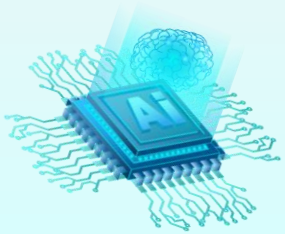
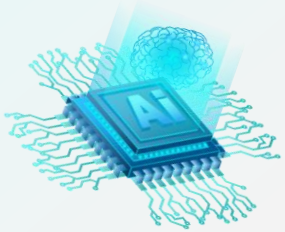
第7章 指令系统及汇编语言基本语法



本章导读：

在微型计算机原理部分，本书根据微处理器的最新发展，选择RISC-V架构作为教学蓝本，以简捷、透明见底、可实践的方式阐述微型计算机系统的基本原理。本章给出RISC-V架构的基本指令系统及汇编语言基本语法，通过汇编环境了解指令对应的机器码，直观的基本理解助记符与机器指令的对应关系；给出汇编工程框架及GPIO构件。基本掌握任何一种CPU的指令系统，当遇到新的CPU时就不会感到陌生，其本质不变。学习指令系统的基本方法是：理解寻址方式、记住几个简单指令、利用汇编语言编程练习。



7.1 RISC-V概述

RISC-V是一个基于精简指令集计算机原则而开源的指令集架构，随着RISC-V生态系统的发展，它将在微型计算机领域占有重要地位。

7.1.1 RISC-V与ISA名词解释

1．精简指令集计算机RISC

精简指令集计算机（Reduced Instruction Set Computer，RISC）的特点是指令数目少、格式一致、执行周期一致、执行时间短，采用流水线技术等。其技术背景是：CPU实现复杂指令功能的目的是让用户代码更加便捷，但复杂指令通常需要几个指令周期才能实现，且实际使用较少。RISC是对比于复杂指令计算机（Complex Instruction Set Computer，CISC）而言的，可以粗略地认为，RISC只保留了CISC常用的指令，并进行了设计优化，更适合设计嵌入式处理器。

2．指令集架构ISA

所谓指令集架构（Instruction Set Architecture，ISA）是与程序设计相关的计算机架构部分，包括数据类型、指令、寄存器、地址模式、内存架构、外部I/O、中断和异常处理等。

7.1.2 RISC-V简介

1．RISC-V的由来

RISC-V的读音为risk-five，它是由美国加州大学伯克利分校于2010年推出。早在1980年，美国加州大学伯克莱分校的帕特逊教授团队设计了RISC-I，这就是RISC这个名称的由来，到2010年，设计了RISC-V指令集架构之前，该团队已经具有四代RISC指令集架构设计经验，正因为有相关的技术沉淀，该团队才能在短期内设计出了RISC-V。

2．RISC-V在中国的发展

2018年8月，SiFive将业务扩展到中国，并在中国注册独立子公司赛昉科技（SaiFan）用于为国内客户提供服务。2018年10月，中国RISC-V产业联盟（CRVIC）在上海成立，同年11月，中国开放指令生态联盟（CRVA）宣布成立。随后的一年，华米黄山一号、阿里巴巴玄铁910，沁恒青稞V3，青稞V4等基于RISC-V开发的处理器相继发布，中国科技公司在对RISC-V的开发中渐渐找到各自的方式。

3．RISC-V与X86、ARM架构的简明比较

表7-1 RISC-V与X86、ARM架构的简明比较

比较指标	X86	ARM	RISC-V
指令集类型	CISC	RISC	RISC
寄存器宽度	32、64	32、64	32、64
源码	不开源	不开源	开源
用户可控性	难以满足需求	现阶段满足需求，未来存在变数	可望满足需求
生态系统	比较成熟	比较成熟	逐步发展
授权费用	缺乏成熟的授权模式	架构授权费用高	无

7.2 寄存器与寻址方式

CPU内部的寄存器是其内部数据暂存的地方，数量一般不会很多，每个寄存器都有自己的名字，有的具备特殊功能。寻址方式是指汇编程序的一条指令中操作数在哪里。本节给出寄存器通用基础知识、RISC-V架构主要寄存器、指令保留字简表与寻址方式，还给出如何能知道一条汇编指令的机器码。

7.2.1 寄存器通用基础知识

对CPU内部寄存器的操作与对内存的操作不同之处在于，使用汇编语言编程时，对CPU内部寄存器的访问直接使用寄存器的名字，不需经过地址、数据、控制三总线，对内存的访问涉及地址，需要经过三总线，因此对寄存器的访问比对内存的访问速度快。

从共性知识角度及功能来看，CPU内至少应该有数据缓冲类寄存器、栈指针类寄存器、程序指针类寄存器、程序状态类寄存器及其他功能寄存器。

1．数据缓冲类寄存器

CPU内数量最多的寄存器是用于数据缓冲的寄存器，名字通常用寄存器英文Register的首字母加数字组成，如R0、R1、R2等。

2．栈指针类寄存器（重点、难点）

在微型计算机的编程中，有全局变量与局部变量的概念。从存储器角度看，对一个具有独立功能的完整程序来说，全局变量具有固定的地址，每次读写都是那个地址。而在一个子程序中开辟的局部变量不是，用RAM中的哪个地址不是固定的，采用“后进先出（Last In First Out，LIFO）”原则使用一段RAM区域，这段RAM区域被称为栈区。它有个栈底的地址，是一开始就确定的，当有数据进栈或出栈时，地址会自动连续变动，不然就放到同一个存储地址中了，CPU中需要有个地方保存这个不断变化的地址，这就是栈指针（Stack Pointer）寄存器，简称SP。地址变动方向是增还是减，取决于不同计算机。

（深入理解栈）

3．程序指针类寄存器（重点、难点）

计算机的程序存储在存储器中，CPU中有个寄存器指示将要执行的指令在存储器中位置，这就是程序指针类寄存器。在许多CPU中，它的名字叫做程序计数寄存器（Program Counter，PC），PC负责告诉CPU将要执行的指令在存储器的什么地方。

4．程序运行状态类寄存器

CPU在进行计算过程中，会出现诸如进位、借位、结果为0、溢出等等情况，CPU内需要有个地方把它们保存下来，以便下一条指令结合这些情况进行处理，这类寄存器就是程序运行状态类寄存器。不同CPU其名称不同，有的叫做标志寄存器、有的叫做程序状态字寄存器等等，大同小异。在这类寄存器中，常用单个英文字母表示其含义，例如，N表示有符号运算中结果为负（Negative）、Z表示结果为零（Zero）、C表示有进位（Carry）、V表示溢出（Overflow）等。

5．其他功能寄存器

不同CPU中，除了具有数据缓冲、栈指针、程序指针、程序运行状态类等寄存器之外，还有表示浮点数运算、中断屏蔽等寄存器。

中断是暂停当前正在执行的程序，先去执行一段更加紧急程序的一种技术，它是计算机中的一个重要概念。中断屏蔽标志，就是表示是否允许某种中断进来的标志。

7.2.2 RISC-V架构主要寄存器（难点）

寄存器名	ABI 接口名称	英文描述	中文描述
x0	zero	Hardwired zero	常数 0
x1	ra	Return address	返回地址
x2	sp	Stack pointer	堆栈指针
x3	gp	Global pointer	全局指针
x4	tp	Thread pointer	线程指针
x5～x7	t0～t2	Temporary	临时寄存器
x8	s0/fp	Saved register，frame pointer	保存寄存器或帧指针
x9	s1	Saved register	保存寄存器
x10～x11	a0～a1	Function argument，return value	函数参数或返回值
x12～x17	a2～a7	Function argument	函数参数
x18～x27	s2～s11	Saved register	保存寄存器
x28～x31	t3～t6	Temporary	临时寄存器

注意点：临时寄存器、ABI接口名称含义、gp、tp、fp的作用域

7.2.3 指令保留字简表与寻址方式

指令: 是指示计算机执行某种操作的命令，是计算机运行的最小功能单位。一条指令就是机器语言的一个语句，它是一组有意义的二进制代码。

指令系统：一台计算机的所有指令的集合构成该机的指令系统，也称为指令集。

RISC-V的指令集使用模块化的方式进行组织，每一个模块使用一个英文字母来表示。

表7-3 RISC-V的模块化指令集

基本/扩展	类 型	指令数	描 述
基本指令集	RV32I	47	32 位地址空间与整数指令，支持 32 个通用整数寄存器
	RV32E	47	RV32I 的子集，仅支持 16 个通用整数寄存器
	RV64I	59	64 位地址空间与整数指令及一部分 32 位的整数指令
	RV128I	71	128 位地址空间与整数指令及一部分 64 位和 32 位的指令
扩展指令集	M	8	整数乘法与除法指令
	A	11	存储器原子操作指令，Load/Store 指令
	F	26	单精度（32 比特）浮点指令
	D	26	双精度（64 比特）浮点指令，必须支持 F 扩展指令
	C	46	压缩指令，指令长度为 16 位

1．指令保留字简表（记忆几个常用的保留字，是学习指令的基本方法）

类 型		保 留 字	含 义
数据传送类		auipc	生成与 PC 指针相关的地址
		la、lb、lh、li、lw、lhu、lbu	将存储器中的内容加载到寄存器中
		SB、SW、SH、MV	将寄存器中的内容存储到存储器中
		lui	将立即数存储到寄存器中
数据操作类	算术运算类	add、addi、sub、mul、div	加、减、乘、除指令
		slt、slti、sltu、sltui	比较指令
	逻辑运算类	and、andi、or、ori、xor、xori	按位与、或、异或
	移位类	sra、srai、sll、sll、srl、srli	算术右移、逻辑左移、逻辑右移
	csr 类	csrrw、csrrs、csrrc、csrrwi、csrrsi、csrrci	用于读写 CSR 寄存器
跳转类	无条件类	jal、jalr	无条件跳转指令
	有条件类	beq、bne、blt、bltu、bge、bgeu	有条件跳转指令
其他指令		call、ret、fence、fengei、ecall、ebreak	调用指令、返回指令、存储器屏障指令、特殊指令

2．寻址方式（重点掌握）

弄清楚指令中要操作的数据在何处

指令是对数据的操作，通常把指令中所要操作的数据称为操作数，可能来自寄存器、指令代码、存储单元。而确定指令中所需操作数的各种方法称为寻址方式（addressing mode）。

1)立即数寻址

```
lui  rd，imm              /*将 20 位立即数左移 12 位，低位补 0 写入 rd 寄存器*/
addi  rd，rs1，imm[11:0]  /*将立即数的低 12 位与 rs1 中整数相加，结果写会 rd 寄存器*/
```

2)寄存器寻址

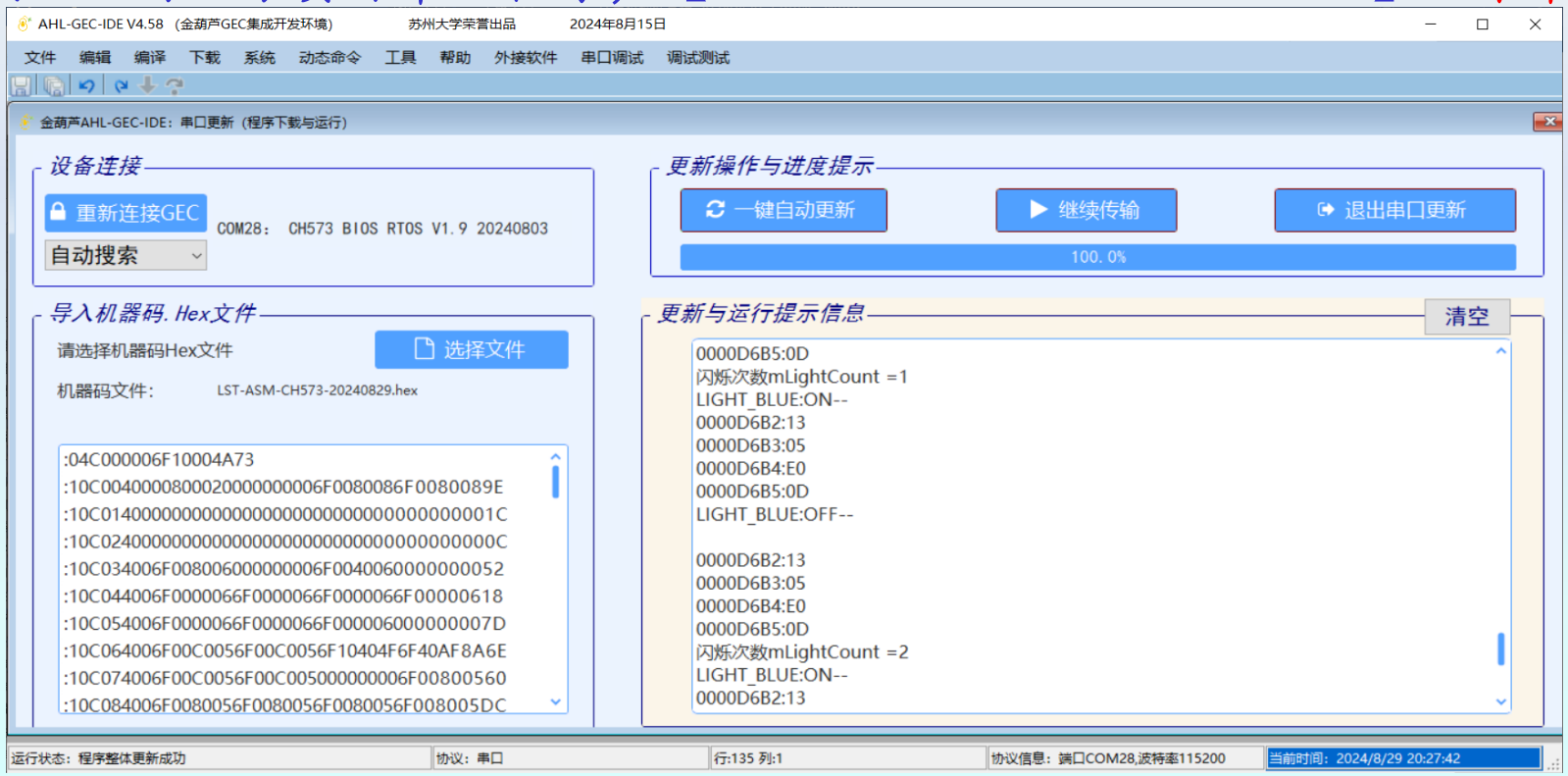
```
add rd，rs1，rs2          /*将寄存器 rs1 中整数值与 rs2 中整数值相加结果写回 rd 寄存器*/
```

3)偏移寻址及寄存器间接寻址

```
lw  rd，offset[11:0](rs1)   /*从地址 x[rs1]+offset[11:0]处读取 32 位数据写入 rd*/
lh  rd，offset[11:0](rs1)   /*从地址 x[rs1]+offset[11:0]处读取 16 位数据写入 rd*/
lhu  rd，offset[11:0](rs1)  /*从地址 x[rs1]+offset[11:0]处读取 16 位数据高位补 0 后写入 rd*/
lb  rd，offset[11:0](rs1)   /*从地址 x[rs1]+offset[11:0]处读取 8 位数据写入 rd*/
lbu  rd，offset[11:0](rs1)  /*从地址 x[rs1]+offset[11:0]处读取 8 位数据高位补 0 后写入 rd*/
sw  rs2，offset[11:0](rs1)  /*将地址 rs2 处的 32 位数据写入地址 x[rs1]+offset[11:0]处*/
```

7.2.4 机器码的获取方法

转入实际运行电子资源中的程序，【03-Software\CH07\LST-ASM】 讲解



7.3 RISC-V基本指令分类解析

理解助记符的含义与来源

(1)	la rd, symbol	将symbol的地址加载到 rd中

la load的缩写 取数，加载

lb l=load b=byte 取一个字节

lh l=Load h=halfword 取半个字（16位）

(4)	li rd, imm	将立即数加载到 rd中

li l=load i=immediate data （立即数）

（要求：根据一个具体程序，记几条，（填空题））

（对照课本分类分析）

7.4 汇编语言的基本语法

通过具体程序学习

注意点：

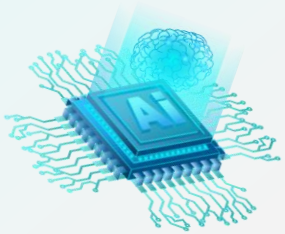
标号

常量

变量

字符串

调用子程序（函数）



7.5 汇编语言工程举例：控制小灯闪烁

【03-Software\CH07\GPIO-ASM】

通过具体程序学习（运行与基本理解程序）

```
/* 初始化蓝灯， a0、a1、a2是gpio_init的入口参数 */
LI  a0, LIGHT_RED          /* a0指明端口和引脚 */
LI  a1, GPIO_OUTPUT        /* a1指明引脚方向为输出 */
LI  a2, LIGHT_OFF          /* a2指明引脚的初始状态为亮 */
CALL  gpio_init            /* 调用gpio初始化函数 */

 LI  a0, LIGHT_RED         /* 亮灯 */
 LI  a1, LIGHT_ON
 CALL  gpio_set
```

如何使用 CALL printf 通过源代码“照葫芦画瓢”地使用，分几种情况

7.6 理解汇编工程中的GPIO构件

7.6.1 GPIO通用基础知识

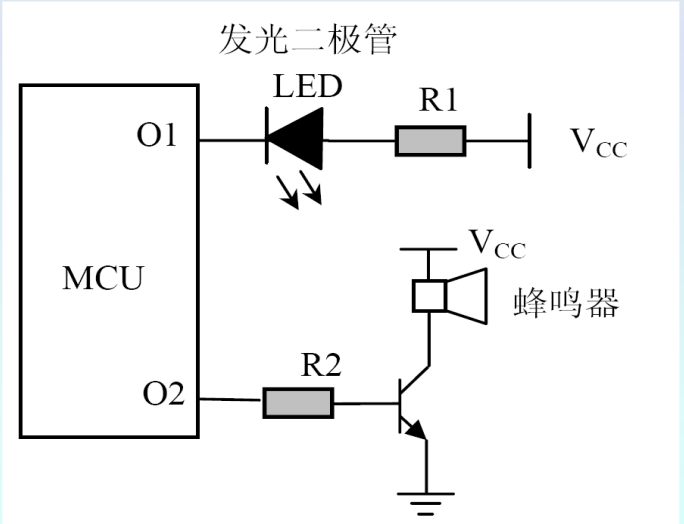
1．GPIO概念

输入/输出（Input/Output，I/O）接口是一电子电路，其内部有若干专用寄存器和相应的控制逻辑电路构成，用于MCU与外界打交道。

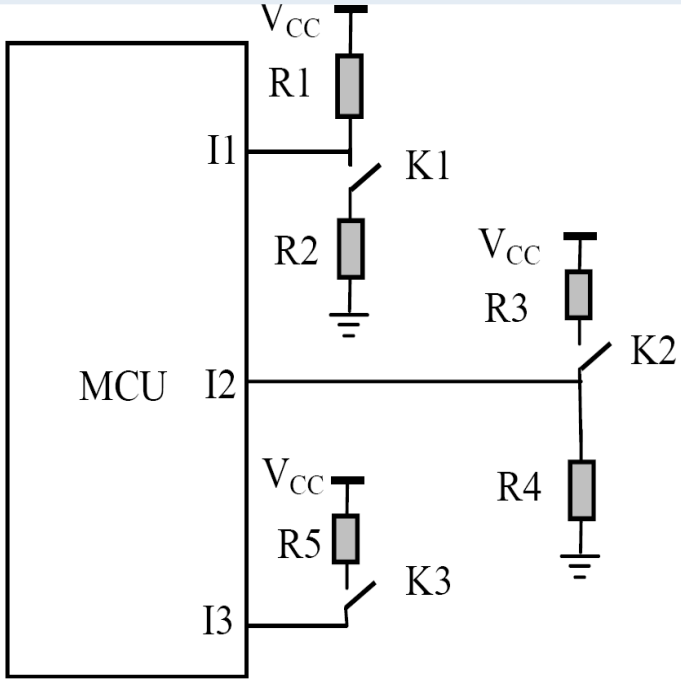
通用I/O也记为GPIO（General Purpose I/O）是I/O最基本形式。作为通用输出引脚，MCU内部程序通过端口寄存器控制该引脚状态，使得引脚输出“1”（高电平）或“0”（低电平），即开关量输出。作为通用输入引脚，MCU内部程序可以通过端口寄存器获取该引脚状态，以确定该引脚是“1”（高电平）或“0”（低电平），即开关量输入。大多数通用I/O引脚可以通过编程来设定其工作方式为输入或输出，称之为双向通用I/O。

2．输出引脚的基本接法

作为通用输出引脚，MCU内部程序向该引脚输出高电平或低电平来驱动器件工作，即开关量输出。



3．上拉下拉电阻与输入引脚的基本接法



引脚I1通过上拉电阻R1接到Vcc，选择R1>>R2，K1断开时，引脚I1为高电平，K1闭合时，引脚I1为低电平。

引脚I2通过下拉电阻R4接到地，选择R3<<R4，K2断开时，引脚I2为低电平，K2闭合时，引脚I2为高电平。

引脚I3处于悬空状态，K3断开时，引脚I3的电平不确定（这样不好）。

7.6.2 GPIO构件API

嵌入式人工智能的重要特点是软件硬件相结合，通过软件获得硬件的状态，通过软件控制硬件的动作。通常情况下，软件与某一硬件模块打交道通过其底层驱动构件，也就是封装好的一些函数，编程时通过调用这些函数，干预硬件。这样就把制作构件与使用构件的工作分成不同过程。就像建设桥梁，先做标准预制板一样，这个标准预制板就是构件。

1．软件是如何干预硬件?

现在先来看看软件是如何干预硬件的。例如想点亮图3-4中的蓝色LED小灯，由该电路原理图可以看出，只要使得标识LIGHT_BLUE的引脚为低电平，蓝色LED就可以亮起来。为了能够做到软件干预硬件，必须将该引脚与MCU的一个具有GPIO功能的引脚连接起来，通过编程使得MCU的该引脚电平为电平（逻辑0），蓝色LED就亮起来了，这就是软件干预硬件的基本过程。

2．GPIO构件的常用函数

GPIO构件主要API有：GPIO的初始化、设置引脚状态、获取引脚状态、设置引脚中断等等。

表7-18 GPIO常用接口函数简明列表

序号	函数名	简明功能	描述
1	gpio_init	引脚初始化	引脚复用为 GPIO 功能；定义其为输入或输出；若为输出，还给出其初始状态
2	gpio_set	设定引脚状态	在 GPIO 输出情况下，设定引脚状态（高/低电平）
3	gpio_get	获取引脚状态	在 GPIO 输入情况下，获取引脚状态（1/0）
4	gpio_reverse	反转引脚状态	在 GPIO 输出情况下，反转引脚状态
5	gpio_pull	设置引脚上/下拉	当 GPIO 输入情况下，设置引脚上/下拉
6	gpio_enable_int	使能引脚中断	当 GPIO 输入情况下，使能引脚中断
7	gpio_disable_int	关闭引脚中断	当 GPIO 输入情况下，关闭引脚中断
	…		

3．GPIO构件的头文件gpio.h

头文件gpio.h中包含的主要内容有：头文件说明、防止重复包含的条件编译代码结构“#ifndef ...#define ...#endif”、有关宏定义、构件中各函数的API及使用说明等。

```
// GPIO 引脚方向宏定义
#define  GPIO_INPUT   (0)     // GPIO 输入
#define  GPIO_OUTPUT  (1)     // GPIO 输出
```

```
//=========================================================================
// 函数名称：gpio_init
// 函数返回：无
// 参数说明：port_pin：(端口号)|(引脚号)（如：(PTB_NUM)|(9) 表示为 B 口 9 号脚）
//         dir：引脚方向（0=输入，1=输出，可用引脚方向宏定义）
//         state：端口引脚初始状态（0=低电平，1=高电平）
// 功能概要：初始化指定端口引脚作为 GPIO 引脚功能，并定义为输入或输出，若是输出，
//         还指定初始状态是低电平或高电平
//=========================================================================
void gpio_init(uint16_t port_pin, uint8_t dir, uint8_t state);
```

7.6.3 GPIO构件的使用方法

现在，以控制一盏小灯闪烁为例，你必须知道两点：一是由芯片的哪个引脚，二是高电平点亮还是低电平点亮。这样你就可使用 gpio 构件控制小灯了，使用步骤如下：

1．给小灯取名

在 user.inc 文件中给小灯起名字，并确定与 MCU 连接的引脚，进行宏定义。

```
.equ LIGHT_RED, (PTC_NUM|0)     /*红色 RUN 灯使用的端口/引脚*/
```

2．给灯灯的亮暗取名

在 user.inc 文件中对小灯亮、暗进行宏定义，方便编程。

```
/*灯状态宏定义（灯亮、灯暗对应的物理电平由硬件接法决定）*/
.equ LIGHT_ON,1             /*灯亮*/
.equ LIGHT_OFF,0            /*灯暗*/
```

3．初始化小灯

在 main.s 文件中初始化小灯的初始状态为输出，并点亮

```
/*初始化红灯, a0、a1、a2 是 gpio_init 的入口参数*/
li   a0,LIGHT_RED          /* 第一个入口参数：a0=端口号|引脚号 */
li   a1,GPIO_OUTPUT        /* 第二个入口参数：a1=输出模式 */
li   a2,LIGHT_ON           /* 第三个入口参数：a2=灯亮 */
call   gpio_init           /* 调用 gpio_init 函数 */
```

4．点亮小灯

在 main.s 文件中调用 gpio_set 函数点亮小灯。

```
li    a0,LIGHT_RED         /* 第一个入口参数：a0=端口号|引脚号 */
li    a1,LIGHT_ON          /* 第二个入口参数： a1 = 灯的亮/暗 */
call    gpio_set           /* 调用 gpio_set 函数 */
```

Thank you