

ISA (Instruction Set Architecture)

ISA指令集架构：是底层硬件电路面向上层软件程序提供的一层接口规范

可以把他看做一种标准

ISA之下是微架构（Microarchitecture），ISA之上是操作系统（Operating System），微架构对不同公司来说不同，它关心的是制造方面的问题（硬件问题），正是由于微架构的不同造就了CPU的不同（比如X86下不同的CPU性能的不同）

ISA定义了：

- 基本数据类型（BYTE/HALFWORD/WORD/.....）
- 寄存器（Register）
- 指令
- 寻址模式
- 异常或者中断的处理方式
- 等等...

CISC（Complex Instruction Set Computing）复杂指令集：针对特定的功能实现特定的长度，导致指令数目比较多，但生成的程序长度相对较短。

RISC（Reduced Instruction Set Computing）精简指令集：只定义常用指令，对复杂的功能采用常用指令组合实现，这导致指令数目比较精简，但生成的程序长度相对较长。

ISA的宽度指的是CPU中通用寄存器的宽度（二进制的位数），这决定了寻址范围的大小、以及数据运算的能力。

通用寄存器的宽度	寻址范围	应用场景
8位	$2^8 = 256$	早期的单片机 8051
16位	$2^{16} = 65536$	X86系列的鼻祖 8086, MSP430系列单片机
32位	$2^{32} = 4294967296$	早期的终端, 个人计算机和服务器
64位	$2^{64} = 18446744073709551616$	目前主流的移动智能终端, 个人计算机和服务器

但是注意ISA的宽度和指令编码长度无关。

RISC-V ISA

特点：

- 简单
- 清晰的分层设计
- 模块化
- 稳定
- 社区化

ISA命名格式：

RV[###][abc....xyz]

- RV：用于标识RISC-V体系架构的前缀
- [###]：{32,64,128}用于标识处理器的字宽
- [abc...xyz]：标识该处理器支持的指令集模块集合

例如：RV32IMA，RV64GC

增量ISA：计算机体系结构的传统方法，同一体系架构下的新一代处理器不仅实现了新的ISA扩展，还必须实现过去的所有扩展，目的是为了保持向后的二进制兼容性。典型的就是x86

模块化ISA：由1个基本整数（Integer）指令集+多个可选的扩展指令集组成。基础指令集是固定的，永远不会改变。

基本整数（Integer）指令集

基本指令集	描述
RV32I	32位整数指令集
RV32E	RV32I的子集，用于小型的嵌入式场景
RV64I	64位整数指令集，兼容RV32I
RV128I	128位整数指令集，兼容RV64I和RV32I

（embeded：嵌入式）

扩展模块指令集：

扩展指令集	描述
M	整数乘法（Multiplication）与除法指令集
A	存储器原子（Atomic）指令集
F	单精度（32bit）浮点（Float）指令集
D	双精度（64bit）浮点（Double）指令，兼容F
C	压缩（Compressed）指令集
.....	其他标准化和为标准化的指令集

- 特定组合"IMAFD"被称为“通用（General）”组合，用英文字母G表示。

通用寄存器（General Purpose Registers）

一、RISC-V的Unprivileged Specification定义了32个通用寄存器以及一个PC

注：RISC-V的PC寄存器是无法被访问的，是被屏蔽的，但是可以用特殊的方式去读

- 对RV32I/RV64I/RV128I都是一样的
- 如果实现支持F/D扩展则需要额外支持32个浮点（Float Point）寄存器
- RV32E将32个通用寄存器缩减为16个

二、寄存器的宽度由ISA指定

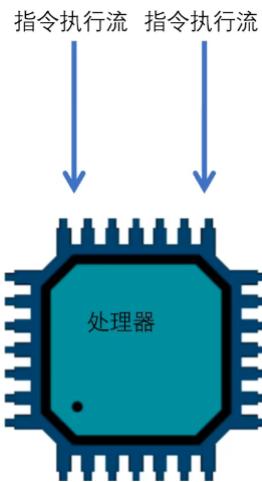
RV32的寄存器宽度为32位，依次类推

三、每个寄存器具体编程是有特定的用途以及各自的别名。由RISC-V Application Binary Interface（ABI）定义

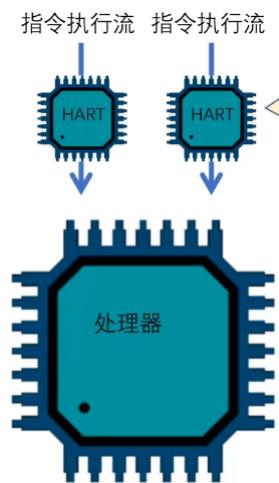
Hart

HART = HARDware Thread（硬件线程）

一个CPU提供两个指令执行流



一个HART便是一个执行流或者说是一个虚拟的CPU



所以以后可以不提处理器的概念

特权级别 (Privileged Level)

- RISC-V的Privileged Specification定义了三个特权级别 (privilege level)
- Machine级别是最高的级别，所有的实现都需要支持，刚上电的CPU运行在Machine中，此时不开放虚拟地址

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	<i>Reserved</i>	
3	11	Machine	M

- 可选的Debug级别

同时通常会有下面三种类型的芯片

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

Control and Status Registers (CSR)

在CPU的角度来说，他在不同的级别下各有一套独立的寄存器，这些不同的寄存器在不同的模式下是不能互相访问的

- 不同的特权级别下分别对应各自的一套Registers (CSR)，用于控制和获取相应Level下的处理器工作状态
- 高级别的特权级别下可以访问低级别的CSR，反之则不可以
- RISC-V定义了专门用于操作CSR的指令（参考1中的"Zicsr"扩展）
- RISC-V定义了特定的指令可以用于在不同特权级别之间切换 (ECALL/EBREAK命令)

内存管理与保护

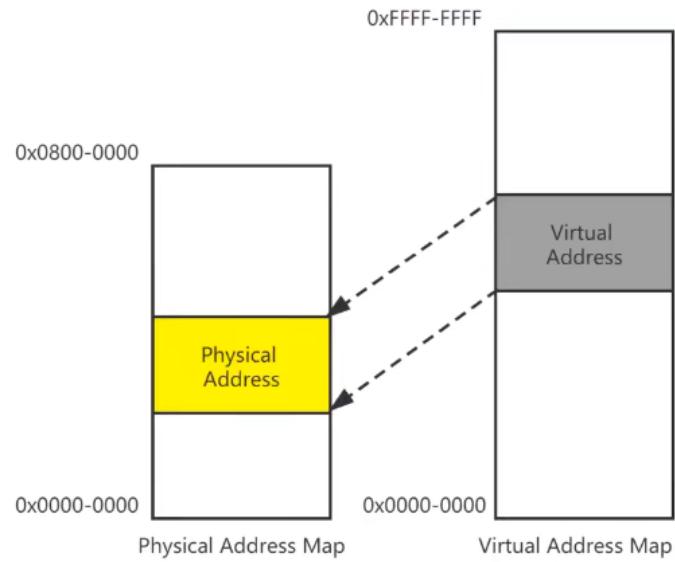
物理内存保护 (Physical Memory Protection, PMP)

- 允许M模式指定U模式可以访问的内存地址
- 支持R/W/X，以及Lock



虚拟内存 (Virtual Memory)

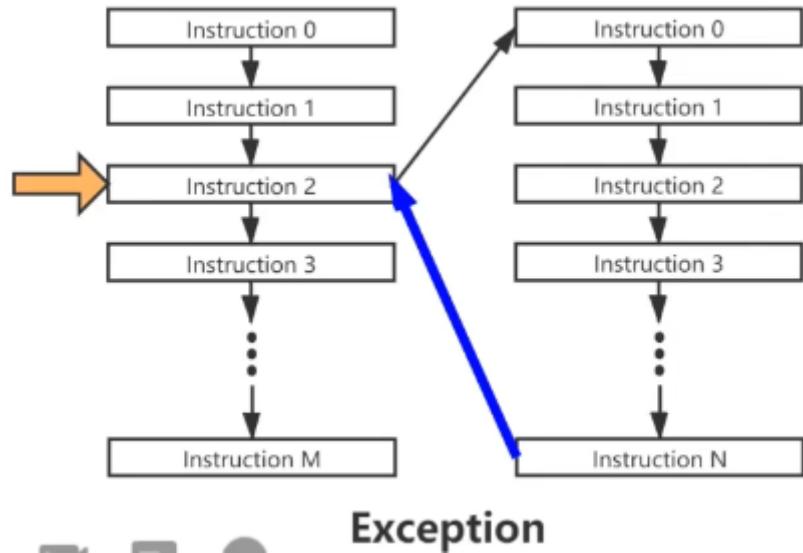
- 需要支持Supervisor Level
- 用于实现高级的操作系统特性 (Unix/Linux)
- 多种映射方式 Sv32/Sv39/Sv48



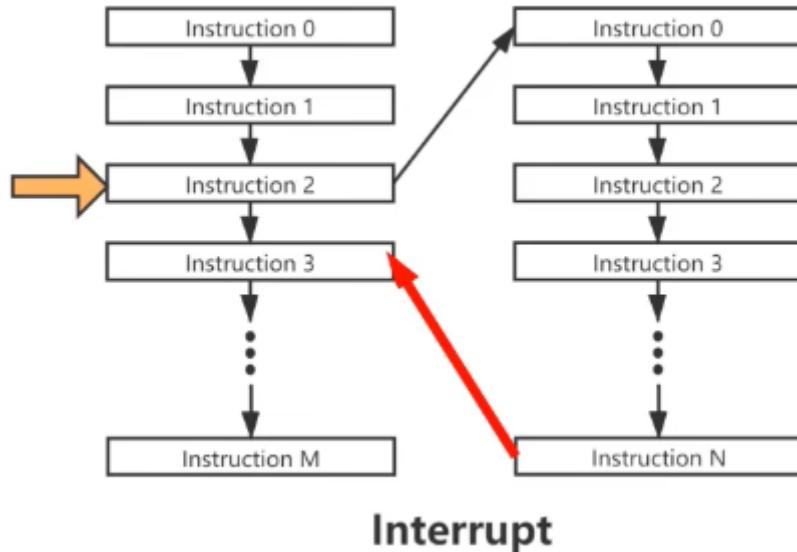
虚实转换是由MMU来实现

异常和中断

异常（Exception）：当指令造成了一下异常的错误时（比如访问了不该访问的地址）此时CPU会停掉当前指令流，跳到一个新的异常处理程序进行处理，然后CPU会回到刚才的异常指令再次执行这条指令



中断（Interrupt）：当执行到这条指令时中断发生，此时转到中断处理程序，处理完后返回到刚才执行指令的下一条指令去执行



编译与链接

GCC (GNU Compiler Collection)

有GNU开发，遵循GPL许可证发行的编译器套件，初衷是为了GNU操作系统专门编写的一款编译器。主要运行在"Unix-like"操作系统

GCC命令格式

gcc [options] [filenames]

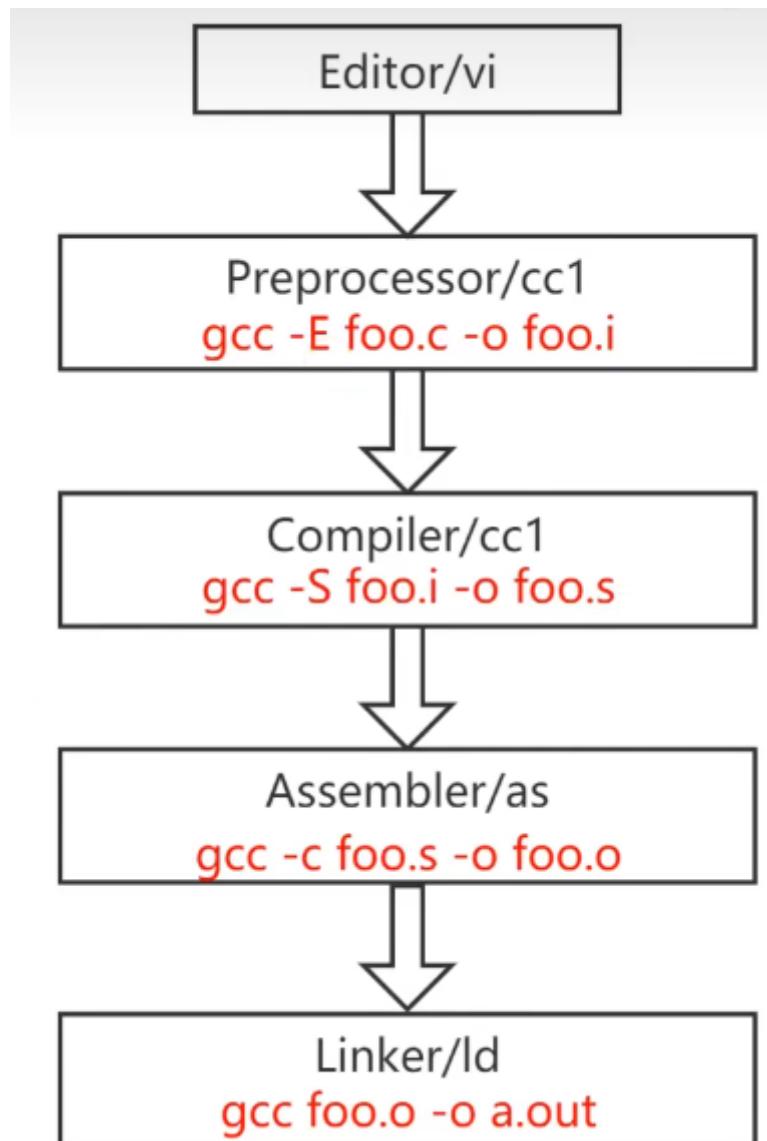
- -E: 只做预处理
- -c: 只编译不链接，生成目标文件".o"
- -S: 生成汇编代码
- -o file: 把输出生成到由file指定文件名的文件中
- -g: 把输出的文件中加入支持调试的信息
- -v: 显示输出详细的命令执行过程信息

GCC的主要执行步骤

编译（cc1，这里针对C语言，不同的语言有自己的编译器）：编译器完成“预处理”和“编译”，“预处理”指处理源文件中以“#”开头的预处理指令，譬如#include,#define等；“编译”则针对预处理的结果进行一系列的词法分析、语法分析、语义分析、优化后生成汇编指令，存放在.o为后缀的目标文件中

汇编 (as)：汇编器将汇编语言代码转换为机器 (CPU) 可以执行的指令

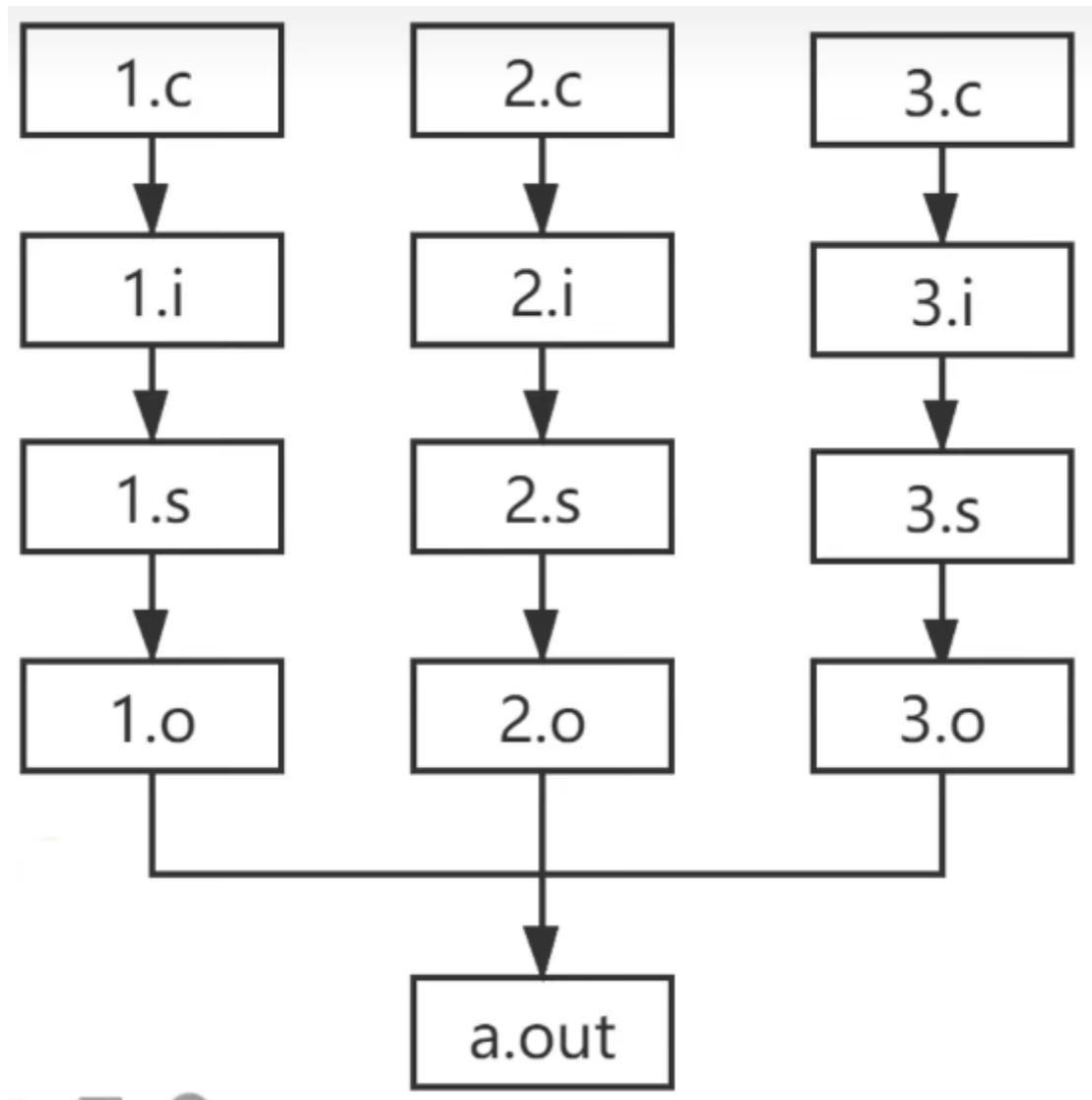
链接 (ld)：链接器将汇编器生成的目标文件和一些标准库（譬如libc）文件组合，形成最终可执行的应用程序



GCC涉及的文件类型

- .c: C源文件
- .cc/.cxx/.cpp: C++源文件
- .i: 经过预处理的C源文件
- .s/.S: 汇编语言源文件（大写S可能会包含预处理语句）
- .h: 头 (header) 文件
- .o: 目标 (object) 文件
- .a/.so: 编译后的静态库 (archive) 文件和共享库 (shared object) 文件
- a.out: 可执行文件

针对多个源文件的处理



对每一个单独的c文件执行这个过程，最后将所有.o文件链接起来

ELF (Executable Linkable Format)

是一种Unix-like系统上的二进制文件格式标准

ELF中文件有四类：

一、可重定位文件 (Relocatable File)

内容包含了代码和数据，可以被链接成可执行文件或共享目标文件。

例如：Linux上的.o文件

二、可执行文件 (Executable File)

可以执行的程序

例如：Linux上的a.out

三、共享目标文件（Shared Object File）

内容包含了代码和数据，可以作为链接器的输入，在链接阶段和其他的Relocatable File或者Shared Object File一起连接成新的Object File；或者在运行阶段，作为动态链接器的输入，和Executable File结合，作为进程的一部分来运行

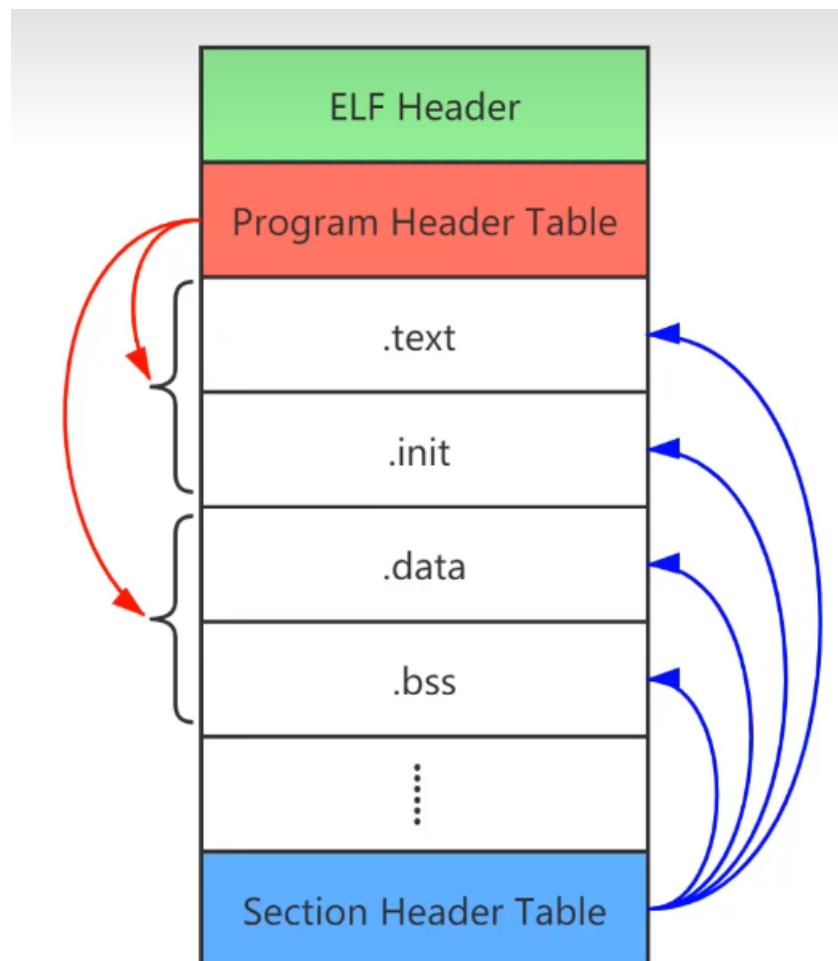
例如：Linux上的.so

四、核心存储文件（Core Dump File）

进程意外终止时，系统可以将该进程的部分内容和终止时的其他状态信息保存在该文件中以供调试分析

例如：Linux上的core文件

ELF文件格式



ELF Header: ELF的基本信息，比如ELF运行在哪些体系架构中，版本号信息等

Program Header Table (运行视图)：将节根据其属性做归并，比如将.text和.init归并成一个段 (**Segment**) 然后将段加载到内存 (为了节省内存) Segment要多大，具体加载到哪里也在运行视图中

白色部分叫做节 (**Section**)

.text: 程序的指令

.init: 初始化的指令

.data: 程序的数据 (比如全局变量)

.bss:

Section Header Table (链接视图)：决定文件中由哪些节，节的含义和节的位置

ELF文件处理相关工具：Binutils

- ar: 归档文件，将多个文件打包成一个大文件
- as: 被gcc调用，输入汇编文件，输出目标文件供链接器ld连接
- ld: GNU链接器。被gcc调用，它把目标文件和各种库文件结合在一起，重定位数据，并链接符号引用
- objcopy: 执行文件格式转换
- objdump: 显示ELF文件的信息
- readelf: 显示更多ELF格式文件的信息 (包括DWARF调试信息)

嵌入式开发

交叉编译

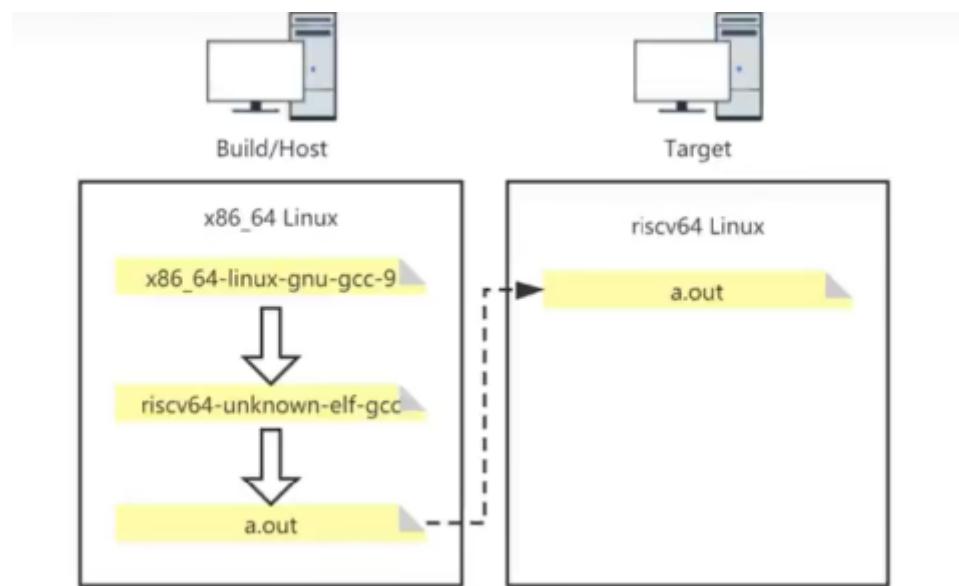
参与编译和运行的机器根据其角色可以分成以下三类：

- 构建 (**build**) 系统：执行编译构建动作的计算机
- 主机 (**host**) 系统：运行build系统生成的可执行程序的计算机系统
- 目标 (**target**) 系统：特别地，当以上生成的可执行程序是GCC时，我们用target来描述用来运行GCC生成的可执行程序的计算机系统

根据build/host/target的不同组合我们可以得到如下的编译方式分类：

- 本地 (**native**) 编译：build == host == target

- 交叉 (cross) 编译: build == host != target



GNU交叉编译工具链 (Toolchain)

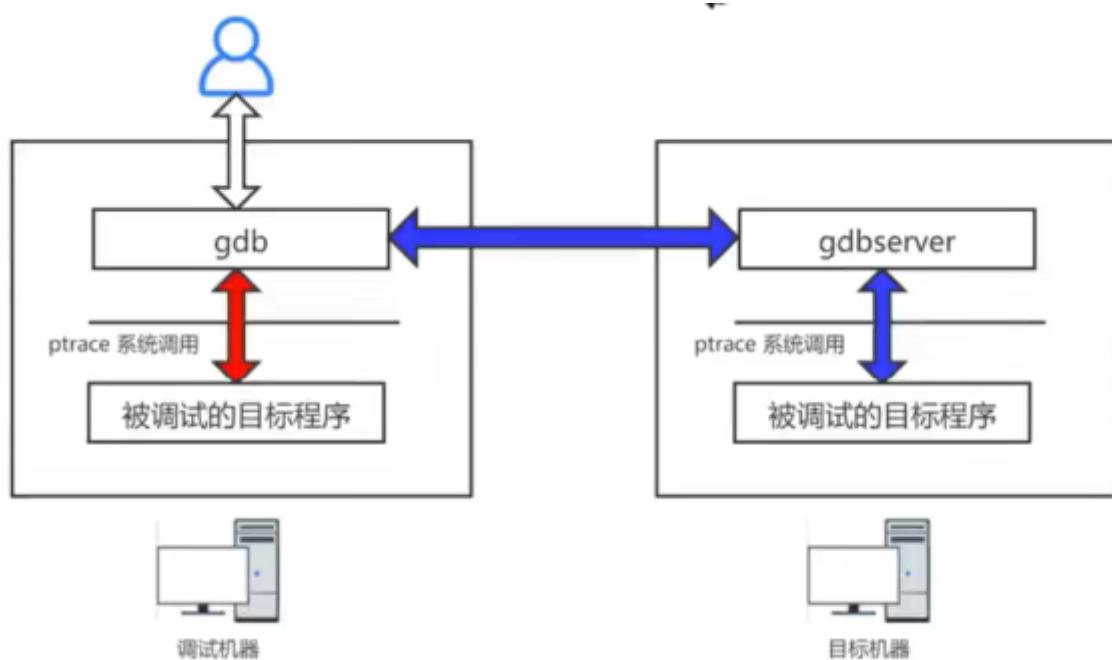
命名格式: arch-vendor-os1-[os2-]xxx

例如:

- x86-64-linux-gnu-gcc
- riscv64-unknown-elf-gcc

调试器 GDB

GDB (The GNU Project Debugger) : GNU项目调试器，用于查看另一个程序在执行过程中正在执行的操作，或该程序崩溃时正在执行的操作



本地调试：GDB会先fork一个子进程运行被调试的目标程序，原进程运行gdb，两个进程之间通过ptrace系统调用进行交互

远程调试：会在目标机上启动一个gdbserver的服务，由这个服务代理gdb行使功能

基本流程：

- 重新编译程序并在编译选项中加入 “-g”
 \$ gcc -g test.c
- 运行 gdb 和程序
 \$ gdb a.out
- 设置断点
 (gdb) b 6
- 运行程序
 (gdb) r
- 程序暂停在断点处，执行查看
 (gdb) p xxx
- 继续、单步或者恢复程序运行
 (gdb) s/n/c

模拟器QEMU

以GPL许可证分发源码的计算机系统模拟软件

主要有两种运作模式：

- User mode：直接运行应用程序
- System mode：模拟整个计算机系统，包括中央处理器及其他周边设备

项目构造工具Make

make是一种自动化工厂管理工具

Makefile是配合make，用于描述构建工程过程中所管理的对象以及如何构造工程的过程

make如何找到Makefile:

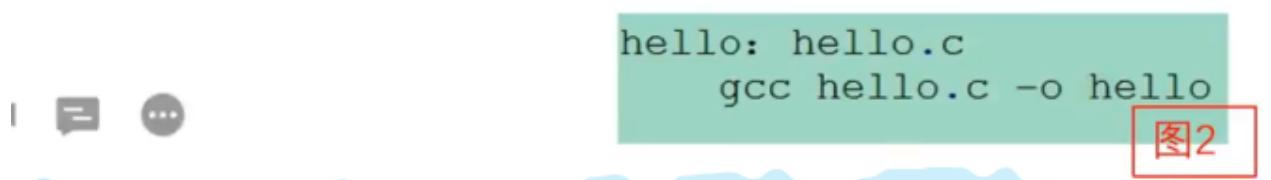
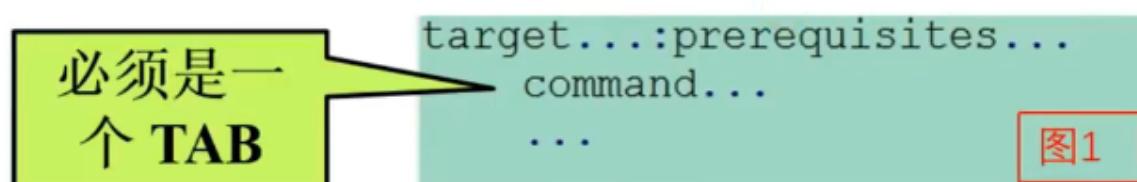
- 隐式查找: 当前目录下按顺序找寻文件名为"GNUmakefile"、 "makefile"、 "Makefile"的文件
- 显式查找: -f

Makefile由一条或者多条规则 (rule) 组成

每条规则由三要素构成:

- target: 目标, 可以是obj文件也可以是可执行文件
- prerequisites: 生成target所需要的依赖
- command: 为了生成target需要执行的命令, 可以有多条

例如:



其它元素:

- 缺省规则: 一个文件中有多个规则时, 不特别指定则从第一个开始执行

```
.DEFAULT_GOAL := all  
all :
```

```
.PHONY : clean
```

- 违规则: clean:

```
rm -f *.o
```

- 注释: 行注释, 以“#”开头

make的运行



RISC-V汇编语言编程

汇编语言（Assembly Language）

汇编缺点：

- 难读
- 难写
- 难移植

汇编优点：

- 灵活
- 强大

应用场景：

- 需要直接访问底层硬件的地方
- 需要对性能执行极致优化的地方

汇编语言语法介绍（GNU版本）

一、一个完整的RISC-V汇编程序有多条语句（statement）组成

二、一条典型的RISC-V汇编语句由3部分组成：

[label:] [operation] [comment]

标签+操作+注释，三部分都是可选的

- **label**（标号）： GNU汇编中，任何以冒号结尾的标识符都被认为是一个标号，
label在汇编里边相当于给地址起了一个名字
- **operation**可以有以下多种类型：
 - **instruction**（指令）：直接对应二进制机器指令的字符串
 - **pseudo-instruction**（伪指令）：为了提高编写代码的效率。可以用一天
伪指令指示汇编器产生多条实际的指令（instruction）
 - **directive**（指示/伪操作）：通过类似指令的形式（以"."开头），通知汇
编器如何控制代码的产生等，不对应具体的指令
 - **macro**：采用.macro/.endm自定义的宏
- **comment**（注释）：常用方式， "#"开始到当前行结束

directive（汇编指示符/指示/伪操作）

指示符	作用
.text	代码段，之后跟的符号都在.text内
.data	数据段，之后跟的符号都在.data内
.bss	未初始化数据段，之后跟的符号都在.bss中
.section .foo	自定义段，之后跟的符号都在.foo段中，.foo段名可以做修改
.align n	按2的n次幂字节对齐
.balign n	按n字节对齐
.globl sym	声明sym为全局符号，其它文件可以访问
.string "str"	将字符串str放入内存
.byte b1,...,bn	在内存中连续存储n个单字节
.half w1,...,wn	在内存中连续存储n个半字(2字节)

指示符	作用
.word w1,...,wn	在内存中连续存储n个字(4字节)
.dword w1,...,wn	在内存中连续存储n个双字(8字节)
.float f1,...,fn	在内存中连续存储n个单精度浮点数
.double d1,...,dn	在内存中连续存储n个双精度浮点数
.option rvc	使用压缩指令(risc-v c)
.option norvc	不压缩指令
.option relax	允许链接器松弛(linker relaxation, 链接时多次扫描代码, 尽可能将跳转两条指令替换为一条)
.option norelax	不允许链接松弛
.option pic	与位置无关代码段
.option nopic	与位置有关代码段
.option push	将所有.option设置存入栈
.option pop	从栈中弹出上次存入的.option设置

补充:

一:

```
1 .rept count
```

重复.rept指令和下一个.endr指令之间的行序列count次。

例如:

```
1 .rept 3
2 .long 0
3 .endr
```

这段代码等于:

```
1 .long 0
2 .long 0
3 .long 0
```

二：

GNU汇编程序中的宏定义

格式如下：

.macro 宏名 参数名列表 @伪指令.macro定义一个宏

宏体

.endm @.endm表示宏结束

如果宏使用参数,那么在宏体中使用该参数时添加前缀“\”。宏定义时的参数还可以使用默认值。可以使用.exitm伪指令来退出宏。

例：宏定义

```
1 .macro SHIFTLEFT a, b
2
3 .if \b < 0
4 MOV \a, \a, ASR #-\b
5 .exitm
6 .endif
7 MOV \a, \a, LSL #\b
8
9 .endm
```

三：

.equ is like #define in C:

```
1 #define bob 10
2
3 .equ bob, 10
```

RISC-V汇编指令总览

RISC-V汇编指令操作对象

一、寄存器

- 32个通用寄存器，x0~x31（注意：本章节课程仅涉及RV32I的通用寄存器组）
- 在RISC-V中，Hart在执行算术逻辑运算时所操作的数据必须直接来自寄存器

XLEN-1	0
x0 / zero	
x1	
x2	
x3	
x4	
x5	
x6	
x7	
x8	
x9	
x10	
x11	
x12	
x13	
x14	
x15	
x16	
x17	
x18	
x19	
x20	
x21	
x22	
x23	
x24	
x25	
x26	
x27	
x28	
x29	
x30	
x31	
XLEN	

XLEN-1	0
pc	
XLEN	

ps：对应规范第14页

注：

- 无法对x0进行写操作，对其读的值永远是0
- pc寄存器无法访问

通用寄存器组功能

寄存器名称	汇编名称	功能描述	调用返回后其值是否会保证不变
x0	zero	零寄存器	未定义
x1	ra	返回地址	否
x2	sp	栈指针	是
x3	gp	全局指针	未定义
x4	tp	线程指针	未定义
x5	t0	临时寄存器, 或者用作替代链接寄存器 (见后续章节详述)	否
x6	t1	临时寄存器	否
x7	t2	临时寄存器	否
x8	s0/fp	该寄存器需要被调函数予以保存, 或也可用作调用栈的帧指针	是
x9	s1	该寄存器需要被调函数予以保存	是
x10~x11	a0~ a1	函数参数或返回值	否
x12~x17	a2~a7	函数参数	否
x18~x27	s2~s11	该寄存器需要被调函数予以保存	是
x28~x31	t3~t6	临时寄存器	否

CSDIP@Cerman

二、内存

- Hart可以执行在寄存器和内存之间的数据读写操作
- 读写操作使用字节（Byte）为基本单位进行寻址
- RV32可以访问最多 2^{32} 个字节（4G）的内存空间

RISC-V汇编指令编码格式

注：下述所有基本指令均在第130页可查

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode				R-type
	imm[11:0]			rs1		funct3		rd		opcode				I-type
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode				S-type
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode				B-type
	imm[31:12]							rd		opcode				U-type
	imm[20 10:1 11 19:12]							rd		opcode				J-type

ps：对应规范第130页

- 指令长度：ILEN1=32bits (RV32I)
- 指令对齐：IALIGN=32bits (RV32I)

- 32个bit划分成不同的“域（field）”
- funct3/funct7和opcode一起决定最终的指令类型
- 指令在内存中按照小端序排列

注：

- funct3、funct7后面的数字表示占了几个bit
- opcode后两位一定是11（也可以说是低两位）
- rd：目标寄存器
- rs：源寄存器

6种指令格式（format）

- R-type (Register)：每条指令中有三个fields，用于指定3个寄存器参数
- I-type (Immediate, 立即数)：每条指令除了带有两个寄存器参数外，还带有一个立即数参数（宽度为12bits）
- S-type (Store)：每条指令除了带有两个寄存器参数外，还带有一个立即数参数（宽度为12bits，但fields的组织方式不同于I-type）
- B-type (Branch)：每条指令除了带有两个寄存器参数外，还带有一个立即数参数（宽度为12bits，但取值为2的倍数）
- U-type (Upper)：每条指令含有一个寄存器参数再加上一个立即数参数（宽度为20bits，用于表示一个立即数的高20位）
- J-type (Jump)：每条指令含有一个寄存器参数再加上一个立即数参数（宽度为20bits）

RISC-V汇编指令分类

	Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
算术运算指令	add	ADD	R	0110011	0x0	0x00	$rd = rs1 + rs2$
	sub	SUB	R	0110011	0x0	0x20	$rd = rs1 - rs2$
	addi	ADD Immediate	I	0010011	0x0		$rd = rs1 + imm$
	lui	Load Upper Imm	U	0110111			$rd = imm << 12$
	auipc	Add Upper Imm to PC	U	0010111			$rd = PC + (imm << 12)$
逻辑运算指令	xor	XOR	R	0110011	0x4	0x00	$rd = rs1 ^ rs2$
	or	OR	R	0110011	0x6	0x00	$rd = rs1 rs2$
	and	AND	R	0110011	0x7	0x00	$rd = rs1 \& rs2$
	xori	XOR Immediate	I	0010011	0x4		$rd = rs1 ^ imm$
	ori	OR Immediate	I	0010011	0x6		$rd = rs1 imm$
	andi	AND Immediate	I	0010011	0x7		$rd = rs1 \& imm$
移位运算指令	sll	Shift Left Logical	R	0110011	0x1	0x00	$rd = rs1 << rs2$
	srl	Shift Right Logical	R	0110011	0x5	0x00	$rd = rs1 >> rs2$
	sra	Shift Right Arith*	R	0110011	0x5	0x20	$rd = rs1 >> rs2$
	slt	Set Less Than	R	0110011	0x2	0x00	$rd = (rs1 < rs2)?1:0$
	sltu	Set Less Than (U)	R	0110011	0x3	0x00	$rd = (rs1 < rs2)?1:0$
	slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	$rd = rs1 << imm[0:4]$
	srlti	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	$rd = rs1 >> imm[0:4]$
	srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	$rd = rs1 >> imm[0:4]$
	slti	Set Less Than Imm	I	0010011	0x2		$rd = (rs1 < imm)?1:0$
	sltiu	Set Less Than Imm (U)	I	0010011	0x3		$rd = (rs1 < imm)?1:0$
内存读写指令	lb	Load Byte	I	0000011	0x0		$rd = M[rs1+imm][0:7]$
	lh	Load Half	I	0000011	0x1		$rd = M[rs1+imm][0:15]$
	lw	Load Word	I	0000011	0x2		$rd = M[rs1+imm][0:31]$
	lbu	Load Byte (U)	I	0000011	0x4		$rd = M[rs1+imm][0:7]$
	lhu	Load Half (U)	I	0000011	0x5		$rd = M[rs1+imm][0:15]$
分支与跳转指令	sb	Store Byte	S	0100011	0x0		$M[rs1+imm][0:7] = rs2[0:7]$
	sh	Store Half	S	0100011	0x1		$M[rs1+imm][0:15] = rs2[0:15]$
	sw	Store Word	S	0100011	0x2		$M[rs1+imm][0:31] = rs2[0:31]$
	beq	Branch ==	B	1100011	0x0		$if(rs1 == rs2) PC += imm$
	bne	Branch !=	B	1100011	0x1		$if(rs1 != rs2) PC += imm$
	blt	Branch <	B	1100011	0x4		$if(rs1 < rs2) PC += imm$
	bge	Branch ≤	B	1100011	0x5		$if(rs1 >= rs2) PC += imm$
	bltu	Branch < (U)	B	1100011	0x6		$if(rs1 < rs2) PC += imm$
	bgou	Branch ≥ (U)	B	1100011	0x7		$if(rs1 >= rs2) PC += imm$
	jal	Jump And Link	J	1101111			$rd = PC+4; PC += imm$
	jalr	Jump And Link Reg	I	1100111	0x0		$rd = PC+4; PC = rs1 + imm$

注：部分指令因篇幅原因未列出，譬如Compare/Synch/Change Level指令等

RISC-V汇编伪指令一览

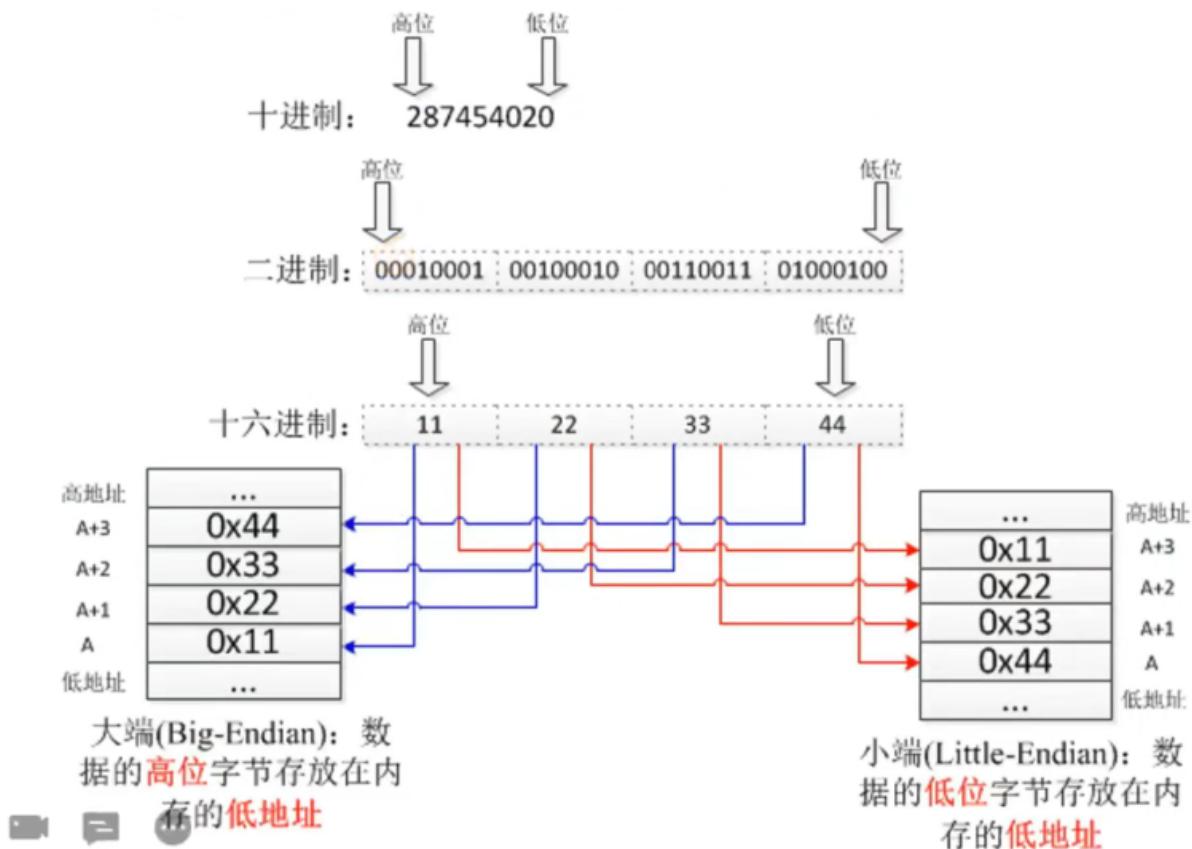
Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load address
l{b h w d} rd, symbol	auipc rd, symbol[31:12] l{b h w d} rd, symbol[11:0] (rd)	Load global
s{b h w d} rd, symbol, rt	auipc rt, symbol[31:12] s{b h w d} rd, symbol[11:0] (rt)	Store global
f{l w d} rd, symbol, rt	auipc rt, symbol[31:12] f{l w d} rd, symbol[11:0] (rt)	Floating-point load global
fs{w d} rd, symbol, rt	auipc rt, symbol[31:12] fs{w d} rd, symbol[11:0] (rt)	Floating-point store global
nop	addi x0, x0, 0	No operation
li rd, immediate	Myriad sequences	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
negw rd, rs	subw rd, x0, rs	Two's complement word
sext.w rd, rs	addiw rd, rs, 0	Sign extend word
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if \neq zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero
fmv.s rd, rs	fsgnj.s rd, rs, rs	Copy single-precision register
fabs.s rd, rs	fsgnjx.s rd, rs, rs	Single-precision absolute value
fneg.s rd, rs	fsgnjn.s rd, rs, rs	Single-precision negate
fmv.d rd, rs	fsgnj.d rd, rs, rs	Copy double-precision register
fabs.d rd, rs	fsgnjx.d rd, rs, rs	Double-precision absolute value
fneg.d rd, rs	fsgnjn.d rd, rs, rs	Double-precision negate
beqz rs, offset	beq rs, x0, offset	Branch if = zero
bnez rs, offset	bne rs, x0, offset	Branch if \neq zero
blez rs, offset	bge x0, rs, offset	Branch if \leq zero
bgez rs, offset	bge rs, x0, offset	Branch if \geq zero
bltz rs, offset	blt rs, x0, offset	Branch if < zero
bgtz rs, offset	blt x0, rs, offset	Branch if > zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if >
ble rs, rt, offset	bge rt, rs, offset	Branch if \leq
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if >, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if \leq , unsigned
j offset	jal x0, offset	Jump
jal offset	jal x1, offset	Jump and link
jr rs	jalr x0, rs, 0	Jump register
jalr rs	jalr x1, rs, 0	Jump and link register
ret	jalr x0, x1, 0	Return from subroutine
call offset	auipc x1, offset[31:12] jalr x1, x1, offset[11:0]	Call far-away subroutine
tail offset	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]	Tail call far-away subroutine
fence	fence iorw, iorw	Fence on all memory and I/O

ps: 对应规范第139页

小端序

主机字节序（HBO-Host Byte Order）：一个多位整数在计算机内存中存储的字节顺序称为主机字节序（本地字节序）

不同类型CPU的HBO不同，这与CPU的设计有关。分为大端序（Big-Endian）和小端序（Little-Endian）



通俗点说就是：

大端序：高位放在内存低地址位

小端序：低位放在内存低地址位

算术运算指令（Arithmetic Instructions）

ADD

语法	ADD RD, RS1, RS2						
例子	add x5, x6, x7						
31 27 26 25 24 20 19 15 14 12 11 7 6 0	funct7	rs2	rs1	funct3	rd	opcode	R-type

编码格式：R-Type

- opcode (7) : 0110011 (OP)
- funct3取值000; funct7取值0000000
- rs1 (5) : 第一个operand ("source register1")
- rs2 (5) : 第二个operand ("source register2")
- rd (5) : "destination register"用于存放求和的结果

注：

- gdb单步执行si
- 有符号数在计算机中的表示：二进制补码（two's complement）
- 符号扩展（Sign extension）：当一个数在字节中需要补位，那么就用符号位进行扩展，符号位为1则补1，为0则补0
- 零扩展（Zero extension）：不管符号位，全补0

SUB (subtract)

SUB (Subtract)

语法	SUB RD, RS1, RS2									
例子	sub x5, x6, x7									
	x5 = x6 - x7									

31 27 26 25 24 20 19 15 14 12 11 7 6 0
funct7 rs2 rs1 funct3 rd opcode R-type

ADDI (ADD Immediate)

语法	ADDI RD, RS1, IMM									
例子	addi x5, x6, 1									
	x5 = x6 + 1									

31 27 26 25 24 20 19 15 14 12 11 7 6 0
imm[11:0] rs1 funct3 rd opcode I-type

- imm (12) : "immediate", 立即数占12位
- 在参与算术运算前该immediate会被“符号扩展”为一个32位的数（注意符号扩展）
- 这个立即数可以表达的数值范围为：[-2^11, +2^11], 即[-2028,2047)

编码格式：I-type

- opcode (7) : 0b0010011 (OP-IMM)
- funct3 (3) : 和opcode一起决定最终的指令类型
- rs1 (5) : 第一个operand ("source register 1")
- rd (5) : "destination register"用于存放求和的结果
- imm (12) : "immediate", 立即数，代替了R-type的第三个寄存器参数和funct7

注意：RISC-V ISA并没有提供SUBI指令

基于算术运算指令实现的其他伪指令

伪指令	语法	等价指令	指令描述	例子
NEG	NEG RD, RS	SUB RD, x0, RS	对 RS 中的值取反，并将结果存放在 RD 中	neg x5, x6
MV	MV RD, RS	ADDI RD, RS, 0	将 RS 中的值拷贝到 RD 中	mv x5, x6
NOP	NOP	ADDI x0, x0, 0	什么也不做	nop

ADDI的局限性

如果要赋值一个大数（32位）该怎么办？

解决方法：

1. 先引入一个新的命令设置高20位，存放在rs1
2. 然后复用现有的ADDI命令补上剩余的低12位即可

LUI (Load Upper Immediate)

语法	LUI RD, IMM											
例子	lui x5, 0x12345											
31 27 26 25 24 20 19 15 14 12 11 7 6 0	imm[31:12]		rd	opcode	U-type							

编码格式： U-type

- opcode (7) : 0b0110111 (LUI)
- rd (5) : "destination register"用于存放结果
- imm (20) : "immediate"，立即数

LUI指令会构造一个32bits的立即数，这个立即数的高20位对应指令中的imm，低12位清零。这个立即数作为结果存放在RD中

注意一种特殊情况

➤ 利用 LUI + ADDI 来为寄存器加载一个大数

0x12345FFF

lui x1, 0x12345 # x1 = 0x12345000

addi x1, x1, 0xFFFF # x1 = 0x12345FFFF



注意 在参与算术运算前 addi 命令中的 immediate 会
被“符号扩展”为一个 32 位的数

➤ 利用 LUI + ADDI 来为寄存器加载一个大数

0x12345FFF

lui x1, 0x12346 # x1 = 0x12346000

addi x1, x1, -1 # x1 = 0x12345FFF

0xffff在符号扩展之后变成

Li (Load Immediate)

语法	LI RD, IMM	
例子	li x5, 0x12345678	x5 = 0x12345678

- Li是一个伪指令 (pseudo-instruction)
- 汇编器会根据IMM的实际情况自动生成正确的真实指令 (instruction)

AUIPC

语法	AUIPC RD, IMM												
例子	auipc x5, 0x12345												
31	27	26	25	24	20	19	15	14	12	11	7	6	0
	imm[31:12]						rd			opcode		U-type	

- AUIPC指令采用U-type
- 和LUI类似， AUIPC指令也会构造一个32bits的立即数，这个立即数的高20位对应指令中的imm，低12位清零。但和LUI不同的是， AUIPC会先将这个立即数和PC值相加，将相加后的结果存放在RD中

LA (Load Address)

语法	LA RD, LABEL
例子	la x5, foo

- LA是一个伪指令 (pseudo-instruction)
- 具体编程时给出需要加载的label，编译器会根据实际情况利用auipc和其他指令自动生成正确的指令序列。
- 常用于加载一个函数或者变量的地址

总表

指令	语法	描述		例子
ADD	ADD RD, RS1, RS2	RS1 和 RS2 的值相加，结果保存到 RD		add x5, x6, x7
SUB	SUB RD, RS1, RS2	RS1 的值减去 RS2 的值，结果保存到 RD		sub x5, x6, x7
ADDI	ADDI RD, RS1, IMM	RS1 的值和 IMM 相加，结果保存到 RD		addi x5, x6, 100
LUI	LUI RD, IMM	构造一个 32 位的数，高 20 位存放 IMM，低 12 位清零。结果保存到 RD		lui x5, 0x12345
AUIPC	AUIPC RD, IMM	构造一个 32 位的数，高 20 位存放 IMM，低 12 位清零。结果和 PC 相加后保存到 RD		auipc x5, 0x12345

伪指令	语法	等价指令	描述	例子
LI	LI RD, IMM	LUI 和 ADDI 的组合	将立即数 IMM 加载到 RD 中	li x5, 0x12345678
LA	LA RD, LABEL	AUIPC 和 ADDI的组合	为 RD 加载一个地址值	la x5 label
NEG	NEG RD, RS	SUB RD, x0, RS	对 RS 中的值取反并将结果存放在 RD 中	neg x5, x6
MV	MV RD, RS	ADDI RD, RS, 0	将 RS 中的值拷贝到 RD 中	mv x5, x6
NOP	NOP	ADDI x0, x0, 0	什么也不做	nop

逻辑运算指令 (Logical Instructions)

指令	格式	语法	描述	例子
AND	R-type	AND RD, RS1, RS2	$RD = RS1 \& RS2$	and x5, x6, x7
OR	R-type	OR RD, RS1, RS2	$RD = RS1 RS2$	or x5, x6, x7
XOR	R-type	XOR RD, RS1, RS2	$RD = RS1 \wedge RS2$	xor x5, x6, x7
ANDI	I-type	ANDI RD, RS1, IMM	$RD = RS1 \& IMM$	andi x5, x6, 20
ORI	I-type	ORI RD, RS1, IMM	$RD = RS1 IMM$	ori x5, x6, 20
XORI	I-type	XORI RD, RS1, IMM	$RD = RS1 \wedge IMM$	xori x5, x6, 20

- 所有的逻辑指令都是按位操作
- XOR (eXclusive OR, “异或”)：两个bit值不同（异）则取值为1（达到类似取1为OR的效果）；如果两个bit相同则取值为0

取非指令用伪指令来实现

伪指令	语法	等价指令	描述	例子
NOT	NOT RD, RS	XORI RD, RS, -1	对 RS 的值按位取反，结果存放在 RD 中	not x5, x6

移位运算指令 (Shifting Instructions)

逻辑移位

指令	格式	语法	描述	例子
SLL	R-type	SLL RD, RS1, RS2	逻辑左移 (Shift Left Logical) , $RD = RS1 \ll RS2$	sll x5, x6, x7
SRL	R-type	SRL RD, RS1, RS2	逻辑右移 (Shift Right Logical) $RD = RS1 \gg RS2$	srl x5, x6, x7
SLLI	I-type	SLLI RD, RS1, IMM	逻辑左移立即数 (Shift Left Logical Immediate) $RD = RS1 \ll IMM$	slli x5, x6, 3
SRLI	I-type	SRLI RD, RS1, IMM	逻辑右移立即数 (Shift Right Logical Immediate) $RD = RS1 \gg IMM$	srali x5, x6, 3

无论是逻辑左移还是逻辑右移，补足的都是0

算术移位

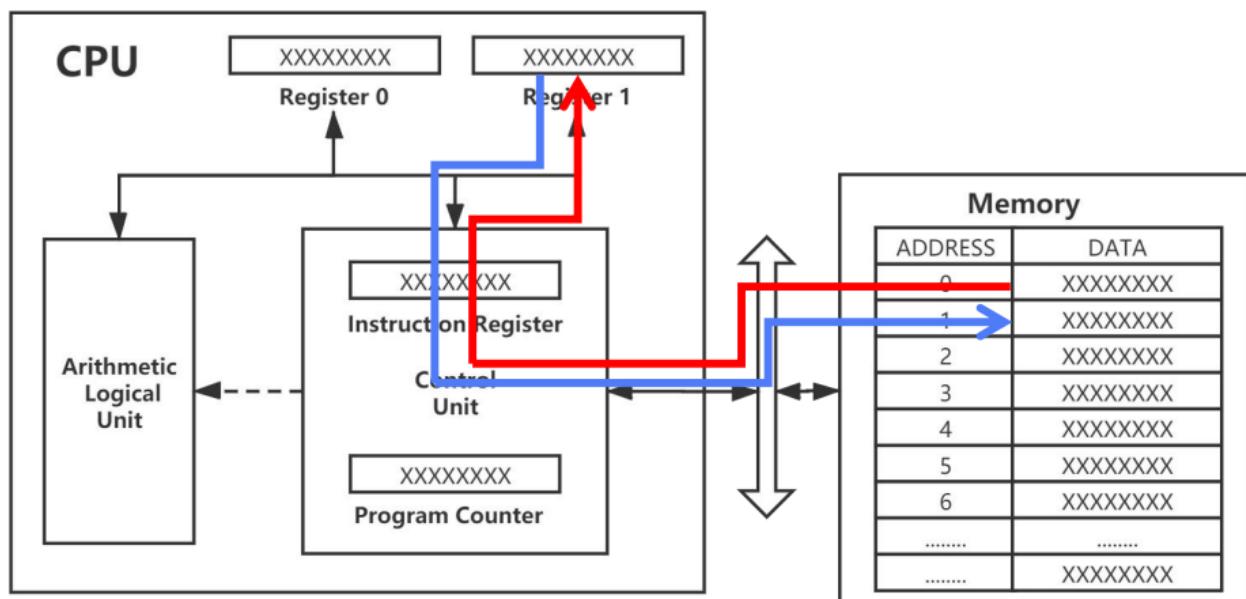
指令	格式	语法	描述	例子
SRA	R-type	SRA RD,RS1,RS2	算术右移 (Shift Right Arithmetic) RD= RS1 >> RS2	sra x5, x6, x7
SRAI	I-type	SRAI RD, RS1, IMM	算术右移立即数 (Shift Right Arithmetic Immediate) RD= RS1 >> IMM	srai x5, x6, 3

- 算术移位只有右移没有左移
- 算术右移时按照符号位值补足

内存读写指令 (Load and Store Instructions)

内存读指令: Load, 将数据从内存读入寄存器

内存写指令: Store, 将数据从寄存器写出到内存



内存读 (Load)

指令	格式	语法	描述	例子
LB	I-type	LB RD, IMM(RS1)	Load Byte, 从内存中读取一个 8 bits 的数据到 RD 中, 内存地址 = RS1 + IMM, 数据在保存到 RD 之前会执行 sign-extended 。	lb x5, 40(x6)
LBU	I-type	LBU RD, IMM(RS1)	Load Byte Unsigned, 从内存中读取一个 8 bits 的数据到 RD 中, 内存地址 = RS1 + IMM, 数据在保存到 RD 之前会执行 zero-extended 。	lbu x5, 40(x6)
LH	I-type	LH RD, IMM(RS1)	Load Halfword, 从内存中读取一个 16 bits 的数据到 RD 中, 内存地址 = RS1 + IMM, 数据在保存到 RD 之前会执行 sign-extended 。	lh x5, 40(x6)
LHU	I-type	LHU RD, IMM(RS1)	Load Halfword Unsigned, 从内存中读取一个 16 bits 的数据到 RD 中, 内存地址 = RS1 + IMM, 数据在保存到 RD 之前会执行 zero-extended 。	luh x5, 40(x6)
LW	I-type	LW RD, IMM(RS1)	Load Word, 从内存中读取一个 32 bits 的数据到 RD 中, 内存地址 = RS1 + IMM	lw x5, 40(x6)

注意：IMM给出的偏移量范围是[-2048,2047]

简单地说：

- B代表单字节， H双字节， W四字节
- 不带U使用符号扩展， 带U使用零扩展
- 同时注意IMM是基于RS1寄存器中的值的偏移地址， 最终内存地址是RS1+IMM计算出

内存写（Store）

指令	格式	语法	描述	例子
SB	S-type	SB RS2, IMM(RS1)	Store Byte, 将 RS2 寄存器中低 8 bits 的数据写出到内存中，内存地址 = RS1 + IMM。	sb x5, 40(x6)
SH	S-type	SH RS2, IMM(RS1)	Store Halfword, 将 RS2 寄存器中低 16 bits 的数据写出到内存中，内存地址 = RS1 + IMM。	sh x5, 40(x6)
SW	S-type	SW RS2, IMM(RS1)	Store Word, 将 RS2 寄存器中的 32 bits 的数据写出到内存中，内存地址 = RS1 + IMM。	sw x5, 40(x6)

31 27 26 25 24 20 19 15 14 12 11 7 6 0	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
---	-----------	-----	-----	--------	----------	--------	--------

注意：IMM给出的偏移量范围是[-2048,2047]

注：写是不需要扩展的，寻址最小单位就是字节，一个字节一个字节得看就好了

条件分支指令（Conditional Branch Instructions）

指令	格式	语法	描述	例子
BEQ	B-type	BEQ RS1,RS2,IMM	Branch if Equal。比较 RS1 和 RS2 的值，如果相等，则执行路径跳转到一个新的地址。	beq x5, x6, 100
BNE	B-type	BNE RS1,RS2,IMM	Branch if Not Equal。比较 RS1 和 RS2 的值，如果不相等，则执行路径跳转到一个新的地址。	bne x5, x6, 100
BLT	B-type	BLT RS1,RS2,IMM	Branch if Less Than。按照有符号方式比较 RS1 和 RS2 的值，如果 RS1 < RS2，则执行路径跳转到一个新的地址。	blt x5, x6, 100
BLTU	B-type	BLTU RS1,RS2,IMM	Branch if Less Than (Unsigned)。按照无符号方式比较 RS1 和 RS2 的值，如果 RS1 < RS2，则执行路径跳转到一个新的地址。	bltu x5, x6, 100
BGE	B-type	BGE RS1,RS2,IMM	Branch if Greater than or Equal。按照有符号方式比较 RS1 和 RS2 的值，如果 RS1 >= RS2，则执行路径跳转到一个新的地址。	bge x5, x6, 100
BGEU	B-type	BGEU RS1,RS2,IMM	Branch if Greater than or Equal (Unsigned)。按照无符号方式比较 RS1 和 RS2 的值，如果 RS1 >= RS2，则执行路径跳转到一个新的地址。	bgeu x5, x6, 100

31 27 26 25 24 20 19 15 14 12 11 7 6 0	imm[12:10:5]	rs2	rs1	funct3	imm[4:1 11]	opcode	B-type
---	--------------	-----	-----	--------	-------------	--------	--------

- 跳转的目标地址计算方法：先将IMM×2， 符号扩展后和PC值相加得到最终的目标地址， 所以跳转范围是以PC为基准， +/-4KB左右（[-4096, 4094]）
- 具体编程时， 不会直接写IMM， 而是用标号代替， 交由链接器来最终决定IMM的值。

常用伪指令：

伪指令	语法	等价指令	描述
BLE	BLE RS, RT, OFFSET	BGE RT, RS, OFFSET	Branch if Less & Equal, 有符号方式比较, 如果 RS <= RT, 跳转到 OFFSET
BLEU	BLEU RS, RT, OFFSET	BGEU RT, RS, OFFSET	Branch if Less or Equal Unsigned, 无符号方式比较, 如果 RS <= RT, 跳转到 OFFSET
BGT	BGT RS, RT, OFFSET	BLT RT, RS, OFFSET	Branch if Greater Than, 有符号方式比较, 如果 RS > RT, 跳转到 OFFSET
BGTU	BGTU RS, RT, OFFSET	BLTU RT, RS, OFFSET	Branch if Greater Than Unsigned, 无符号方式比较, 如果 RS > RT, 跳转到 OFFSET
BEQZ	BEQZ RS, OFFSET	BEQ RS, x0, OFFSET	Branch if EQual Zero, 如果 RS == 0, 跳转到 OFFSET
BNEZ	BNEZ RS, OFFSET	BNE RS, x0, OFFSET	Branch if Not Equal Zero, 如果 RS != 0, 跳转到 OFFSET
BLTZ	BLTZ RS, OFFSET	BLT RS, x0, OFFSET	Branch if Less Than Zero, 如果 RS < 0, 跳转到 OFFSET
BLEZ	BLEZ RS, OFFSET	BGE x0, RS, OFFSET	Branch if Less or Equal Zero, 如果 RS <= 0, 跳转到 OFFSET
BGTZ	BGTZ RS, OFFSET	BLT x0, RS, OFFSET	Branch if Greater Than Zero, 如果 RS > 0, 跳转到 OFFSET
BGEZ	BGEZ RS, OFFSET	BGE RS, x0, OFFSET	Branch if Greater or Equal Zero, 如果 RS >= 0, 跳转到 OFFSET

无条件跳转指令（Unconditional Jump Instructions）

JAL (Jump And Link)

语法	JAL RD, LABEL
例子	jal x1, label
31 27 26 25 24 20 19 15 14 12 11 7 6 0	imm[20 10:1 11 19:12] rd opcode J-type

- JAL指令使用J-type编码格式
- JAL指令用于调用子过程（subroutine/function）
- 子过程的地址计算方式：首先对20bits宽的IMM×2后进行符号扩展（sign-extended），然后将符号扩展后的值和PC的值相加。因此该函数跳转的范围是以PC为基准，上下~+/-1MB）
- JAL指令的下一条指令的地址写入RD，保存为返回地址
- 实际编程时，用label给出跳转的目标，具体IMM值由编译器和链接器最终负责生成

如何解决更远距离的跳转？

**AUIPC X6, IMM-20
JALR X1, X6, IMM-12**

JALR (Jump And Link Register)

语法	JALR RD, IMM (RS1)							
例子	jalr x0, 0(x5)							
31 27 26 25 24 20 19 15 14 12 11 7 6 0	imm[11:0]	rs1	funct3	rd	opcode	I-type		

- JALR指令使用I-type编码格式、
- JALR指令用于调用子过程（subroutine/function）
- 子过程的地址计算方法：首先对12bits宽的IMM进行符号扩展，然后将符号扩展后的值和RS1的值相加，得到最终的结果后将其最低位设置为0（确保地址按2字节对其）。因此该函数跳转的范围是以RS1为基准，上线~+/-2KB
- JALR指令的下一条指令的地址写入RD，保存为返回地址

如果跳转后不需要返回，可以利用x0代替JAL和JALR中的RD，此时可以用伪指令：

伪指令	语法	等价指令	例子
J	J OFFSET	JAL X0, OFFSET	j leap
JR	JR RS	JALR X0, 0(RS)	jr x2

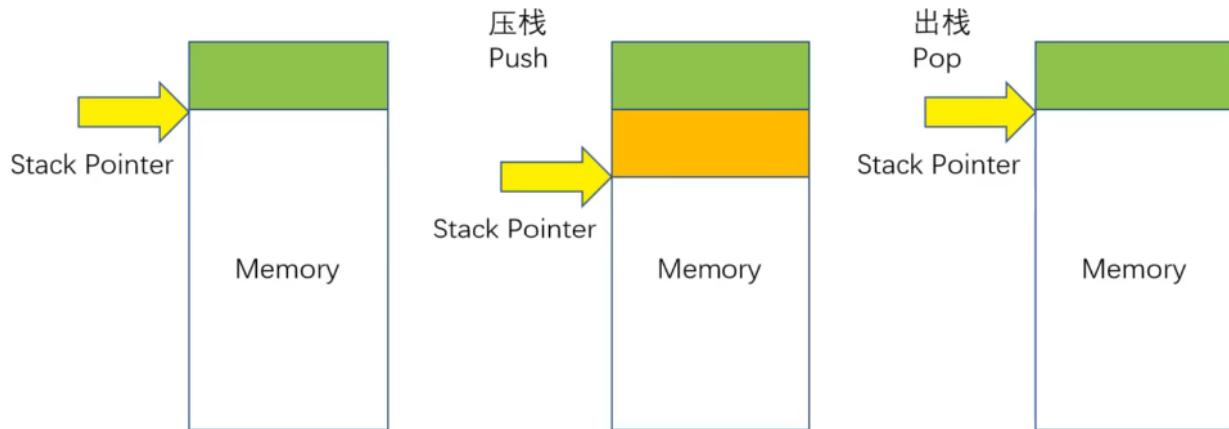
RICS-V指令寻址模式总结

所谓的寻址模式指的是指令中定位操作数（oprand）或者地址的方式

寻址模式	解释	例子
立即数寻址	操作数是指令本身的一部分	addi x5, x6, 20
寄存器寻址	操作数存放在寄存器中，指令中指定访问的寄存器从而获取该操作数。	add x5, x6, x7
基址寻址	操作数在内存中，指令中通过指定寄存器（基址 base）和立即数（偏移量 offset），通过 base + offset 的方式获得操作数在内存中的地址从而获取该操作数。	sw x5, 40(x6)
PC 相对寻址	在指令中通过 PC 和指令中的立即数相加获得目标地址的值	beq x5, x6, 100

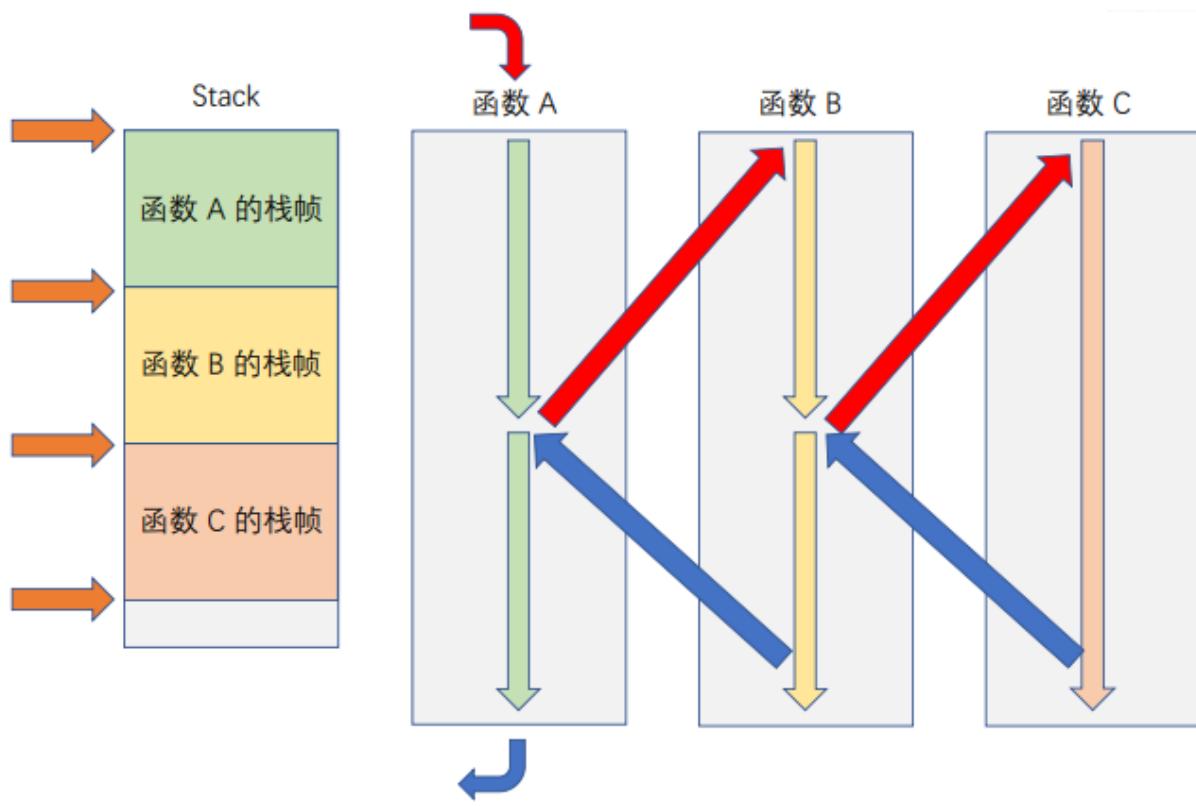
RISC-V汇编函数调用约定

函数调用过程概述



Stack (栈)：有栈底 (bottom)，栈顶 (top)，压栈 (push)，出栈 (pop)

函数栈：



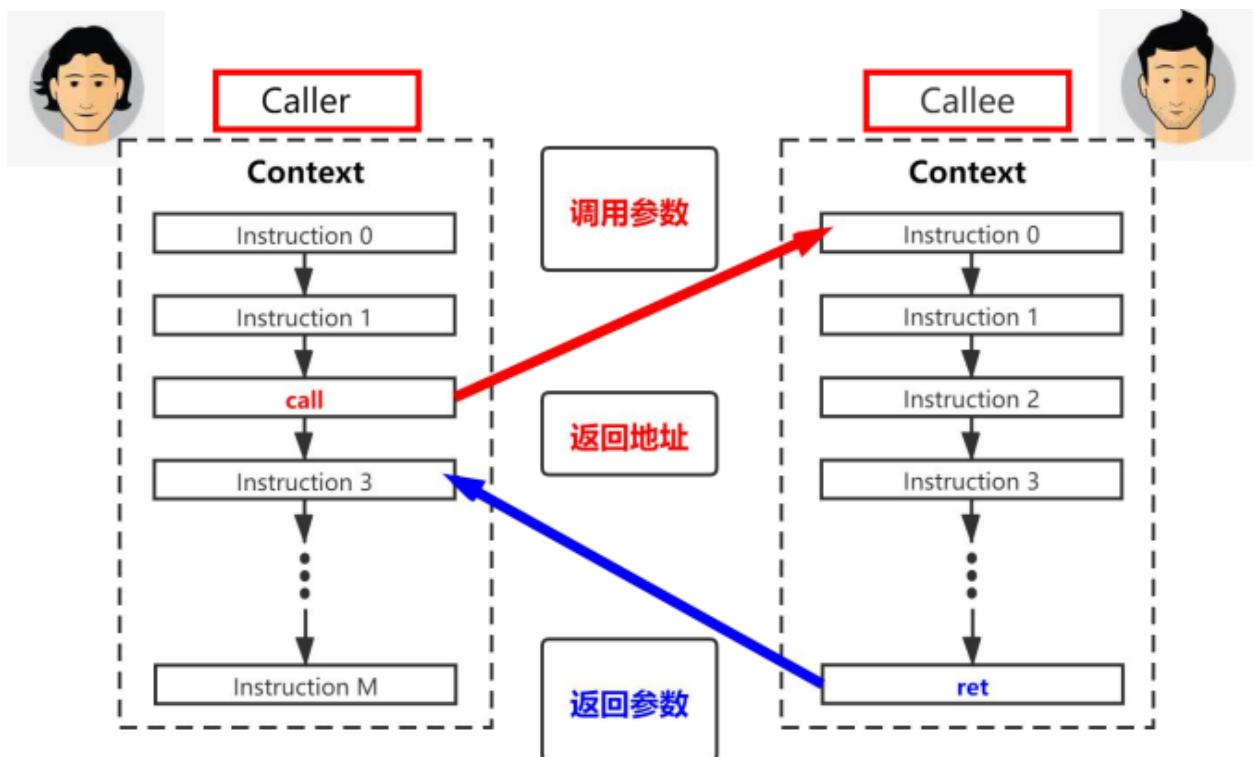
栈帧：存放函数的信息（局部变量等，全局变量在段中）

每调用一个函数，就会分配一个栈帧压入栈中

注：当栈帧大小溢出栈的大小时，就会发生栈溢出

函数调用过程中的编程约定

汇编编程时为何需要制定函数调用约定（Calling Conventions）



A函数调B函数，那么A函数叫做Caller， B函数叫做Callee

调用参数，返回地址，返回参数需要有一个约定好的地方存放

有关寄存器的编程约定

寄存器名	ABI名(编程名)	用途约定	谁负责在函数调用过程中维护这些寄存器
x0	zero	读取时总为 0, 写入时不起任何效果	N/A
x1	ra	存放函数返回值 (return address)	Caller
x2	sp	存放栈指针 (stack pointer)	Callee
x5~x7, x28~x31	t0~t2, t3~t6	临时 (temporaries) 寄存器, Callee 可能会使用这些寄存器, 所以 Callee 不保证这些寄存器中的值在函数调用过程中保持不变, 这意味着对于 Caller 来说, 如果需要的话, Caller 需要自己在调用 Callee 之前保存临时寄存器中的值。	Caller
x8, x9, x18~x27	s0, s1, s2~s11	保存 (saved) 寄存器, Callee 需要保证这些寄存器的值在函数返回后仍然维持函数调用之前的原值, 所以一旦 Callee 在自己的函数中会用到这些寄存器则需要在栈中备份并在退出函数时进行恢复。	Callee
x10, x11	a0, a1	参数 (argument) 寄存器, 用于在函数调用过程中保存第一个和第二个参数, 以及在函数返回时传递返回值。	Caller
x12 ~ x17	a2 ~ a7	参数 (argument) 寄存器, 如果函数调用时需要传递更多的参数, 则可以用这些寄存器, 但注意用于传递参数的寄存器最多只有 8 个 (a0 ~ a7), 如果还有更多的参数则要利用栈。	Caller

注：

1. ra存放函数地址的返回值
2. 通常来讲，谁维护这些寄存器，谁把这个寄存器的值保存在自己的栈中，生命周期结束要恢复原状

函数跳转和返回指令的编程约定

伪指令	等价指令	描述	例子
jal offset	jal x1 , offset	跳转到 offset 制定位置，返回地址保存在 x1 (ra)	jal foo
jalr rs	jalr x1 , 0(rs)	跳转到 rs 中值所指定的位置，返回地址保存在 x1 (ra)	jalr s1
j offset	jal x0 , offset	跳转到 offset 制定位置， 不保存返回地址	j loop
jr rs	jalr x0 , 0(rs)	跳转到 rs 中值所指定的位置， 不保存返回地址	jr s1
call offset	auipc x1, offset[31 : 12] + offset[11] jalr x1 , offset[11:0](x1)	长跳转调用函数	call foo
tail offset	auipc x6, offset[31 : 12] + offset[11] jalr x0 , offset[11:0](x6)	长跳转 尾调用	tail foo
ret	jalr x0, 0(x1)	从 Callee 返回	ret

注：

1. tail：尾调用，当A函数调B函数，在B函数中发生return func_c()时，可以看出C函数执行完不需要返回到B函数直接返回到A函数，此时这个过程就叫做尾调用，可以看如下代码

```

1 void func_A(){
2     func_B();
3 }
4
5 void func_B(){
6     return func_C();
7 }
```

2. 通常我们用call调用函数，用ret作为return返回函数

实现被调用函数的编程约定



注:

1. 在我们写C语言时, 除了函数执行体, 其它部分其实是编译器帮我们完成, 他会自己自动插入这些指令
2. 减少sp的值其实就是为函数开辟栈帧
3. ra的值保存是为了当其它函数调用时不影响我当前函数的返回

RISC-V汇编与C混合编程

RISC-V汇编调用C函数

遵守**ABI (Abstract Binary Interface)** 的规定

- 数据类型的大小, 布局和对齐
- 函数调用约定 (Calling Convention)
- 系统调用约定
-

RISC-V函数调用约定规定

- 函数参数采用寄存器a0~a7传递
- 函数返回值采用寄存器a0和a1传递

C函数中嵌入RISC-V汇编

```
asm [volatile] (
    “汇编指令”
    : 输出操作数列表 (可选)
    : 输入操作数列表 (可选)
    : 可能影响的寄存器或者存储器 (可选)
);
```

```
int foo(int a, int b)
{
    int c;
    asm volatile (
        "add %[sum], %[add1], %[add2]"
        :[sum]"=r"(c)
        :[add1]"r"(a), [add2]"r"(b)
    );
    return c;
}
```

- 想要优化就加volatile
- 汇编指令用双引号括起来，多条指令之间用";"或者"\n"分割
- "输出操作数列表"和"输入操作数列表"用于将需要操作的C变量和汇编指令的操作数对应起来，多个操作数之间用","分隔
- "可能影响的寄存器或者存储器"用于告知编译器当前嵌入的汇编语句可能修改的寄存器或者内存，方便编译器执行优化

也可以有以下写法

```

int foo(int a, int b)
{
    int c;
    asm volatile (
        "add %0, %1, %2"
        :"=r"(c)
        :"r"(a), "r"(b)
    );
    return c;
}

```

这样默认会指定

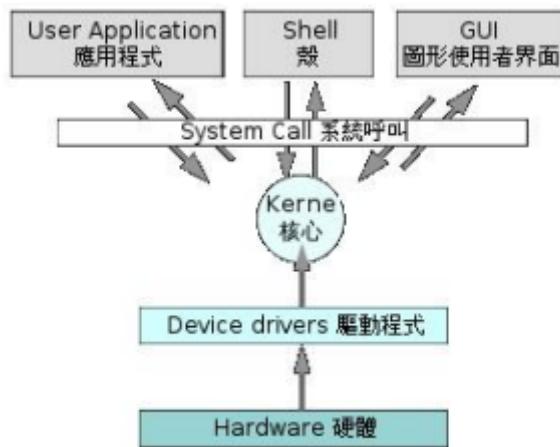
RVOS

RVOS介绍

操作系统的定义

操作系统（Operating System，缩写：OS）是一组系统软件程序：

- 主管并控制计算机操作、运用和运行硬件、软件资源
- 提供公共服务来组织用户交互



操作系统有广义和狭义之分

- 狹义：内核
- 广义：发行包=内核+一组软件

操作系统分类

分类	特点	应用场景	RISC-V ISA 对其支持
裸机系统 (Bare Metal)	非常小，没有明显的分层设计，没有通用性。通常为单任务+中断处理	微型控制器，简单外设，简单实时任务。	简单的 Machine 模式支持。
实时操作系统 (Real-Time Operating Systems)	中等规模，支持多任务，具备一定的通用性，和通用性相比更强调实时性。	比较复杂的多任务和实时场景，丰富的外设。	Machine + User；或许需要支持物理内存保护 (Physical Memory Protection, PMP)。
高级操作系统 (Rich Operating Systems)	大型规模，强调用户体验或者复杂通用性。	智能手持设备，PC工作站，云计算服务器	Machine + Supervisor + User，需要支持虚拟内存机制。

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

典型的RTOS

FreeRTOS (<https://www.freertos.org/>) 是一个很流行的应用在嵌入式设备上的实时操作系统内核。诞生于 2003 年。采用 MIT 许可证发布。

- 设计小巧，整个核心代码只有 3 到 4 个 C 文件
- 可读性强，易维护，大部分的代码都是 C 语言编写，很少的部分采用汇编语言。
- 支持优先级多线程 (threads)、互斥锁 (mutex)、信号量 (semaphore) 和软件计时器 (software timer)，支持低功耗处理以及一定程度的内存保护。
- 支持多种平台架构，包括 ARM, x86, RISC-V 等
- 已经被移植到多款微处理器上。

RT-Thread (<https://www.rt-thread.org/>) “是一个集实时操作系统 (RTOS) 内核、中间件组件和开发者社区于一体的技术平台，..... 也是一个组件完整丰富、高度可伸缩、简易开发、超低功耗、高安全性的物联网操作系统”。诞生于 2006 年。采用 Apache 2.0 许可证发布。

- 面向对象的实时内核；
- 8, 32 或 256 个优先级的多线程调度。对于同优先级线程使用时间片轮转调度法；
- 提供信号量，也提供互斥信号量以防止优先级反转；
- 支持其他高效通信方式，比如邮箱、消息队列和事件标志；
- 支持静态内存分配方法，也支持线程安全的动态内存管理；
- 对高层应用提供设备框架。
- 支持多种平台架构，包括 ARM, MIPS, X86, Xtensa, C-Sky, RISC-V 等
- 几乎支持市场上所有主流的 MCU 和 Wi-Fi 芯片。

课程的项目如下

RVOS

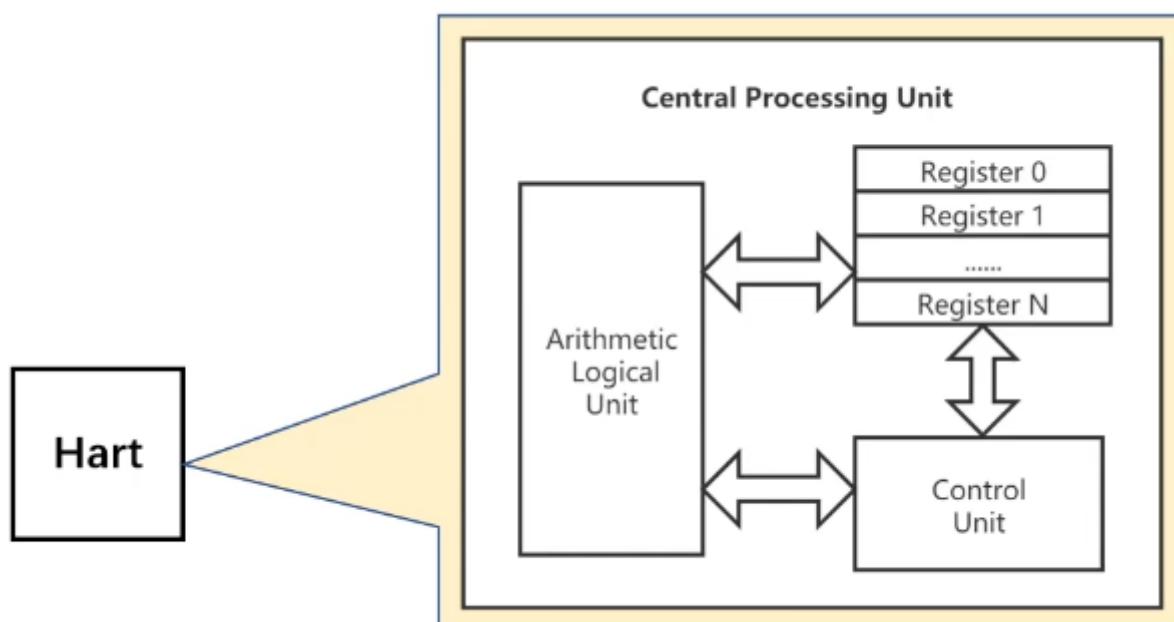
RVOS (<https://github.com/plctlab/riscv-operating-system-mooc>) 是一个用于教学演示的操作系统内核。诞生于 2021 年。采用 BSD 2-Clause 许可证发布。

- 设计小巧，整个核心有效代码 ~ 1000 行；
- 可读性强，易维护，绝大部分代码为 C 语言，很少部分采用汇编；
- 演示了简单的内存分配管理实现；
- 演示了可抢占多线程调度实现，线程调度采用轮转调度法；
- 演示了简单的任务互斥实现；
- 演示了软件定时器实现；
- 演示了系统调用实现（M + U 模式）；
- 支持 RV32；
- 支持 QEMU-virt 平台。

Hello RVOS

系统引导过程

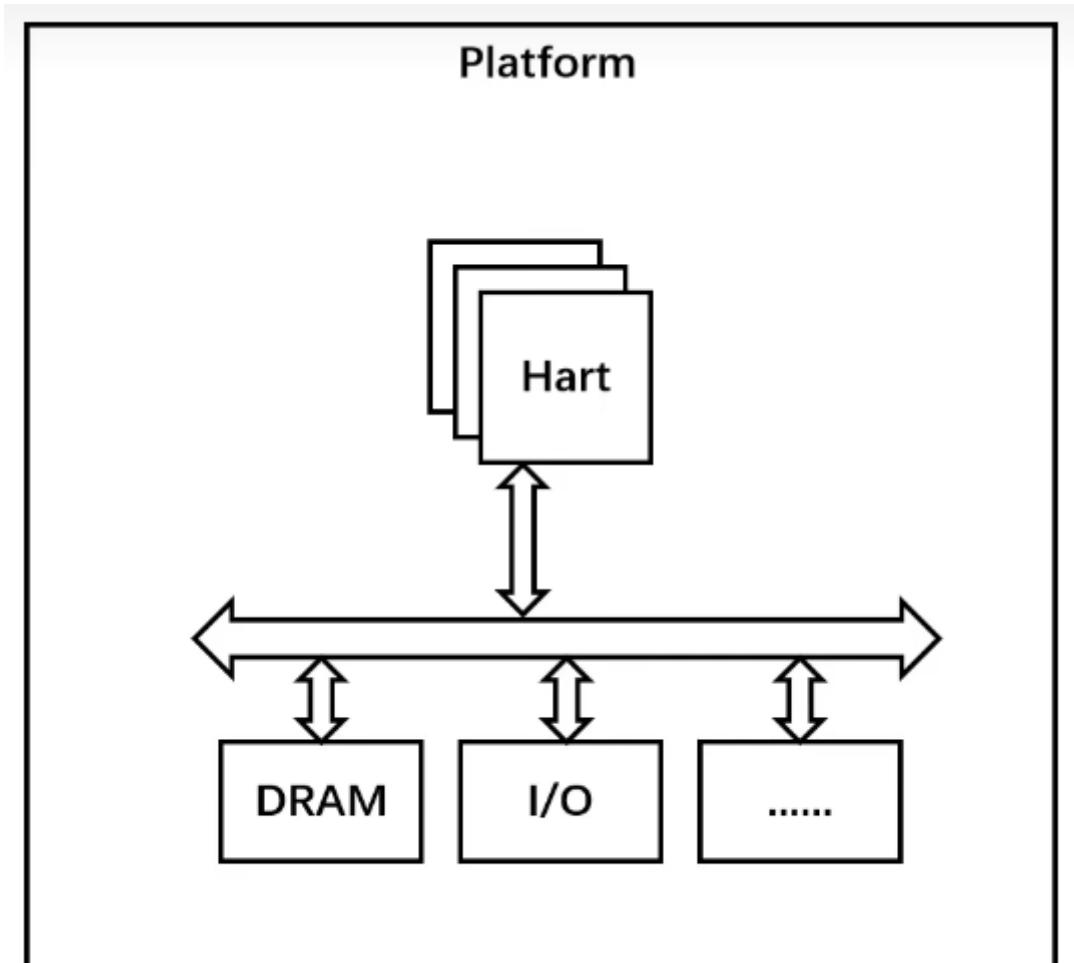
硬件的一些基本概念



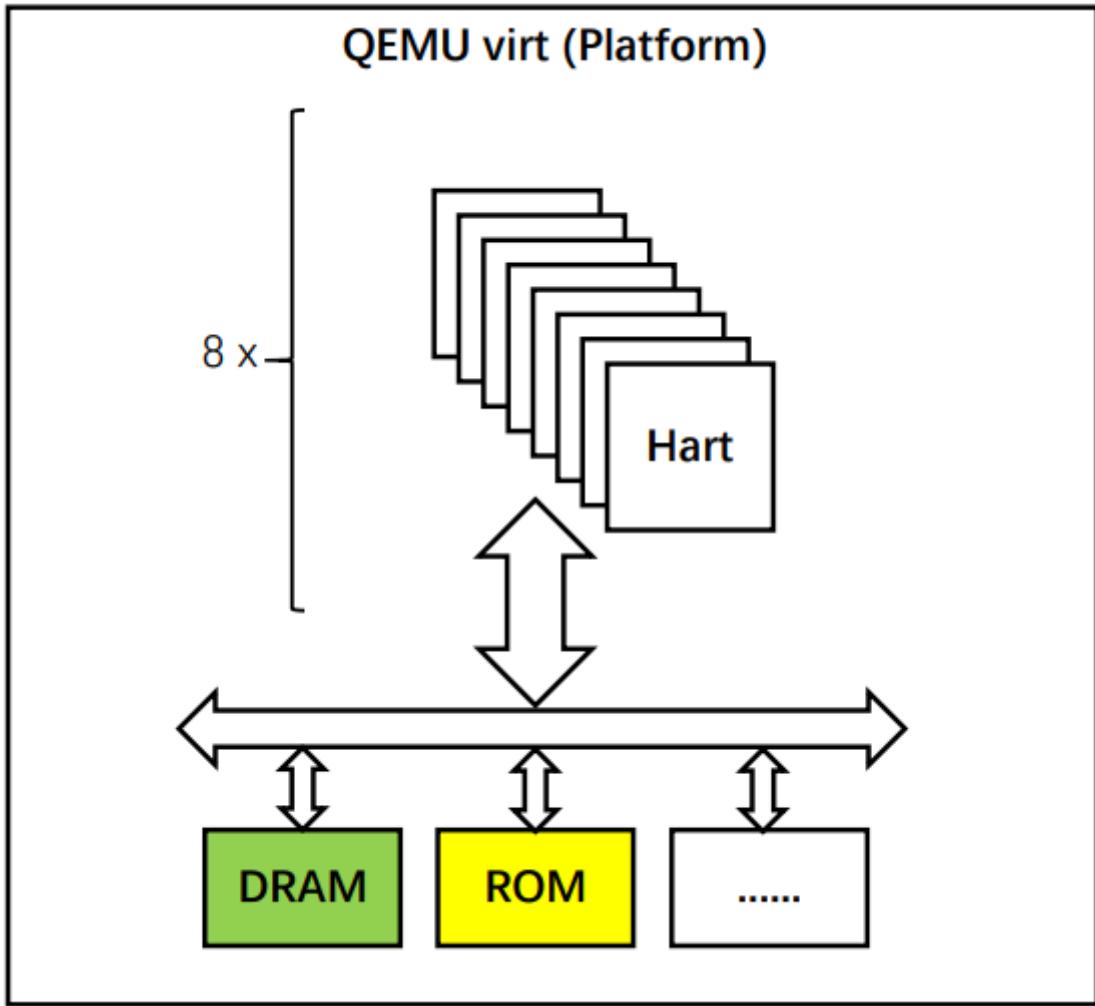
Hart: 在RISC-V中可以把他看做一个处理器

通常操作的硬件由多个Hart构成

将其封装起来称为一个Platform（可以叫做一个芯片了）



模拟器



QEMU virt就是模拟了一个Platform

QEMU-virt地址映射：

<https://github.com/qemu/qemu/blob/master/hw/riscv/virt.c>

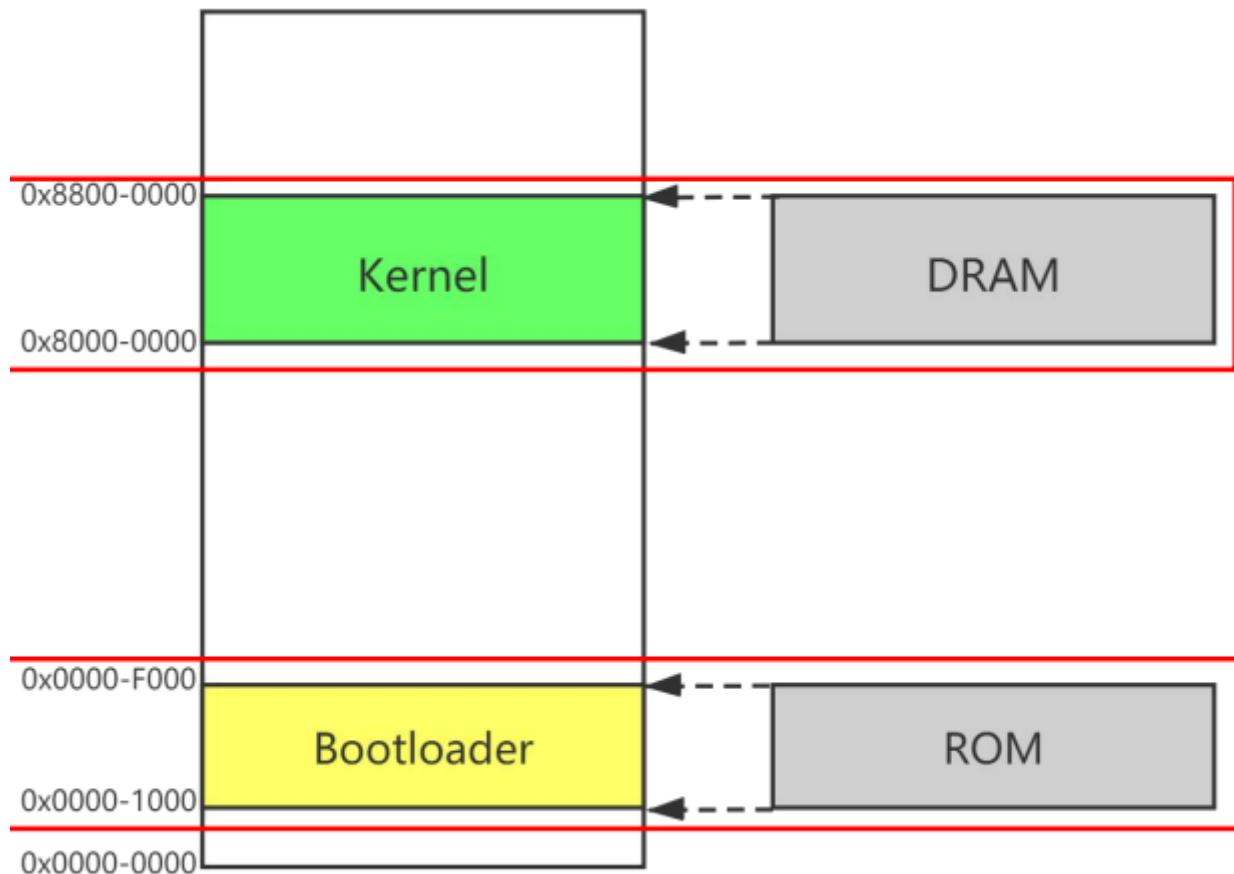
```

static const MemMapEntry virt_memmap[] = {
    [VIRT_DEBUG] = { 0x0, 0x100 },
    [VIRT_MROM] = { 0x1000, 0xf000 },
    [VIRT_TEST] = { 0x10000, 0x1000 },
    [VIRT_RTC] = { 0x10100, 0x1000 },
    [VIRT_CLINT] = { 0x2000000, 0x10000 },
    [VIRT_PCIE_PIO] = { 0x3000000, 0x10000 },
    [VIRT_PLIC] = { 0xc000000, VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 2) },
    [VIRT_UART0] = { 0x10000000, 0x100 },
    [VIRT_VIRTIO] = { 0x10001000, 0x1000 },
    [VIRT_FLASH] = { 0x20000000, 0x4000000 },
    [VIRT_PCIE_ECAM] = { 0x30000000, 0x10000000 },
    [VIRT_PCIE_MMIO] = { 0x40000000, 0x40000000 },
    [VIRT_DRAM] = { 0x80000000, 0x0 },
};

```

每一个元器件都有一个统一的物理地址映射值，上图对应下图

注：ROM，掉电不会丢失



引导过程

首先上电，CPU开始运行，硬件首先会跑到0x0000-1000上，启动Bootloader

<https://github.com/qemu/qemu/blob/master/hw/riscv/boot.c>

```
auipc    t0,0x0
addi     a2,t0,40
csrr     a0,mhartid
lw       a1,32(t0)
lw       t0,24(t0)
jr      t0
start: .word
```

该段代码会将程序跳转到0x8000运行

```

run: all
    @${QEMU} -M ? | grep virt >/dev/null || exit
    @echo "Press Ctrl-A and then X to exit QEMU"
    @echo "-----"
    @${QEMU} ${QFLAGS} -kernel os.elf

```

-kernel就是告诉QEMU加载os.elf到kernel地址上

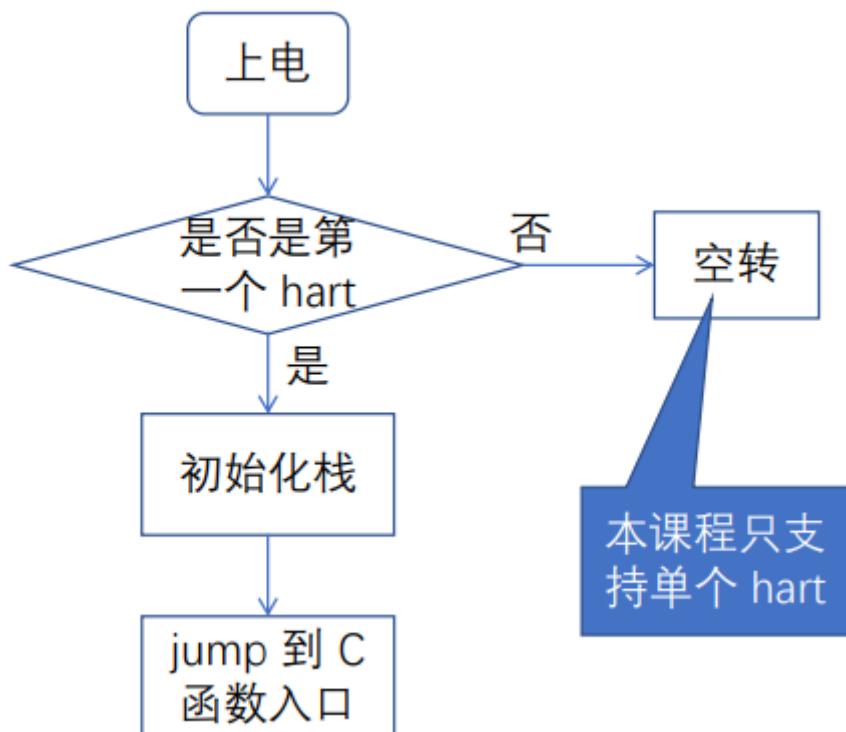
```

# start.o must be the first in dependency!
os.elf: ${OBJS}
    ${CC} ${CFLAGS} -Ttext=0x80000000 -o os.elf $^
    ${OBJCOPY} -O binary os.elf os.bin

```

所以链接的时候，编译时要加-Ttext=0x80000000，告诉链接器起始地址

引导程序要做哪些事情



注：因为硬件中有多个核，而实际我们目前的RVOS只支持一个核，所以就只跑第一个Hart，其它核让他空转

那么下面就会有三个问题：

1. 如何判断当前hart是不是第一个hart

2. 如何初始化栈
3. 如何跳转到C语言的执行环境

如何判断当前**hart**是不是第一个**hart**

通过查看CSRs寄存器组

Control and Status Registers (CSRs)

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	<i>Reserved</i>	
3	11	Machine	M

【参考 2】 Table 1.1: RISC-V privilege levels.

- 除了所有Level下都可以访问的通用寄存器组之外，每个Level都有自己对应的一组寄存器
- 高Level可以访问低Level的CSR，反之不可以
- ISA Specification ("Zicsr"扩展) 定义了特殊的CSR指令来访问这些CSR

Machine模式下的CSR列表：

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	<code>mvendorid</code>	Vendor ID.
0xF12	MRO	<code>marchid</code>	Architecture ID.
0xF13	MRO	<code>mimpid</code>	Implementation ID.
0xF14	MRO	<code>mhartid</code>	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	<code>mstatus</code>	Machine status register.
0x301	MRW	<code>misa</code>	ISA and extensions
0x302	MRW	<code>medeleg</code>	Machine exception delegation register.
0x303	MRW	<code>mideleg</code>	Machine interrupt delegation register.
0x304	MRW	<code>mie</code>	Machine interrupt-enable register.
0x305	MRW	<code>mtvec</code>	Machine trap-handler base address.
0x306	MRW	<code>mcounteren</code>	Machine counter enable.
Machine Trap Handling			
0x340	MRW	<code>mscratch</code>	Scratch register for machine trap handlers.
0x341	MRW	<code>mepc</code>	Machine exception program counter.
0x342	MRW	<code>mcause</code>	Machine trap cause.
0x343	MRW	<code>mtval</code>	Machine bad address or instruction.
0x344	MRW	<code>mip</code>	Machine interrupt pending.
Machine Memory Protection			
0x3A0	MRW	<code>pmpcfg0</code> ⋮ <code>pmpaddr15</code>	Physical memory protection configuration.
0x3BF	MRW		Physical memory protection address register.

【参考 2】 Table 2.4: Currently allocated RISC-V machine-level CSR addresses.

红框处就是一些信息值

相关命令：

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	
source/dest	source	CSRRW	dest	SYSTEM	
source/dest	source	CSRRS	dest	SYSTEM	
source/dest	source	CSRRC	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM	

【参考 1】 9.1 CSR Instructions

- CSRRW (Read/Write CSR)
- CSRRS (Read and Set bits in CSR)
- CSRRC (Read and Clear bits in CSR)
- CSRRWI/CSRRSI/CSRRCI和以上三个命令的区别是用5bit的无符号立即数（zero-extending, 零扩展）代替了rs1
- opcode取值为SYSTEM (值为1110011)

CSR指令

CSRRW (Atomic Read/Write CSR)

语法	CSRRW RD, CSR, RS1				
例子	csrrw t6, mscratch, t6				
31	20 19	15 14	12 11	7 6	0
	csr	rs1	funct3	rd	opcode
12	5	3	5	7	
source/dest	source	CSRRW	dest	SYSTEM	

- CSRRW先读出CSR中的值，将其按XLEN位的宽度进行“零扩展（zero-extend）”然后写入RD；然后将RS1中的值写入CSR
- 以上两步操作以“原子性（atomically）”方式完成
- 如果RD是x0，则不对CSR执行读的操作，只进行写操作

pseudoinstruction	Base Instruction	Meaning
csrw csr, rs	csrrw x0, csr, rs	Write CSR

CSRRS (Atomic Read and Set Bits in CSR)

语法	CSRRS RD, CSR, RS1				
例子	csrrs x5, mie, x6				
31	20 19	15 14	12 11	7 6	0
	csr	rs1	funct3	rd	opcode
12	5	3	5	7	
source/dest	source	CSRRS	dest	SYSTEM	

- CSRRS先读出CSR中的值，将其按XLEN位的宽度进行“零扩展（zero-extend）”后写入RD；然后逐个检查RS1中的值，如果某一位为1则对CSR的对应位置1，否则保持不变
- 以上两步操作以“原子性（atomically）”方式完成
- 可以只用这条命令执行读操作：

pseudoinstruction	Base Instruction	Meaning
csrr rd, csr	csrrs rd, csr, x0	Read CSR

mhartid

MXLEN-1	0
Hart ID	
MXLEN	

【参考 2】 Figure 3.5: Hart ID register (**mhartid**).

- 该CSR只读

- 包含了运行当前指令的hart的ID
- 多个hart的ID必须是唯一的，且必须有一个hart的ID值为0（第一个hart的ID）

如何实现？

```
_start:  
    # park harts with id != 0  
    csrr    t0, mhartid  
    mv     tp, t0  
    bnez   t0, park
```

⋮

```
park:  
    wfi  
    j      park
```

csrr读指令，然后到bnez里边判断跳转，跳到park中便死循环

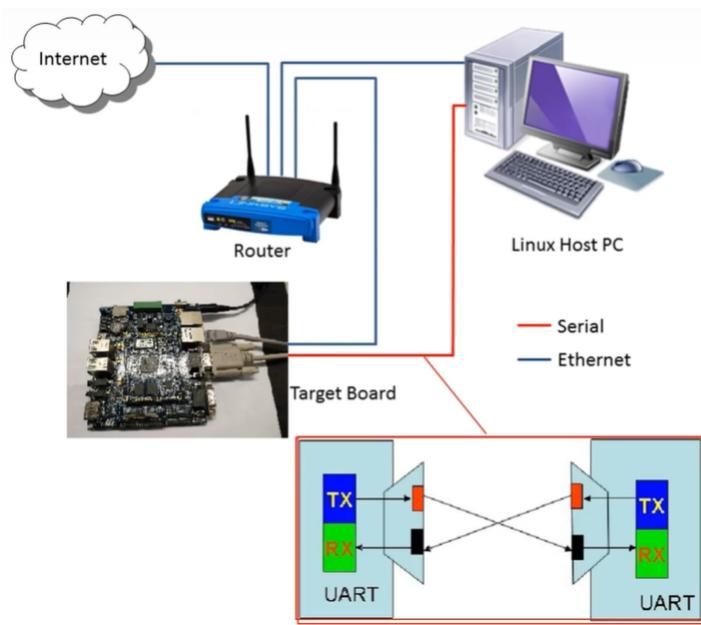
注：wfi指令

Wait for Interrupt instruction (WFI) 是 RISC-V架构定义的一条休眠指令。当处理器执行到 WFI 指令之后，将会停止执行当前的指令流，进入一种空闲状态。这种空闲状态可以被称为“休眠”状态，直到处理器接收到中断，【参考2】 3.2.3

UART

UART的硬件连接

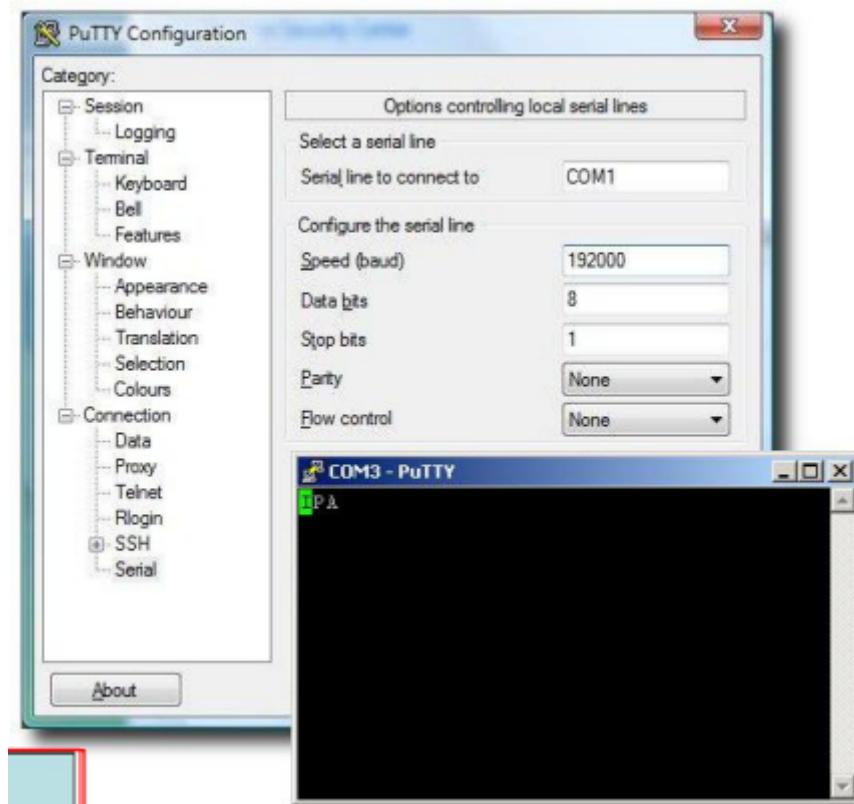
使用串口线使硬件与开发主机连接



T: transfer

R: receive

如图所示：

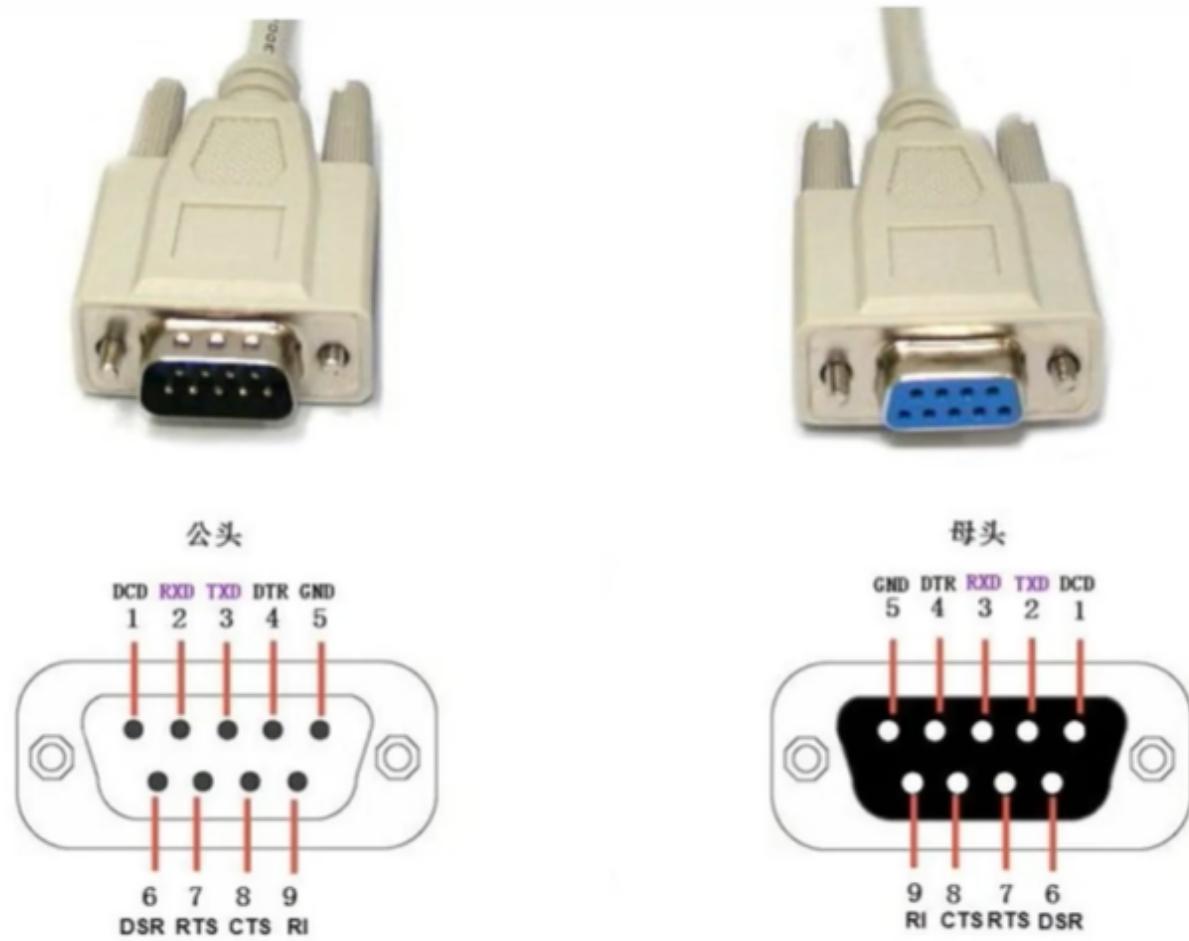


UART的特点

UART (Universal Asynchronous Receiver and Transmitter)

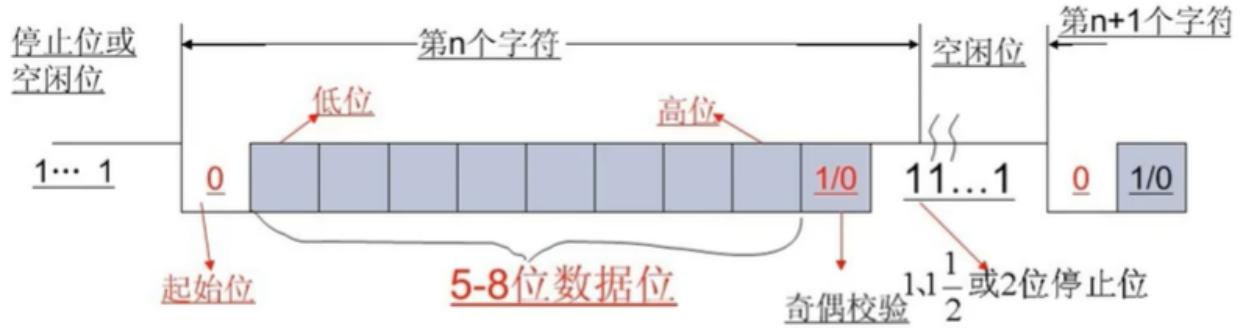
- 串行：相对于并行，串行是按位来进行传递，即一位一位的发送和接受。
- 波特率（baud rate）：每秒传输的二进制位数，单位为bps（bit per second）
- 异步（Asynchronous）：相对于同步，异步数据传输的过程中，不需要时钟线，直接发送数据，但需要约定通讯协议格式
- 全双工：相对于单工和半双工，全双工值可以同时进行收发两方向的数据传递、
- UART发送数据是按照低位到高位发送的

物理接口：



注意一点，现在很少这样用，现在一般USB

UART的通信协议

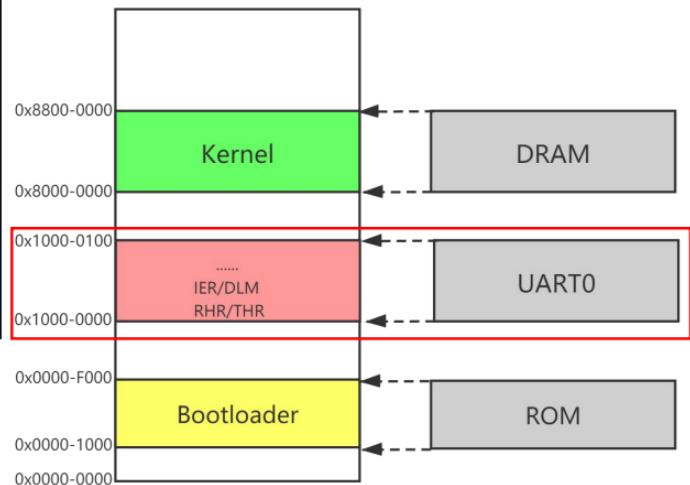


- 空闲位：总线处于空闲状态时信号线的状态为'1'即高电平
- 起始位：发送方要先发出一个低电平'0'来表示传输字符的开始
- 数据位：起始位之后就是要传输的，数据长度（word length）可以是5/6/7/8/9位，构成一个字符，一般都是8位。先发送最低位然后发送最高位
- 奇偶校验位（parity）
 - 串口校验分几种方式：
 - 无校验（no parity）
 - 奇校验（odd parity）：如果数据位中'1'的数目是偶数，则校验位为'1'，如果'1'的数目为奇数，校验位为'0'
 - 偶校验（even parity）：如果数据位中'1'的数目是偶数，则校验位为'0'，如果为奇数，校验位为'1'
 - mark parity：校验位始终为1
 - space parity：校验位始终为0
 - 停止（stop）位：数据结束标志，可以是1位，1.5位，2位的高电平

NS16550a编程接口介绍

<https://github.com/qemu/qemu/blob/master/hw/riscv/virt.c>

```
static const MemMapEntry virt_memmap[] = {
    [VIRT_DEBUG] = { 0x0, 0x100 },
    [VIRT_MROM] = { 0x1000, 0xf000 },
    [VIRT_TEST] = { 0x10000, 0x1000 },
    [VIRT_RTC] = { 0x101000, 0x1000 },
    [VIRT_CLINT] = { 0x2000000, 0x1000 },
    [VIRT_PCIE_PIO] = { 0x3000000, 0x1000 },
    [VIRT_PLIC] = { 0xc000000, VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 2) },
    [VIRT_UART0] = { 0x10000000, 0x100 }, // Address range for UART0
    [VIRT_VIRTIO] = { 0x10001000, 0x1000 },
    [VIRT_FLASH] = { 0x20000000, 0x4000000 },
    [VIRT_PCIE_ECAM] = { 0x30000000, 0x10000000 },
    [VIRT_PCIE_MMIO] = { 0x40000000, 0x40000000 },
    [VIRT_DRAM] = { 0x8000000, 0x8 },
};
```



图中前边是地址起始，后面是长度

从图中可以看出，内存中从0x1000开始，给了0x100大小的内存区域让UART0使用

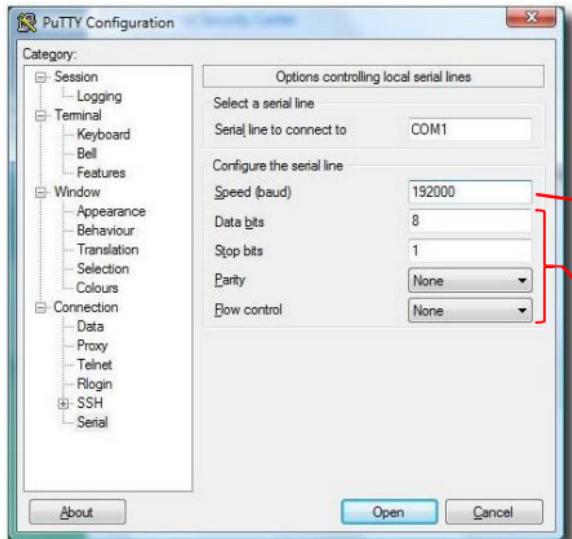
同时，可以看到DRAM的起始地址是在0x8000，DRAM就是我们使用的内存，所以我们编程时地址都是从0x8000开始的

A2	A1	A0	READ MODE	WRITE MODE
0	0	0	• Receive Holding Register	• Transmit Holding Register
0	0	0	N/A	• LSB of Divisor Latch when Enabled
0	0	1	N/A	• Interrupt Enable Register
0	0	1	N/A	• MSB of Divisor Latch when Enabled
0	1	0	• Interrupt Status Register	• FIFO control Register
0	1	1	N/A	• Line Control Register
1	0	0	N/A	• Modem Control Register
1	0	1	• Line Status Register	N/A
1	1	0	• Modem Status Register	N/A
1	1	1	• Scratchpad Register Read	• Scratchpad Register Write

【参考 3】 PROGRAMMING TABLE

- A0, A1, A2: 偏移量
- 重复出现的寄存器: 该寄存器可以复用

NS16550a的初始化



```

void uart_init()
{
    /* disable interrupts. */
    uart_write_reg(IER, 0x00);

    /* Setting baud rate. */
    uint8_t lcr = uart_read_reg(LCR);
    uart_write_reg(LCR, lcr | (1 << 7));
    uart_write_reg(DLL, 0x03);
    uart_write_reg(DLM, 0x00);

    /* Setting communication format */
    lcr = 0;
    uart_write_reg(LCR, lcr | (3 << 0));
}

```

1. 首先禁止中断功能

2. 设置LCR确定DLL于DLM寄存器的功能（DLL和DLM寄存器存在复用情况）

3. 设置串口波特率（设置方法参考手册中的波特率产生编程表）

类型于 $y=f(x)$ 通过一个比率产生对应的波特率，这个比率储存在DLL和DLM中（因为是一个16进制数，共16位），DLL存放低位，DLM存放高位，图中代码设置成0x03就是因为我们需要38.4K的波特率（和左图有出入）

4. 设计奇偶校验位

5. 该设置仅配置了发送功能

NS16550a的数据读写

UART工作方式为全双工，分发送（TX）和接受（RX）两个独立的方向进行数据传输

对数据的TX/RX有两种处理方式：

- 轮询处理方式：不停地查看该寄存器，当其空闲时放入值
- 中断处理方式：当发送寄存器空闲时就触发中断告诉CPU

A2	A1	A0	READ MODE	WRITE MODE
0	0	0	• Receive Holding Register	• Transmit Holding Register
0	0	0	N/A	• LSB of Divisor Latch when Enabled
0	0	1	N/A	• Interrupt Enable Register
0	0	1	N/A	• MSB of Divisor Latch when Enabled
0	1	0	• Interrupt Status Register	• FIFO control Register
0	1	1	N/A	• Line Control Register
1	0	0	N/A	• Modem Control Register
1	0	1	• Line Status Register	N/A
1	1	0	• Modem Status Register	N/A
1	1	1	• Scratchpad Register Read	• Scratchpad Register Write

轮询方式中，Line Status Register告诉我们发送寄存器是否空闲（为1时空闲）

RVOS

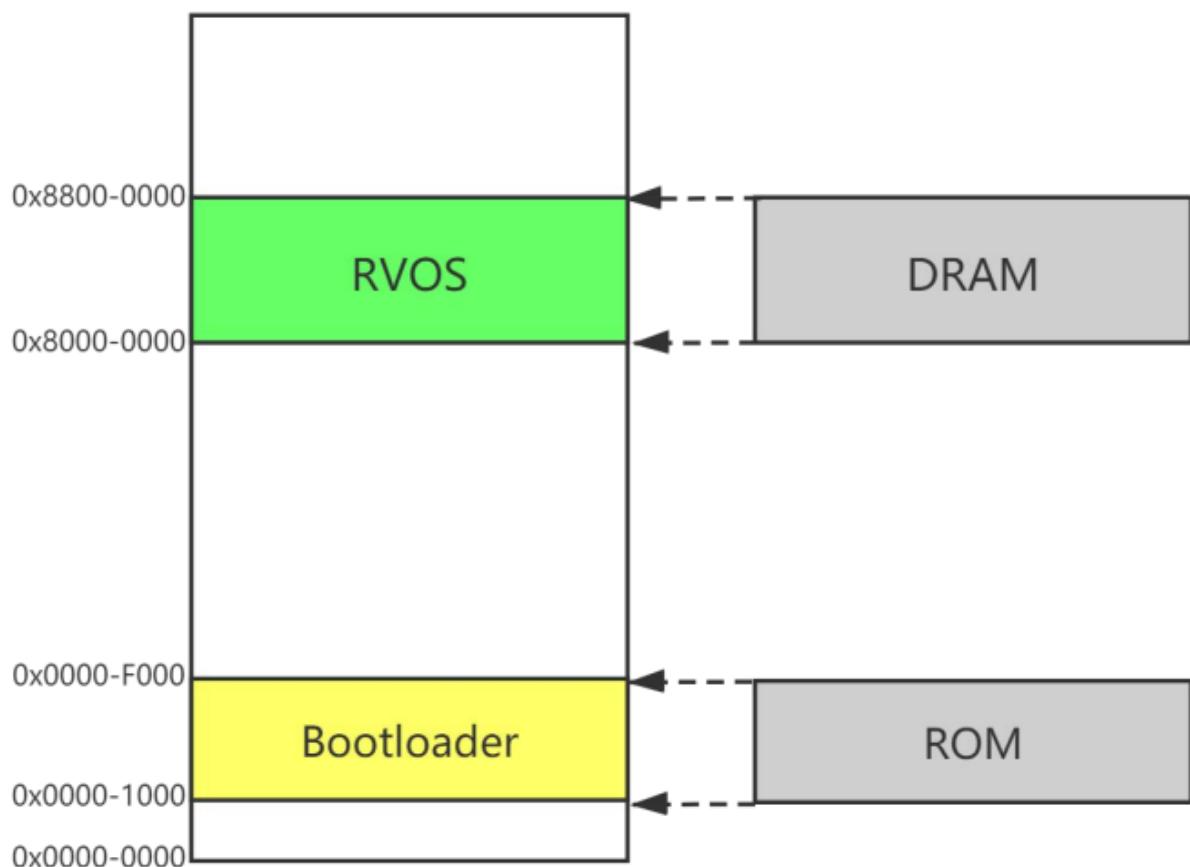
下面开始介绍RVOS的实现，为保证空间足够，下面开始使用头标题作为每一章的开始

内存管理

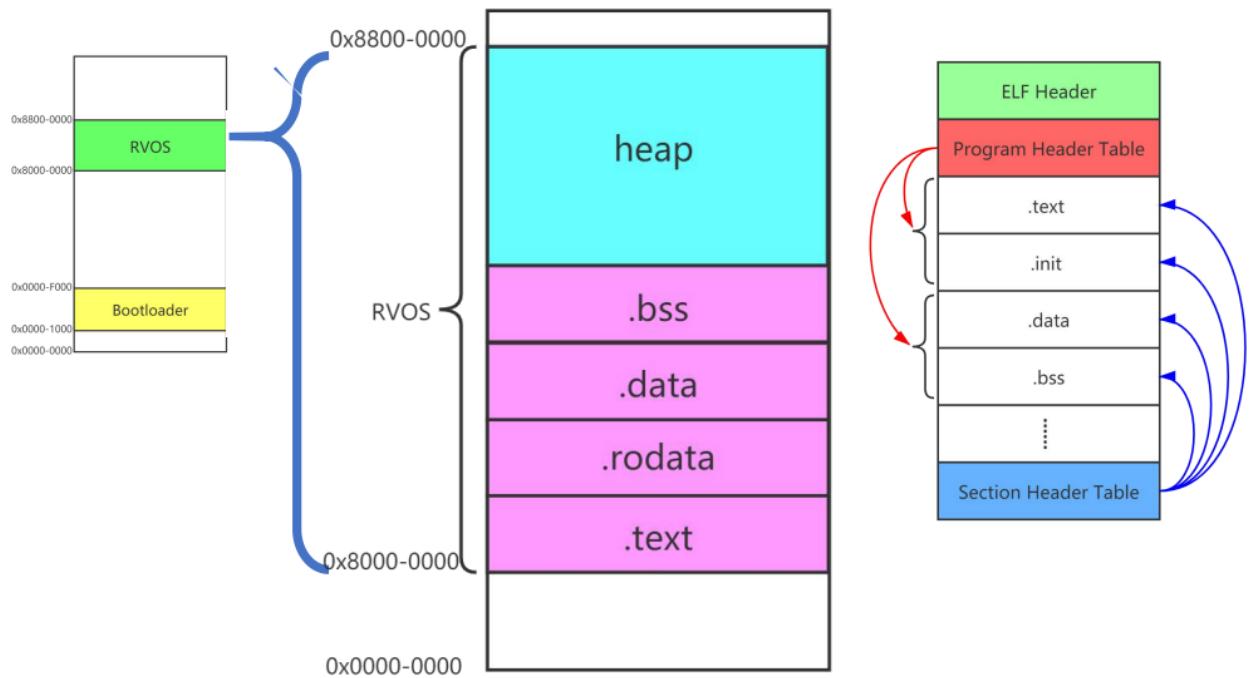
内存管理分类

- 自动管理内存：栈（stack）
- 静态内存：全局变量/静态变量
- 动态管理内存：堆（heap）

内存映射表（Memory Map）



ROM通常无法操作，我们操作的是DRAM即内存

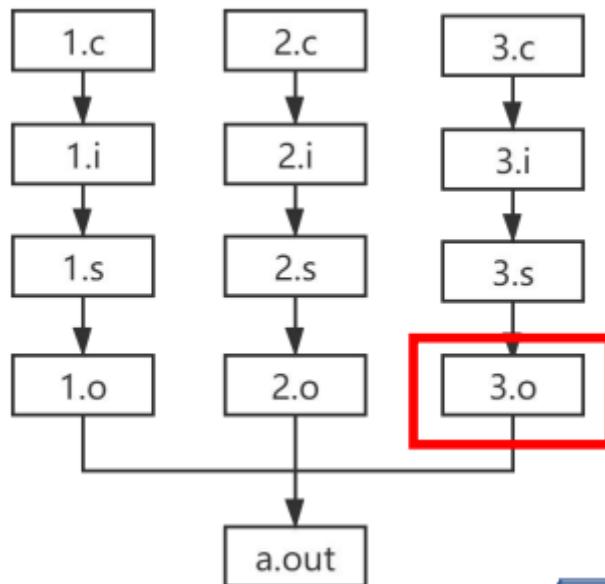


动态分配要做的就是如何利用Heap这段空间

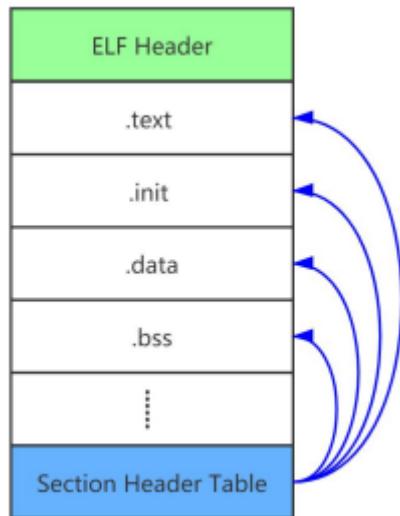
首先需要知道heap的起始地址，当然可以自己算，但是通常，编译器和链接器会帮我们计算出来

Linker Script链接脚本

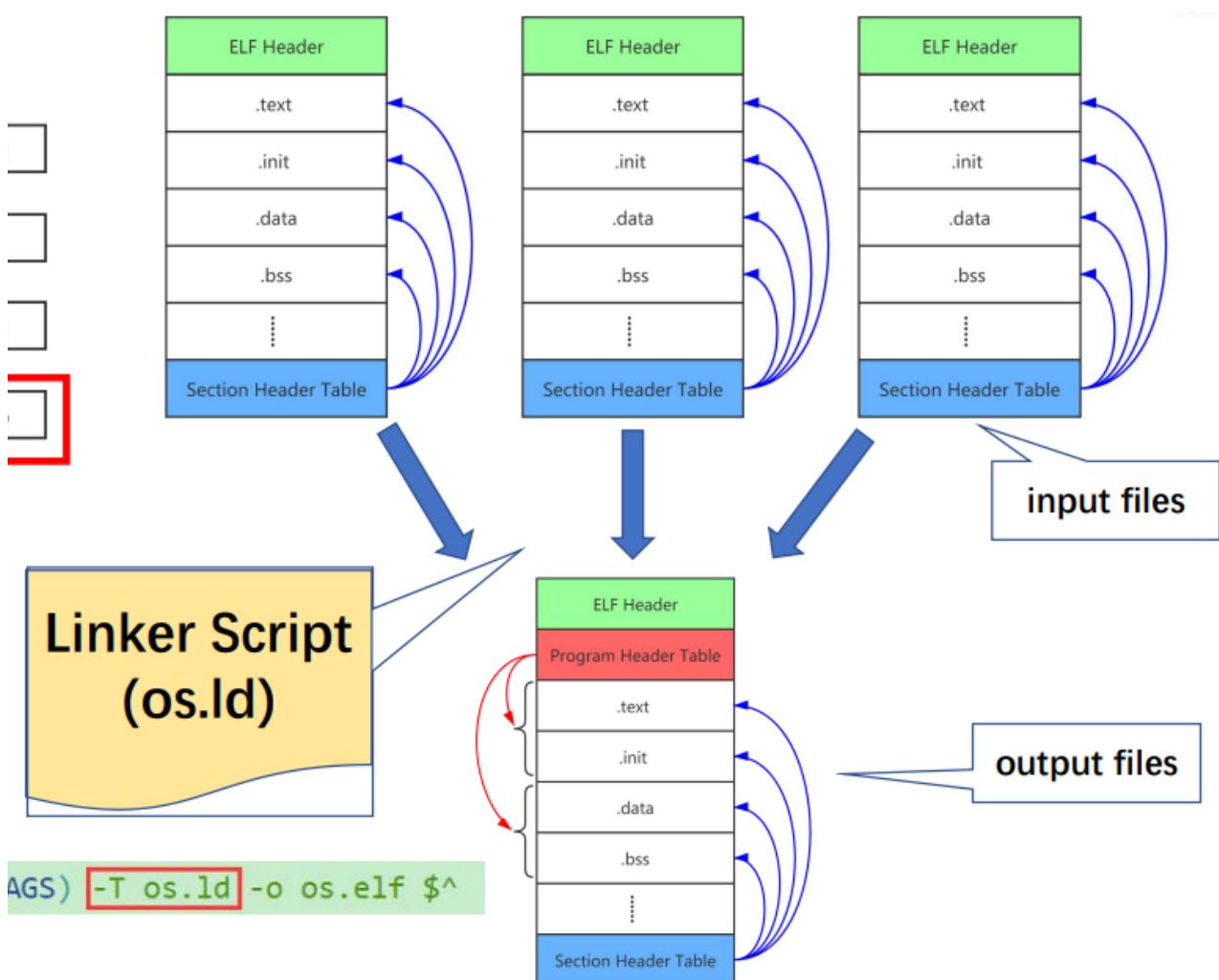
这是通常的编译链接过程：



在这其中，.o文件的格式是这样的：



可以发现，没有Program Header Table，这个部分只有在最终的.out文件才会出现，那么下面就会经过这样一个过程：



几个.o文件在Linker Script (os.ld) 脚本的链接下最终生成.out文件，我们可以在命令中这样调用这个脚本：

```
 ${CC} $ ${CC} $(CFLAGS) -T os.ld -o os.elf $^
```

-T: 告诉链接器一些相关的设置，保存在os.ld脚本中

os.ld书写方法

- GNU ld使用 Linker Script 来描述和控制链接过程
- Linker Script 是最简单的纯文本文件，采用特定的脚本描述语言编写
- 每个Linker Script中包含多条命令（Command）
- 注释采用" /* "和" */ "括起来
- 使用方法: gcc -T os.ld
- 参考文献: ld.pdf

一个简单的例子：

```
SECTIONS
{
    . = 0x10000;
    .text : { *(.text) }
    . = 0x80000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

常用命令

ENTRY

语法	ENTRY(symbol)
例子	ENTRY(_start)

- ENTRY命令用于设置“入口点（entry point）”，即程序中执行的第一条指令
- ENTRY命令的参数是一个符号（symbol）的名称

OUTPUT_ARCH

语法	OUTPUT_ARCH(bfdarch)
例子	OUTPUT_ARCH("riscv")

- OUTPUT_ARCH命令指定输出文件所适用的计算机体系架构
- 其它参数会在Makefile中提供:

-march=rv32ima -mabi=ilp32

补充一点:

```
CROSS_COMPILE = riscv64-unknown-elf-
```

为什么这里是riscv64呢? 明明我们是编译32位的程序?

是因为我们是在64位的host上编译

MEMORY

语法	MEMORY { name [(attr)] : ORIGIN = origin, LENGTH = len }
例子	MEMORY { rom (rx) : ORIGIN = 0, LENGTH = 256K ram (!rx) : org = 0x40000000, l = 4M }

- MEMORY用于描述目标机器上的内存区域、大小和相关
- 每一行可以认为定义了一个内存的区域, ORIGIN是起始地址, LENGTH是长度
- rx: 当一个节没有被section描述时, 如果它是只读, 就放在这块内存区域中

SECTIONS

语法	例子
<pre>SECTIONS { sections-command sections-command }</pre>	<pre>SECTIONS { . = 0x10000; .text : { *(.text) } . = 0x8000000; .data : { *(.data) } .bss : { *(.bss) } } >ram</pre>

- SECTIONS告诉链接器如何将input sections映射到output sections，以及如何将output sections放置到内存中
- section-command除了可以是对out section的描述外还可以是符号赋值命令等其他形式
- `1 >ram`

意思是这段的内容最终会放到ram中

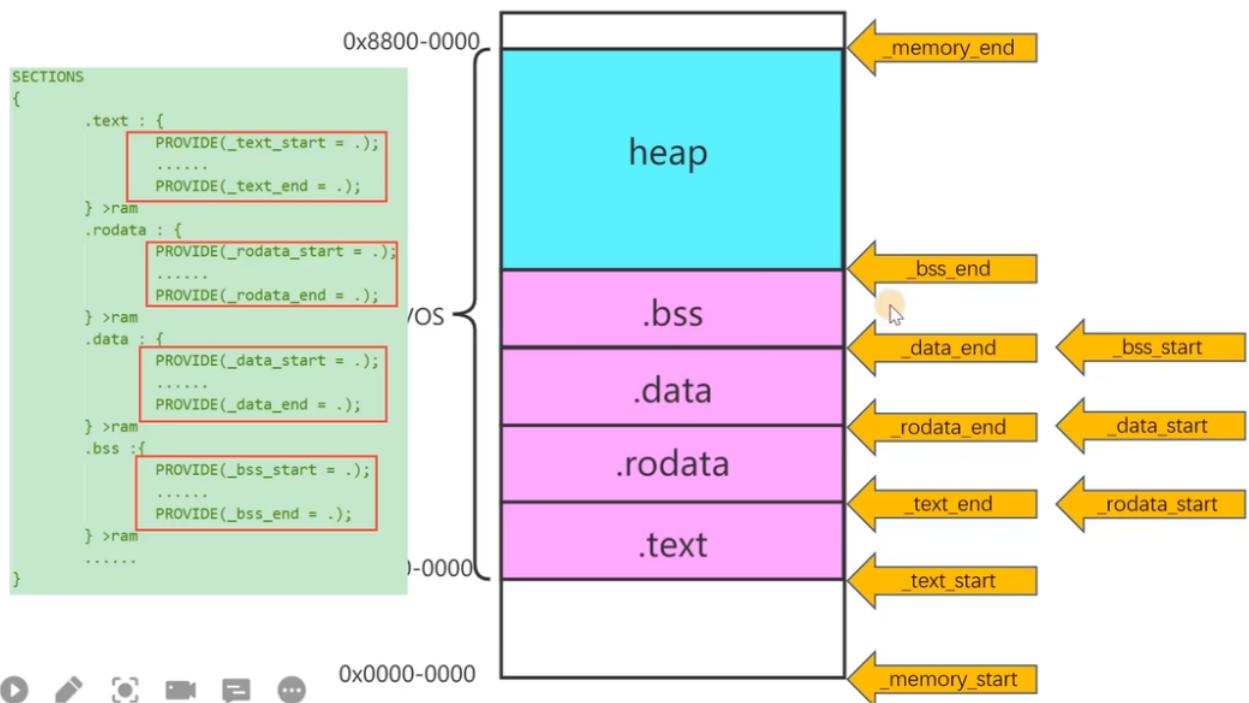
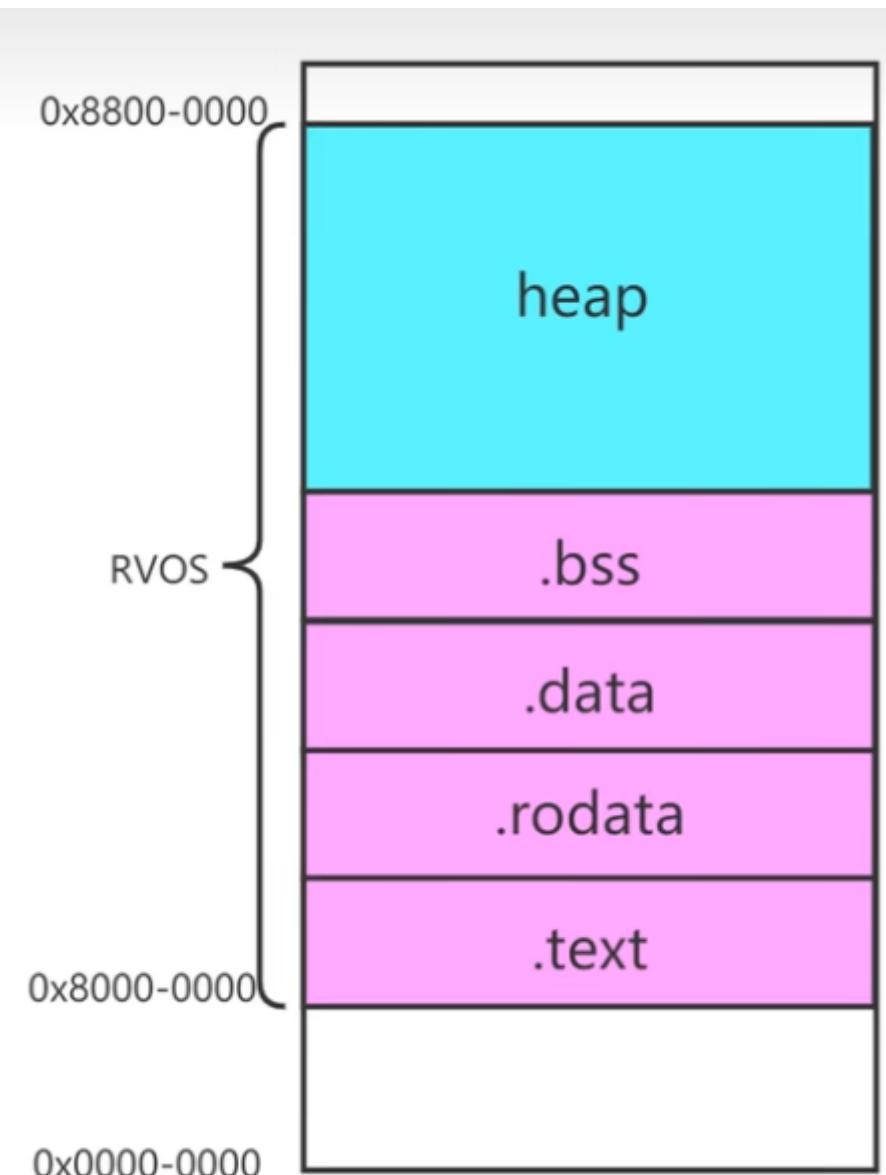
- .开头的表示内存中这些节排放的位置
- *(.text)表示所有输入文件的.text节都放到输出文件的.text节中

PROVIDE

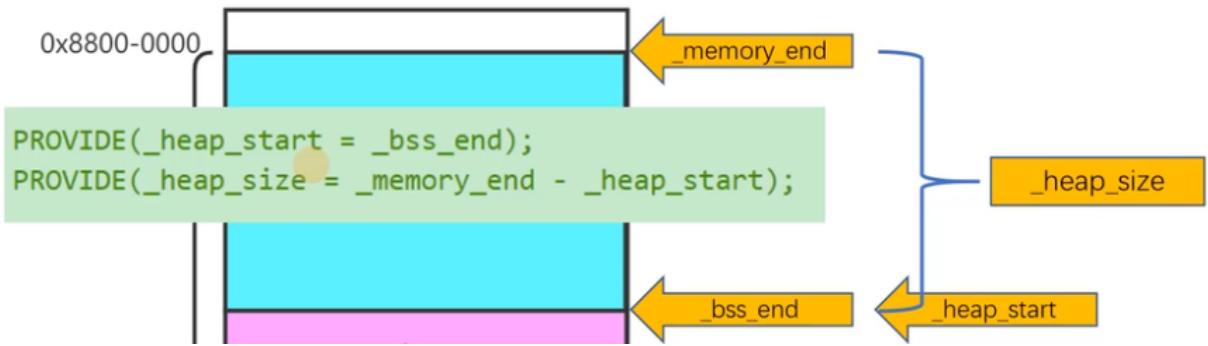
语法	PROVIDE(symbol = expression)
例子	PROVIDE(_text_start = .)

- 可以在Linker Script中定义符号（Symbols）
- 每个符号包括一个名字（name）和一个对应的地址值（address）
- 在代码中可以访问这些符号，等同于访问一个地址
- .表示当前位置

通过符号获取各个**output sections**在内存中的地址范围

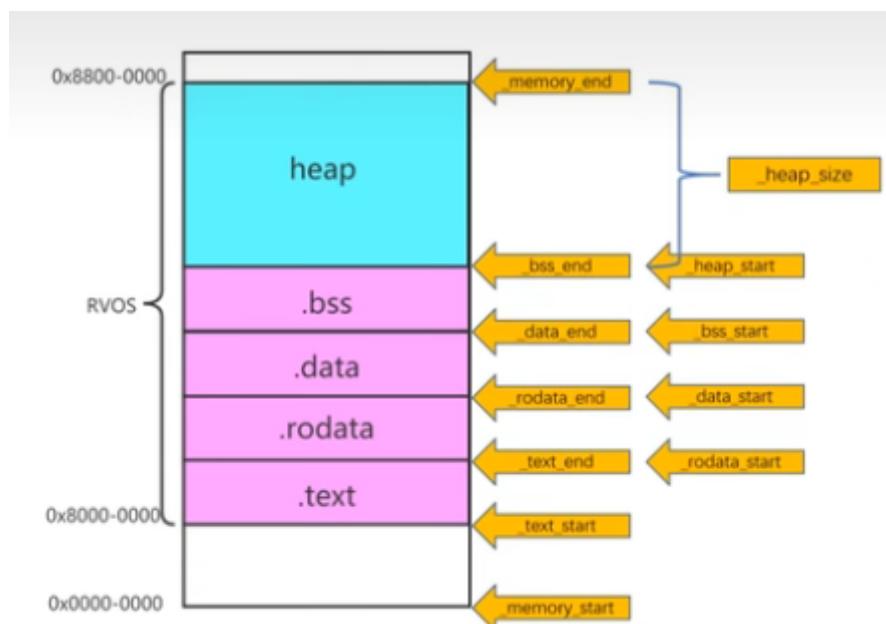


通过这些信息，我们可以得到堆的大小和起始位置



从Linker Script到Code

当我们获得链接器脚本信息后



我们就可以在代码中定义相应的变量，将相关的值赋给对应的全局变量

```

.section .rodata
.global HEAP_START
HEAP_START: .word _heap_start

.global HEAP_SIZE
HEAP_SIZE: .word _heap_size

.global TEXT_START
TEXT_START: .word _text_start

.global TEXT_END
TEXT_END: .word _text_end

.global DATA_START

```

接下来我们就可以在C语言中使用了：

```
extern uint32_t TEXT_START;
extern uint32_t TEXT_END;
extern uint32_t DATA_START;
extern uint32_t DATA_END;
extern uint32_t RODATA_START;
extern uint32_t RODATA_END;
extern uint32_t BSS_START;
extern uint32_t BSS_END;
extern uint32_t HEAP_START;
extern uint32_t HEAP_SIZE;
```

```
_num_pages = (HEAP_SIZE / PAGE_SIZE) - 8;

struct Page *page = (struct Page *)HEAP_START;
.....
_alloc_start = _align_page(HEAP_START + 8 * PAGE_SIZE);
_alloc_end = _alloc_start + (PAGE_SIZE * _num_pages);
```

基于**Page**实现动态内存分配