

项目进阶

stm32

DMA -
flash - spi-外接 内部
freeRTOS - RTOS 实时操作系统
cubeIDE / cubeMX - 软件 hal库 ll库
图像处理库 - 屏幕 - 显示

云平台

物联网lot - 阿里云-设备开发平台 mqtt http
oneNET - 中国移动-设备开发平台 mqtt http tcp

百度云 - 人脸识别/车牌/物品/二维码/语音/文字....
天气
ai

QT

自己利用一些现有的库实现
opencv

qt自己接入大模型

ARM接口技术

arm - 公司/芯片技术 - 芯片的学习
三类：A-高性能 R-实时性高 M-低功耗

exynos4412 - cortex-A9
从0开始 - 汇编，寄存器编程，点灯

接口技术
GPIO
ADC
TIM
GIC
UART
RTC
WDT

芯片 - 芯片的组成 - 硬件相关知识

门电路：由 与 或 非 异或 等门电路组成

可以是单独的芯片：741s138 74HC594 (辅助芯片 半加器 选择器 38译码器 串转并 锁存器....)

可编程的芯片：计算机 加法

门电路 - 运算器

cpu：芯片的大脑

cpu的组成：运算器 控制器 存储器(寄存器) 总线

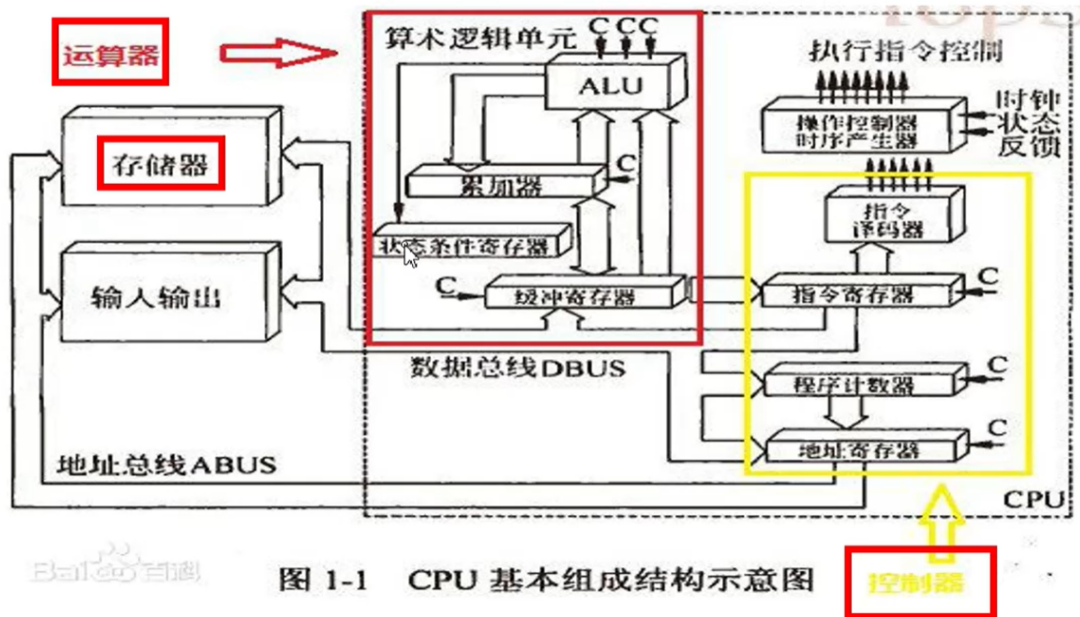


图 1-1 CPU 基本组成结构示意图

芯片：有特定功能的模块单元

芯片的组成：(soc芯片 - system on chip)

cpu + 存储器 + 专用控制器(GPIO TIM ADC DMA UART IIC SPI RTC WDT)

电脑芯片：cpu + 存储器 + 专用控制器 + GPU + NPU + GPS

计算机的组成：

核心芯片 + 输入 + 输出 + 存储器 + 总线 + 运算器 + 控制器

输入输出：键盘 鼠标 麦克风 摄像头 屏幕 喇叭

存储器：register cache ram 硬盘 网盘

RISC: reduced instruction set computer (精简指令集计算机) arm

CISC: complex instruction set computer (复杂指令集计算机) x86

常见芯片架构：芯片的图纸 (cpu架构)

c51/52: 低功耗、价格低、成本低、简单

x86: 性能高、速度快、兼容性好

ARM: 成本低、低功耗 在这基础上最求高性能

MIPS: 简洁、优化方便、高拓展性 天然64位

PowerPC: 方便灵活、可伸缩性好、功耗极高

RISC-V: 模块化、极简、可拓展、开源

了解arm芯片架构发展

三级流水线：
最佳流水线：

ARM 有8个基本工作模式：

- User : 非特权模式，大部分任务执行在这种模式
- FIQ : 当一个高优先级（fast）中断产生时将会进入这种模式
- IRQ : 当一个低优先级（normal）中断产生时将会进入这种模式
- Supervisor : 当复位或软中断指令执行时将会进入这种模式(SVC)
- Abort : 当存取异常时将会进入这种模式
- Undef : 当执行未定义指令时会进入这种模式
- System : 使用>User模式相同寄存器集的特权模式
- Cortex-A特有模式：
 - Monitor : 是为了安全而扩展出的用于执行安全监控代码的模式；也是一种特权模式

寄存器：存储数据 32位

每种模式有自己的寄存器集

寄存器的名字： 40个

基础寄存器：r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15

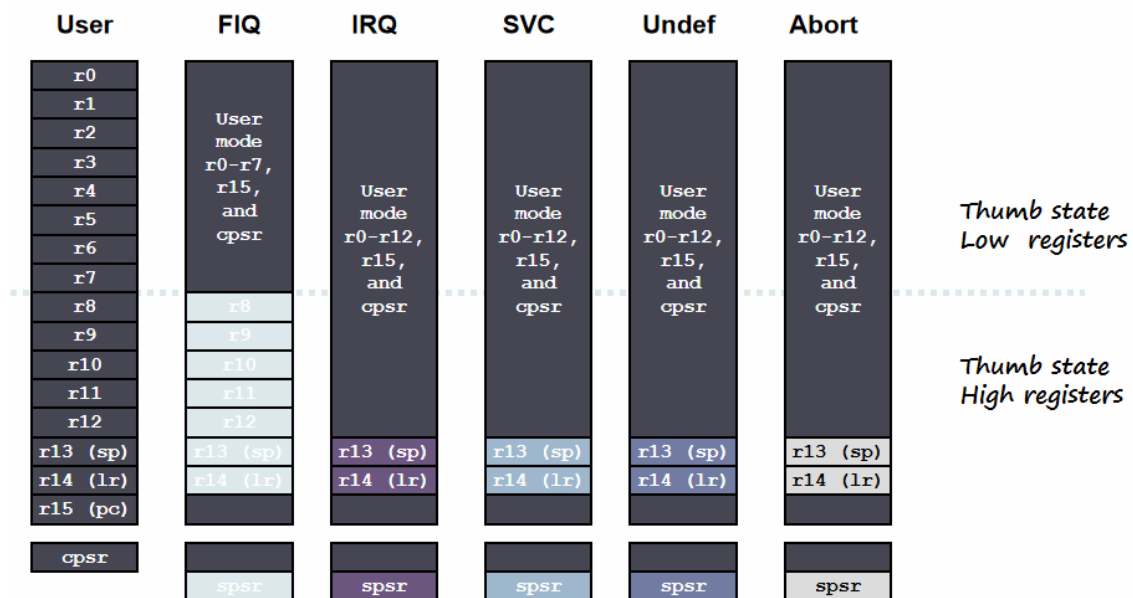
r13(sp)：栈指针寄存器：保存(内存)栈地址

r14(lr)：链接寄存器：保存程序的返回地址(函数的返回地址)

r15(pc)：程序计数器寄存器：保存的是正被取指的指令地址，而非正在执行的指令


当前程序状态寄存器(current program staius register)：cpsr (模式 状态 计算结果 中断禁止 大小端)

保存程序状态寄存器(save program staius register)：spsr (进行模式切换时，保存上一个模式的状态)



| ARM state general registers and program counter | | | | | | |
|---|----------|------------|----------|----------|-----------|----------------|
| System and User | FIQ | Supervisor | Abort | IRQ | Undefined | Secure monitor |
| r0 | r0 | r0 | r0 | r0 | r0 | r0 |
| r1 | r1 | r1 | r1 | r1 | r1 | r1 |
| r2 | r2 | r2 | r2 | r2 | r2 | r2 |
| r3 | r3 | r3 | r3 | r3 | r3 | r3 |
| r4 | r4 | r4 | r4 | r4 | r4 | r4 |
| r5 | r5 | r5 | r5 | r5 | r5 | r5 |
| r6 | r6 | r6 | r6 | r6 | r6 | r6 |
| r7 | r7 | r7 | r7 | r7 | r7 | r7 |
| r8 | r8_fiq | r8 | r8 | r8 | r8 | r8 |
| r9 | r9_fiq | r9 | r9 | r9 | r9 | r9 |
| r10 | r10_fiq | r10 | r10 | r10 | r10 | r10 |
| r11 | r11_fiq | r11 | r11 | r11 | r11 | r11 |
| r12 | r12_fiq | r12 | r12 | r12 | r12 | r12 |
| r13 | r13_fiq | r13_svc | r13_abt | r13_irq | r13_und | r13_mon |
| r14 | r14_fiq | r14_svc | r14_abt | r14_irq | r14_und | r14_mon |
| r15 | r15 (PC) | r15 (PC) | r15 (PC) | r15 (PC) | r15 (PC) | r15 (PC) |

| ARM state program status registers | | | | | | |
|------------------------------------|----------|----------|----------|----------|----------|----------|
| CPSR | CPSR | CPSR | CPSR | CPSR | CPSR | CPSR |
| | SPSR_fiq | SPSR_svc | SPSR_abt | SPSR_irq | SPSR_und | SPSR_mon |

 = banked register

操作寄存器：写代码

C语言：register int i = 0; x

汇编：指令

搬移指令：

MOV R0, #1

MOV R1, #2

看效果：

模拟开发板 - Ubuntu内部 gdb调试，查看寄存器的值

模拟开发板 - windows上 keil 4 软件，

环境搭建

1.交叉编译工具安装

ubuntu 输入： arm-linux-gcc -v 查看交叉编译工具是否安装好了

自己安装：

方法一：

先安装32位的兼容包

1、解压 gcc-4.6.4.tar.xz 到 主目录（家目录：/home/hqyj/ 根据自己Ubuntu命名）

tar -xvf gcc-4.6.4.tar.xz

2、确认解压完成，可以进入~/gcc-4.6.4/bin 下面看到 arm-linux-gcc 命令

cd ~/gcc-4.6.4/bin

pwd 获取绝对路径 /home/hqyj/gcc-4.6.4/bin

3、修改配置文件~/.bashrc，添加环境变量

(1).vi ~/.bashrc ,在最后一行添加 路径消息

export PATH=\$PATH:/home/hqyj/gcc-4.6.4/bin

(2).保存退出后:wq，执行 source ~/.bashrc

(3).关闭所有终端

方法二：

安装32位的兼容包

在线安装： 自己查一下 Ubuntu 20.04 安装arm-linux-gcc交叉编译工具

2.模拟开发板的安装

```
sudo apt-get update
sudo apt-get install qemu-system
```

3.测试

【 写代码 汇编文件 .S的后缀 】

新建一个目录: `vim test.S`

```
.global _start
```

```
_start:
```

```
    mov r1, #3
```

```
    nop
```

```
    nop
```

【 程序编译 】

```
arm-linux-gcc test.S -o test.o -c -g
```

```
arm-linux-ld test.o -o test.elf -Ttext 0x00000000
```

`-Ttext 0x00000000` 程序运行的起始地址，模拟开发板的地址是0x0

【 开启虚拟目标板 】

在第一个终端执行下面命令: 可封装成脚本

```
qemu-system-arm -machine xilinx-zynq-a9 -m 256M -serial stdio -kernel
test.elf -S -s
```

【 调试端 - 看执行结果】

再开一个终端，在另外一个终端 执行下面命令:

```
export LC_ALL=C
```

```
arm-none-linux-gnueabi-gdb test.elf
```

进入GDB后，执行

```
(gdb) target remote 127.0.0.1:1234
```

```
(gdb) s
```

命令解释:

`l:` 查看代码

`s:` 单步调试，会进入函数体

`n:` 单步调试，不进入函数体

`b 11:` 在11行设置断点

`c:` 继续运行，运行到断点位置停止

`p:` 显示变量值 `p $r1` 十进制打印; `p/x val` 十六进制打印

`x:` 显示内存值 `x/4 0x10` 显示内存地址0x10 开始的4块连续地址

`info locals:` 查看当前栈中所有局部变量的值

`q:` 退出

ARM汇编指令集

汇编 - 精简指令集

1. 底层语言 - 了解机器的运行过程
2. 代码优化，提高效率 - C代码中嵌入汇编语言 `__ASM("mov r0, #1")`
3. 调试和排除故障 - 反汇编
4. 逆向工程 -

汇编中的一些符号

@ 用来做注释。可以在行首也可以在代码后面同一行直接跟，和C语言中//类似。

做注释，一般放在行首，表示这一行都是注释而不是代码。

#数字 `add r0,r1,#4` 表示立即数 `MOV R0, #4`

: 以冒号结尾的是标号。

指令

指令的组成:

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|-----------|----|----|----|-----|----|----|--------|----|----|----|----|----|----|----|----|----|----|----|----|-------------|----|---|---|-------|---|----|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| condition | | | | O/O | | I | opcode | | | | S | Rn | | | | Rd | | | | shift mount | | | | shift | 0 | Rm | | | | | |

一条指令 32 位

`mov r1,r2,ls1 #2` 指令机器码 `0xe1a01102`
`@r1 = (r2 << 2)`

Cond: 指令的条件码。

I :用于区别Operand2(第2个操作数)是立即数(I=1)，还是寄存器移位(I=0)

Opcode: 指令操作码 `mov add sub...`

S: 操作是否影响cpsr, S=0不影响, S=1影响。运算结果是否改变cpsr的NZCV位

Rn: 包含第一个操作数的寄存器编码。(r2)

Rd: 目标寄存器编码。(r1)

剩下的12位: Operand2: 第2操作数、Rm(第二个操作数-寄存器的表达方式)

Shift amount :移位数(2)

Shift : 移位方式(ls1-左移)

`add r0, r1, #4`

立即数

为了在有限的 12位 中表达更多的数，发明立即数

高4位 - 移位数

低8位 - 数

立即数 = (低8位) ror (高4位*2); @将 一个8位数 循环右移 偶数位

如何判断一个数是否是立即数:

`0x00ff00ff`

`0xff000000 <- 0xff ror 8`

`0xf000000f <- 0xff ror 4`

`0x00010800`

1. 将数据转换为32位2进制数

```

0x00ff00ff -> 0000 0000 1111 1111 0000 0000 1111 1111 x
0xf000000f -> 1111 0000 0000 0000 0000 0000 0000 1111 v
0x00010800 -> 0000 0000 0000 0001 0000 1000 0000 0000 v
10000       -> 0000 0000 0000 0000 0010 0111 0001 0000 x
65536       -> 0000 0000 0000 0001 0000 0000 0000 0000 v

```

2. 观察其中1的位置:

将数据想象成一个圈, 首位相连

一次性去掉数据当中连续的偶数个0, 剩下的数<=8位, 就是立即数

基础指令

指令{**s**}{条件码} 目标寄存器, 第一个操作数, 第二个操作数

目标寄存器, 第一个操作数: 只能是寄存器 r0 ~ r15 sp lr pc

第二个操作数:

可以是寄存器: r0 ~ r15 sp lr pc

也可以是寄存器移位: 寄存器, 移位方式 #移位数 r2, lsl #2

也可以是立即数: #5

s: 代表运算结果要影响cpsr的nzcvc位

条件码: 这一条指令会条件执行 if()

移位方式:

LSL 逻辑左移, 右边补0

LSR 逻辑右移, 左边补0

ASL 算数左移, 右边补0

ASR 算数右移, 左边补符号位

ROR 循环右移

寄存器, 移位方式 #移位数

r0, lsr #3

搬移指令:

```

mov r0, #4      @r0 = 4
mov r0, r1      @r0 = r1
mov r0, r1, lsr #5 @r0 = r1 >> 5

```

```

mvn r0, #4      @r0 = ~4

```

算数指令:

```

add r0, r1, #1   @r0 = r1 + 1
add r0, r1, r2   @r0 = r1 + r2
adc

```

```

sub r0, r1, #1   @r0 = r1 - 1
sbc
RSB
RSC

```

```

MUL

```

位操作指令:

AND 与
ORR 或
EOR 异或
BIC 位清除

比较指令:

CMP CMN TST TEQ

```
cmp  r0, #3      等价于 subs  r0, #3
mov{条件码} r1, #4
```

```
if(r0 < 3)
    r1 = 4;
```

```
cmp  r0, #3
movlt r1, #4
```

```
if(r0 == 0)
    r1 = 4;
```

```
cmp r0, #0
moveq r1, #4
```

```
subs r0, #0
moveq r1, #4
```

```
cmp  r0, #3
cmpne r0, #4
```

条件码助记符:

| | | |
|------|----|------------|
| 相等 | eq | |
| 小于 | lt | less than |
| 大于 | gt | great than |
| 小于等于 | le | |
| 大于等于 | ge | |
| 不相等 | ne | |

| 条件码 | 助记符后缀 | 标 志 | 含 义 |
|------|-------|-------------|-----------|
| 0000 | EQ | Z置位 | 相等 |
| 0001 | NE | Z清零 | 不相等 |
| 0010 | CS | C置位 | 无符号数大于或等于 |
| 0011 | CC | C清零 | 无符号数小于 |
| 0100 | MI | N置位 | 负数 |
| 0101 | PL | N清零 | 正数或零 |
| 0110 | VS | V置位 | 溢出 |
| 0111 | VC | V清零 | 未溢出 |
| 1000 | HI | C置位Z清零 | 无符号数大于 |
| 1001 | LS | C清零Z置位 | 无符号数小于或等于 |
| 1010 | GE | N等于V | 带符号数大于或等于 |
| 1011 | LT | N不等于V | 带符号数小于 |
| 1100 | GT | Z清零且（N等于V） | 带符号数大于 |
| 1101 | LE | Z置位或（N不等于V） | 带符号数小于或等于 |
| 1110 | AL | 忽略 | 无条件执行 |

跳转指令

```
Branch :          B{<cond>} label
Branch with Link : BL{<cond>} subroutine_label
```

```
B{条件码}    label
BL{条件码}    label
```

BL 跳转的同时，lr会记录返回地址，cpu主动完成 $lr = pc - 4$ 的动作

只能在+-32M空间内进行跳转

如何进行长跳转？

```
PC = 地址
ldr pc, =0x000fffff
```

```
loop:
    nop
    nop
    b loop
```

```
nop
nop
b label
nop
nop
label:
    nop
    nop
```

BL - 在跳转的时候会记录返回地址
lr会记录返回地址，cpu主动完成 $lr = pc - 4$ 的动作

函数的实现

@实现一个延时函数

```

.global _start
_start:
    nop
    nop
    nop
    bl delay
    mov r0, #100
    nop
    nop

delay:
    mov r0, #0
loop:
    cmp r0, #100
    addle r0, #1
    ble loop
    mov pc, lr    @pc = lr    mov r15, r14

    nop
    nop

```

@实现一个延时函数

```

.global _start
_start:
    mov sp, #0x100
    nop
    nop
    nop
    bl delay_ms
    mov r0, #100
    nop
    nop

delay_ms:
    str lr, [sp, #4]!
    mov r0, #2
loop:
    subs r0, r0, #1
    bl delay_us
    bne loop
    ldr pc, [sp], #-4
    @ mov pc, lr    @pc = lr    mov r15, r14

delay_us:
    str lr, [sp, #4]!
    mov r0, #2
loop:
    subs r0, r0, #1
    bne loop
    ldr pc, [sp], #-4

    nop
    nop

```

内存操作指令

单内存操作

load 加载/读取 内存->寄存器
store 存储/写入 寄存器->内存

寄存器: r0 ~ r15
内存: 地址

ldr r0, [r1] @r0 = *(int *)r1
str r0, [r1] @*(int *)r1 = r0

ldr r0, [r1, #4] @r0 = *(int *)(r1+4)
str r0, [r1, #8] @*(int *)(r1+8) = r0

ldr r0, [r1, #4]! @r0 = *(int *)(r1+4) r1+=4
str r0, [r1, #8]! @*(int *)(r1+8) = r0 r1+=8

ldr r0, [r1], #4 @r0 = *(int *)r1 r1+=4
str r0, [r1], #8 @*(int *)r1 = r0 r1+=8

多内存操作

stmxx--store much, 多数据存储, 将寄存器的值存到地址上
ldmxx--load much, 多数据加载, 将地址上的值加载到寄存器上

stmia **stmib** **stmda** **stmdb** **stmfd** **stmfa** **stmed** **stmea**

xx有下面8种类型:

- (1) **IA**: (Increase After) 每次传送后地址加4, 其中的寄存器从左到右执行
eg: **stmia r0, {r1, r4}** //先存r1, 将r0里面的地址加上4再存r4
eg: **ldmia r0, {r0, r1, r2}** //将r0地址中的值逐个写入到寄存器r0 r1 r2中
- (2) **IB**: (Increase Before) 每次传送前地址加4, 同上
- (3) **DA**: (Decrease After) 每次传送后地址减4, 其中的寄存器从右到左执行
eg: **STMDA R0, {R1, LR}** //先存LR, 再存R1
- (4) **DB**: (Decrease Before) 每次传送前地址减4, 同上
- (5) **FD**: 满递减堆栈(每次传送前地址减4) (**LDMFD--LDMIA**; **STMFD--STMDB**)
- (6) **FA**: 满递增堆栈(每次传送后地址减4) (**LDMFA--LMDMA**; **STMFA--STMIB**)
- (7) **ED**: 空递减堆栈(每次传送前地址加4) (**LDMED--LDMIB**; **STMED--STMDA**)
- (8) **EA**: 空递增堆栈(每次传送后地址加4) (**LDMEA--LDMDB**; **STMEA--STMIA**)

四种栈:

空栈: 栈指针指向空位, 每次存入时可以直接存入然后栈指针移动一格; 而取出时需要先移动一格才能取出

满栈: 栈指针指向栈中最后一格数据, 每次存入时需要先移动栈指针一格再存入; 取出时可以直接取出, 然后再移动栈指针

增栈: 栈指针移动时向地址增加的方向移动的栈

减栈: 栈指针移动时向地址减小的方向移动的栈

栈顶: 地址大的一端

栈底: 地址小的一端

交换

```
SWP{条件}{B} <dest>, <op 1>, [<op 2>]
```

```
SWP r0, r1, [r2]  
@ r0 = *(int *)r2    *(int *)r2 = r1
```

```
SWP r0, r0, [r2]  
@ r0 = *(int *)r2    *(int *)r2 = r0
```

协处理器指令

- CDP 协处理器数操作指令
- LDC 协处理器数据加载指令
- STC 协处理器数据存储指令
- MCR ARM处理器寄存器到协处理器寄存器的数据传送指令
- MRC 协处理器寄存器到ARM处理器寄存器的数据传送指令

状态寄存器操作指令

```
cpsr    spsr
```

```
mrs : register = Status  
msr : Status = register
```

```
mrs r0~r15, cpsr/spsr  
msr cpsr/spsr, r0~r15
```

```
cpsr 后五位: 模式  
SVC:10011 -> user:10000
```

```
mrs r0, cpsr  
bic r0, #3  
msr cpsr, r0
```

软中断指令

```
swi 中断号
```

软件内部产生一个中断信号

伪指令

arm-linux-gcc 编译工具 gnu编译工具链
对代码做了一定的优化

伪指令：假的指令 arm-linux-gcc能翻译 ARM官方没有这条指令

nop -空指令 -> mov r0, r0

mov r0, #0xffffffff x

mvn r0, #0xf v

mov r0, #0xffffffff 伪指令v -> 自动翻译: mvn r0, #0xf

mov r0, #10000 x

ldr r0, =10000 v 伪指令

.global _start 定义一个全局符号

.space 32 开辟内存空间 32字节 地址?

buf: .space 32 开辟内存空间 32字节 地址: buf相当于首地址

ldr r0, =buf r0 = buf, r0存入空间的首地址

ldr r0, buf r0 = *buf, r0存入buf地址中的内容

.word 开辟空间的同时存入数据

.byte

num: .word 12 int num = 12;

num: .word 12,13,14,15 int num[4] = {12,13,14,15};

buf: .byte 1,2,3,4 char buf[4] = {1,2,3,4};

num: .word 22,66,11,88,33,99,44,55

.global _start

_start:

nop

nop

nop

.space 32

nop

nop

寻址方式

寻址：拿到数据的方式

立即数寻址: mov r0, #4

寄存器寻址: mov r1, r0

寄存器偏移寻址: mov r1, r0, lsl #3

寄存器间接寻址: ldr r0, [r1]

基址变址寻址: ldr r0, [r1, #4]

堆栈寻址: ldmfd sp, {r0-r12}

相对寻址: b label

异常

寄存器

arm模式：8种

- user** : 一般程序需要运行在**user**模式
- IRQ** : 有一个低优先级中断产生
- FIQ** : 有一个高优先级中断产生
- SVC** : 复位
- Abort** : 出现存储异常
- Undef** : 出现未定义指令异常
- Monitor** : 安全扩展模式
- System** : 和**user**一致的特权模式

异常：cpu调度的方式 7种

异常一旦产生，就会切换对应的模式去做处理

1. 复位异常 **reset** : 从头开始执行，所有数据都刷新了，进入**SVC**模式
2. 未定义指令异常 **undefined instruction** : 执行未定义的指令，执行时产生异常，进入**Undef**模式
如: **user**模式下执行 **msr cpsr, r1 mrs r3, spsr** 指令
3. 软中断异常 **swi** : 指令本身产生的异常，执行时产生，进入**SVC**模式
4. 预取指异常 **prefetch abort** : 取值发生了异常, 不会影响正在执行的指令, 执行完后处理异常，进入**Abort**模式
5. 数据异常 **data abort** : 执行指令时用到的数据有问题, 会更新**PC**寄存器的值, 处理完异常后, **cpu**认为需要重新执行这条指令，进入**Abort**模式 如: **ldr r0, [r1]** 可能**r1**这个地址不存在
6. **IRQ**异常 **irq** : 中断，在执行指令时来了异常，会在语句执行完后再去处理异常，进入**IRQ**模式
7. **FIQ**异常 **fiq** : 中断，在执行指令时来了异常，会在语句执行完后再去处理异常，进入**FIQ**模式
中断响应尽可能的快

异常优先级:

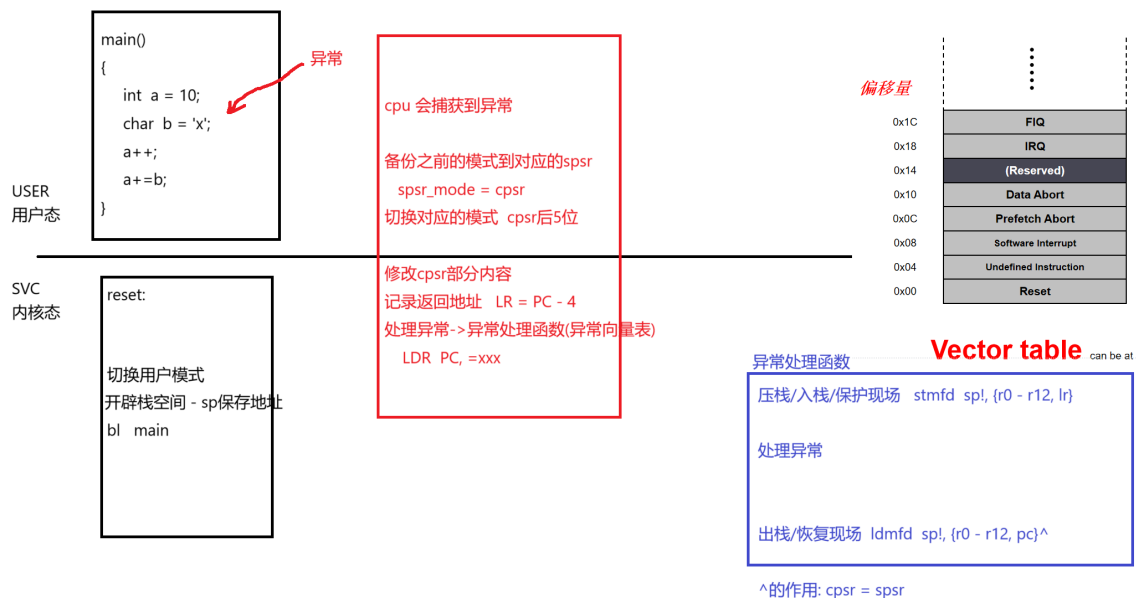
异常在当前指令执行完成之后才被响应

多个异常可以在同一时间产生

异常指定了优先级和固定的服务顺序:

- Reset**
- Data Abort**
- FIQ**
- IRQ**
- Prefetch Abort**
- SWI**
- Undefined instruction**

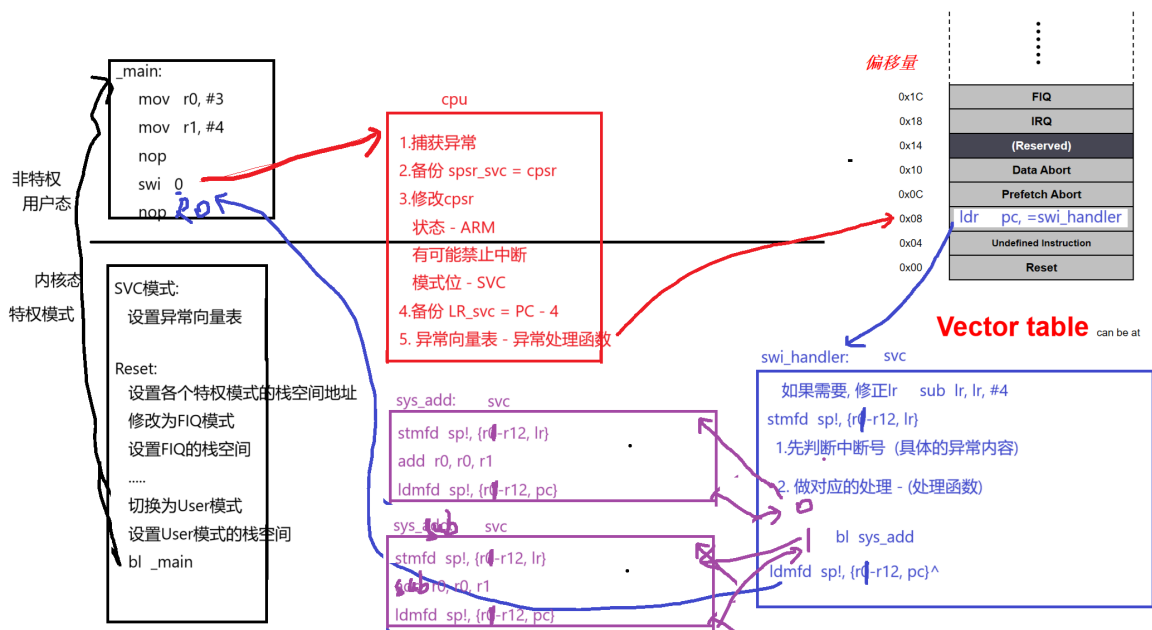
异常的处理:



FIQ 比 IRQ 快 的原因?

1. 优先级更高-响应快
2. FIQ有自己的R8-R12-效率高
3. FIQ位于异常向量表的最末端

SWI异常处理流程



```
.global _start
_start: b reset
    ldr pc, =_undef_handler
    ldr pc, =_swi_handler
    ldr pc, =_pre_handler
    ldr pc, =_data_handler
    ldr pc, =_reserved_handler
    ldr pc, =_irq_handler
    ldr pc, =_fiq_handler

_undef_handler: .word _undef_handler
_swi_handler: .word swi_handler
_pre_handler: .word _pre_handler
```

```

_data_handler: .word _data_handler
_reseerved_handler: .word _reseerved_handler
_irq_handler: .word _irq_handler
_fiq_handler: .word _fiq_handler

reset:
    /* set the cpu to SVC32 mode 禁止fiq irq*/
    mrs r0, cpsr
    bic r0, r0, #0x1f
    orr r0, r0, #0xd3
    msr cpsr,r0

    /* Set vector address in CP15 VBAR register */
    ldr r0, =_start
    mcr p15, 0, r0, c12, c0, 0 @Set VBAR

    @ 默认处于 SVC 模式 sp = 栈顶地址(地址大的一端)
    ldr sp, svc_stack_top

    @ 修改模式 fiq 10001: FIQ (cpsr后5位), 设置对应模式的栈地址
    mrs r0, cpsr
    bic r0, #0x1f
    orr r0, #0x11
    msr cpsr, r0
    @ ldr sp, fiq_stack_top

    @ 修改模式 10000: User
    mrs r0, cpsr
    bic r0, #0x1f
    orr r0, #0x10
    msr cpsr, r0
    ldr sp, user_stack_top

    bl _main

_main:
    stmfd sp!, {r1-r12, lr}
    mov r0, #3
    mov r1, #4
    nop
    swi 0
    nop
    loop:
        b loop
    ldmfd sp!, {r1-r12, pc}

swi_handler:
    stmfd sp!, {r1-r12, lr}

    ldr r4, [lr, #-4]
    bic r4, #0xff000000

    cmp r4, #0
    bleq sys_add

    cmp r4, #1

```



```
@ bleq sys_sub

cmp r4, #2
@ bleq xxx

ldmfd sp!, {r1-r12, pc}^

sys_add:
    stmfd sp!, {r1-r12, lr}
    add r0, r0, r1
    ldmfd sp!, {r1-r12, pc}

svc_stack: .space 128*4
svc_stack_top: .word svc_stack+128*4

user_stack: .space 128*4
user_stack_top: .word user_stack+128*4
```

cpu: lr = pc - 4

reset: b reset

swi: pc = lr

undef: pc = lr

data: lr = lr - 8 pc = lr

pre: lr = lr - 4 pc = lr

IRQ FIQ: lr = lr - 4 pc = lr

| | | | | | | | | |
|---|-------|----|----|----|----|----|----|----|
| 1 | 0x100 | 取指 | 译码 | 执行 | | | | |
| 2 | 0x104 | | 取指 | 译码 | 执行 | | | |
| 3 | 0x108 | | | 取指 | 译码 | 执行 | | |
| 4 | 0x10c | | | | 取指 | 译码 | 执行 | |
| 5 | 0x110 | | | | | 取指 | 译码 | 执行 |

异常处理函数中:
通过代码修正

练习

练习 - CMP

```
r0 r1 r2 中存入三个任意值

找到其中最大值存入 r3 寄存器
```

```

mov r0, #4
mov r1, #6
mov r2, #2

cmp r0, r1
movge r3, r0
movlt r3, r1

cmp r3, r2
movlt r3, r2

```

```

mov r0, #4
mov r1, #6
mov r2, #2

mov r3, r0

cmp r3, r1
movlt r3, r1
cmp r3, r2
movlt r3, r2

```

练习 - B

实现 $1+2+3+4+\dots+100$

```

sum = 0;
for(i=1; i<=100; i++)
{
    sum = sum + i;
}

r1 = 0;
for(r0=1; r0<=100; r0++)
{
    r1 = r1 + r0;
}

```

```

.global _start
_start:
    mov r0, #1
    mov r1, #0

loop:
    cmp r0, #100
    addle r1, r1, r0
    addle r0, r0, #1
    ble loop

    nop
    nop

```

练习:

1.思考 64 位数据的加减法如何实现?

2. r0 r1 r2 中存入三个任意值，将他们按照从小到大排序

尝试封装函数

思考：冒泡算法怎么写？ - 利用内存

label:

```
mov r0, #4
```

```
mov r1, #8
```

```
mov r2, #2
```

```
cmp r0, r1
```

```
movgt r4, r0
```

```
movgt r0, r1
```

```
movgt r1, r4
```

```
cmp    r1, r2
```

```
movgt r4, r1
```

```
movgt r1, r2
```

```
movgt r2, r4
```

```
mov    pc, 1r
```

[illegible]

冒泡排序思路：

```
num: .word 22,66,11,88,33,99,44,55
```

```
1dr r0, =num
```

```
ldr r1, [r0]    @22
```

```
ldr r2, [r0, #4] @66
```

```
cmp r1, r2
```

```
strgt r1, [r0, #4]
```

```
strgt r2, [r0]
```

接口编程

1. 点亮一个led灯 - 引脚 - 看原理图

高电平灯亮

LED2 GPX2_7

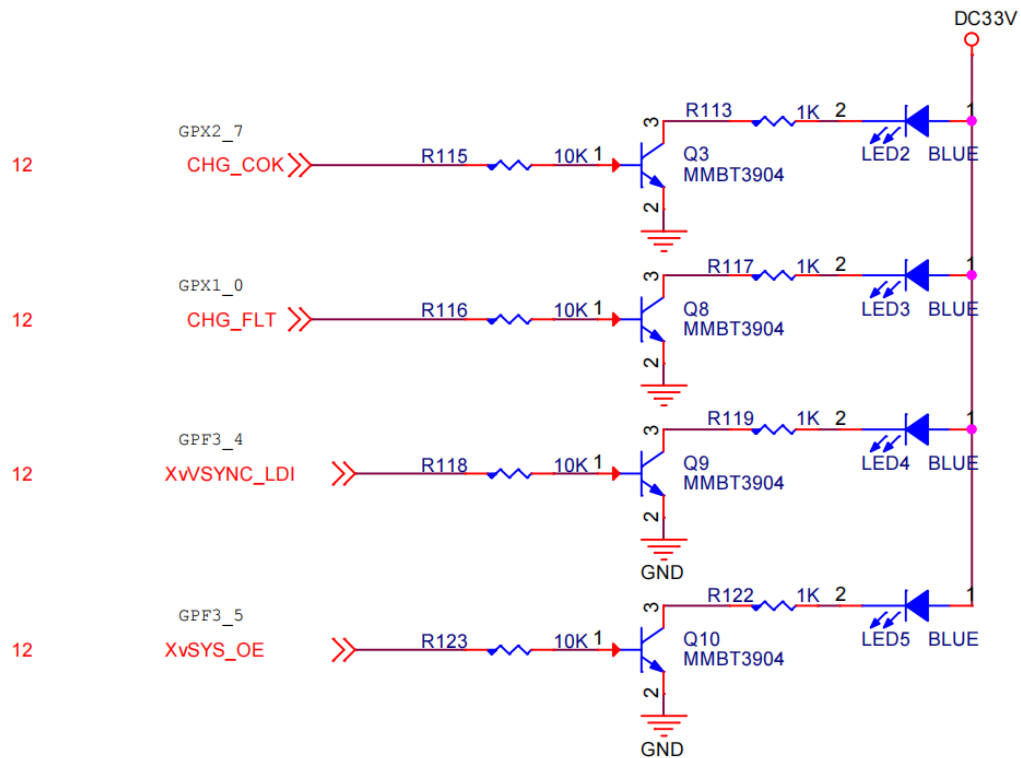
LED3 GPX1_0

LED4 GPF3_4

LED5 GPF3_5

2. 写代码 - 看用户手册

看懂 寄存器(SFR) -> 内存上的地址 -> 找准要操作bit位 -> 找准要写入的数据
确认寄存器是否是我们需要配置的



LED

6.2.2.61 GPF3CON

- Base Address: 0x1140_0000
- Address = Base Address + 0x01E0, Reset Value = 0x0000_0000

| Name | Bit | Type | Description | Reset Value |
|------------|---------|------|---|-------------|
| GPF3CON[5] | [23:20] | RW | 0x0 = Input 0x1 = Output 0x2 = SYS_OE 0x3 to 0xE = Reserved 0xF = EXT_INT16[5] | 0x00 |
| GPF3CON[4] | [19:16] | RW | 0x0 = Input 0x1 = Output 0x2 = VSYNC_LDI 0x3 to 0xE = Reserved 0xF = EXT_INT16[4] | 0x00 |

6.2.2.62 GPF3DAT

- Base Address: 0x1140_0000
- Address = Base Address + 0x01E4, Reset Value = 0x00

| Name | Bit | Type | Description | Reset Value |
|--------------|-------|------|--|-------------|
| GPF3DAT[5:0] | [5:0] | RWX | When you configure port as input port then corresponding bit is pin state. When configuring as output port the pin state should be same as the corresponding bit. When the port is configured as functional pin, the undefined value will be read. | 0x00 |

[illegible]

汇编代码:

```
ldr r0, =0x114001e0
mov r1, #0x10000
str r1, [r0]

mov r1, #0x10
str r1, [r0, #4]
```

```
@ LED2      GPX2_7
ldr r0, =0x11000c40
ldr r1, [r0]
bic r1, #0xf0000000
orr r1, #0x10000000
str r1, [r0]

ldr r0, =0x11000c44
ldr r1, [r0]
orr r1, #0x80      @ bic r1, #0x80
str r1, [r0]

@ LED3      GPX1_0
ldr r0, =0x11000c20
ldr r1, [r0]
bic r1, #0xf
orr r1, #0x1
str r1, [r0]

ldr r0, =0x11000c24
ldr r1, [r0]
orr r1, #0x01      @ bic r1, #0x01
str r1, [r0]
```

```

@ LED4      GPF3_4
ldr r0, =0x114001e0
ldr r1, [r0]
bic r1, #0xf0000
orr r1, #0x10000
str r1, [r0]

ldr r0, =0x114001e4
ldr r1, [r0]
orr r1, #0x10    @ bic r1, #0x10
str r1, [r0]

@ LED5      GPF3_5
ldr r0, =0x114001e0
ldr r1, [r0]
bic r1, #0xf00000
orr r1, #0x100000
str r1, [r0]

ldr r0, =0x114001e4
ldr r1, [r0]
orr r1, #0x20    @ bic r1, #0x20
str r1, [r0]

```

```

_main:
    @ LED5      GPF3_5
    ldr r0, =0x114001e0
    ldr r1, [r0]
    bic r1, #0xf00000
    orr r1, #0x100000
    str r1, [r0]

    ldr r0, =0x114001e4
    ldr r1, [r0]
    bic r1, #0x20
    str r1, [r0]

    while:
        ldr r0, =0x114001e4
        ldr r1, [r0]
        orr r1, #0x20    @灯亮
        str r1, [r0]

        bl delay

        ldr r0, =0x114001e4
        ldr r1, [r0]
        bic r1, #0x20    @灯灭
        str r1, [r0]

        bl delay

        b while

delay:

```

```
stmfd sp!, {lr}
mov r0, #0xff00000
loop:
    subs r0, r0, #1
    bge loop
ldmfd sp!, {pc}
```

开发板操作:

接好线后(接开发板中间的串口), 打开win7超级终端
 开发板断电, 重新上电的同时, 敲回车, 进入UBOOT启动界面
 输入 **loadb** 准备下载文件
 点击传送 -> 发送文件-> 选择待传送文件(如**test.bin**) -> 注意改变协议为**kermit** -> 发送
 等待下载完成
 输入 **go 43e00000** //去到43e00000内存地址运行代码

蜂鸣器

有源蜂鸣器: 通电就响 GPIO-高电平
 无源蜂鸣器: 有振荡才响 GPIO-高低高低切变-频率[20~20000Hz]

GPD0_0 XpwmTOUT0/LCD_FRM/GPD0_0

无源蜂鸣器: 有振荡才响 GPIO-高低高低切变-频率[20~20000Hz] - PWM TIMER

PCLK : system_clock : 100MHz

一级预分频: 8位 0~255

二级预分频: 2 4 8 16

TCNTBn : 寄存器: 存放计数值 - 递减计数

定时器输出PWM频率: 1000Hz

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--------|----------------|----|----|----|----|----|----|----|-------|----|----|-------------|-------------|----|----|----|------------------------|----|----|----|-------------|----|---|---|------------------|------|-----|--------|-------------|---|------|-----|---------|
| TCFG0 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| | 保留位 | | | | | | | | 死区长度x | | | | | | | | 预分频器 Timer 2 3 4 x | | | | | | | | 预分频器 Timer 0 1 v | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 ~ 255 |
| TCFG1 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| | 保留位 | | | | | | | | | | | | MUX4(Tim 4) | | | | MUX3(Tim 3) | | | | MUX3(Tim 2) | | | | MUX1(Tim 1) | | | | MUX0(Tim 0) | | | | |
| | | | | | | | | | | | | | 0000 = 1/1 | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | 0001 = 1/2 | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | 0010 = 1/4 | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | 0011 = 1/8 | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | 0100 = 1/16 | | | | | | | | | | | | | | | | | | | | | |
| TCON | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| | 保留位 | | | | | | | | | | | | 保留位 | | | | | | | | | | | | 死区 | auto | 反相器 | manual | start/stop | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 1 | 0 | 1->0 | 1/0 | |
| TCNTB0 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| | count 计数值 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | 0 ~ 0xffffffff | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| TCMPB0 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| | compare 比较值 | | | | | | | | | | | | | | | | CNT == CMP -> 输出电平进行翻转 | | | | | | | | | | | | | | | | |
| | 0 ~ count | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

必须预 uart

后续 RTC WDT ADC

