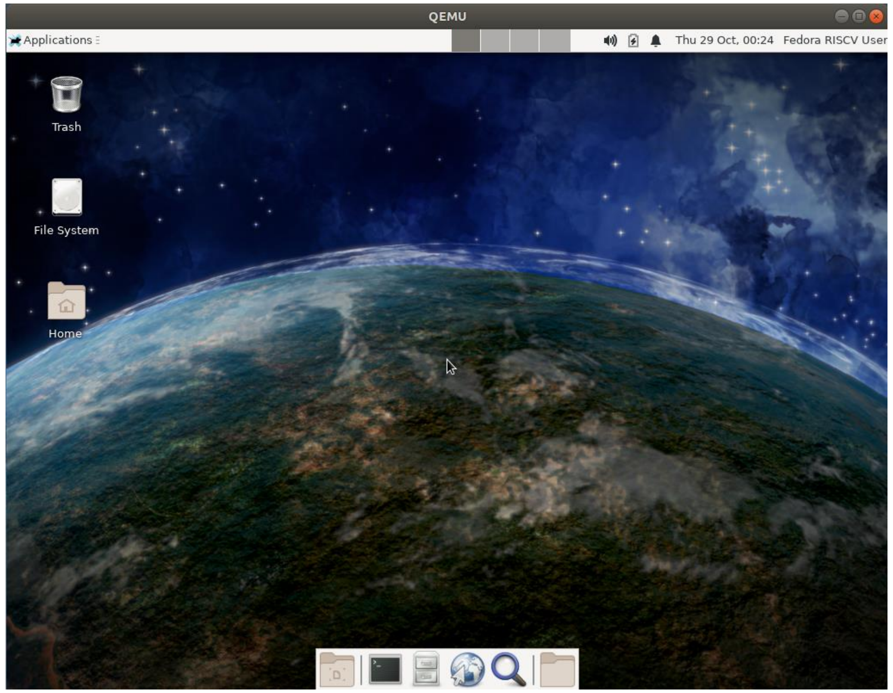
https://doc.nucleisys.com/nuclei_sdk/design/board/gd32vf103v_rvstar.html

软件环境(开发对象)

QEMU是一款开源的模拟器及虚拟机监管器(Virtual Machine Monitor, VMM)，通过动态二进制翻译来模拟CPU，并提供一系列的硬件模型，使guest os认为自己和硬件直接打交道，其实是同QEMU模拟出来的硬件打交道，QEMU再将这些指令翻译给真正硬件进行操作。

RISCV支持状态:

32bit Supported machines are:		**cpu type:**
none	empty machine	any
opentitan	RISC-V Board compatible with OpenTitan	lowrisc-ibex
sifive_e	RISC-V Board compatible with SiFive E SDK	rv32
sifive_u	RISC-V Board compatible with SiFive U SDK	sifive-e31
spike	RISC-V Spike board (default)	sifive-e34
virt	RISC-V VirtIO board	sifive-u34
64bit Supported machines are:		**cpu type:**
microchip-icicle-kit	Microchip PolarFire SoC Icicle Kit	any
none	empty machine	rv64
sifive_e	RISC-V Board compatible with SiFive E SDK	sifive-e51
sifive_u	RISC-V Board compatible with SiFive U SDK	sifive-u54
spike	RISC-V Spike board (default)	
virt	RISC-V VirtIO board	



Fedora Desktop for RISC-V

QEMU与RISC-V支持

Extension	Version
I	2.1
E	2.1
M	2.0
A	2.1
F	2.2
D	2.2
V	0.7.1/1.0
C	2.0
P	0.93?
H	0.3?
Counters	2.0
Zifencei	2.0
Zicsr	2.0

privileged 1.10.0
privileged 1.11.0
v extension 0.7.1

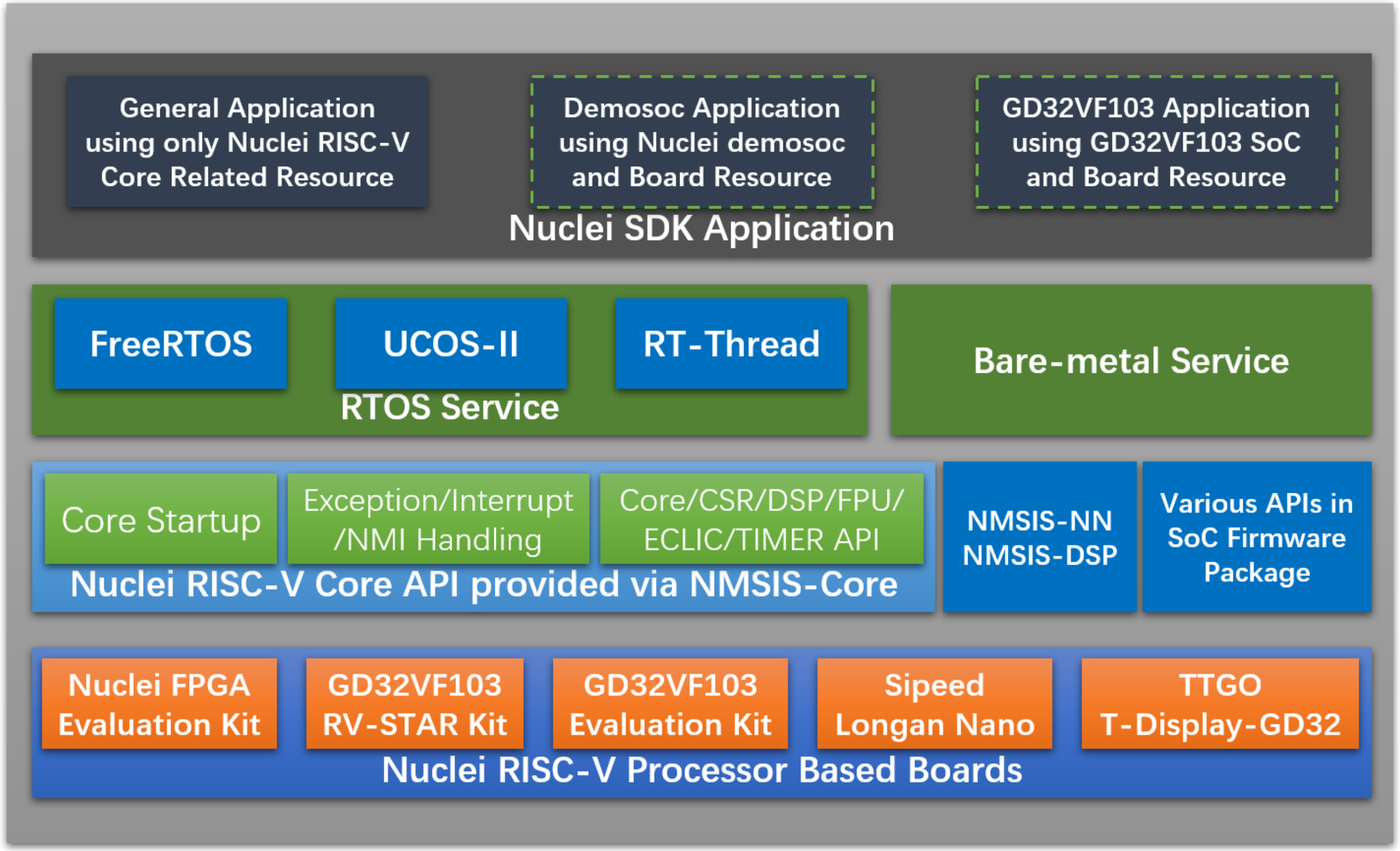
CORE系列	RISC-V支持状态
N200	RV32I/E/M/A/C
N300	RV32I/E/M/A/C/F/D/P
N600	RV32I/M/A/C/F/D/P
NX600	RV64I/M/A/C/F/D/P
UX600(MMU)	RV64I/M/A/C/F/D/P
N900	RV32I/M/A/C/F/D/P/V
NX900	RV64I/M/A/C/F/D/P/V
UX900(MMU)	RV64I/M/A/C/F/D/P/V

NUCLEI NICE for Demo

软件环境(验证环境)

- ➢ 准备环境(见第一课)
  芯来工具链: Nuclei Toolchain
  参考文档:Nuclei User Center

➢验证环境
- ➢Nuclei SDK
- ➢Nuclei Linux SDK
- ➢RT-Thread Nuclei BSP


General Application using only Nuclei RISC-V Core Related Resource
Demosoc Application using Nuclei demosoc and Board Resource
GD32VF103 Application using GD32VF103 SoC and Board Resource
Nuclei SDK Application
FreeRTOS
UCOS-II
RT-Thread
RTOS Service
Bare-metal Service
Core Startup
Exception/Interrupt /NMI Handling
Core/CSR/DSP/FPU/ ECLIC/TIMER API
Nuclei RISC-V Core API provided via NMSIS-Core
NMSIS-NN NMSIS-DSP
Various APIs in SoC Firmware Package
Nuclei FPGA Evaluation Kit
GD32VF103 RV-STAR Kit
GD32VF103 Evaluation Kit
Sipeed Longan Nano
TTGO T-Display-GD32
Nuclei RISC-V Processor Based Boards

有用的在线手册
SDK使用:https://doc.nucleisys.com/nuclei_sdk/index.html
ISA说明: https://doc.nucleisys.com/nuclei_spec/#

验证环境(Nuclei SDK)

bare metal

demo名	验证点
helloworld	简单运行，UART测试
demo_timer	timer Interrupt and timer software interrupt.
demo_eclic	timer interrupt and timer software interrupt
demo_nice	NICE自定义指令支持
coremark dhrystone whetstone	benchmark性能测试

注意：baremetal ELF 与 ELF区别
另：SPI IIC 测试case不包含，后续提供

RTOS

demo名	验证点
FreeRTOS	2 tasks创建和切换
UCOSII	4 tasks创建调度
RT-Thread	5 test threads调度

```
编译示例: make SOC=demosoc \
               BOARD=nuclei_fpga_eval \
               CORE=n307 \
               DOWNLOAD=ilm
```

Tag	VAL	描述
SOC	gd32vf103	gd32vf103v系列
	demosoc	200T系列
BOARD	nuclei_fpga_eval	200T系列
	gd32vf103v_rvstar	gd32vf103v系列
	gd32vf103v_eval	gd32vf103v系列
DOWNLOAD	ilm	ilm/ram
	flash	ilm/ram
	flashxip	flash
	ddr	ddr

```
gcc_demosoc_ddr.ld
gcc_demosoc_flash.ld
gcc_demosoc_flashxip.ld
gcc_demosoc_ilm.ld
```

可切换download测试内存分配

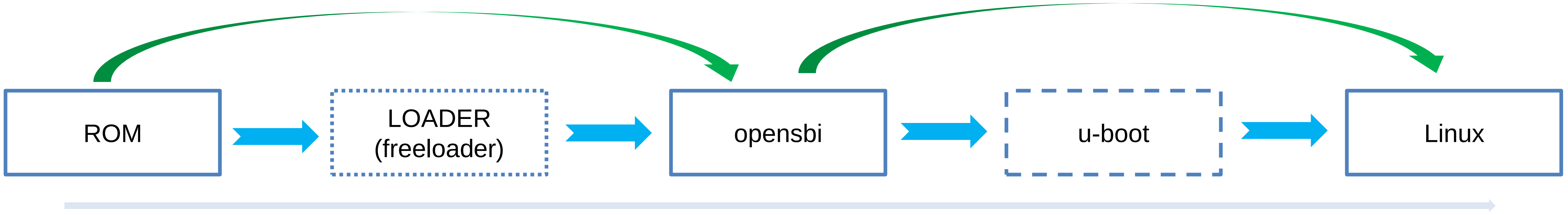
测试过程：GDB，反汇编，源码阅读定位问题

源码: https://github.com/Nuclei-Software/nuclei-sdk

验证环境(Nuclei Linux SDK)

包含组件	主要用途
freeloader	opensbi u-boot搬运
opensbi	初始化设置
u-boot	初始化设置/引导OS
linux	OS初始化并挂载文件系统
buildroot	生成文件系统

目前支持外设：
Nuclei UART/USART
Nuclei SPI
Nuclei GPIO
目前支持CPU：
ux600 and ux900
ux600fd and ux900fd



opensbi模式:https://github.com/riscv/opensbi/blob/master/docs/platform/qemu_virt.md 源码: https://github.com/Nuclei-Software/nuclei-linux-sdk.git

QOM机制

The QEMU Object Model (QOM)

The QEMU Object Model provides a framework for registering user creatable types and instantiating objects from those types. QOM provides the following features:

Features:
- System for dynamically registering types
- Support for single-inheritance of types
- Multiple inheritance of stateless interfaces

```
class MyClass {
public:
    int a;
    void set_A(int a);
}
```

⇩

```
struct MyClass {
    int a;
    void (*set_A)(MyClass *this, int a);
};
```

OOP IN C
- ✓ 封装
- ✓ 继承
- ✓ 多态

编程语言X
设计方法/设计思想√

QOM描述: https://qemu.readthedocs.io/en/latest/devel/qom.html

OOP阅读: https://planetpdf.com/codecuts/pdfs/ooc.pdf

QEMU Object Model (QOM)

```c
static void my_device_class_init(ObjectClass *oc, void *data)
{
}

static void my_device_init(Object *obj)
{
}
```

```c
typedef struct MyDevice
{
    DeviceState parent;
    int reg0, reg1, reg2;
} MyDevice;
```

```c
typedef struct MyDeviceClass
{
    DeviceClass parent;
    void (*init) (MyDevice *obj);
} MyDeviceClass;
```

⇒

```c
#define TYPE_MY_DEVICE "my-device"
static const TypeInfo my_device_info = {
    .parent = TYPE_DEVICE,
    .name = TYPE_MY_DEVICE,
    .instance_size = sizeof(MyDevice),
    .instance_init = my_device_init,
    .instance_finalize = my_device_finalize,
    .class_size = sizeof(MyDeviceClass),
    .class_init = my_device_class_init,
};
```

```c
static void my_device_register_types(void)
{
    type_register_static(&my_device_info);
}
type_init(my_device_register_types)
```

等同于

```c
DEFINE_TYPES(my_device_info)
```

来自: https://qemu.readthedocs.io/en/latest/devel/qom.html

QEMU Object Model (QOM)

```
static void my_device_class_init(ObjectClass *oc, void *data)
{
}

static void my_device_init(Object *obj)
{
}
```

函数构造

```
typedef struct MyDevice
{
    DeviceState parent;    继承
    int reg0, reg1, reg2; 私有属性
} MyDevice;
```

属性封装

```
typedef struct MyDeviceClass
{
    DeviceClass parent;  继承
    void (*init) (MyDevice *obj);
} MyDeviceClass;         私有属性
```

构造函数

```
#define TYPE_MY_DEVICE "my-device"  自定义类型名
static const TypeInfo my_device_info = {
    .parent = TYPE_DEVICE,
    .name = TYPE_MY_DEVICE,
    .instance_size = sizeof(MyDevice),  TypeInfo定义
    .instance_init = my_device_init,
    .instance_finalize = my_device_finalize,
    .class_size = sizeof(MyDeviceClass),
    .class_init = my_device_class_init,
};
```

```
static void my_device_register_types(void)
{
    type_register_static(&my_device_info);
}
type_init(my_device_register_types)
```

等同于

```
DEFINE_TYPES(my_device_infos)
```

类型注册

类型调用?

关注类型:
TypeInfo
DeviceState 与 DeviceClass

Object 与 ObjectClass

Object

关注过程:
类型注册?
DEFINE_TYPES(my_device_info)
与
`type_init`
`type_register_static`
类型调用? 何时 and 如何

来自: https://qemu.readthedocs.io/en/latest/devel/qom.html

QEMU Object Model (QOM)之类型

```
typedef struct MyDevice
{
    DeviceState parent;   继承
    int reg0, reg1, reg2; 私有属性
} MyDevice;
```

属性封装

```
typedef struct MyDeviceClass
{
    DeviceClass parent;  继承
    void (*init) (MyDevice *obj);
} MyDeviceClass;        私有属性
```

构造函数

关注类型:
DeviceState 与 DeviceClass

↓

Object 与 ObjectClass

```
struct DeviceState {
    /*< private >*/
    Object parent_obj;
    /*< public >*/

    const char *id;
    char *canonical_path;
    bool realized;
    bool pending_deleted_event;
    QemuOpts *opts;
    int hotplugged;
    bool allow_unplug_during_migration;
    BusState *parent_bus;
    QLIST_HEAD(, NamedGPIOList) gpios;
    QLIST_HEAD(, NamedClockList) clocks;
    QLIST_HEAD(, BusState) child_bus;
    int num_child_bus;
    int instance_id_alias;
    int alias_required_for_version;
    ResettableState reset;
};
```

```
struct DeviceClass {
    /*< private >*/
    ObjectClass parent_class;
    /*< public >*/

    DECLARE_BITMAP(categories, DEVICE_CATEGORY_MAX);
    const char *fw_name;
    const char *desc;

    Property *props_;

    bool user_creatable;
    bool hotpluggable;

    /* callbacks */
     */
    DeviceReset reset;
    DeviceRealize realize;
    DeviceUnrealize unrealize;

    /* device state */
    const VMStateDescription *vmsd;

    /* Private to qdev / bus.  */
    const char *bus_type;
};
```

QEMU Object Model (QOM)之类型

```
#define TYPE_MY_DEVICE "my-device"  自定义类型名
static const TypeInfo my_device_info = {
    .parent = TYPE_DEVICE,
    .name = TYPE_MY_DEVICE,
    .instance_size = sizeof(MyDevice), TypeInfo定义
    .instance_init = my_device_init,
    .instance_finalize = my_device_finalize,
    .class_size = sizeof(MyDeviceClass),
    .class_init = my_device_class_init,
};
```

关注类型：
TypeInfo

```
/**
 * struct InterfaceInfo:
 * @type: The name of the interface.
 */
struct InterfaceInfo {
    const char *type;
};
```

信息描述:

```
struct TypeInfo
{
    const char *name; //类型名称
    const char *parent; //父类型名称

    size_t instance_size; //对象实例的大小
    size_t instance_align;
    void (*instance_init)(Object *obj); //对象实例的init函数
    void (*instance_post_init)(Object *obj); //对象实例的post_init函数, 在init执行完后执行
    void (*instance_finalize)(Object *obj); //对象实例销毁时执行, 释放动态资源

    bool abstract; //如为true则是抽象的类型, 不能直接实例
    size_t class_size;

    void (*class_init)(ObjectClass *klass, void *data);
    //类对象实例的init函数, 初始化自己的方法指针, 也可覆盖父类的方法指针
    void (*class_base_init)(ObjectClass *klass, void *data);
    //在父类的class_init执行完, 自己的class_init执行之前执行, 做一些清理工作。
    void *class_data;

    InterfaceInfo *interfaces; //类型定义的接口信息名称数组
};
```

静态定义 or 动态生成

QEMU Object Model (QOM)之注册

关注过程：
类型注册？
DEFINE_TYPES(my_device_info)
与
type_init
type_register_static

```
static void my_device_register_types(void)
{
    type_register_static(&my_device_info);
}
type_init(my_device_register_types)
```

```
DEFINE_TYPES(my_device_infos)
```

类型注册

```
#define DEFINE_TYPES(type_array)                                        \
static void do_qemu_init_ ## type_array(void)                           \
{                                                                       \
    type_register_static_array(type_array, ARRAY_SIZE(type_array));     \
}                                                                       \
type_init(do_qemu_init_ ## type_array)
```

C run-time(CRT)

注册：

type_init(function)
module_init(function, type)
register_module_init(function, type);
QTAILQ_INSERT_TAIL(l, e, node);

```
typedef enum {
    MODULE_INIT_MIGRATION,
    MODULE_INIT_BLOCK,
    MODULE_INIT_OPTS,
    MODULE_INIT_QOM,
    MODULE_INIT_TRACE,
    ……
    MODULE_INIT_MAX
} module_init_type;
```

全局变量

```
static ModuleTypeList init_type_list[MODULE_INIT_MAX];
```

```
typedef struct ModuleEntry
{
    void (*init)(void);
    QTAILQ_ENTRY(ModuleEntry) node;
    module_init_type type;
} ModuleEntry;
```

$$\begin{tikzcd}\text{TypeInfo} \arrow[r] & \text{type\_init} \arrow[r] & \text{ModuleEntry} \arrow[r] & \text{ModuleTypeList}\end{tikzcd}$$

QEMU Object Model (QOM)之调用注册

main入口：

```
void module_call_init(module_init_type type);
```

MODULE_INIT_QOM

```
void module_call_init(module_init_type type)
{
    ModuleTypeList *l;
    ModuleEntry *e;

    if (modules_init_done[type]) {
        return;
    }

    l = find_type(type);

    QTAILQ_FOREACH(e, l, node) {
        e->init();
    }

    modules_init_done[type] = true;
}
```

```
TypeImpl *type_register_static(const TypeInfo *info)
```

$$\begin{tikzcd}\boxed{\texttt{TypeInfo}} \arrow[r] & \boxed{\texttt{type\_new}} \arrow[r] & \boxed{\texttt{TypeImpl}}\end{tikzcd}$$

```
struct TypeImpl
{
    const char *name;

    size_t class_size;

    size_t instance_size;
    size_t instance_align;

    void (*class_init)(ObjectClass *klass, void *data);
    void (*class_base_init)(ObjectClass *klass, void *data);

    void *class_data;

    void (*instance_init)(Object *obj);
    void (*instance_post_init)(Object *obj);
    void (*instance_finalize)(Object *obj);

    bool abstract;

    const char *parent;
    TypeImpl *parent_type;

    ObjectClass *class;

    int num_interfaces;
    InterfaceImpl interfaces[MAX_INTERFACES];
};
```

GLIB GHashTable	
KEY	VAL
Name	TypeImpl

QEMU Object Model (QOM)之类型基础

```
//The base for all objects
struct Object
{
    /* private: */
    ObjectClass *class; //指向类对象
    ObjectFree *free;
    GHashTable *properties;
    uint32_t ref;
    Object *parent;
};
```

是否注册？

关注类型:
TypeInfo
DeviceState 与 DeviceClass

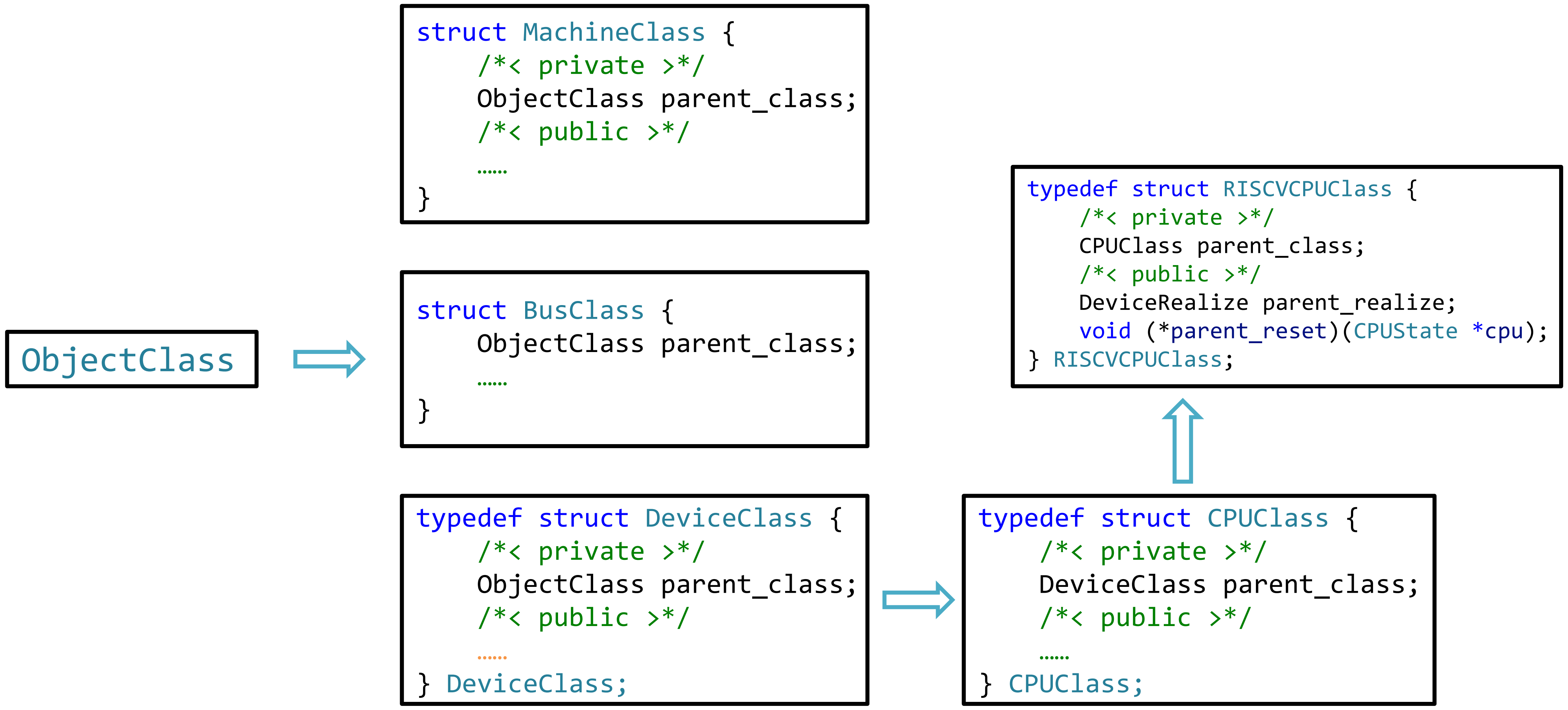
Object 与 ObjectClass

```
//The base for all classes.
struct ObjectClass
{
    /* private: */
    Type type;
    GSList *interfaces;

    const char *object_cast_cache[OBJECT_CLASS_CAST_CACHE];
    const char *class_cast_cache[OBJECT_CLASS_CAST_CACHE];

    ObjectUnparent *unparent;

    GHashTable *properties;
};
```

```
static void register_types(void)
{
    static TypeInfo interface_info = {
        .name = TYPE_INTERFACE,
        .class_size = sizeof(InterfaceClass),
        .abstract = true,
    };

    static TypeInfo object_info = {
        .name = TYPE_OBJECT,
        .instance_size = sizeof(Object),
        .class_init = object_class_init,
        .abstract = true,
    };

    type_interface = type_register_internal(&interface_info);
    type_register_internal(&object_info);
}

type_init(register_types)
```

parent的根
- TYPE_OBJECT
- TYPE_INTERFACE

QEMU Object Model (QOM)之类对象



```
ObjectClass
```

```
struct MachineClass {
    /*< private >*/
    ObjectClass parent_class;
    /*< public >*/
    ……
}
```

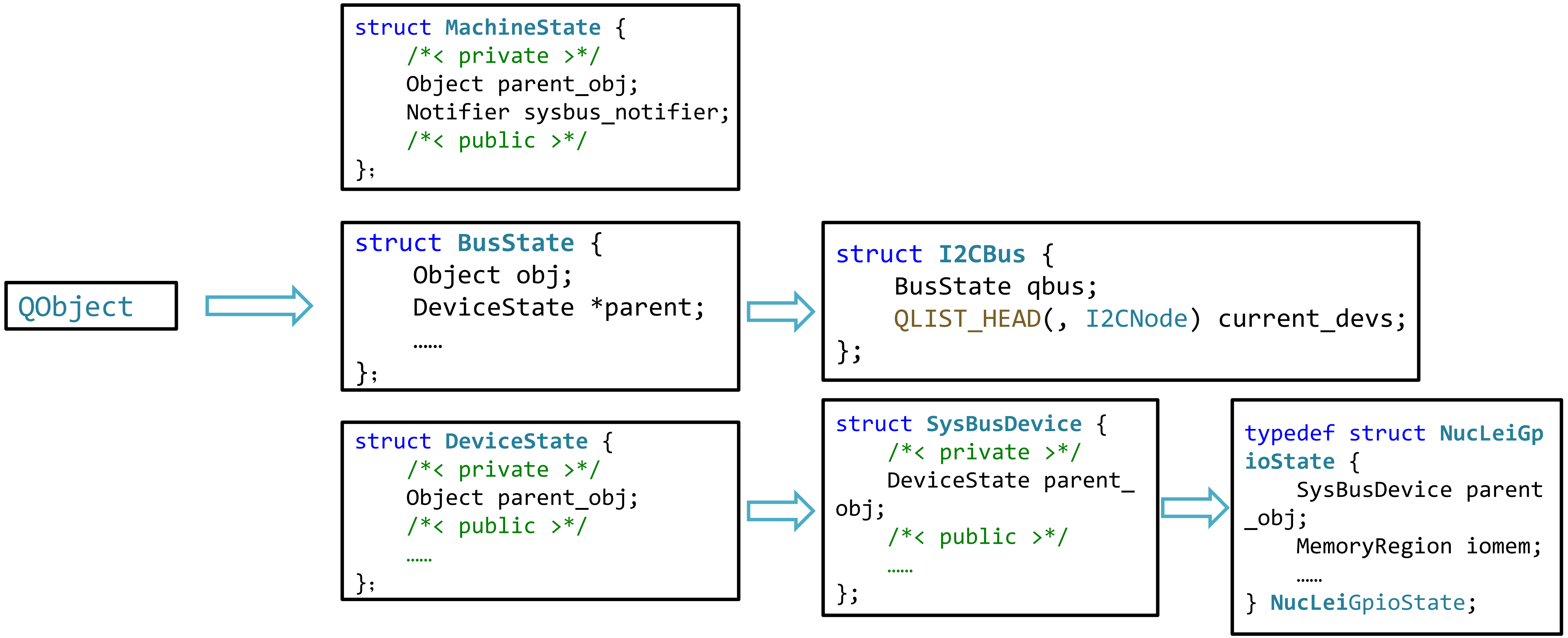
```
struct BusClass {
    ObjectClass parent_class;
    ……
}
```

```
typedef struct DeviceClass {
    /*< private >*/
    ObjectClass parent_class;
    /*< public >*/
    ……
} DeviceClass;
```

```
typedef struct CPUClass {
    /*< private >*/
    DeviceClass parent_class;
    /*< public >*/
    ……
} CPUClass;
```

```
typedef struct RISCVCPUClass {
    /*< private >*/
    CPUClass parent_class;
    /*< public >*/
    DeviceRealize parent_realize;
    void (*parent_reset)(CPUState *cpu);
} RISCVCPUClass;
```

QEMU Object Model (QOM)之类实例对象



```
QObject
```

```
struct MachineState {
    /*< private >*/
    Object parent_obj;
    Notifier sysbus_notifier;
    /*< public >*/
};
```

```
struct BusState {
    Object obj;
    DeviceState *parent;
    ……
};
```

```
struct I2CBus {
    BusState qbus;
    QLIST_HEAD(, I2CNode) current_devs;
};
```

```
struct DeviceState {
    /*< private >*/
    Object parent_obj;
    /*< public >*/
    ……
};
```

```
struct SysBusDevice {
    /*< private >*/
    DeviceState parent_
obj;
    /*< public >*/
    ……
};
```

```
typedef struct NucLeiGp
ioState {
    SysBusDevice parent
_obj;
    MemoryRegion iomem;
    ……
} NucLeiGpioState;
```

QEMU Object Model (QOM)之TYPE/NAME

```
parent的根为TYPE_OBJECT和TYPE_INTERFACE

TYPE_OBJECT ->TYPE_IOTHREAD
              TYPE_ACCEL
              TYPE_CHARDEV
              TYPE_CRYPTODEV_BACKEND
              TYPE_RNG_BACKEND
              TYPE_CHARDEV
              TYPE_BUS -> TYPE_SYSTEM_BUS
                          TYPE_I2C_BUS
              TYPE_IRQ
              TYPE_MACHINE-> 各种RISC-V boards

              TYPE_DEVICE -> TYPE_I2C_SLAVE
                             TYPE_RISCV_U_SOC

              TYPE_CAN_BUS
              TYPE_CAN_HOST
```

Class调用：
- ➢ object_class_by_name
- ➢ object_class_get_parent
- ➢ object_new_with_type
- ➢ object_initialize_with_type

```
type_initialize(type);
```

```
g_malloc0
type_get_parent
parent->class_base_init
class_init
```

Instance调用：
- ➢ object_init_with_type
- ➢ object_post_init_with_type

```
instance_init
instance_post_init
```

QEMU Object Model (QOM)

主要结构体：

- Object(**The base for all objects**)
- ObjectClass(**The base for all classes**)
- TypeInfo
- TypeImpl
- InterfaceInfo
- InterfaceClass

原始版本:
TypeInfo 创建了 TypeImpl
TypeImpl 创建了对应的 ObjectClass
TypeImpl 创建了对应的 Object
ObjectClass 带有属性
Object 带有属性

实际版本:
TypeInfo 创建了 TypeImpl
TypeImpl 创建了对应的 派生Class，父Class，ObjectClass
TypeImpl 创建了对应的 派生Object，父，Object
派生Class 带有属性和接口
派生Object 带有属性

初始化Machine

第一步：定义设备

第二步：SOC设备注册

第三步: Machine 设备注册

第四步: 修改编译文件

第五步: 运行

```
#define TYPE_NUCLEI_HBIRD_SOC "riscv.nuclei.hbird.soc"
#define RISCV_NUCLEI_HBIRD_SOC(obj) \
    OBJECT_CHECK(NucleiHBSoCState, (obj), TYPE_NUCLEI_HBIRD_SO
C)

typedef struct NucleiHBSoCState
{
    /*< private >*/
    SysBusDevice parent_obj;
    /*< public >*/

} NucleiHBSoCState;
```

SOC state定义

```
#define TYPE_HBIRD_FPGA_MACHINE MACHINE_TYPE_NAME("hbird_fpga")
#define HBIRD_FPGA_MACHINE(obj) \
    OBJECT_CHECK(NucleiHBState, (obj), TYPE_HBIRD_FPGA_MACHINE)

typedef struct
{
    /*< private >*/
    SysBusDevice parent_obj;

    /*< public >*/
    NucleiHBSoCState soc;

} NucleiHBState;
```

Machine state定义

空Machine创建

第一步:
定义设备

**第二步:
SOC设备注册**

第三步:
Machine
设备注册

第四步:
修改编译文件

第五步:
运行

```
static void nuclei_soc_init(Object *obj)
{
    qemu_log(">>nuclei_soc_init \n");
}

static void nuclei_soc_realize(DeviceState *dev, Error **errp)
{
    qemu_log(">>nuclei_soc_realize \n");
}

static void nuclei_soc_class_init(ObjectClass *oc, void *data)
{
    qemu_log(">>nuclei_soc_class_init \n");
    DeviceClass *dc = DEVICE_CLASS(oc);
    dc->realize = nuclei_soc_realize;
    dc->user_creatable = false;
}
```

```
static const TypeInfo nuclei_soc_type_info = {
    .name = TYPE_NUCLEI_HBIRD_SOC,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(NucleiHBSoCState),
    .instance_init = nuclei_soc_init,
    .class_init = nuclei_soc_class_init,
};

static void nuclei_soc_register_types(void)
{
    type_register_static(&nuclei_soc_type_info);
}

type_init(nuclei_soc_register_types)
```

空Machine创建

第一步:
定义设备

第二步:
SOC设备注册

第三步:
Machine
设备注册

第四步:
修改编译文件

第五步:
运行

```
static void nuclei_board_init(MachineState *machine)
{
    NucleiHBState *s = HBIRD_FPGA_MACHINE(machine);
    qemu_log(">>nuclei_board_init \n");
    /* Initialize SOC */
    object_initialize_child(OBJECT(machine), "soc", &s->soc, TYPE_NUCLEI_HBIRD_SOC);
    qdev_realize(DEVICE(&s->soc), NULL, &error_abort);
}

static void nuclei_machine_instance_init(Object *obj)
{
    qemu_log(">>nuclei_machine_instance_init \n");
}

static void nuclei_machine_class_init(ObjectClass *oc, void *data)
{
    qemu_log(">>nuclei_machine_class_init \n");
    MachineClass *mc = MACHINE_CLASS(oc);
    mc->desc = "Nuclei HummingBird Evaluation Kit";
    mc->init = nuclei_board_init;
}
```

```
static const TypeInfo nuclei_machine_typeinfo = {
    .name = MACHINE_TYPE_NAME("hbird_fpga"),
    .parent = TYPE_MACHINE,
    .class_init = nuclei_machine_class_init,
    .instance_init = nuclei_machine_instance_init,
    .instance_size = sizeof(NucleiHBState),
};

static void nuclei_machine_init_register_types(void)
{
    type_register_static(&nuclei_machine_typeinfo);
}

type_init(nuclei_machine_init_register_types)
```

空Machine创建

第一步：
定义设备

第二步：
SOC设备注册

第三步:
Machine
设备注册

第四步:
修改编译文件

第五步:
运行

配置编译

hw/riscv/Kconfig

```
config NUCLEI_N
    bool
    select MSI_NONBROKEN
    select UNIMP
```

hw/riscv/meson.build

```
riscv_ss = ss.source_set()
riscv_ss.add(files('boot.c'), fdt)
riscv_ss.add(files('numa.c'))
riscv_ss.add(files('riscv_hart.c'))
……
riscv_ss.add(when: 'CONFIG_NUCLEI_N', if_true: files('nuclei_n.c'))
riscv_ss.add(when: 'CONFIG_NUCLEI_U', if_true: files('nuclei_u.c'))

hw_arch += {'riscv': riscv_ss}
```

编译参数：

```
./configure --target-list=riscv32-softmmu,riscv64-softmmu,riscv32-linux-user,riscv64-linux-user
make -j $(nproc)
```

空Machine创建

第一步:
定义设备

第二步:
SOC设备注册

第三步:
Machine
设备注册

第四步:
修改编译文件

**第五步:
运行**

运行测试

```
$ ./build/qemu-system-riscv32 -M ?
>>nuclei_soc_class_init
>>nuclei_machine_class_init
Supported machines are:
hbird_fpga           Nuclei HummingBird Evaluation Kit
none                 empty machine
opentitan            RISC-V Board compatible with OpenTitan
sifive_e             RISC-V Board compatible with SiFive E SDK
sifive_u             RISC-V Board compatible with SiFive U SDK
spike                RISC-V Spike board (default)
virt                 RISC-V VirtIO board
$ ./build/qemu-system-riscv64 -M ?
>>nuclei_soc_class_init
>>nuclei_machine_class_init
Supported machines are:
hbird_fpga           Nuclei HummingBird Evaluation Kit
microchip-icicle-kit Microchip PolarFire SoC Icicle Kit
none                 empty machine
nuclei_u             RISC-V NUCLEI U Board
sifive_e             RISC-V Board compatible with SiFive E SDK
sifive_u             RISC-V Board compatible with SiFive U SDK
spike                RISC-V Spike board (default)
virt                 RISC-V VirtIO board
```

```
$ ./build/qemu-system-riscv64 -nographic -M hbird_fpga
>>nuclei_soc_class_init
>>nuclei_machine_class_init
>>nuclei_machine_instance_init
>>nuclei_board_init
>>nuclei_soc_init
>>nuclei_soc_realize
QEMU 5.2.90 monitor - type 'help' for more information
(qemu) info qom-tree
/machine (hbird_fpga-machine)
  /peripheral (container)
  /peripheral-anon (container)
  /soc (riscv.nuclei.hbird.soc)
  /unattached (container)
    /io[0] (memory-region)
    /sysbus (System)
    /system[0] (memory-region)
```

Machine创建过程

```
main()
    qemu_init
    qemu_main_loop
    qemu_cleanup
```

```
qemu_add_opts(&qemu_machine_opts);
module_call_init(MODULE_INIT_OPTS);
qemu_init_subsystems();
    module_call_init(MODULE_INIT_QOM);
```

```
qemu_create_machine(select_machine());
    object_class_get_list
    find_default_machine
    qemu_opt_get(opts, "type");
    machine_parse(optarg, machines)
```

```
>>nuclei_soc_class_init
>>nuclei_machine_class_init
```

```
object_new_with_class
    type_initialize  ->  ti->class_init
    object_initialize_with_type
    object_init_with_type
    ti->instance_init
```

```
>>nuclei_machine_instance_init
```

```
qemu_init_board
    machine_run_board_init(current_machine)
        machine_class->init(machine);
```

```
>>nuclei_board_init
```

```
object_initialize_child
qdev_realize
```

```
ti->instance_init
nuclei_soc_realize
```

```
>>nuclei_soc_init
>>nuclei_soc_realize
```

打印流程：

```
>>nuclei_soc_class_init
>>nuclei_machine_class_init
>>nuclei_machine_instance_init
>>nuclei_board_init
>>nuclei_soc_init
>>nuclei_soc_realize
```

过程简析

下节课内容：

CPU虚拟化

- RISCV CPU实现分析
- 新RISCV CPU建立
- CSR寄存器实现添加
- TCG原理与指令翻译
- DecodeTree与指令添加

谢谢

wangjunqiang@iscas.ac.cn