

Uboot中start.S源码的指令级的详尽解析

目录

1. 正文之前.....	3
1.1. 本文内容.....	3
1.2. 本文目标.....	3
1.3. 代码来源.....	3
1.4. 关于本文内容的组织形式.....	3
1.5. 阅读此文所要具有的前提知识.....	3
1.6. 声明.....	3
2. start.S 详解	4
2.1. 设置 CPU 模式	4
2.2. 关闭看门狗.....	16
2.3. 关闭中断.....	20
2.4. 设置堆栈 sp 指针	24
2.5. 清除 bss 段.....	32
2.6. 异常中断处理.....	44
3. 相关知识点详解.....	57
3.1. uboot 初始化中, 为何要设置 CPU 为 SVC 模式而不是设置为其他模式.....	57
3.2. 什么是 watchdog + 为何在要系统初始化的时候关闭 watchdog.....	58
3.2.1. 什么是 watchdog	58
3.2.2. 为何在要系统初始化的时候关闭 watchdog.....	59
3.3. 为何 ARM7 中 PC=PC+8	59
3.3.1. 为何 ARM9 和 ARM7 一样, 也是 PC=PC+8.....	60
3.4. AMR 寄存器的别名 + APCS	65
3.4.1. ARM 中的寄存器的别名.....	65
3.4.2. 什么是 APCS.....	67
3.5. 为何 C 语言 (的函数调用) 需要堆栈, 而汇编语言却不需要堆栈.....	67
3.5.1. 保存现场/上下文.....	67
3.5.1.1. 什么叫做上下文 context	68
3.5.2. 传递参数.....	68
3.5.3. 举例分析 C 语言函数调用是如何使用堆栈的	68
3.6. 关于为何不直接用 mov 指令, 而非要用 adr 伪指令.....	70
3.7. mov 指令的操作数的取值范围到底是多少.....	71
4. 引用.....	74

图表

图表 1 global 的语法.....	5
图表 2 LDR 指令的语法.....	7
图表 3 .word 的语法	8
图表 4 balignl 的语法.....	9
图表 5 CPSR/SPSR 的位域结构.....	13
图表 6 pWTCN.....	16
图表 7 INTMOD.....	16
图表 8 INTMSK	17
图表 9 INTSUBMSK.....	17
图表 10 CLKDIVN.....	17
图表 11 WTCN 寄存器的位域	20
图表 12 INTMSK 寄存器的位域	21
图表 13 INTSUBMSK 寄存器的位域.....	22
图表 14 CLKDIVN 的位域.....	23
图表 15 控制寄存器 1 的位域含义.....	40
图表 16 时钟模式.....	41
图表 17 关于访问控制位在域访问控制寄存器中的含义	41
图表 18 关于访问允许(AP)位的含义	42
图表 19 macro 的语法.....	45
图表 20 LDM/STM 的语法	46
图表 21 条件码的含义.....	46
图表 22 ARM 中 CPU 的模式.....	57
图表 23 AMR7 三级流水线.....	59
图表 24 ARM7 三级流水线状态.....	60
图表 25 ARM7 三级流水线示例	60
图表 26 ARM7 三级流水线 vs ARM9 五级流水线.....	61
图表 27 ARM7 三级流水线到 ARM9 五级流水线的映射	61
图表 28 ARM9 的五级流水线示例.....	62
图表 29 ARM9 五级流水线的执行机构的细节.....	65
图表 30 ARM 寄存器的别名.....	66
图表 31 数据处理指令的指令格式.....	72
图表 32 mov 指令 0xe3a00453 的位域含义解析	73

版本历史:

版本	时间	内容
1.0	2011-04-17	详细解释了 uboot 的 start.s 中的每行代码 添加了相关知识点的详细解释

1. 正文之前

1.1. 本文内容

此文主要内容就是分析start.S这个汇编文件的内容，即ARM上电后的最开始那一段的启动过程。

1.2. 本文目标

本文的目标是，希望看完此文的读者，可以达到：

微观上，对此start.S的每一行，都有了基本的了解。

宏观上，对基于ARM核的S3C24X0的CPU的启动过程，有更加清楚的概念。

这样的目的，是为了读者看完本文后，再去看其他类似的启动相关的源码，能明白需要做什么事情，然后再看别的系统是如何实现相关的内容的，达到一定程度的触类旁通。

总体说就是，要做哪些，为何要这么做，如何实现的，即英语中常说的：do what, why do, how do, 此三方面都清楚理解了，那么也才能算真正懂了。

1.3. 代码来源

所用代码来自TQ2440官网，天嵌的bbs上下载下来的uboot中的源码：

[u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\cpu\arm920t\start.S](http://u-boot-1.1.6_20100601/opt/EmbedSky/u-boot-1.1.6/cpu/arm920t/start.S)

下载地址为：

2010年6月 最新TQ2440光盘下载（Linux内核，WinCE的eboot，uboot均有更新）

<http://bbs.embedsky.net/viewthread.php?tid=859>

1.4. 关于本文内容的组织形式

1. 类似于这样的代码框：

Start.S的代码。。。

中的内容，是源文件start.S的汇编代码，紧接着代码框的内容，是代码的解释。

2. 在

和

之间的代码，是转自别的地方，非start.S的代码。

1.5. 阅读此文所要具有的前提知识

阅读此文之前，你至少要对TQ2440的板子有个基本的了解，

以及要了解开发板初始化的大概要做的事情，比如设置输入频率，设置堆栈等等。

另外，至少要有一定的C语言的基础，这样更利于理解汇编代码。

1.6. 声明

由于水平有限，难免有误，欢迎指正： [green-waste \(at\) 163.com](mailto:green-waste@163.com)

欢迎转载，但请注明作者。

2. start.S 详解

下面将详细解释 uboot 中的 start.S 中的每一行代码。详细到，每个指令的语法和含义，都进行详细讲解，使得此文读者可以真正搞懂具体的含义，即 what。

以及对于一些相关的问题，深入探究为何要这么做，即 why。

对于 uboot 的 start.S，主要做的事情就是系统的各个方面的初始化。

从大的方面分，可以分成这几个部分：

- (1) 设置 CPU 模式
- (2) 关闭看门狗
- (3) 关闭中断
- (4) 设置堆栈 sp 指针
- (5) 清除 bss 段
- (6) 异常中断处理

下面来对 start.S 进行详细分析，看看每一个部分，是如何实现的。

2.1. 设置 CPU 模式

```
/*
 * armboot - Startup Code for ARM920 CPU-core
 *
 * Copyright (c) 2001   Marius Gröber <mag@sysgo.de>
 * Copyright (c) 2002   Alex Zinke <azu@sysgo.de>
 * Copyright (c) 2002   Gary Jennejohn <gj@denx.de>
 *
 * See file CREDITS for list of people who contributed to this
 * project.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
```

```
* MA 02111-1307 USA
*/

#include <config.h>
#include <version.h>

/*

*****
*
* Jump vector table as in table 3.1 in [1]
*
*****
*/

.globl _start
```

globl是个关键字，对应含义为：
<http://re-eject.gbadev.org/files/GasARMRef.pdf>
“

图表 1 global 的语法

Directive	Description	Syntax	Example
.global	Makes <u>symbol</u> visible to the linker	.global symbol	.global MyAsmFunc
.globl	Same as .global	.globl symbol	.globl MyOtherAsmFunc

“
所以，意思很简单，就是相当于C语言中的Extern，声明此变量，并且告诉链接器此变量是全局的，外部可以访问，所以，你可以看到：
[u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\u-boot.lds](#)
中，有用到此变量：
ENTRY(_start)
即指定入口为_start,而由下面的_start的含义可以得知，_start就是整个start.S的最开始，即整个uboot的代码的开始。

```
_start:    b        reset
```

_start后面加上一个冒号':',表示其是一个标号Label，类似于C语言goto后面的标号。而同时，_start的值，也就是这个代码的位置了，此处即为代码的最开始，相对的0的位置。而此处最开始的对的0位置，在程序开始运行的时候，如果是从NorFlash启动，那么其地址是0，

_stat=0

如果是重新relocate代码之后，就是我们定义的值了，即，在

[u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\config.mk](#)

中的：

TEXT_BASE = 0x33D00000

表示是代码段的基地址，即

_start=TEXT_BASE=0x33D00000

关于标号的语法解释：

<http://sourceware.org/binutils/docs-2.20/as/Labels.html#Labels>

"A label is written as a symbol immediately followed by a colon `:`. The symbol then represents the current value of the active location counter, and is, for example, a suitable instruction operand. You are warned if you use the same symbol to represent two different locations: the first definition overrides any other definitions. "

而_start标号后面的：

b reset

就是跳转到对应的标号为reset的位置。

```
ldr pc, _undefined_instruction
ldr pc, _software_interrupt
ldr pc, _prefetch_abort
ldr pc, _data_abort
ldr pc, _not_used
ldr pc, _irq
ldr pc, _fiq
```

ldr命令的语法为：

http://infocenter.arm.com/help/topic/com.arm.doc.dui0206hc/DUI0206HC_rvct_linker_and_utilities_guide.pdf

图表 2 LDR 指令的语法

4.1.20 LDR

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond	0	1	I	P	U	0	W	1	Rn	Rd	addr_mode				

The LDR (Load Register) instruction loads a word from the memory address calculated by `<addressing_mode>` and writes it to register `<Rd>`. If the address is not word-aligned, the loaded value is rotated right by 8 times the value of bits[1:0] of the address. For a little-endian memory system, this rotation causes the addressed byte to occupy the least significant byte of the register. For a big-endian memory system, it causes the addressed byte to occupy bits[31:24] or bits[15:8] of the register, depending on whether bit[0] of the address is 0 or 1 respectively.

If the PC is specified as register `<Rd>`, the instruction loads a data word which it treats as an address, then branches to that address. In ARM architecture version 5 and above, bit[0] of the loaded value determines whether execution continues after this branch in ARM state or in Thumb state, as though a BX (loaded_value) instruction had been executed. In earlier versions of the architecture, bits[1:0] of the loaded value are ignored and execution continues in ARM state, as though a MOV PC, (loaded_value) instruction had been executed.

Syntax

LDR{<cond>} <Rd>, <addressing_mode>

where:

<cond> Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If **<cond>** is omitted, the AL (always) condition is used.

<Rd> Specifies the destination register for the loaded value.

<addressing_mode>

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of **<addressing_mode>** includes a base register `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

<http://wenku.baidu.com/view/f7cc280102020740bele9bea.html>

“LDR指令的格式为:

LDR{ 条件 } 目的寄存器, <存储器地址>

LDR指令用于从存储器中将一个32位的字数据传送到目的寄存器中。该指令通常用于从存储器中读取32位的字数据到通用寄存器, 然后对数据进行处理。当程序计数器PC作为目的寄存器时, 指令从存储器中读取的字数据被当作目的地址, 从而可以实现程序流程的跳转。该指令在程序设计中比较常用, 且寻址方式灵活多样, 请读者认真掌握。

指令示例:

LDR R0, [R1] ; 将存储器地址为R1的字数据读入寄存器R0。

LDR R0, [R1, R2] ; 将存储器地址为R1+R2的字数据读入寄存器R0。

LDR R0, [R1, #8] ; 将存储器地址为R1+8的字数据读入寄存器R0。

LDR R0, [R1, R2]! ; 将存储器地址为R1+R2的字数据读入寄存器R0, 并将新地址R1+R2写入R1。

LDR R0, [R1, #8]! ; 将存储器地址为R1+8的字数据读入寄存器R0, 并将新地址R1+8写入R1。

LDR R0, [R1], R2 ; 将存储器地址为R1的字数据读入寄存器R0, 并将新地址R1+R2写入

R1。

LDR R0, [R1, R2, LSL#2]! ; 将存储器地址为 $R1 + R2 \times 4$ 的字数据读入寄存器R0, 并将新地址 $R1 + R2 \times 4$ 写入R1。

LDRR0, [R1], R2, LSL#2 ; 将存储器地址为R1的字数据读入寄存器R0, 并将新地址 $R1 + R2 \times 4$ 写入R1。”

<http://www.pczpg.com/a/2010/0607/11062.html>

“ARM是RISC结构, 数据从内存到CPU之间的移动只能通过L/S指令来完成, 也就是ldr/str指令。

比如想把数据从内存中某处读取到寄存器中, 只能使用ldr

比如:

```
ldr r0, 0x12345678
```

就是把0x12345678这个地址中的值存放到r0中。”

上面那些ldr的作用, 以第一个_undefined_instruction为例, 就是将地址为_undefined_instruction中的一个word的值, 赋值给pc。

```
_undefined_instruction: .word undefined_instruction
_software_interrupt: .word software_interrupt
_prefetch_abort: .word prefetch_abort
_data_abort: .word data_abort
_not_used: .word not_used
_irq: .word irq
_fiq: .word fiq
```

http://blogold.chinaunix.net/u3/115924/showart_2280163.html

“.word .word expr {,expr}... 分配一段字内存单元, 并用expr初始化字内存单元(32bit)”

<http://re-eject.gbadev.org/files/GasARMRef.pdf>

图表 3 .word 的语法

Directive	Description	Syntax	Example
.word	Define word <u>expr</u> (32bit numbers)	.word expr {, ...}	.word 144511, 0x11223

所以上面的含义, 以_undefined_instruction为例, 就是, 此处分配了一个word=32bit=4字节的地址空间, 里面存放的值是undefined_instruction。

而此处_undefined_instruction也就是该地址空间的地址了。用C语言来表达就是:

```
_undefined_instruction = &undefined_instruction
```

或

```
*_undefined_instruction = undefined_instruction
```

在后面的代码, 我们可以看到, undefined_instruction也是一个标号, 即一个地址值, 对应着就是在发生“未定义指令”的时候, 系统所要去执行的代码。

（其他几个对应的“软件中断”，“预取指错误”，“数据错误”，“未定义”，“（普通）中断”，“快速中断”，也是同样的做法，跳转到对应的位置执行对应的代码。）

所以：

```
ldr pc, 标号1
...
标号1: .word 标号2
```

标号2:
...（具体要执行的代码）

的意思就是，将地址为标号1中内容载入到pc，而地址为标号1中的内容，正好装的是标号2。

用C语言表达其实很简单：

```
PC = *(标号1) = 标号2
```

对PC赋值，即是实现代码跳转，所以整个这段汇编代码的意思就是：

跳转到标号2的位置，执行对应的代码。

```
.balignl 16,0xdeadbeef
```

balignl这个标号的语法及含义：
<http://re-eject.gbadev.org/files/GasARMRef.pdf>
“

图表 4 balignl 的语法

Directive	Description	Syntax	Example
.balignl	Word align the following code to <u>alignment</u> byte boundary (default=4). Fill skipped words with <u>fill</u> (default=0 or NOP). If the number of bytes skipped is greater than max, then don't align (default= <u>alignment</u>).	.balignl {alignment } {, fill} {, max}	.balignl

”
所以意思就是，接下来的代码，都要16字节对齐，不足之处，用0xdeadbeef填充。
其中0xdeadbeef其实没啥特别意思，就是要填充的内容，你把其换成其他的也行，当然最好也和0xdeadbeef一样是4个字节。

```
/*
*****
****
*
* Startup Code (reset vector)
*
* do important init only if we don't start from memory!
```

```
* relocate armboot to ram
* setup stack
* jump to second stage
*

*****
****
*/

_TEXT_BASE:
    .word TEXT_BASE
```

此处和上面的类似，_TEXT_BASE是一个标号地址，此地址中是一个word类型的变量，变量名是TEXT_BASE，此值见名知意，是text的base，即代码的基地址，可以在

[u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\config.mk](#)

中找到其定义：

```
TEXT_BASE = 0x33D00000
```

```
.globl _armboot_start
_armboot_start:
    .word _start
```

同理，此含义可用C语言表示为：

```
*( _armboot_start) = _start
```

```
/*
 * These are defined in the board-specific linker script.
 */
.globl _bss_start
_bss_start:
    .word __bss_start

.globl _bss_end
_bss_end:
    .word _end
```

关于_bss_start和_bss_end都只是两个标号，对应着此处的地址。

而两个地址里面分别存放的值是__bss_start和_end，这两个的值，根据注释所说，是定义在开发板相关的链接脚本里面的，我们此处的开发板相关的链接脚本是：

[u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\u-boot.lds](#)

其中可以找到__bss_start和_end的定义：

```
__bss_start = .;
.bss : { *(.bss) }
_end = .;
```

而关于_bss_start和_bss_end定义为.globl即全局变量，是因为uboot的其他源码中要用到这两个变量，详情请自己去搜索源码。

```
.globl FREE_RAM_END
FREE_RAM_END:
    .word 0x0badc0de

.globl FREE_RAM_SIZE
FREE_RAM_SIZE:
    .word 0x0badc0de
```

关于FREE_RAM_END和FREE_RAM_SIZE，这里只是两个标号，之所以也是声明为全局变量，是因为uboot的源码中会用到这两个变量。

但是这里有点特别的是，这两个变量，将在本源码start.S中的后面要用到，而在后面用到这两个变量之前，uboot的C源码中，会先去修改这两个值，具体的逻辑是：

本文件start.S中，后面有这两句：

```
ldr pc, _start_armboot
```

```
_start_armboot: .word start_armboot
```

意思很明显，就是去调用start_armboot函数。

而start_armboot函数是在：

[u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\lib_arm\board.c](#)

中：

```
init_fnc_t *init_sequence[] = {
    cpu_init,      /* basic cpu dependent setup */
    . . .
    NULL,
};

void start_armboot (void)
{
    init_fnc_t **init_fnc_ptr;
    . . .

    for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
        if ((*init_fnc_ptr)() != 0) {
            hang ();
        }
    }
    . . .
}
```

即在start_armboot去调用了cpu_init。

cpu_init函数是在:

u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\cpu\arm920t\cpu.c

中:

cpu_init源码

```
int cpu_init (void)
{
    /*
     * setup up stacks if necessary
     */
#ifdef CONFIG_USE_IRQ
    IRQ_STACK_START = _armboot_start - CFG_MALLOC_LEN - CFG_GBL_DATA_SIZE
- 4;
    FIQ_STACK_START = IRQ_STACK_START - CONFIG_STACKSIZE_IRQ;
    FREE_RAM_END = FIQ_STACK_START - CONFIG_STACKSIZE_FIQ -
CONFIG_STACKSIZE;
    FREE_RAM_SIZE = FREE_RAM_END - PHYS_SDRAM_1;
#else
    FREE_RAM_END = _armboot_start - CFG_MALLOC_LEN - CFG_GBL_DATA_SIZE
- 4 - CONFIG_STACKSIZE;
    FREE_RAM_SIZE = FREE_RAM_END - PHYS_SDRAM_1;
#endif
    return 0;
}
```

在cpu_init中, 根据我们的一些定义, 比如堆栈大小等等, 去修改了IRQ_STACK_START , FIQ_STACK_START , FREE_RAM_END和FREE_RAM_SIZE的值。
至于为何这么修改, 后面遇到的时候会具体再解释。

```
#ifdef CONFIG_USE_IRQ
/* IRQ stack memory (calculated at run-time) */
.globl IRQ_STACK_START
IRQ_STACK_START:
    .word 0x0badc0de

/* FIQ stack memory (calculated at run-time) */
.globl FIQ_STACK_START
FIQ_STACK_START:
    .word 0x0badc0de
#endif
```

同上, IRQ_STACK_START和FIQ_STACK_START, 也是在cpu_init中用到了。

不过此处，是只有当定义了宏CONFIG_USE_IRQ的时候，才用到这两个变量，其含义也很明显，只有用到了中断IRQ，才会用到中断的堆栈，才有中端堆栈的起始地址。快速中断FIQ，同理。

```
/*
 * the actual reset code
 */

reset:
    /*
     * set the cpu to SVC32 mode
     */
    mrs r0,cpsr
```

CPSR:

CPSR 是当前的程序状态寄存器 (Current Program Status Register)，
而 SPSR 是保存的程序状态寄存器 (Saved Program Status Register)。
具体细节为：

[ARM7体系结构](#)

<http://www.docin.com/p-73665362.html>

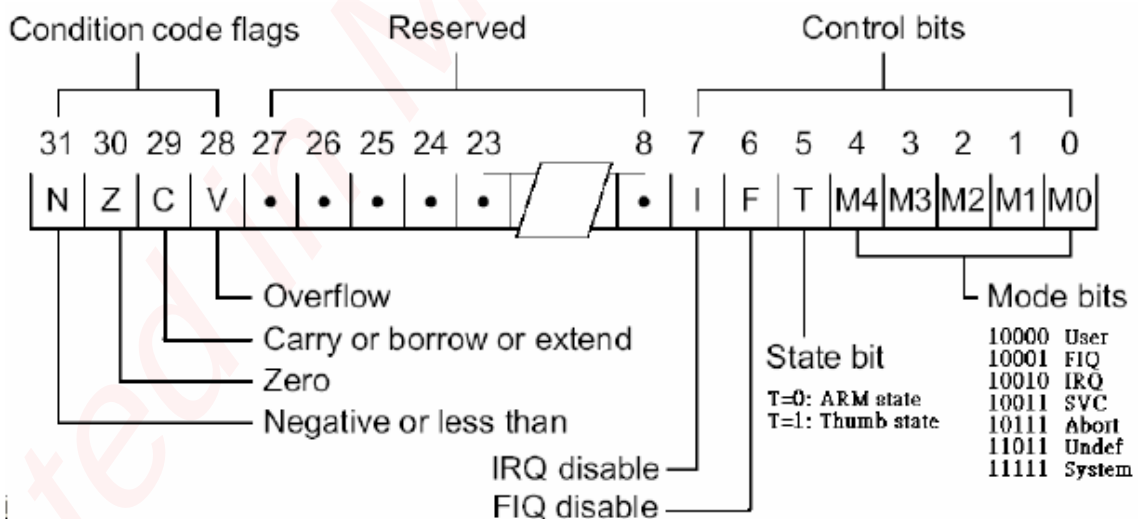
或

<http://www.csie.nctu.edu.tw/~wjtsai/EmbeddedSystemDesign/Ch2-bootloader.pdf>

图表 5 CPSR/SPSR 的位域结构

• Program Status Register (PSR)

- **CPSR**: Current Program Status Register
- **SPSR**: Saved Program Status Register



或：

31 30 29 28 --- 7 6 - 4 3 2 1 0

N	Z	C	V	I	F	M4	M3	M2	M1	M0	
0	0	0	0	0	0	0	0	0	0	0	User26 模式
0	0	0	0	0	1	0	0	0	0	1	FIQ26 模式
0	0	0	0	1	0	0	0	1	0	0	IRQ26 模式
0	0	0	0	1	1	0	0	1	1	0	SVC26 模式
1	0	0	0	0	0	1	0	0	0	0	User 模式
1	0	0	0	0	1	1	0	0	0	1	FIQ 模式
1	0	0	1	0	0	1	0	0	1	0	IRQ 模式
1	0	0	1	1	1	1	0	0	1	1	svc 模式
1	0	1	1	1	1	1	1	0	1	1	ABT 模式
1	1	0	1	1	1	1	1	1	1	1	UND 模式

MRS:

MRS - Move From Status Register

MRS指令的语法为:

“四、程序状态寄存器访问指令

1、 MRS指令

MRS指令的格式为:

MRS{条件} 通用寄存器, 程序状态寄存器 (CPSR或SPSR)

MRS指令用于将程序状态寄存器的内容传送到通用寄存器中。该指令一般用在以下两种情况:

I . 当需要改变程序状态寄存器的内容时, 可用MRS将程序状态寄存器的内容读入通用寄存器, 修改后再写回程序状态寄存器。

II . 当在异常处理或进程切换时, 需要保存程序状态寄存器的值, 可先用该指令读出程序状态寄存器的值, 然后保存。

指令示例:

MRS R0, CPSR ; 传送CPSR的内容到R0

MRS R0, SPSR ; 传送SPSR的内容到R0”

所以, 上述汇编代码含义为, 将CPSR的值赋给R0寄存器。

```
bic r0,r0,#0x1f
```

bic指令的语法是:

“16、BIC指令

BIC指令的格式为:

BIC{条件}{S} 目的寄存器, 操作数1, 操作数2

BIC指令用于清除操作数1的某些位, 并把结果放置到目的寄存器中。操作数1应是一个寄存器, 操作数2可以是一个寄存器, 被移位的寄存器, 或一个立即数。操作数2为32位的掩码, 如果在掩码中设置了某一位, 则清除这一位。未设置的掩码位保持不变。”

而0x1f=11111b

所以, 此行代码的含义就是, 清除r0的bit[4:0]位。

```
orr r0,r0,#0xd3
```

orr指令的语法是:

“14、ORR指令

ORR指令的格式为：

ORR{条件}{S} 目的寄存器，操作数1，操作数2

ORR指令用于在两个操作数上进行逻辑或运算，并把结果放置到目的寄存器中。操作数1应是一个寄存器，操作数2可以是一个寄存器，被移位的寄存器，或一个立即数。该指令常用于设置操作数1的某些位。

指令示例：

ORR R0, R0, #3 ; 该指令设置R0的0、1位，其余位保持不变。”

所以此行汇编代码的含义为：

而0xd3=1101 0111

将r0与0xd3算数或运算，然后将结果给r0,即把r0的bit[7:6]和bit[4]和bit[2:0]置为1。

```
msr cpsr,r0
```

MSR - Move to Status Register

msr的指令格式是：

“四、程序状态寄存器访问指令

。 。 。

2、 MSR指令

MSR指令的格式为：

MSR{条件} 程序状态寄存器（CPSR或SPSR）_<域>，操作数

MSR指令用于将操作数的内容传送到程序状态寄存器的特定域中。其中，操作数可以为通用寄存器或立即数。<域>用于设置程序状态寄存器中需要操作的位，32位的程序状态寄存器可分为4个域：

位[31: 24]为条件标志位域，用f表示；

位[23: 16]为状态位域，用s表示；

位[15: 8]为扩展位域，用x表示；

位[7: 0]为控制位域，用c表示；

该指令通常用于恢复或改变程序状态寄存器的内容，在使用时，一般要在MSR指令中指明将要操作的域。

指令示例：

MSR CPSR, R0 ; 传送R0的内容到CPSR

MSR SPSR, R0 ; 传送R0的内容到SPSR

MSR CPSR_c, R0 ; 传送R0的内容到SPSR，但仅仅修改CPSR中的控制位域

”

此行汇编代码含义为，将r0的值赋给CPSR。

【总结】

所以，上面四行汇编代码的含义就很清楚了。

先是把CPSR的值放到r0寄存器中，然后清楚bit[4:0]，然后再或上

0xd3=11 0 10111b

CPSR位域	7	6	-	4	3	2	1	0
	I	F		M4	M3	M2	M1	M0

0xd3= 1 1 1 0 0 1 1 svc 模式

即:

bit[7]=1 -> 设置I位为1 -> 关闭中断IRQ

bit[6]=1 -> 设置F位为1 -> 关闭快速中断FIQ

bit[4:0]=10011b -> 设置CPU为svc模式, 这和上面代码注释中的“set the cpu to SVC32 mode”, 也是一致的。

2.2. 关闭看门狗

```
/* turn off the watchdog */
#ifdef CONFIG_S3C2400
# define pWTCON      0x15300000
# define INTMSK      0x14400008 /* Interrupt-Controller base addresses */
# define CLKDIVN     0x14800014 /* clock divisor register */
#elif defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
# define pWTCON      0x53000000
# define INTMOD      0x4A000004
# define INTMSK      0x4A000008 /* Interrupt-Controller base addresses */
# define INTSUBMSK    0x4A00001C
# define CLKDIVN     0x4C000014 /* clock divisor register */
#endif
```

上面几个宏定义所对应的地址, 都可以在对应的datasheet中找到对应的定义:

其中, S3C2410和TQ2440开发板所用的CPU S3C2440, 两者在这部分的寄存器定义, 都是一样的, 所以此处, 采用CONFIG_S3C2410所对应的定义。

关于**S3C2440**相关的**软硬件资料**, 这个网站提供的很全面:

<http://just4you.springnote.com/pages/1052612>

其中有S3C2440的CPU的datasheet:

s3c2440a_um_rev014_040712.pdf

其中有对应的寄存器定义:

图表 6 pWTCON

Watchdog Timer					
WTCON	0x53000000	←	W	R/W	Watchdog Timer Mode
WTDAT	0x53000004				Watchdog Timer Data
WTCNT	0x53000008				Watchdog Timer Count

图表 7 INTMOD

Register	Address	R/W	Description	Reset Value
INTMOD	0X4A000004	R/W	Interrupt mode register. 0 = IRQ mode 1 = FIQ mode	0x00000000

图表 8 INTMSK

Register	Address	R/W	Description	Reset Value
INTMSK	0X4A000008	R/W	Determine which interrupt source is masked. The masked interrupt source will not be serviced. 0 = Interrupt service is available. 1 = Interrupt service is masked.	0xFFFFFFFF

图表 9 INTSUBMSK

Register	Address	R/W	Description	Reset Value
INTSUBMSK	0X4A00001C	R/W	Determine which interrupt source is masked. The masked interrupt source will not be serviced. 0 = Interrupt service is available. 1 = Interrupt service is masked.	0xFFFF

图表 10 CLKDIVN

Register Name	Address (B. Endian)	Address (L. Endian)	Acc. Unit	Read/Write	Function
Clock & Power Management					
LOCKTIME	0x4C000000	←	W	R/W	PLL Lock Time Counter
MPLLCON	0x4C000004				MPLL Control
UPLLCON	0x4C000008				UPLL Control
CLKCON	0x4C00000C				Clock Generator Control
CLKSLOW	0x4C000010				Slow Clock Control
CLKDIVN	0x4C000014				Clock divider Control
CAMDIVN	0x4C000018				Camera Clock divider Control

而关于每个寄存器的具体含义，见后面的分析。

```
#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410) ||
defined(CONFIG_S3C2440)
    ldr    r0, =pWTCON
```

```
ldr    r0, =pWTCON
```

这里的ldr和前面介绍的ldr指令不是一个意思。

这里的ldr是伪指令ldr。

【什么是伪指令】

伪指令，就是“伪”的指令，是针对“真”的指令而言的。

真的指令就是那些常见的指令，比如上面说的arm的ldr, bic, msr等等指令，是arm体系架构中真正存在的指令，你在arm汇编指令集中找得到对应的含义。

而伪指令是写出来给汇编程序看的，汇编程序能看的伪指令具体表示的是啥意思，然后将其翻译成真正的指令或者进行相应的处理。

伪指令ldr语法和含义：

<http://blog.csdn.net/lihaoweiV/archive/2010/11/24/6033003.aspx>

“另外还有一个就是ldr伪指令，虽然ldr伪指令和ARM的ldr指令很像，但是作用不太一样。

ldr伪指令可以在立即数前加上=，以表示把一个地址写到某寄存器中，比如：

```
ldr r0, =0x12345678
```

这样，就把 0x12345678 这个地址写到 r0 中了。所以，ldr 伪指令和 mov 是比较相似的。”

只不过 mov 指令后面的立即数是有限制的，这个立即数，能够**必须由一个 8 位的二进制数，即属于 0x00-0xFF 内的某个值，经过偶数次右移后得到，这样才是合法数据**，而 ldr 伪指令没有这个限制。

那为何 ldr 伪指令的操作数没有限制呢，那是因为其是伪指令，写出来的伪指令，最终会被编译器解释成为真正的，合法的指令的，一般都是对应的 mov 指令。

这样的话，写汇编程序的时候，使用 MOV 指令是比较麻烦的，因为有些简单的数据比较容易看出来，有些数据即不容易看出来是否是合法数据。所以，对此，ldr 伪指令的出现，就是为了解决这个问题的，你只管放心用 ldr 伪指令，不用关心操作数，而写出的 ldr 伪指令，编译器会帮你翻译成对应的真正的汇编指令的。

而关于编译器是如何将这些ldr伪指令翻译成为真正的汇编指令的，我的理解是，其会自动去算出来对应的操作数，是否是合法的mov 的操作数，如果是，就将该ldr伪指令翻译成mov指令，否则就用别的方式处理，我所观察到的，其中一种方式就是，单独申请一个4字节的空间用于存放操作数，然后用ldr指令实现。

在uboot中，最后make完毕之后，会生产u-boot，通过：

```
arm-linux-objdump -d u-boot > dump_u-boot.txt
```

就可以把对应的汇编代码输出到该txt文件了，其中就能找到伪指令：

```
ldr    r0, =0x53000000
```

所对应的，真正的汇编代码：

```
33d00068: e3a00453  mov r0, #1392508928 ; 0x53000000
```

这里伪指令ldr的操作数，0x53000000就是有效的mov的指令的操作数，

所以被翻译成了mov指令。

而经过我的尝试，故意将0x53000000改为0x53000010，对应的生产的汇编代码为：

```
33d00068: e59f0408  ldr r0, [pc, #1032] ; 33d00478 <fiq+0x58>
```

```
...
```

```
33d00478: 53000010  .word 0x53000010
```

其中可以看到，由于0x53000010不是有效的mov的操作数，没法找到合适的0x00-0xFF去通过偶数次循环右移而得到，所以只能换成此处这种方式，即在另外申请一个word的空间用于存放这个值：

```
33d00478: 53000010  .word 0x53000010
```

然后通过计算出相对当前PC的偏移，得到的地址，用ldr指令去除该地址中的值，即

0x53000010，送给r0，比起mov指令，要复杂的多，也多消耗了一个word的空间。

对应地，其他方式，个人理解，好像也可以通过MVN指令来实现，具体细节，有待进一步探索。

【总结】

而这里的：

```
ldr    r0, =pWTCN
```

意思就很清楚了，就是把宏pWTCON的值赋值给r0寄存器，即

r0=0x53000000

```
mov    r1, #0x0
```

mov指令语法：

“1、MOV指令

MOV指令的格式为：

MOV{条件}{S} 目的寄存器，源操作数

MOV指令可完成从另一个寄存器、被移位的寄存器或将一个立即数加载到目的寄存器。其中S选项决定指令的操作是否影响CPSR中条件标志位的值，当没有S时指令不更新CPSR中条件标志位的值。

指令示例：

MOV R1, R0 ; 将寄存器R0的值传送到寄存器R1

MOV PC, R14 ; 将寄存器R14的值传送到PC，常用于子程序返回

MOV R1, R0, LSL # 3 ; 将寄存器R0的值左移3位后传送到R1”

不过对于MOV指令多说一句，那就是，一般可以用类似于：

MOV R0, R0

的指令来实现NOP操作。

上面这句mov指令很简单，就是把0x0赋值给r1，即

r1=0x0

```
str    r1, [r0]
```

str指令语法：

“4、STR指令

STR指令的格式为：

STR{条件} 源寄存器，<存储器地址>

STR指令用于从源寄存器中将一个32位的字数据传送到存储器中。该指令在程序设计中比较常用，且寻址方式灵活多样，使用方式可参考指令LDR。

指令示例：

STR R0, [R1], # 8 ; 将R0中的字数据写入以R1为地址的存储器中，并将新地址R1+8写入R1。

STR R0, [R1, # 8] ; 将R0中的字数据写入以R1+8为地址的存储器中。”

所以这句str的作用也很简单，那就是将r1寄存器的值，传送到地址值为r0的（存储器）内存中。

用C语言表示就是：

*** r0 = r1**

【总结】

所以，上面几行代码意思也很清楚：

先是用r0寄存器存pWTCON的值，然后r1=0，再将r1中的0写入到pWTCON中，其实就是

```
pWTCON = 0;
```

而pWTCON寄存器的具体含义是什么呢？下面就了解其详细含义：

图表 11 WTCON 寄存器的位域

WTCON	Bit	Description	Initial State
Prescaler value	[15:8]	Prescaler value. The valid range is from 0 to 255(2 ⁸ -1).	0x80
Reserved	[7:6]	Reserved. These two bits must be 00 in normal operation.	00
Watchdog timer	[5]	Enable or disable bit of Watchdog timer. 0 = Disable 1 = Enable	1
Clock select	[4:3]	Determine the clock division factor. 00: 16 01: 32 10: 64 11: 128	00
Interrupt generation	[2]	Enable or disable bit of the interrupt. 0 = Disable 1 = Enable	0
Reserved	[1]	Reserved. This bit must be 0 in normal operation.	0
Reset enable/disable	[0]	Enable or disable bit of Watchdog timer output for reset signal. 1: Assert reset signal of the S3C2440A at watchdog time-out 0: Disable the reset function of the watchdog timer.	1

注意到bit[0]是Reset Enable/Disable，而设置为0的话，那就是关闭Watchdog的reset了，所以其他位的配置选项，就更不需要看了。
我们只需要了解，**在此处禁止了看门狗watchDog（的复位功能）**，即可。

关于看门狗的作用，以及为何要在系统初始化的时候关闭看门狗，请参见本文档后面的章节：
[什么是watchdog + 为何在要系统初始化的时候关闭watchdog](#)

2.3. 关闭中断

```
/*  
 * mask all IRQs by setting all bits in the INTMR - default  
 */  
mov r1, #0xffffffff  
ldr r0, =INTMSK  
str r1, [r0]
```

上面这几行代码，和前面的很类似，作用很简单，就是将INTMSK寄存器设置为0xffffffff，即，将所有中断都mask了。
关于每一位的定义，其实可以不看的，反正此处都已mask了，不过还是贴出来，以备后用：

图表 12 INTMSK 寄存器的位域

INTMSK	Bit	Description	Initial State
INT_ADC	[31]	0 = Service available, 1 = Masked	1
INT_RTC	[30]	0 = Service available, 1 = Masked	1
INT_SPI1	[29]	0 = Service available, 1 = Masked	1
INT_UART0	[28]	0 = Service available, 1 = Masked	1
INT_IIC	[27]	0 = Service available, 1 = Masked	1
INT_USBH	[26]	0 = Service available, 1 = Masked	1
INT_USBD	[25]	0 = Service available, 1 = Masked	1
INT_NFCON	[24]	0 = Service available, 1 = Masked	1
INT_UART1	[23]	0 = Service available, 1 = Masked	1
INT_SPI0	[22]	0 = Service available, 1 = Masked	1
INT_SDI	[21]	0 = Service available, 1 = Masked	1
INT_DMA3	[20]	0 = Service available, 1 = Masked	1
INT_DMA2	[19]	0 = Service available, 1 = Masked	1
INT_DMA1	[18]	0 = Service available, 1 = Masked	1
INT_DMA0	[17]	0 = Service available, 1 = Masked	1
INT_LCD	[16]	0 = Service available, 1 = Masked	1
INT_UART2	[15]	0 = Service available, 1 = Masked	1
INT_TIMER4	[14]	0 = Service available, 1 = Masked	1
INT_TIMER3	[13]	0 = Service available, 1 = Masked	1
INT_TIMER2	[12]	0 = Service available, 1 = Masked	1
INT_TIMER1	[11]	0 = Service available, 1 = Masked	1
INT_TIMER0	[10]	0 = Service available, 1 = Masked	1
INT_WDT_AC97	[9]	0 = Service available, 1 = Masked	1
INT_TICK	[8]	0 = Service available, 1 = Masked	1
nBATT_FLT	[7]	0 = Service available, 1 = Masked	1
INT_CAM	[6]	0 = Service available, 1 = Masked	1
EINT8_23	[5]	0 = Service available, 1 = Masked	1
EINT4_7	[4]	0 = Service available, 1 = Masked	1
EINT3	[3]	0 = Service available, 1 = Masked	1
EINT2	[2]	0 = Service available, 1 = Masked	1
EINT1	[1]	0 = Service available, 1 = Masked	1
EINT0	[0]	0 = Service available, 1 = Masked	1

此处，关于mask这个词，解释一下。

mask这个单词，是面具的意思，而中断被mask了，就是中断被掩盖了，即虽然硬件上中断发生了，但是此处被屏蔽了，所以从效果上来说，就相当于中断被禁止了，硬件上即使发生了中断，CPU也不会去执行对应中断服务程序ISR了。

关于中断的内容的详细解释，推荐看这个，解释的很通俗易懂：

[【转】ARM9 2410移植之ARM中断原理，中断嵌套的误区，中断号的怎么来的](#)

<http://againinput4.blog.163.com/blog/static/17279949120113882341352/>

```
# if defined(CONFIG_S3C2410)
    ldr r1, =0x3ff
    ldr r0, =INTSUBMSK
    str r1, [r0]
```



```
# elif defined(CONFIG_S3C2440)
    ldr r1, =0x7fff
    ldr r0, =INTSUBMSK
    str r1, [r0]
# endif
```

此处CPU是S3C2440，所以用到0x7fff这段代码。
其意思也很容易看懂，就是将INTSUBMSK寄存器的值设置为0x7fff。
先贴出对应每一位的含义：

图表 13 INTSUBMSK 寄存器的位域

INTSUBMSK	Bit	Description	Initial State
Reserved	[31:15]	Not used	0
INT_AC97	[14]	0 = Service available, 1 = Masked	1
INT_WDT	[13]	0 = Service available, 1 = Masked	1
INT_CAM_P	[12]	0 = Service available, 1 = Masked	1
INT_CAM_C	[11]	0 = Service available, 1 = Masked	1
INT_ADC_S	[10]	0 = Service available, 1 = Masked	1
INT_TC	[9]	0 = Service available, 1 = Masked	1
INT_ERR2	[8]	0 = Service available, 1 = Masked	1
INT_TXD2	[7]	0 = Service available, 1 = Masked	1
INT_RXD2	[6]	0 = Service available, 1 = Masked	1
INT_ERR1	[5]	0 = Service available, 1 = Masked	1
INT_TXD1	[4]	0 = Service available, 1 = Masked	1
INT_RXD1	[3]	0 = Service available, 1 = Masked	1
INT_ERR0	[2]	0 = Service available, 1 = Masked	1
INT_TXD0	[1]	0 = Service available, 1 = Masked	1
INT_RXD0	[0]	0 = Service available, 1 = Masked	1

然后我们再来分析对应的0x7fff是啥含义。
其实也很简单，意思就是：
0x7fff = bit[14:0]全是1 = 上表中的全部中断都被屏蔽（mask）。

```
#if 0
    /* FCLK:HCLK:PCLK = 1:2:4 */
    /* default FCLK is 120 MHz ! */
    ldr r0, =CLKDIVN
    mov r1, #3
    str r1, [r0]
#endif
```

此处，关于CLKDIVN的具体含义，参见下表：

图表 14 CLKDIVN 的位域

CLKDIVN	Bit	Description	Initial State
DIVN_UPLL	[3]	UCLK select register(UCLK must be 48MHz for USB) 0: UCLK = UPLL clock 1: UCLK = UPLL clock / 2 Set to 0, when UPLL clock is set as 48Mhz Set to 1, when UPLL clock is set as 96Mhz.	0
HDIVN	[2:1]	00 : HCLK = FCLK/1. 01 : HCLK = FCLK/2. 10 : HCLK = FCLK/4 when CAMDIVN[9] = 0. HCLK= FCLK/8 when CAMDIVN[9] = 1. 11 : HCLK = FCLK/3 when CAMDIVN[8] = 0. HCLK = FCLK/6 when CAMDIVN[8] = 1.	00
PDIVN	[0]	0: PCLK has the clock same as the HCLK/1. 1: PCLK has the clock same as the HCLK/2.	0

而此处代码被**#if 0**注释掉了。

问：为何要注释掉，难道想要使用其默认的值，即HDIVN和PDIVN上电后，默认值Initial State，都是0，对应的含义为，FCLK:HCLK:PCLK = 1:1:1 ???

答：不是，是因为我们在其他地方会去初始化时钟，去设置对应的CLKDIVN，详情参考后面的代码：

b1 clock_init

的部分。

```
#endif /* CONFIG_S3C2400 || CONFIG_S3C2410 || CONFIG_S3C2440 */
```

此处是结束上面的**#ifdef**。

```
/*
 * we do sys-critical inits only at reboot,
 * not when booting from ram!
 */
#ifdef CONFIG_SKIP_LOWLEVEL_INIT
    b1 cpu_init_crit
#endif
```

关于b1指令的含义：

b指令，是单纯的跳转指令，即CPU直接跳转到某地址继续执行。

而BL是Branch with Link，带分支的跳转，而Link指的是Link Register，链接寄存器R14，即lr，所以，b1的含义是，除了包含b指令的单纯的跳转功能，在跳转之前，还把r15寄存器=PC=cpu地址，赋值给r14=lr，然后跳转到对应位置，等要做的事情执行完毕之后，再用mov pc, lr

使得cpu再跳转回来，所以整个逻辑就是调用子程序的意思。

b1的语法为：

“2、 BL指令

BL指令的格式为：

BL{条件} 目标地址

BL 是另一个跳转指令，但跳转之前，会在寄存器R14中保存PC的当前内容，因此，可以通过将R14 的内容重新加载到PC中，来返回到跳转指令之后的那个指令处执行。该指令是实现子程序调用的一个基本但常用的手段。以下指令：

BL Label ; 当程序无条件跳转到标号Label处执行时，同时将当前的PC值保存到R14中”

对于上面的代码来说，意思就很清晰了，就是当没有定义CONFIG_SKIP_LOWLEVEL_INIT的时候，就掉转到cpu_init_crit的位置，而在后面的代码cpu_init_crit中，你可以看到最后一行汇编代码就是

```
mov pc, lr,
```

又将PC跳转回来，所以整个的含义就是，调用子程序cpu_init_crit,等cpu_init_crit执行完毕，再返回此处继续执行下面的代码。

于此对应地b指令，就只是单纯的掉转到某处继续执行，而不能再通过mov pc, lr跳转回来了。

2.4.设置堆栈 sp 指针

```
/* Set up the stack */
stack_setup:
    ldr r0, _TEXT_BASE /* upper 128 KiB: relocated uboot */
    sub r0, r0, #CFG_MALLOC_LEN /* malloc area */
    sub r0, r0, #CFG_GBL_DATA_SIZE /* bdfinfo */
```

上面代码中，第一行的意思很简单，就是把地址为_TEXT_BASE的内存中的内容给r0，而查看前面的相关部分的代码，即：

_TEXT_BASE:

.word TEXT_BASE

得知，地址为_TEXT_BASE的内存中的内容，就是

[u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\config.mk](#)

中的：

TEXT_BASE = 0x33D00000

所以，此处即：

r0

= TEXT_BASE

= 0x33D00000

而关于sub指令：

“SUB : 减法

(Subtraction)

SUB{条件}{S} <dest>, <op 1>, <op 2>

dest = op_1 - op_2

SUB 用操作数 **one** 减去操作数 **two**，把结果放置到目的寄存器中。操作数 1 是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即值：

SUB R0, R1, R2 ; R0 = R1 - R2

SUB R0, R1, #256 ; R0 = R1 - 256

SUB R0, R2, R3, LSL#1 ; R0 = R2 - (R3 << 1)

减法可以在有符号和无符号数上进行。”

所以对应含义为:

```
r0 = r0 - #CFG_MALLOC_LEN
```

```
r0 = r0 - #CFG_GBL_DATA_SIZE
```

其中, 对应的两个宏的值是:

```
u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\include\configs\EmbedSky.h
```

中:

```
_____  
#define CONFIG_64MB_Nand      0           //添加了对64MB Nand Flash支持  
  
/*  
 * Size of malloc() pool  
 */  
#define CFG_MALLOC_LEN        (CFG_ENV_SIZE + 128*1024)  
#define CFG_GBL_DATA_SIZE     128 /* size in bytes reserved for  
initial data */  
  
#if(CONFIG_64MB_Nand == 1)  
#define CFG_ENV_SIZE          0xc000 /* Total Size of Environment Sector  
 */  
#else  
#define CFG_ENV_SIZE          0x20000 /* Total Size of Environment  
Sector */  
#endif
```

所以, 从源码中的宏定义中可以看出,

```
CFG_MALLOC_LEN  
= (CFG_ENV_SIZE + 128*1024)  
= 0x20000 + 128*1024  
= 0x40000  
= 256*1024  
= 256KB
```

```
CFG_GBL_DATA_SIZE  
= 128
```

所以, 此三行的含义就是算出r0的值:

```
r0  
= (r0 - #CFG_MALLOC_LEN) - #CFG_GBL_DATA_SIZE  
= r0 - 0x40000 - 128  
= r0 - 0x40080  
= 33CBFF80
```

```
#ifndef CONFIG_USE_IRQ
    sub r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
```

如果定义了CONFIG_USE_IRQ，即如果使用中断的话，那么再把r0的值减去IRQ和FIQ的堆栈的值，

而对应的宏的值也是在

u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\include\configs\EmbedSky.h
中：

```
/*-----
 * Stack sizes
 *
 * The stack sizes are set up in start.S using the settings below
 */
#define CONFIG_STACKSIZE      (128*1024) /* regular stack */
#ifdef CONFIG_USE_IRQ
#define CONFIG_STACKSIZE_IRQ   (4*1024) /* IRQ stack */
#define CONFIG_STACKSIZE_FIQ   (4*1024) /* FIQ stack */
#endif
```

所以，此时r0的值就是：

```
#ifndef CONFIG_USE_IRQ
r0
= r0 - #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
= r0 - (4*1024 + 4*1024)
= r0 - 8*1024
= 33CBFF80 - 8*1024
= 33CBDF80
#endif
```

```
sub sp, r0, #12 /* leave 3 words for abort-stack */
```

最后，再减去终止异常所用到的堆栈大小，即12个字节。

现在r0的值为：

```
#ifndef CONFIG_USE_IRQ
r0
= r0 - 12
= 33CBDF80 - 12
= 33CBDF74
#else
r0
```

```
= r0 - 12
= 33CBFF80 - 12
= 33CBFF74
#endif
```

然后将r0的值赋值给sp,即堆栈指针。

关于:

sp代表stack pointer, 堆栈指针;

和后面要提到的ip寄存器:

ip代表instruction pointer, 指令指针。

更多详情参见下面的解释。

关于ARM的寄存器的别名和相关的APCS, 参见本文后面的内容:

[AMR寄存器的别名 + APCS](#)

```
b1 clock_init
```

在上面, 经过计算, 算出了堆栈的地址, 然后赋值给了sp, 此处, 接着才去调用函数clock_init去初始化时钟。

其中此函数是在C文件:

u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\boot_init.c

中:

```
void clock_init(void)
{
    ... 设置系统时钟clock的相关代码...
}
```

看到这里, 让我想起, 关于其他人的关于此start.S代码解释中说到的, 此处是先去设置好堆栈, 即初始化sp指针, 然后才去调用C语言的函数clock_init的。

而我们可以看到, 前面那行代码:

```
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
    b1 cpu_init_crit
#endif
```

就不需要先设置好堆栈, 再去进行函数调用。

其中cpu_init_crit对应的代码也在start.S中(详见后面对应部分的代码), 是用汇编实现的。

而对于C语言, 为何就需要堆栈, 而汇编却不需要堆栈的原因, 请参见本文后面的内容:

[为何C语言\(的函数调用\)需要堆栈, 而汇编语言却不需要堆栈](#)

```
#ifndef CONFIG_SKIP_RELOCATE_UBOOT
relocate:                /* relocate U-Boot to RAM          */
    adr r0, _start        /* r0 <- current position of code */
```

adr指令的语法和含义:

http://blog.mcuol.com/User/cdkfGao/article/8057_1.htm

“1、ADR伪指令--- 小范围的地址读取

ADR伪指令将基于PC相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中。在汇编编译器编译源程序时，ADR伪指令被编译器替换成一条合适的指令。通常，编译器用一条ADD指令或SUB指令来实现该ADR伪指令的功能，若不能用一条指令实现，则产生错误，编译失败。

ADR伪指令格式：**ADR{cond} register, expr**

地址表达式expr的取值范围：

当地址值是字节对齐时，其取值范围为： $+255 \sim 255B$ ；

当地址值是字对齐时，其取值范围为： $-1020 \sim 1020B$ ；

”

所以，上述：

```
adr r0, _start
```

的意思其实很简单，就是将_start的地址赋值给r0。但是具体实现的方式就有点复杂了，对于用adr指令实现的话，说明_start这个地址，相对当前PC的偏移，应该是很小的，意思就是向前一段或者向后一段去找，肯定能找到_start这个标号地址的，此处，自己通过看代码也可以看到_start，就是在当前指令的前面，距离很近，编译后，对应汇编代码，也可以猜得出，应该是上面所说的，用sub来实现，即当前PC减去某个值，得到_start的值，参照前面介绍的内容，去：

```
arm-inux-objdump -d u-boot > dump_u-boot.txt
```

然后打开dump_u-boot.txt，可以找到对应的汇编代码，如下：

```
33d00000 <_start>:
```

```
33d00000: ea000014  b  33d00058 <reset>
```

```
...
```

```
33d000a4 <relocate>:
```

```
33d000a4: e24f00ac  sub r0, pc, #172 ; 0xac
```

可以看到，这个相对当前PC的距离是0xac=172，细心的读者可以看到，那条指令的地址减去0xac，却并不等于_start的值，即

$$33d000a4 - 33d00000 = 0xa4 \neq 0xac$$

而 $0xac - 0xa4 = 8$ ，

那是因为，由于ARM920T的五级流水线的缘故导致指令执行那一时刻的PC的值等于该条指令PC的值加上8，即

sub r0, pc, #172中的PC的值是

```
sub r0, pc, #172
```

指令地址：33d000a4，再加上8，即 $33d000a4 + 8 = 33d000ac$ ，

所以， $33d000ac - 0xac$ ，才等于我们看到的33d00000，才是_start的地址。

这个由于流水线导致的PC的值和当前指令地址不同的现象，就是我们常说的，ARM中， $PC=PC+8$ 。

对于为何是 $PC=PC+8$ ，请参见后面的内容：

[为何ARM7中PC=PC+8](#)

对于此处为何不直接用mov指令，却使用adr指令，请参见后面内容：

[关于为何不直接用mov指令，而非要用adr伪指令](#)

对于mov指令的操作数的取值范围，请参见后面内容：

mov指令的操作数的取值范围到底是多少

【总结】

```
adr r0, _start
```

的伪代码，被翻译成实际汇编代码为：

```
33d000a4: e24f00ac  sub r0, pc, #172 ; 0xac
```

其含义就是，通过计算PC+8-172 => _start的地址，

而_start的地址，即相对代码段的0地址，是这个地址在运行时刻的值，而当ARM920T加电启动后，此处是从Nor Flash启动，对应的代码，也是在Nor Flash中，对应的物理地址是0x0，所以，此时_start的值就是0，而不是0x33d00000。

所以，此时：

r0 = 0x0

```
ldr r1, _TEXT_BASE /* test if we run from flash or RAM */
```

这里的_TEXT_BASE的含义，前面已经说过了，那就是：

“_TEXT_BASE:

```
.word TEXT_BASE
```

得知，地址为_TEXT_BASE的内存中的内容，就是

[u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\config.mk](#)

中的：

```
TEXT_BASE = 0x33D00000”
```

所以，此处就是

r1 = 0x33D00000

```
cmp    r0, r1 /* don't reloc during debug */
beq     clear_bss
```

这两句很简单，就是比较r0和r1。而

r0 = 0x0

r1 = 0x33D00000

所以不相等，因此beq发现两者不相等，就不会去跳转到clear_bss，不会去执行对应的将bss段清零的动作了。

```
ldr r2, _armboot_start
ldr r3, _bss_start
```

这两行代码意思也很清楚，分别装载_armboot_start和_bss_start地址中的值，赋值给r2和r3。

而_armboot_start和_bss_start的值，前面都已经提到过了，就是：

```
.globl _armboot_start
```

```
_armboot_start:
    .word _start
```

```
.globl __bss_start
__bss_start:
    .word __bss_start
```

而其中的_start，是我们uboot的代码的最开始的位置，而__bss_start的值，是在
[u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\u-boot.lds](#)
中的：

```
SECTIONS
{
    . = 0x00000000;

    . = ALIGN(4);
    .text :

    ...

    . = ALIGN(4);
    .rodata : { *(.rodata) }

    . = ALIGN(4);
    .data : { *(.data) }

    ...

    . = ALIGN(4);
    __bss_start = .;
    .bss : { *(.bss) }
    _end = .;
}
```

所以，可以看出，__bss_start的位置，是bss的start开始位置，同时也是
text+rodata+data的结束位置，即代码段，只读数据和已初始化的可写的的最末尾的位
置。

其实我们也可以通过前面的方法，objdump出来，看到对应的值：

```
33d00048 <__bss_start>:
33d00048: 33d339d4    .word  0x33d339d4
是0x33d339d4。
```

【总结】

```
r2 = _start      = 0x33d00000
r3 = __bss_start = 0x33d339d4
```

```
sub r2, r3, r2    /* r2 <- size of armboot
```

```
*/
```

而此处的意思就很清楚了，就是 $r2 = r3 - r2$ ，计算出

`text + rodata + data`

的大小，即整个需要载入的数据量是多少，用于下面的函数去拷贝这么多的数据到对应的内存的位置。

这里的实际的值是

`r2`

`= r3 - r2`

`= 0x33d339d4 - 0x33d00000`

`= 0x000339d4`

【总结】

到此刻位置，假定是从Nor Flash启动的话：

`r0 = 0x0` = 我们代码此刻所在的位置

`r1 = 0x33d00000` = 我们想要把我们的代码放到哪里

`r2 = 0x000339d4` = 对应的代码的大小（此处的代码 = `text + rodata + data`）

```
#if 1
    bl CopyCode2Ram      /* r0: source, r1: dest, r2: size */
#else
    add r2, r0, r2      /* r2 <- source end address */

copy_loop:
    ldmia r0!, {r3-r10} /* copy from source address [r0] */
    stmia r1!, {r3-r10} /* copy to target address [r1] */
    cmp r0, r2          /* until source end address [r2] */
    ble copy_loop
#endif
#endif /* CONFIG_SKIP_RELOCATE_UBOOT */
```

此处，代码很简单，只是注释掉了原先的那些代码，而单纯的只是去调用CopyCode2Ram这个函数。

CopyCode2Ram函数，前面也提到过了，是在：

`u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\boot_init.c`

中：

```
int CopyCode2Ram(unsigned long start_addr, unsigned char *buf, int size)
{
    unsigned int *pdwDest;
    unsigned int *pdwSrc;
    int i;

    if (bBootFrmNORFlash())
    {
        pdwDest = (unsigned int *)buf;
```

```
pdwSrc = (unsigned int *)start_addr;
/* 从 NOR Flash启动 */
for (i = 0; i < size / 4; i++)
{
    pdwDest[i] = pdwSrc[i];
}
return 0;
}
else
{
    /* 初始化NAND Flash */
    nand_init_ll();

    /* 从 NAND Flash启动 */
    if (NF_ReadID() == 0x76 )
        nand_read_ll(buf, start_addr, (size +
NAND_BLOCK_MASK) &~(NAND_BLOCK_MASK));
    else
        nand_read_ll_lp(buf, start_addr, (size +
NAND_BLOCK_MASK_LP) &~(NAND_BLOCK_MASK_LP));
    return 0;
}
}
```

可以看到，其有三个参数，start_addr，*buf和size，这三个参数，分别正好对应着我们刚才所总结的r0,r1和r2。

这些寄存器和参数的对应关系，也是APSC中定义的：

“实际参数

APCS 没有定义记录、数组、和类似的格局。这样语言可以自由的定义如何进行这些活动。但是，如果你自己的实现实际上不符合 APCS 的精神，那么将不允许来自你的编译器的代码与来自其他编译器的代码连接在一起。典型的，使用 C 语言的惯例。

- 前 4 个整数实参 (或者更少!) 被装载到 a1 - a4。
- 前 4 个浮点实参 (或者更少!) 被装载到 f0 - f3。
- 其他任何实参 (如果有的话) 存储在内存中,用进入函数时紧接在 sp 的值上面的字来指向。换句话说，其余的参数被压入栈顶。所以要想简单。最好定义接受 4 个或更少的参数的函数。

”

上面说的a1-a4,就是寄存器r0-r3。

而CopyCode2Ram函数的逻辑也很清晰，就是先去判断是从Nor Flash启动还是从Nand Flash启动，然后决定从哪里拷贝所需要的代码到对应的目标地址中。

2.5.清除 bss 段

```
clear_bss:
    ldr r0, _bss_start          /* find start of bss segment */
```

```
ldr r1, _bss_end      /* stop here */
mov r2, #0x00000000    /* clear */
```

上面代码中的_bss_start, 是:

```
.globl _bss_start
```

```
_bss_start:
```

```
.word __bss_start
```

而_bss_end, 是:

```
.globl _bss_end
```

```
_bss_end:
```

```
.word _end
```

对应的, __bss_start和_end, 都在前面提到过的那个链接脚本里面:

```
u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\u-boot.lds
```

中的:

```
__bss_start = .;
.bss : { *(.bss) }
_end = .;
```

即bss段的起始地址和结束地址。

```
clbss_1: str r2, [r0]      /* clear loop... */
add r0, r0, #4
cmp r0, r1
ble clbss_1
```

而此段代码, 含义也很清晰, 那就是,

先将r2, 即0x0, 存到地址为r0的内存中去, 然后r0地址加上4, 比较r0地址和r1地址, 即比较当前地址是否到了bss段的结束位置, 如果le, little or equal, 小于或等于, 那么就跳到clbss_1, 即接着这几个步骤, 直到地址超过了bss的_end位置, 即实现了将整个bss段, 都清零。

```
#if 0
/* try doing this stuff after the relocation */
ldr r0, =PWTCN
mov r1, #0x0
str r1, [r0]

/*
 * mask all IRQs by setting all bits in the INTMR - default
 */
mov r1, #0xffffffff
ldr r0, =INTMR
str r1, [r0]

/* FCLK:HCLK:PCLK = 1:2:4 */
```

```
/* default FCLK is 120 MHz ! */
ldr r0, =CLKDIVN
mov r1, #3
str r1, [r0]
/* END stuff after relocation */
#endif

ldr pc, _start_armboot

_start_armboot: .word start_armboot
```

上面已经注释掉的代码，此处忽略。

只看最后的那两行，意思也很简单，那就是将地址为_start_armboot中的内容，即start_armboot，赋值给PC，即调用start_armboot函数。至此，汇编语言的start.S的整个工作，就完成了。

而start_armboot函数，在C文件中：

u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\EmbedSky.c
中：

```
void start_armboot (void)
{
    ...
}
```

这就是传说中的，调用第二层次，即C语言级别的初始化了，去初始化各个设备了。

其中包括了CPU，内存等，以及串口，正常初始化后，就可以从串口看到uboot的打印信息了。

```
/*
*****
****
*
* CPU_init_critical registers
*
* setup important registers
* setup memory timing
*
*****
****
*/
```

```
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
cpu_init_crit:
```



```
/*
 * flush v4 I/D caches
 */
mov r0, #0
mcr p15, 0, r0, c7, c7, 0 /* flush v3/v4 cache */
mcr p15, 0, r0, c8, c7, 0 /* flush v4 TLB */
```

关于mcr的来龙去脉:

<http://apps.hi.baidu.com/share/detail/32319228>

“

ARM 微处理器可支持多达 16 个协处理器，用于各种协处理操作，在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略 ARM 处理器和其他协处理器的指令。ARM 的协处理器指令主要用于 ARM 处理器初始化 ARM 协处理器的数据处理操作，以及在ARM 处理器的寄存器和协处理器的寄存器之间传送数据，和在 ARM 协处理器的寄存器和存储器之间传送数据。ARM 协处理器指令包括以下 5 条：

- 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
- ...

CP15系统控制协处理器

CP15 —系统控制协处理器 (the system control coprocessor) 他通过协处理器指令 MCR和MRC提供具体的寄存器来配置和控制caches、MMU、保护系统、配置时钟模式（在 bootloader时钟初始化用到）

CP15的寄存器只能被MRC和MCR (Move to Coprocessor from ARM Register) 指令访问”

一些要说明的内容，见下:

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0151c/ARM920T_TRM1_S.pdf

“you can only access CP15 registers with MRC and MCR instructions in a privileged mode. The assembler for these instructions is:

MCR/MRC{cond} P15,opcode_1,Rd,CRn,CRm,opcode_2

The CRn field of MRC and MCR

instructions specifies the coprocessor register to access. The CRm field and opcode_2 fields specify a particular action when addressing registers. The L bit distinguishes between an MRC (L=1) and an MCR (L=0).

Note:

Attempting to read from a nonreadable register, or to write to a nonwritable register causes unpredictable results.

The opcode_1, opcode_2, and CRm fields should be zero, except when the values specified are used to select the desired operations, in all

instructions that access CP15. Using other values results in unpredictable behavior”

CP15有很多个寄存器，分别叫做寄存器0 (Register 0)，到寄存器15 (Register 15)，每个寄存器分别控制不同的功能，而且有的是只读，有的是只写，有的是可读写。

而且这些寄存器的含义，随着版本ARM内核版本变化而不断扩展，详情请参考这个网址：

<http://www.heyrick.co.uk/assembler/coprocmand.html>

其中，根据我们此处关心的内容，摘录部分内容如下：

“ARM 710

- **Register 7 - IDC flush** (write only)

Any data written to this location will cause the IDC (Instruction/Data cache) to be flushed.

.....

StrongARM SA110

....

- **Register 7 - Cache control** (write only)

Any data written to this location will cause the selected cache to be flushed.

The OPC_2 and CRm co-processor fields select which cache operation should occur:

o	<u>Function</u>	<u>OPC_2</u>	<u>CRm</u>	<u>Data</u>
o	Flush I + D	%0000	%0111	-
o	Flush I	%0000	%0101	-
o	Flush D	%0000	%0110	-
o	Flush D single	%0001	%0110	Virtual address
o	Clean D entry	%0001	%1010	Virtual address
o	Drain write buf.	%0100	%1010	-

- **Register 8 - TLB operations** (write only)

Any data written to this location will cause the selected TLB flush operation.

The OPC_2 and CRm co-processor fields select which cache operation should occur:

o	<u>Function</u>	<u>OPC_2</u>	<u>CRm</u>	<u>Data</u>
o	Flush I + D	%0000	%0111	-
o	Flush I	%0000	%0101	-
o	Flush D	%0000	%0110	-
o	Flush D single	%0001	%0110	Virtual address”

而MCR的详细语法为：

“MCR指令

MCR指令将ARM处理器的寄存器中的数据传送到协处理器寄存器中。如果协处理器不能成功地执

行该操作，将产生未定义的指令异常中断。

指令语法格式

`MCR{<cond>} <p>, < opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}`

`MCR{<cond>} p15, 0, <Rd>, <CRn>, <CRm>{, <opcode_2>}`

其中，<cond>为指令执行的条件码。当<cond>忽略时指令为无条件执行。

< opcode_1>为协处理器将执行的的操作的操作码。对于CP15协处理器来说，< opcode_1>永远为0b000，当< opcode_1>不为0b000时，该指令操作结果不可预知。

<Rd>作为源寄存器的ARM寄存器，其值将被传送到协处理器寄存器中。

<CRn>作为目标寄存器的协处理器寄存器，其编号可能是C0, C1, ..., C15。

<CRm>和<opcode_2>两者组合决定对协处理器寄存器进行所需要的操作，如果没有指定，则将为<CRm>为C0，opcode_2为0”

对照上面的那行代码：

`mcr p15, 0, r0, c7, c7, 0 /* flush v3/v4 cache */`

可以看出，其中

rd为r0=0

CRn为C7

CRm为C7

opcode_2为0，

对于这行代码的作用，以此按照语法，来一点点解释如下：

首先，mcr做的事情，其实很简单，就是“ARM处理器的寄存器中的数据传送到协处理器寄存器中”，此处即是，将ARM的寄存器r0中的数据，此时r0=0，所以就是把0这个数据，传送到协处理器CP15中。而对应就是写入到“<CRn>”这个“目标寄存器的协处理器寄存器”，此处CRn为C7，即将0写入到寄存器7（Register 7）中去。

而上面关于Register 7的含义中也说了，“Any data written to this location will cause the selected cache to be flushed”，即你往这个寄存器7中写入任何数据，都会导致对应的缓存被清空。而到底那个缓存被清空呢，即我们这行指令

“mcr p15, 0, r0, c7, c7, 0”，起了什么作用呢，

那是由“<CRm>和<opcode_2>两者组合决定”的。

而此处CRm为C7，opcode_2为0，而对于C7和0的组合的作用，参见上面的那个表中，关于Register 7的定义：

“

	<u>Function</u>	<u>OPC 2</u>	<u>CRm</u>	<u>Data</u>
o	Flush I + D	%0000	%0111	-
o	Flush I	%0000	%0101	-
o	Flush D	%0000	%0110	-
o	Flush D single	%0001	%0110	Virtual address
o	Clean D entry	%0001	%1010	Virtual address
	Drain write buf.	%0100	%1010	-

”

可以看出，上述表中，红色哪一行，当opcode_2为0，CRm为0111=7，就是我们要找的，其作用是“Flush I + D”，即清空指令缓存I Cache和数据缓存D Cache。

根据该表，同理，如果是opcode_2=0，而CRm=0101b=5，那么对应的就是去“Flush I”，即只清除指令缓存I Cache了。

而对应的指令也就是“mcr p15, 0, r0, c7, c5, 0”了。

而关于注释“/* flush v3/v4 cache */”，说此行代码的作用是，清理v3或v4的缓存。其中v4，我们很好理解，因为我们此处的CPU是ARM920T的核心，是属于ARM V4的，而为何又说，也可以清除v3的cache呢？

那是因为，本身这些寄存器位域的定义，都是向下兼容的，参见上面引用的内容，也写到了：

“ARM 710

- Register 7 - IDC flush (write only)

Any data written to this location will cause the IDC (Instruction/Data cache) to be flushed.”

即，对于ARM7的话，你写同样的这行代码“mcr p15, 0, r0, c7, c7, 0”，也还是向register 7中写入了数据0，这也同样满足了其所说的“Any data written to this location”，也会产生同样的效果“cause the IDC (Instruction/Data cache) to be flushed”。

同理，可以看出“mcr p15, 0, r0, c8, c7, 0 /* flush v4 TLB */”是去操作寄存器8，而对应的各个参数为：

rd为r0=0

CRn为C8

CRm为C7

opcode_2为0，

对照寄存器8的表：

“

	Function	OPC 2	CRm	Data
o	Flush I + D	%0000	%0111	-
o	Flush I	%0000	%0101	-
o	Flush D	%0000	%0110	-
o	Flush D single	%0001	%0110	Virtual address

”其含义为：

向寄存器8中写入数据，会导致对应的TLB被清空。具体是哪个TLB，由opcode_2和CRm组合决定，

此处opcode_2为0，CRm为7=0111b，所以对应的作用是“Flush I + D”，即清空指令和数据的TLB。

注：上述两行代码，其实都可以ARM的官方网站上面找到：

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0184b/Chdcfejb.html>

Function	Rd	Instruction
Invalidate ICache and DCache	SBZ	MCR p15,0,Rd,c7,c7,0

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0184b/Chdifbjc.html>

Function	Rd	Instruction
Invalidate TLB(s)	SBZ	MCR p15,0,Rd,c8,c7,0

```
/*
 * disable MMU stuff and caches
 */
mrc p15, 0, r0, c1, c0, 0
```

此处，对应的值为：

rd为r0=0

CRn为C1

CRm为C0

opcode_2为0，

即，此行代码是将r0的值，即0，写入到CP15的寄存器1中。

寄存器1的相关的定义为：

<http://www.heyrick.co.uk/assembler/coproc.md.html>

“StrongARM SA110

- Register 1 - Control (read/write)
All values set to 0 at power-up.
 - o Bit 0 - On-chip MMU turned off (0) or on (1)
 - o Bit 1 - Address alignment fault disabled (0) or enabled (1)
 - o Bit 2 - Data cache turned off (0) or on (1)
 - o Bit 3 - Write buffer turned off (0) or on (1)
 - o Bit 7 - Little-endian operation if 0, big-endian if 1
 - o Bit 8 - System bit - controls the MMU permission system
 - o Bit 9 - ROM bit - controls the MMU permission system
 - o Bit 12 - Instruction cache turned off (0) or on (1)”

所以，对应内容就是，向bit[CRm]中写入opcode_2，即向bit[0]写入0，对应的作用为“On-chip MMU turned off”，即关闭MMU。

```
bic r0, r0, #0x00002300 @ clear bits 13, 9:8 (--V- --RS)
bic r0, r0, #0x00000087 @ clear bits 7, 2:0 (B--- -CAM)
orr r0, r0, #0x00000002 @ set bit 2 (A) Align
orr r0, r0, #0x00001000 @ set bit 12 (I) I-Cache
mcr p15, 0, r0, c1, c0, 0
```

上面几行代码，注释中写的也很清楚了，就是去清楚对应的位和设置对应的位，具体位域的含义，见下：

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0184b/Chdifbjc.html>

“

Table 2.10. Control register 1 bit functions

图表 15 控制寄存器 1 的位域含义

Register bits	Name	Function	Value
31	iA bit	Asynchronous clock select	See Table 2.11.
30	nF bit	notFastBus select	See Table 2.11.
29:15	-	Reserved	Read = Unpredictable. Write = Should be zero.
14	RR bit	Round robin replacement	0 = Random replacement. 1 = Round-robin replacement.
13	V bit	Base location of exception registers	0 = Low addresses = 0x00000000. 1 = High addresses = 0xFFFF0000.
12	I bit	ICache enable	0 = ICache disabled. 1 = ICache enabled.
11:10	-	Reserved	Read = 00. Write = 00.
9	R bit	ROM protection	This bit modifies the MMU protection system. See Domain access control .
8	S bit	System protection	This bit modifies the MMU protection system. See Domain access control .
7	B bit	Endianness	0 = Little-endian operation. 1 = Big-endian operation.
6:3	-	Reserved	Read = 1111. Write = 1111.
2	C bit	DCache enable	0 = DCache disabled. 1 = DCache enabled.
1	A bit	Alignment fault enable	Data address alignment fault checking. 0 = Fault checking disabled. 1 = Fault checking enabled.

Register bits	Name	Function	Value
0	M bit	MMU enable	0 = MMU disabled. 1 = MMU enabled.

Table 2.11. Clocking modes

图表 16 时钟模式

Clocking mode	iA	nF
FastBus mode	0	0
Synchronous	0	1
Reserved	1	0
Asynchronous	1	1

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0151c/I273867.html>

“Domain access control

Table 3.10. Interpreting access control bits in domain access control register

图表 17 关于访问控制位在域访问控制寄存器中的含义

Value	Meaning	Description
00	No access	Any access generates a domain fault
01	Client	Accesses are checked against the access permission bits in the section or page descriptor
10	Reserved	Reserved. Currently behaves like the no access mode
11	Manager	Accesses are <i>not</i> checked against the access permission bits so a permission fault cannot be generated

Table 3.11 shows how to interpret the *Access Permission* (AP) bits and how their interpretation is dependent on the S and R bits (control register bits 8 and 9).

Table 3.11. Interpreting access permission (AP) bits

图表 18 关于访问允许(AP)位的含义

AP	S	R	Supervisor permissions	User permissions	Description
00	0	0	No access	No access	Any access generates a permission fault
00	1	0	Read-only	No access	Only Supervisor read permitted
00	0	1	Read-only	Read-only	Any write generates a permission fault
00	1	1	Reserved	-	-
01	x	x	Read/write	No access	Access allowed only in Supervisor mode
10	x	x	Read/write	Read-only	Writes in User mode cause permission fault
11	x	x	Read/write	Read/write	All access types permitted in both modes
xx	1	1	Reserved	-	-

”

所以：

```
“ bic r0, r0, #0x00002300 @ clear bits 13, 9:8 (--V- --RS)”
```

的作用是：

(1)清除bit[13]：

Base location of exception register (异常寄存器基地址)

0 = Low address = 0x0000 0000

(2)清除bit[9]和bit[8]

此处不是很懂，待后续深入了解。

目前的理解是：

不论是Supervisor还是user，谁都不能访问，否则就出现权限错误“Any access generates a permission fault”。

```
“ bic r0, r0, #0x00000087 @ clear bits 7, 2:0 (B--- -CAM)”
```

的作用是：

(1)清除bit[7]：

使用little endian

(2)清除bit[2-0]：

DCache disabled, 关闭Dcache;

Alignment Fault checking disabled, 关闭地址对齐的错误检查;

MMU disabled, 关闭MMU。

```
“ orr r0, r0, #0x00000002 @ set bit 2 (A) Align”
```

的作用是：

(1) 设置bit[1]：

“Enable Data address alignment fault checking” 打开数据地址对齐的错误检查，即如果数据地址为非法（奇数？）地址，就报错。

```
“ orr r0, r0, #0x0001000 @ set bit 12 (I) I-Cache”
```

(1) 设置bit[12]：

开启指令缓存I cache。

```
“ mcr p15, 0, r0, c1, c0, 0”
```

mcr指令，将刚才设置的r0的值，再写入到寄存器1中。

```
/*
 * before relocating, we have to setup RAM timing
 * because memory timing is board-dependend, you will
 * find a lowlevel_init.S in your board directory.
 */
mov ip, lr
bl lowlevel_init
mov lr, ip
mov pc, lr
#endif /* CONFIG_SKIP_LOWLEVEL_INIT */
```

这里，其实和前面的代码：

```
“
#ifdef CONFIG_SKIP_LOWLEVEL_INIT
    bl cpu_init_crit
#endif
”
```

是对应的，最后一行是：

```
“mov pc, lr”
```

是典型的子函数调用，通过将lr的值赋值给pc，实现函数调用完成后而返回的。

而

```
“bl lowlevel_init”
```

前面的一行：

```
“mov ip, lr”
```

意思是将lr的值给ip，即指令指针r12，此处之所以要保存一下lr是因为此处是在子函数

“cpu_init_crit”中，lr已经保存了待会用于返回主函数的地址，即上次调用时候的pc的值，而此处如果在子函数cpu_init_crit中继续调用其他子函数lowlevel_init，而不保存lr的话，那么调用完lowlevel_init返回来时候，就丢失了cpu_init_crit要返回的位置。说白了就是，每次你要调用函数之前，你自己要确保是否已经正确保存了lr的值，要保证函数调用完毕后，也能正常返回。当然，如果你此处根本不需要返回，那么就不用去保存lr的值了。

2.6. 异常中断处理

```
/*
*****
****
*
* Interrupt handling
*
*****
****
*/

@
@ IRQ stack frame.
@
#define S_FRAME_SIZE 72

#define S_OLD_R0 68
#define S_PSR 64
#define S_PC 60
#define S_LR 56
#define S_SP 52

#define S_IP 48
#define S_FP 44
#define S_R10 40
#define S_R9 36
#define S_R8 32
#define S_R7 28
#define S_R6 24
#define S_R5 20
#define S_R4 16
#define S_R3 12
#define S_R2 8
#define S_R1 4
#define S_R0 0

#define MODE_SVC 0x13
#define I_BIT 0x80
```

此处很简单，只是一些宏定义而已。

后面用到的时候再解释。

```
/*
 * use bad_save_user_regs for abort/prefetch/undef/swi ...
 * use irq_save_user_regs / irq_restore_user_regs for IRQ/FIQ handling
 */
```

```
.macro bad_save_user_regs
```

.macro和后面的.endm相对应，其语法是：

图表 19 macro 的语法

Macro Directives			
Directive	Description	Syntax	Example
.macro	Define a macro.	.macro name {args, ...}	
	A macro can be defined without arguments, and can be called simply by specifying its name.	name {args, ...}	.macro NoArgsMacro NoArgsMacro
	A macro can also be defined with arguments, and can be called the same way with commas seperating its arguments.		.macro ArgMacro arg, arg2 ArgMacro 10, 11
	Arguments can be accessed by their name prefixed with a \.	\arg	mov r0, \arg
	You can define default macro arguments.		.macro ArgMacro arg=1, arg2
	Arguments are omitted by simply placing a comma and no value, or ignoring them all together (trailing only).		ArgMacro , 11
	Arguments can be set in a modified order by referncing them by name.		ArgMacro arg2=11, arg=10
	Macros can be recursive.		
.endm	Mark the end of a macro.	.endm	.endm
.exitm	Exit a macro early.	.exitm	.exitm
\@	Psudo variable that contains the macro number executed. Can be used for a unique number on every macro definition.	\@	MyLabel\@:
.purgem	Undefine a macro, so that further uses do not evaluate.	.purgem name	.purgem NoArgsMacro

所以，此处就相当于一个无参数的宏bad_save_user_regs，也就相当于一个函数了。

```
sub sp, sp, #S_FRAME_SIZE
```

即

```
sp
= sp- S_FRAME_SIZE
= sp - 72
```

```
stmia sp, {r0 - r12} @ Calling r0-r12
```

stmia
stmia的语法为：

图表 20 LDM/STM 的语法

`<LDM|STM>{cond}<FD|ED|FA|EA|IA|IB|DA|DB> Rn{!},<Rlist>{^}`

where:

- `{cond}` is a two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
- `Rn` is an expression evaluating to a valid register number
- `<Rlist>` is a list of registers and register ranges enclosed in `{}` (e.g. `{R0,R2-R7,R10}`).
- `{!}` (if present) requests write-back (`W=1`), otherwise `W=0`
- `{^}` (if present) set `S` bit to load the CPSR along with the PC, or force transfer of user bank when in privileged mode

其中，条件域的具体含义如下：

图表 21 条件码的含义

31	28	27	0
cond			
Condition Field			
0000	=	EQ (equal)	- Z set
0001	=	NE (not equal)	- Z clear
0010	=	CS (unsigned higher or same)	- C set
0011	=	CC (unsigned lower)	- C clear
0100	=	MI (negative)	- N set
0101	=	PL (positive or zero)	- N clear
0110	=	VS (overflow)	- V set
0111	=	VC (no overflow)	- V clear
1000	=	HI (unsigned higher)	- C set and Z clear
1001	=	LS (unsigned lower or same)	- C clear or Z set
1010	=	GE (greater or equal)	- N set and V set, or N clear and V clear
1011	=	LT (less than)	- N set and V clear, or N clear and V set
1100	=	GT (greater than)	- Z clear, and either N set and Vset, or N clear and V clear
1101	=	LE (less than or equal)	- Z set, or N set and V clear, or N clear and V set
1101	=	AL	- always
1111	=	NV	- never

Figure 5-2: Condition codes

更具体的含义：

“

六、批量数据加载/存储指令ARM微处理器所支持批量数据加载/存储指令可以一次在一片连续的存储器单元和多个寄存器之间传送数据，批量加载指令用于将一片连续的存储器中的数据传送到多个寄存器，批量数据存储指令则完成相反的操作。常用的加载存储指令如下：

LDM（或STM）指令

LDM（或STM）指令的格式为：

LDM（或STM）{条件}{类型} 基址寄存器{!}，寄存器列表{^}

LDM（或STM）指令用于从由基址寄存器所指示的一片连续存储器到寄存器列表所指示的多个寄存器之间传送数据，该指令的常见用途是将多个寄存器的内容入栈或出栈。其中，{类型}为以下几种情况：

IA 每次传送后地址加1；

IB 每次传送前地址加1；

DA 每次传送后地址减1;
DB 每次传送前地址减1;
FD 满递减堆栈;
ED 空递减堆栈;
FA 满递增堆栈;
EA 空递增堆栈;

{!}为可选后缀,若选用该后缀,则当数据传送完毕之后,将最后的地址写入基址寄存器,否则基址寄存器的内容不改变。

基址寄存器不允许为R15,寄存器列表可以为R0~R15的任意组合。

{^}为可选后缀,当指令为LDM且寄存器列表中包含R15,选用该后缀时表示:除了正常的数据传送之外,还将SPSR复制到CPSR。同时,该后缀还表示传入或传出的是用户模式下的寄存器,而不是当前模式下的寄存器。

指令示例:

STMFD R13!, {R0, R4-R12, LR} ; 将寄存器列表中的寄存器(R0, R4到R12, LR)存入堆栈。
LDMFD R13!, {R0, R4-R12, PC} ; 将堆栈内容恢复到寄存器(R0, R4到R12, LR)。”

所以,此行的含义是,

将r0到r12的值,一个个地传送到对应的地址上,基地址是sp的值,传完一个,sp的值加4,一直到传送完为止。

此处,可见,前面那行代码:

sp = sp - 72

就是为此处传送r0到r12,共13个寄存器,地址空间需要13*4=72个字节,

即前面sp减去72,就是为了腾出空间,留此处将r0到r12的值,放到对应的位置的。

```
ldr r2, _armboot_start
```

此处的含义就是,将

中的值,参考前面内容,即为_start,

而_start的值:

从 Nor Flash启动时: **_stat=0**

relocate代码之后: **_start=TEXT_BASE=0x33D00000**

此处是已经relocate代码了,所以应该理解为后者,即 **_start=0x33D00000**

所以:

r2=0x33D00000

```
sub r2, r2, #(CONFIG_STACKSIZE+CFG_MALLOC_LEN)
```

此处

r2

= r2 - (CONFIG_STACKSIZE+CFG_MALLOC_LEN)

= r2 - (128*1024 + 256*1024)

= 0x33D00000 - 384KB

= **0x33CA0000**

```
sub r2, r2, #(CFG_GBL_DATA_SIZE+8) @ set base 2 words into abort stack
```

此处:

r2

= r2 - (CFG_GBL_DATA_SIZE + 8)

= 0x33CA0000 - (128 + 8)

= **0x33C9FF78**

```
ldmia r2, {r2 - r3} @ get pc, cpsr
```

分别将地址为r2和r2+4的内容，即地址为 **0x33C9FF78** 和 **0x33C9FF7C** 中的内容，load 载入给r2和r3寄存器。

```
add r0, sp, #S_FRAME_SIZE @ restore sp_SVC
```

将sp的值，加上72，送给r0

```
add r5, sp, #S_SP
```

前面的定义是: “#define S_SP 52”

所以此处就是将sp的值，加上52，送给r5

```
mov r1, lr
```

将lr给r1

```
stmia r5, {r0 - r3} @ save sp_SVC, lr_SVC, pc, cpsr
```

然后将r0到r3中的内容，存储到地址为r5-r5+12中的位置去。

```
mov r0, sp
```

将sp再赋值给r0

```
.endm
```

结束宏bad_save_user_regs

【总结】

此处虽然每行代码基本看懂了，但是到底此bad_save_user_regs函数是做什么的，还是不太清楚，有待以后慢慢深入理解。

```
.macro irq_save_user_regs
```



```
sub sp, sp, #S_FRAME_SIZE
stmia sp, {r0 - r12}           @ Calling r0-r12
add r8, sp, #S_PC
stmdb r8, {sp, lr}^           @ Calling sp, LR
str lr, [r8, #0]               @ Save calling PC
mrs r6, spsr
str r6, [r8, #4]               @ Save CPSR
str r0, [r8, #8]               @ Save OLD_R0
mov r0, sp
.endm
```

```
.macro irq_restore_user_regs
ldmia sp, {r0 - lr}^          @ Calling r0 - lr
mov r0, r0
ldr lr, [sp, #S_PC]           @ Get PC
add sp, sp, #S_FRAME_SIZE
subs pc, lr, #4               @ return & move spsr_svc into cpsr
.endm
```

上面两段代码，基本上和前面很类似，虽然每一行都容易懂，但是整个两个函数的意思，除了看其宏的名字irq_save_user_regs和irq_restore_user_regs，分别对应着中断中，保存和恢复用户模式寄存器，之外，其他的，个人目前还是没有太多了解。

```
.macro get_bad_stack
ldr r13, _armboot_start       @ setup our mode stack
sub r13, r13, #(CONFIG_STACKSIZE+CFG_MALLOC_LEN)
sub r13, r13, #(CFG_GBL_DATA_SIZE+8) @ reserved a couple spots in abort
stack
str lr, [r13]                 @ save caller lr / spsr
mrs lr, spsr
str lr, [r13, #4]
mov r13, #MODE_SVC           @ prepare SVC-Mode
@ msr spsr_c, r13
msr spsr, r13
mov lr, pc
movs pc, lr
.endm
```

此处的get_bad_stack被后面undefined_instruction, software_interrupt等处调用，目前能理解的意思是，在出错的时候，获得对应的堆栈的值。

```
.macro get_irq_stack           @ setup IRQ stack
ldr sp, IRQ_STACK_START
.endm
```

此处的含义很好理解，就是把地址为IRQ_STACK_START中的值赋值给sp。

即获得IRQ的堆栈的起始地址。

而对于IRQ_STACK_START，是我们前面这里[cpu_init源码](#)（[点击可掉转到对应位置](#)）就提到过的：

```
int cpu_init (void)
{
    /*
     * setup up stacks if necessary
     */
#ifdef CONFIG_USE_IRQ
    IRQ_STACK_START = _armboot_start - CFG_MALLOC_LEN - CFG_GBL_DATA_SIZE
- 4;
    FIQ_STACK_START = IRQ_STACK_START - CONFIG_STACKSIZE_IRQ;
    FREE_RAM_END = FIQ_STACK_START - CONFIG_STACKSIZE_FIQ -
CONFIG_STACKSIZE;
    FREE_RAM_SIZE = FREE_RAM_END - PHYS_SDRAM_1;
#else
    FREE_RAM_END = _armboot_start - CFG_MALLOC_LEN - CFG_GBL_DATA_SIZE
- 4 - CONFIG_STACKSIZE;
    FREE_RAM_SIZE = FREE_RAM_END - PHYS_SDRAM_1;
#endif
    return 0;
}
```

而此处，就是用到了，前面已经在cpu_init()中重新计算正确的值了。

即算出IRQ堆栈的起始地址，其算法很简单，就是：

```
IRQ_STACK_START = _armboot_start - CFG_MALLOC_LEN - CFG_GBL_DATA_SIZE
- 4;
```

即，先减去malloc预留的空间，和global data，即在

[u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\board\EmbedSky\board.c](#)

中定义的全局变量：

```
DECLARE_GLOBAL_DATA_PTR;
```

而此宏对应的值在：

```
u-boot-1.1.6\_20100601\opt\EmbedSky\u-boot-1.1.6\include\asm-arm\global\_data.h
```

中：

```
#define DECLARE_GLOBAL_DATA_PTR    register volatile gd_t *gd asm ("r8")
```

即，用一个固定的寄存器r8来存放此结构体的指针。

（注：这也对应着编译uboot的时候，你所看到的编译参数**-ffixed-r8**）

此gd_t的结构体，不同体系结构，用的不一样。

而此处arm的平台中，gd_t的定义在同一文件中：

```
typedef struct global_data {
    bd_t      *bd;
    unsigned long flags;
    unsigned long baudrate;
    unsigned long have_console; /* serial_init() was called */
    unsigned long reloc_off; /* Relocation Offset */
    unsigned long env_addr; /* Address of Environment struct */
    unsigned long env_valid; /* Checksum of Environment valid? */
    unsigned long fb_base; /* base address of frame buffer */
#ifdef CONFIG_VFD
    unsigned char vfd_type; /* display type */
#endif
#if 0
    unsigned long cpu_clk; /* CPU clock in Hz! */
    unsigned long bus_clk;
    unsigned long ram_size; /* RAM size */
    unsigned long reset_status; /* reset status register at boot */
#endif
    void **jt; /* jump table */
} gd_t;
```

而此全局变量`gd_t *gd`会被其他很多文件所引用，详情自己去代码中找。

```
.macro get_fiq_stack @ setup FIQ stack
ldr sp, FIQ_STACK_START
.endm
```

此处和上面类似，把地址为`FIQ_STACK_START`中的内容，给`sp`。

其中：

`FIQ_STACK_START = IRQ_STACK_START - CONFIG_STACKSIZE_IRQ;`

即`FIQ`的堆栈起始地址，是`IRQ`堆栈起始地址减去`IRQ`堆栈的大小。

```
/*
 * exception handlers
 */
.align 5
undefined_instruction:
    get_bad_stack
    bad_save_user_regs
    bl do_undefined_instruction
    .align 5
```

如果发生未定义指令异常，CPU会掉转到start.S开头中对应的位置：

```
ldr pc, _undefined_instruction
```

即把地址为 _undefined_instruction中的内容给pc，即跳转到此处执行对应的代码。

其做的事情依次是：

获得出错时候的堆栈

保存用户模式寄存器

跳转到对应的函数：do_undefined_instruction

而do_undefined_instruction函数是在：

u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\cpu\arm920t\interrupts.c

中：

```
void bad_mode (void)
{
    panic ("Resetting CPU ...\n");
    reset_cpu (0);
}

void do_undefined_instruction (struct pt_regs *pt_regs)
{
    printf ("undefined instruction\n");
    show_regs (pt_regs);
    bad_mode ();
}
```

可以看到，此处起始啥事没错，只是打印一下出错时候的寄存器的值，然后跳转到bad_mode中取reset CPU，直接重启系统了。

```
software_interrupt:
    get_bad_stack
    bad_save_user_regs
    b1 do_software_interrupt

    .align 5
prefetch_abort:
    get_bad_stack
    bad_save_user_regs
    b1 do_prefetch_abort

    .align 5
data_abort:
    get_bad_stack
```

```
bad_save_user_regs
b1 do_data_abort

.align 5
not_used:
get_bad_stack
bad_save_user_regs
b1 do_not_used
```

以上几个宏，和前面的do_undefined_instruction是类似的，就不多说了。

```
@ HJ
.globl Launch
.align 4
Launch:
mov r7, r0
@ disable interrupt
@ disable watch dog timer
mov r1, #0x53000000
mov r2, #0x0
str r2, [r1]

ldr r1,=INTMSK
ldr r2,=0xffffffff @ all interrupt disable
str r2,[r1]

ldr r1,=INTSUBMSK
ldr r2,=0x7fff @ all sub interrupt disable
str r2,[r1]

ldr r1, = INTMOD
mov r2, #0x0 @ set all interrupt as IRQ (not FIQ)
str r2, [r1]

@
mov ip, #0
mcrp15, 0, ip, c13, c0, 0 @ /* zero PID */
mcrp15, 0, ip, c7, c7, 0 @/* invalidate I,D caches */
mcrp15, 0, ip, c7, c10, 4 @ /* drain write buffer */
mcrp15, 0, ip, c8, c7, 0 @/* invalidate I,D TLBs */
mrcp15, 0, ip, c1, c0, 0 @/* get control register */
bic ip, ip, #0x0001 @ /* disable MMU */
mcrp15, 0, ip, c1, c0, 0 @/* write control register */
```

```
@ MMU_EnableICache
@mrc p15,0,r1,c1,c0,0
@orr r1,r1,#(1<<12)
@mcr p15,0,r1,c1,c0,0

#ifdef CONFIG_SURPORT_WINCE
    bl Wince_Port_Init
#endif

    @ clear SDRAM: the end of free mem(has wince on it now) to the end
of SDRAM
    ldr    r3, FREE_RAM_END
    ldr    r4, =PHYS_SDRAM_1+PHYS_SDRAM_1_SIZE    @ must clear all the
memory unused to zero
    mov    r5, #0

    ldr    r1, _armboot_start
    ldr    r2, =On_Steppingstone
    sub    r2, r2, r1
    mov    pc, r2
On_Steppingstone:
2: stmia  r3!, {r5}
    cmp    r3, r4
    bne    2b

    @ set sp = 0 on sys mode
    mov    sp, #0

    @ add by HJ, switch to SVC mode
    msrcpsr_c,    #0xdf    @ set the I-bit = 1, diable the IRQ interrupt
    msrcpsr_c,    #0xd3    @ set the I-bit = 1, diable the IRQ interrupt
    ldr    sp, =0x31ff5800

    nop
    nop
    nop
    nop

    mov    pc, r7    @ Jump to PhysicalAddress
    nop
    mov    pc, lr
```

此处相当于一个叫做Launch的函数，做了也是类似的系统初始化的动作。
但是没找到此函数在哪里被调用的。具体不太清楚。

```
#ifdef CONFIG_USE_IRQ

    .align 5
irq:
/* add by www.embedsky.net to use IRQ for USB and DMA */
    sub lr, lr, #4           @ the return address
    ldr sp, IRQ_STACK_START  @ the stack for irq
    stmdb sp!, { r0-r12,lr } @ save registers

    ldr lr, =int_return      @ set the return addr
    ldr pc, =IRQ_Handle      @ call the isr
int_return:
    ldmba sp!, { r0-r12,pc }^ @ return from interrupt
```

此处，做的事情，很容易看懂，就是中断发生后，掉转到这里，然后保存对应寄存器，然后跳转到对应irq函数IRQ_Handle中去。但是前面为何sp为何去减去4，原因不太懂。

关于IRQ_Handle，是在：

u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\cpu\arm920t\s3c24x0\interrupts.c

中：

```
void IRQ_Handle(void)
{
    unsigned long oft = intregs->INTOFFSET;
    S3C24X0_GPIO * const gpio = S3C24X0_GetBase_GPIO();

    // printk("IRQ_Handle: %d\n", oft);

    //清中断
    if( oft == 4 ) gpio->EINTPEND = 1<<7;
    intregs->SRCPND = 1<<oft;
    intregs->INTPND = intregs->INTPND;

    /* run the isr */
    isr_handle_array[oft]();
}
```

此处细节就不多解释了，大体含义是，找到对应的中断源，然后调用对应的之前已经注册的中断服务函数ISR。

```
.align 5
```



```
fiq:
    get_fiq_stack
    /* someone ought to write a more efficient fiq_save_user_regs */
    irq_save_user_regs
    bl do_fiq
    irq_restore_user_regs
```

此处也很简单，就是发生了快速中断FIQ的时候，保存IRQ的用户模式寄存器，然后调用函数do_fiq，调用完毕后，再恢复IRQ的用户模式寄存器。

其中do_fiq()是在：

u-boot-1.1.6_20100601\opt\EmbedSky\u-boot-1.1.6\cpu\arm920t\interrupts.c

中：

```
void do_fiq (struct pt_regs *pt_regs)
{
    printf ("fast interrupt request\n");
    show_regs (pt_regs);
    bad_mode ();
}
```

和前面提到过的do_undefined_instruction的一样，就是打印寄存器信息，然后跳转到bad_mode()去重启CPU而已。

```
#else

    .align 5
irq:
    get_bad_stack
    bad_save_user_regs
    bl do_irq

    .align 5
fiq:
    get_bad_stack
    bad_save_user_regs
    bl do_fiq

#endif
```

此处就是，如果没有定义CONFIG_USE_IRQ，那么就用这段代码，可以看到，都只是直接调用do_irq和do_fiq，也没做什么实际工作。

3. 相关知识点详解

3.1. uboot 初始化中，为何要设置 CPU 为 SVC 模式而不是设置为其他模式

在看Uboot的start.S文件时候，发现其最开始初始化系统，做的第一件事情，就是将CPU设置为SVC模式，但是S3C2440的CPU的core是ARM920T，其有7种模式，为何非要设置为SVC模式，而不是设置为其他模式呢？对此，经过一些求证，得出如下原因：

首先，先要了解ARM的CPU的7种模式是哪些：

<http://www.docin.com/p-73665362.html>

图表 22 ARM 中 CPU 的模式

处理器模式	说明	备注
用户 (usr)	正常程序工作模式	此模式下程序不能够访问一些受操作系统保护的系统资源，应用程序也不能直接进行处理器模式的切换。
系统 (sys)	用于支持操作系统的特权任务等	与用户模式类似，但具有可以直接切换到其它模式等特权
快中断 (fiq)	支持高速数据传输及通道处理	FIQ异常响应时进入此模式
中断 (irq)	用于通用中断处理	IRQ异常响应时进入此模式
管理 (svc)	操作系统保护代码	系统复位和软件中断响应时进入此模式
中止 (abt)	用于支持虚拟内存和/或存储器保护	在ARM7TDMI没有大用处
未定义 (und)	支持硬件协处理器的软件仿真	未定义指令异常响应时进入此模式

另外，7种模式中，除用户usr模式外，其它模式均为特权模式。

对于为何此处是svc模式，而不是其他某种格式，其原因，可以从两方面来看：

【第一方面】

我们先简单的来分析一下那7种模式：

中止abt和未定义und模式：

首先可以排除的是，中止abt和未定义und模式，那都是不太正常的模式，此处程序是正常运行的，所以不应该设置CPU为其中任何一种模式，所以可以排除。

快中断fiq和中断irq模式：

其次，对于快中断fiq和中断irq来说，此处uboot初始化的时候，也还没啥中断要处理和能够处理，而且即使是注册了终端服务程序后，能够处理中断，那么这两种模式，也是自动切换过去的，所以，此处也不应该设置为其中任何一种模式。

用户usr模式:

虽然从理论上来说,可以设置CPU为用户usr模式,但是由于此模式无法直接访问很多的硬件资源,而uboot初始化,就必须要去访问这类资源,所以此处可以排除,不能设置为用户usr模式。

系统sys模式 vs 管理svc模式:

首先,sys模式和usr模式相比,所用的寄存器组,都是一样的,但是增加了一些访问一些在usr模式下不能访问的资源。

而svc模式本身就属于特权模式,本身就可以访问那些受控资源,而且,比sys模式还多了些自己模式下的影子寄存器,所以,相对sys模式来说,可以访问资源的能力相同,但是拥有更多的硬件资源。

所以,从理论上来说,虽然可以设置为sys和svc模式的任一种,但是从uboot方面考虑,其要做的事情是初始化系统相关硬件资源,需要获取尽量多的权限,以方便操作硬件,初始化硬件。

从uboot的目的是初始化硬件的角度来说,设置为svc模式,更有利于其工作。

因此,此处将CPU设置为SVC模式。

【第二方面】

uboot作为一个bootloader来说,最终目的是为了启动Linux的kernel,在做好准备工作(即初始化硬件,准备好kernel和rootfs等)跳转到kernel之前,本身就要满足一些条件,其中一个条件,就是要求CPU处于SVC模式的。

(关于满足哪些条件,详情请参考:

ARM Linux Kernel Boot Requirements

<http://www.arm.linux.org.uk/developer/booting.php>

或者Linux内核文档:

`kernel_source_root\documentation\arm\booting`

中也是同样的解释:

“The CPU must be in SVC mode”)

所以,uboot在最初的初始化阶段,就将CPU设置为SVC模式,也是最合适的。

综上所述,uboot在初始化阶段,就应该将CPU设置为SVC模式。

3.2.什么是 watchdog + 为何在要系统初始化的时候关闭

watchdog

关于 Uboot 初始化阶段,在 start.S 中,为何要去关闭 watchdog,下面解释具体的原因:

3.2.1.什么是 watchdog

嵌入式系统之WATCHDOG(看门狗)概述

<http://wenku.baidu.com/view/e5cd52ff04a1b0717fd5dd27.html>

简要摘录如下:

watchdog 一般是一个硬件模块,其作用是,在嵌入式操作系统中,很多应用情况是系统长期运行且无人看守,所以难免或者怕万一出现系统死机,那就杯具了,这时,watchdog 就会自动帮你重启系统。

那么其是如何实现此功能的呢？那么就要简单解释一下其实现原理了。

watchdog 硬件的逻辑就是，其硬件上有个记录超时功能，然后要求用户需要每隔一段时间（此时间可以根据自己需求而配置）去对其进行一定操作，比如往里面写一些固定的值，俗称“喂狗”，那么我发现超时了，即过了这么长时间你还不给偶喂食，那么偶就认为你系统是死机了，出问题了，偶就帮你重启系统。说白了就是弄个看家狗 dog，你要定期给其喂食，如果超时不喂食，那么狗就认为你，他的主人，你的系统，死机了，就帮你 reset 重启系统。

3.2.2. 为何在要系统初始化的时候关闭 watchdog

了解了 watchdog 的原理后，此问题就很容易理解了。

如果不禁用 watchdog，那么就要单独写程序去定期“喂狗”，那多麻烦，多无聊啊。

毕竟咱此处只是去用 uboot 初始化必要的硬件资源和系统资源而已，完全用不到这个 watchdog 的机制。需要用到，那也是你 linux 内核跑起来了，是你系统关心的事情，和我 uboot 没啥关系的，所以肯定此处要去关闭 watchdog（的 reset 功能）了。

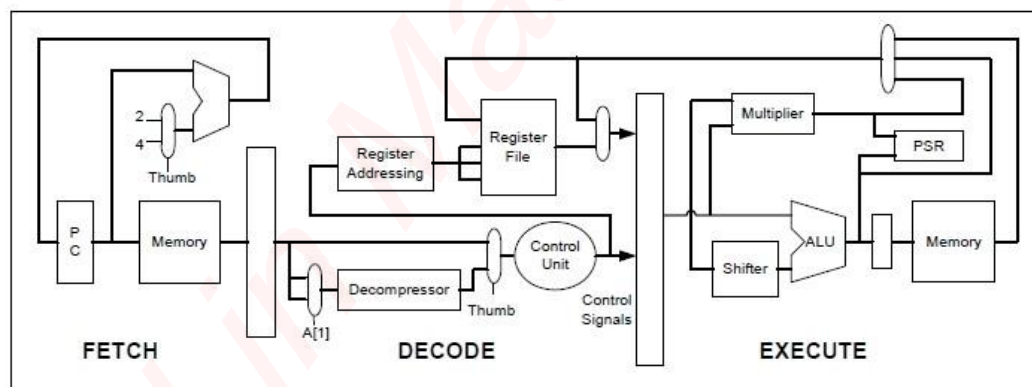
3.3. 为何 ARM7 中 $PC=PC+8$

此处解释为何 ARM7 中，CPU 地址，即 PC，为何有 $PC=PC+8$ 这一说法：

总所周知，ARM7，是三级流水线，其细节见图：

图表 23 ARM7 三级流水线

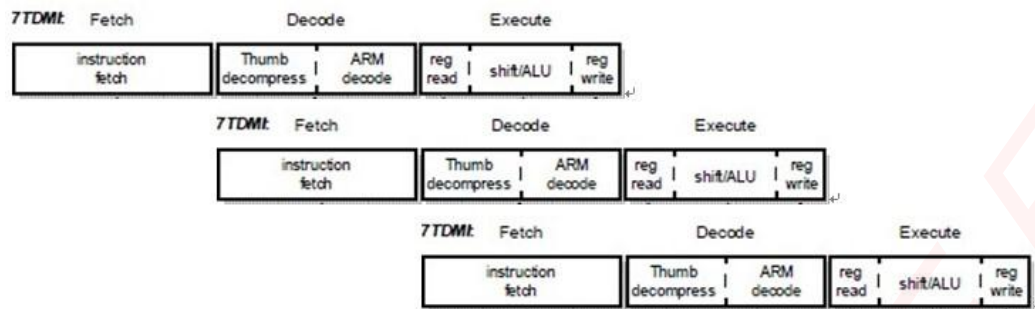
ARM7TM Core and Pipeline



首先，对于 ARM7 对应的流水线的执行情况，如下面这个图所示：

ARM7

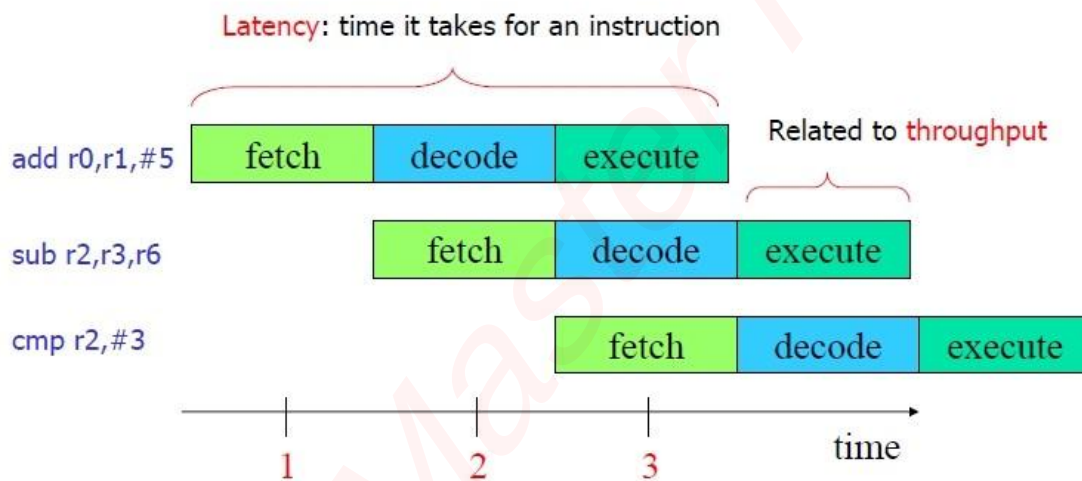
图表 24 ARM7 三级流水线状态



然后对于三级流水线举例如下：

图表 25 ARM7 三级流水线示例

ARM7 Single-Cycle Instruction 3-Stage Pipeline



从上图，其实很容易看出，第一条指令：

`add r0, r1, $5`

执行的时候，此时PC已经指向第三条指令：

`cmp r2, #3`

的地址了，所以，是 $PC=PC+8$ 。

3.3.1. 为何 ARM9 和 ARM7 一样，也是 $PC=PC+8$

ARM7 的三条流水线， $PC=PC+8$ ，很好理解，但是 ARM9 中，是五级流水线，为何还是 $PC=PC+8$ ，而不是

PC

$=PC + (5-1) * 4$

$=PC + 16$ ，

呢？

下面就需要好好解释一番了。

具体解释之前，先贴上 ARM7 和 ARM9 的流水线的区别和联系：

图表 26 ARM7 三级流水线 vs ARM9 五级流水线

Figure 1 : The ARM7TDMI core and ARM7TDMI-S core pipeline

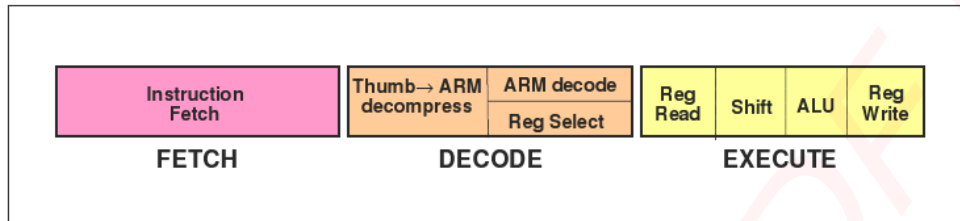
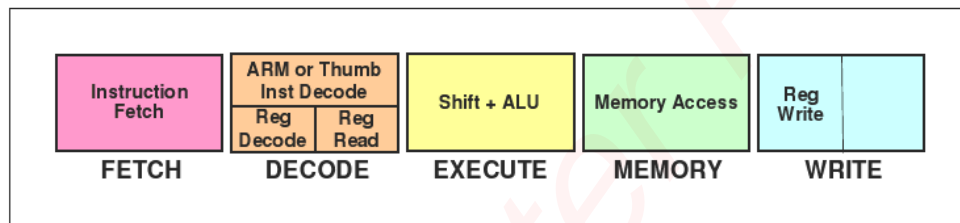
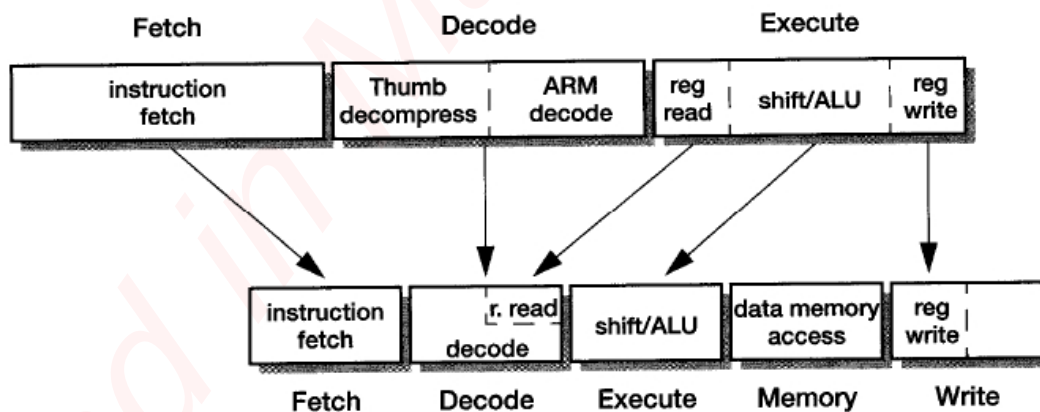


Figure 2 : The ARM9TDMI core pipeline



图表 27 ARM7 三级流水线到 ARM9 五级流水线的映射

ARM7TDMI 3-Stage Pipeline



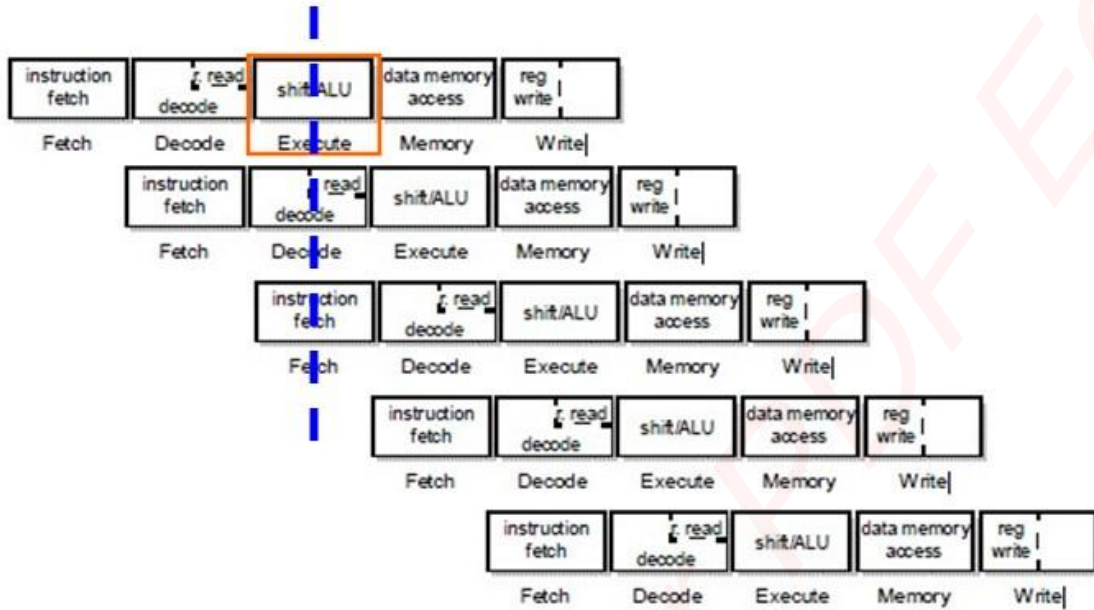
ARM9TDMI 5-Stage Pipeline

下面开始对为何 ARM9 也是 $PC=PC+8$ 进行解释。

先列出 ARM9 的五级流水线的示例：

图表 28 ARM9 的五级流水线示例

ARM9 的五级流水线示例



然后我们以下面 uboot 中的 start.S 的最开始的汇编代码为例来进行解释：

```
00000000 <_start>:
  0:  ea000014  b  58 <reset>
  4:  e59ff014  ldr pc, [pc, #20] ; 20 <_undefined_instruction>
  8:  e59ff014  ldr pc, [pc, #20] ; 24 <_software_interrupt>
 c:  e59ff014  ldr pc, [pc, #20] ; 28 <_prefetch_abort>
10:  e59ff014  ldr pc, [pc, #20] ; 2c <_data_abort>
14:  e59ff014  ldr pc, [pc, #20] ; 30 <_not_used>
18:  e59ff014  ldr pc, [pc, #20] ; 34 <_irq>
1c:  e59ff014  ldr pc, [pc, #20] ; 38 <_fiq>

00000020 <_undefined_instruction>:
 20:  00000120  .word 0x00000120
```

下面对每一个指令周期，CPU 做了哪些事情，分别详细进行阐述：

在看下面具体解释之前，有一句话要牢记，那就是：

PC 不是指向你正在运行的指令，而是

PC 始终指向你要取的指令的地址。

认识清楚了这个前提，后面的举例讲解，就容易懂了。

指令周期 Cycle1

(1) 取指：

PC 总是指向将要读取的指令的地址（即我们常说的，指向下一条指令的地址），而当前 PC=4，所以去取物理地址为 4 对对应的指令“ldr pc, [pc, #20]”，其对应二进制代码为

e59ff014。

此处取指完之后，自动更新 PC 的值，即 $PC=PC+4$ （单个指令占 4 字节，所以加 4） $=4+4=8$

指令周期 Cycle2

(1) **译指**：翻译指令 e59ff014；

(2) 同时再去**取指**：

PC 总是指向将要读取的指令的地址（即我们常说的，指向下一条指令的地址），而当前 $PC=8$ ，所以去物理地址为 8 所对应的指令“ldr pc, [pc, #20]” 其对应二进制代码为 e59ff014。

此处取指完之后，自动更新 PC 的值，即 $PC=PC+4=8+4=12=0xc$

指令周期 Cycle3

(1) **执行**（指令）：执行“e59ff014”，即“ldr pc, [pc, #20]”所表达的含义，即 $PC = PC + 20$

$$= 12 + 20$$

$$= 32$$

$$= 0x20$$

此处，只是计算出待会要赋值给 PC 的值是 0x20，这个 0x20 还只是放在执行单元中内部的缓冲中。

(2) **译指**：翻译 e59ff014。

(3) **取指**：

此步骤由于是和上面（1）中的执行同步做的，所以，未受到影响，继续取指，而取指的那一时刻，PC 为上一 Cycle 更新后的值，即 $PC=0xc$ ，所以是去取物理地址为 0xc 所对应的指令“ldr pc, [pc, #20]”，对应二进制为 e59ff014。

其实，分析到这里，大家就可以看出：

在 Cycle3 的时候，PC 的值，刚好已经在 Cycle1 和 Cycle2，分别加了 4，所以 Cycle3 的时候， $PC=PC+8$ ，而同样道理，对于任何一条指令的，都是在 Cycle3，指令的 Execute 执行阶段，如果用到 PC 的值，那么 PC 那一时刻，就是 $PC=PC+8$ 。

所以，此处虽然是五级流水线，但是却不是 $PC=PC+16$ ，而是 $PC=PC+8$ 。

进一步地，我们发现，其实 $PC=PC+N$ 的 N，是和指令的执行阶段所处于流水线的深度有关，即此处指令的执行 Execute 阶段，是五级流水线中的第三个，而这个第三阶段的 Execute 和指令的第一个阶段的 Fetch 取指，相差的值是 $3-1=2$ ，即两个 CPU 的 Cycle，而每个 Cycle 都会导致 $PC=PC+4$ ，所以，指令到了 Execute 阶段，才会发现，此时 PC 已经变成 $PC=PC+8$ 了。

回过头来反观 ARM7 的三级流水线，也是同样的道理，指令的 Execute 执行阶段，是处于指令的第三个阶段，同理，在指令计算数据的时候，如果用到 PC，就会发现此时 $PC=PC+8$ 。

同理，假如 ARM9 的五级流水线，把指令的 Execute 执行阶段，设计在了第四个阶段，那么就是 $PC=PC+(第4阶段-1)*4个字节 = PC=PC+12$ 了。

【总结】

ARM7 的三级流水线， $PC=PC+8$ ，

ARM9 的五级流水线，也是 $PC=PC+8$ ，

根本的原因是，两者的流水线设计中，指令的 Execute 执行阶段，都是处于流水线的第三级，所以使得 $PC=PC+8$ 。

类似地，可以推导出：

假设，Execute 阶段处于流水线中的第 E 阶段，每条指令是 T 个字节，那么

PC

$$= PC + N * T$$

$$= PC + (E - 1) * T$$

此处 ARM7 和 ARM9：

Execute 阶段都是第 3 阶段 $\rightarrow E=3$

每条指令是 4 个字节 $\rightarrow T=4$

所以：

PC

$$= PC + N * T$$

$$= PC + (3 - 1) * 4$$

$$= PC + 8$$

【关于直接改变 PC 的值，会导致流水线清空的解释】

把 PC 的值直接赋值为 0x20。而 PC 值更改，直接导致流水线的清空，即导致下一个 cycle 中的，对应的流水线中的其他几个步骤，包括接下来的同一个 Cycle 中的取指的工作被取消。在 PC 跳转到 0x20 的位置之后，流水线重新计算，重新一步步地按照流水线的逻辑，去一点点执行。当然要保证当前指令的执行完成，即执行之后，还有两个 cycle，分别做的 Memory 和 Write，会继续执行完成。

【相关资料】

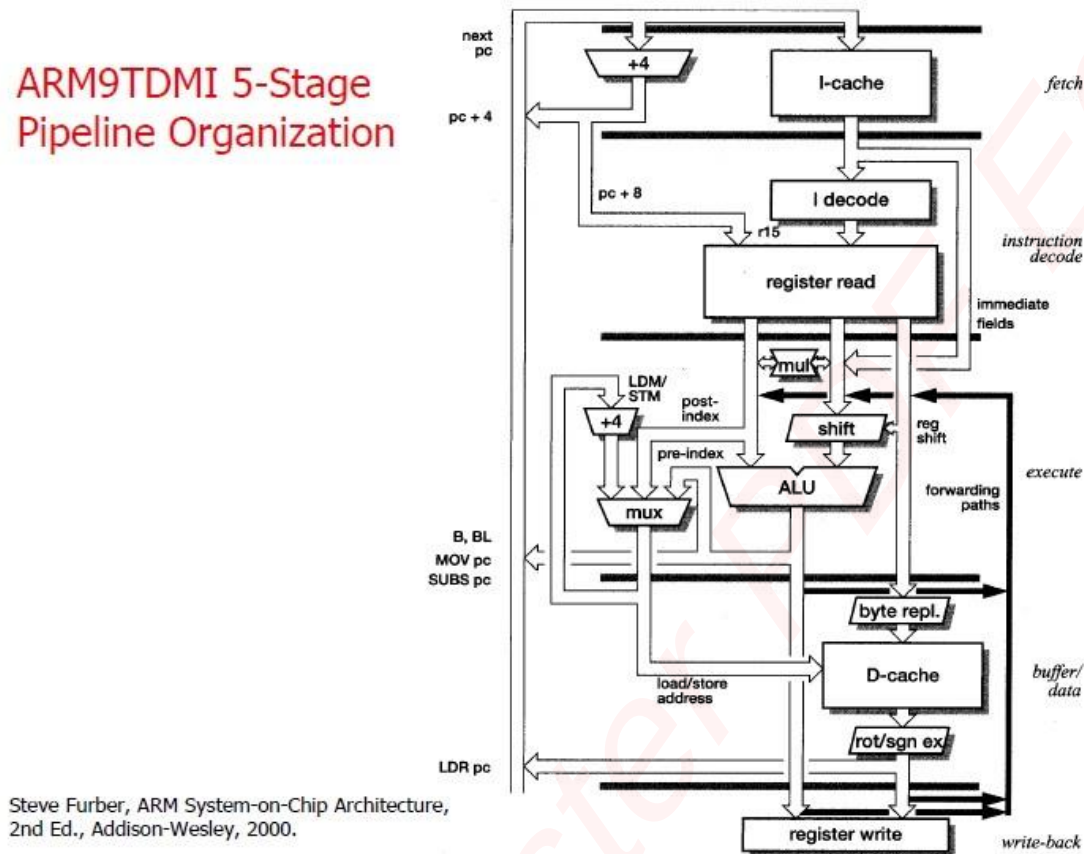
ARM 流水线和 program counter(PC)的增量

<http://hi.baidu.com/istry/blog/item/f823e1438de0a71972f05d0f.html>

最后附上 ARM9 的五级流水线的内置执行结构的细节：

图表 29 ARM9 五级流水线的执行机构的细节

ARM9TDMI 5-Stage Pipeline Organization



3.4.AMR 寄存器的别名 + APCS

此处简单介绍一下，ARM 寄存器的别名，以及什么是 APCS。

3.4.1.ARM 中的寄存器的别名

默认的情况下，这些寄存器只是叫做 $r0, r1, \dots, r14$ 等，而 APCS 对其起了不同的别名。

使用汇编器预处理器的功能，你可以定义 $R0$ 等名字，但在你修改其他人写的的代码的时候，最好还是学习使用 APCS 名字。

一般编程过程中，最好按照其约定，使用对应的名字，这样使得程序可读性更好。

关于不同寄存器所对应的名字，见下表：

APCS寄存器别名定义

图表 30 ARM 寄存器的别名

寄存器名字		
Reg #	APCS	意义
R0	a1	工作寄存器
R1	a2	"
R2	a3	"
R3	a4	"
R4	v1	必须保护
R5	v2	"
R6	v3	"
R7	v4	"
R8	v5	"
R9	v6	"
R10	s1	栈限制
R11	fp	帧指针
R12	ip	内部过程调用寄存器
R13	sp	栈指针
R14	lr	连接寄存器
R15	pc	程序计数器

更加详细一点，见下：

“

Predeclared register names

http://www.keil.com/support/man/docs/armasm/armasm_ch03s03s01.htm

The following register names are predeclared:

- **r0-r15** and **R0-R15**
- **a1-a4** (argument, result, or scratch registers, synonyms for r0 to r3)
- **v1-v8** (variable registers, r4 to r11)
- **sb** and **SB** (static base, r9)
- **ip** and **IP** (intra-procedure-call scratch register, r12)
- **sp** and **SP** (stack pointer, r13)
- **lr** and **LR** (link register, r14)
- **pc** and **PC** (program counter, r15).

Predeclared extension register names

http://www.keil.com/support/man/docs/armasm/armasm_ch03s03s02.htm

The following extension register names are predeclared:

- **d0-d31** and **D0-D31** (VFP double-precision registers)
- **s0-s31** and **S0-S31** (VFP single-precision registers).

Predeclared coprocessor names

http://www.keil.com/support/man/docs/armasm/armasm_ch03s03s03.htm

The following coprocessor names and coprocessor register names are predeclared:

- p0-p15 (coprocessors 0-15)
- c0-c15 (coprocessor registers 0-15).

”

3.4.2. 什么是 APCS

APCS, ARM 过程调用标准 (**ARM Procedure Call Standard**), 提供了紧凑的编写例程的一种机制, 定义的例程可以与其他例程交织在一起。最显著的一点是对这些例程来自哪里没有明确的限制。它们可以编译自 C、Pascal、也可以是用汇编语言写成的。

APCS 定义了:

- 对寄存器使用的限制。
- 使用栈的惯例。
- 在函数调用之间传递/返回参数。
- 可以被‘回溯’的基于栈的结构格式, 用来提供从失败点到程序入口的函数 (和给予的参数) 的列表。

3.5. 为何 C 语言 (的函数调用) 需要堆栈, 而汇编语言却不 需要堆栈

之前看了很多关于uboot的分析, 其中就有说要为C语言的运行, 准备好堆栈。

而自己在Uboot的start.S汇编代码中, 关于系统初始化, 也看到有堆栈指针初始化这个动作。但是, 从来只是看到有人说系统初始化要初始化堆栈, 即正确给堆栈指针sp赋值, 但是却从来没有看到有人解释, 为何要初始化堆栈。所以, 接下来的内容, 就是经过一定的探究, 试图来解释一下, 为何要初始化堆栈, 即:

为何c语言的函数调用要用到堆栈, 而汇编却不需初始化堆栈。

要明白这个问题, 首先要了解堆栈的作用。

关于堆栈的作用, 要详细讲解的话, 要很长的篇幅, 所以此处只是做简略介绍。

总的来说, 堆栈的作用就是: 保存现场/上下文, 传递参数。

3.5.1. 保存现场/上下文

现场, 意思就相当于案发现场, 总有一些现场的情况, 要记录下来的, 否则被别人破坏掉之后, 你就无法恢复现场了。而此处说的现场, 就是指CPU运行的时候, 用到了一些寄存器, 比如r0, r1等等, 对于这些寄存器的值, 如果你不保存而直接跳转到子函数中去执行, 那么很可能就被其破坏了, 因为其函数执行也要用到这些寄存器。

因此, 在函数调用之前, 应该将这些寄存器等现场, 暂时保持起来, 等调用函数执行完毕返回后, 再恢复现场。这样CPU就可以正确的继续执行了。

在计算机中, 你常可以看到上下文这个词, 对应的英文是context。那么:

3.5.1.1. 什么叫做上下文 context

保存现场，也叫保存上下文。

上下文，英文叫做context，就是上面的文章，和下面的文章，即与你此刻，当前CPU运行有关的内容，即那些你用到寄存器。所以，和上面的现场，是一个意思。

保存寄存器的值，一般用的是push指令，将对应的某些寄存器的值，一个个放到堆栈中，把对应的值压入到堆栈里面，即所谓的**压栈**。

然后待被调用的子函数执行完毕的时候，再调用pop，把堆栈中的一个一个的值，赋值给对应的那些你刚开始压栈时用到的寄存器，把对应的值从堆栈中弹出去，即所谓的**出栈**。

其中保存的寄存器中，也包括lr的值（因为用bl指令进行跳转的话，那么之前的pc的值是存在lr中的），然后在子程序执行完毕的时候，再把堆栈中的lr的值pop出来，赋值给pc，这样就实现了子函数的正确的返回。

3.5.2. 传递参数

C语言进行函数调用的时候，常常会传递给被调用的函数一些参数，对于这些C语言级别的参数，被编译器翻译成汇编语言的时候，就要找个地方存放一下，并且让被调用的函数能够访问，否则就没发实现传递参数了。对于找个地方放一下，分两种情况。

一种情况是，本身传递的参数就很少，就可以通过寄存器传送参数。

因为在前面的保存现场的动作中，已经保存好了对应的寄存器的值，那么此时，这些寄存器就是空闲的，可以供我们使用的了，那就可以放参数，而参数少的情况下，就足够存放参数了，比如参数有2个，那么就用r0和r1存放即可。（关于参数1和参数2，具体哪个放在r0，哪个放在r1，就是和APCS中的“在函数调用之间传递/返回参数”相关了，APCS中会有详细的约定。感兴趣的自己去研究。）

但是如果参数太多，寄存器不够用，那么就得把多余的参数堆栈中了。

即，可以用堆栈来传递所有的或寄存器放不下的那些多余的参数。

3.5.3. 举例分析 C 语言函数调用是如何使用堆栈的

对于上面的解释的堆栈的作用显得有些抽象，此处再用例子来简单说明一下，就容易明白了：

用：

```
arm-inux-objdump -d u-boot > dump_u-boot.txt
```

可以得到dump_u-boot.txt文件。该文件就是中，包含了u-boot中的程序的可执行的汇编代码，

其中我们可以看到C语言的函数的源代码，到底对应着那些汇编代码。

下面贴出两个函数的汇编代码，

一个是clock_init，

另一个是与clock_init在同一C源文件中的，另外一个函数CopyCode2Ram：

```
33d0091c <CopyCode2Ram>:
```

```
33d0091c: e92d4070  push    {r4, r5, r6, lr}
```



```
33d00920: e1a06000    mov r6, r0
33d00924: e1a05001    mov r5, r1
33d00928: e1a04002    mov r4, r2
33d0092c: ebffffef    bl 33d008f0 <bBootFrmNORFlash>
... ..
33d00984: ebffff14    bl 33d005dc <nand_read_11>
... ..
33d009a8: e3a00000    mov r0, #0 ; 0x0
33d009ac: e8bd8070    pop {r4, r5, r6, pc}

33d009b0 <clock_init>:
33d009b0: e3a02313    mov r2, #1275068416 ; 0x4c000000
33d009b4: e3a03005    mov r3, #5 ; 0x5
33d009b8: e5823014    str r3, [r2, #20]
... ..
33d009f8: e1a0f00e    mov pc, lr
```

(1) clock_init 部分的代码

可以看到该函数第一行:

```
33d009b0: e3a02313    mov r2, #1275068416 ; 0x4c000000
```

就没有我们所期望的push指令, 没有去将一些寄存器的值放到堆栈中。这是因为, 我们clock_init这部分的内容, 所用到的r2,r3等等寄存器, 和前面调用clock_init之前所用到的寄存器r0, 没有冲突, 所以此处可以不用push去保存这类寄存器的值, 不过有个寄存器要注意, 那就是lr, 即lr, 其是在前面调用clock_init的时候, 用的是bl指令, 所以会自动把跳转时候的pc的值赋值给lr, 所以也不需要push指令去将PC的值保存到堆栈中。

而clock_init的代码的最后一行:

```
33d009f8: e1a0f00e    mov pc, lr
```

就是我们常见的mov pc, lr, 把lr的值, 即之前保存的函数调用时候的PC值, 赋值给现在的PC,

这样就实现了函数的正确的返回, 即返回到了函数调用时候下一个指令的位置。

这样CPU就可以继续执行原先函数内剩下那部分的代码了。

(2) CopyCode2Ram 部分的代码

其第一行:

```
33d0091c: e92d4070    push {r4, r5, r6, lr}
```

就是我们所期望的, 用push指令, 保存了r4,r5,r6以及lr。

用push去保存r4,r5,r6, 那是因为所谓的保存现场, 以后后续函数返回时候再恢复现场,

而用push去保存lr, 那是因为此函数里面, 还有其他函数调用:

```
33d0092c: ebffffef    bl 33d008f0 <bBootFrmNORFlash>
```

```
... ..
```

```
33d00984: ebffff14    bl 33d005dc <nand_read_11>
```

```
... ..
```

也用到了bl指令, 会改变我们最开始进入clock_init时候的lr的值, 所以我们要用push也暂时保存起来。

而对应地，CopyCode2Ram的最后一行：

```
33d009ac: e8bd8070 pop {r4, r5, r6, pc}
```

就是把之前push的值，给pop出来，还给对应的寄存器，其中最后一个是将开始push的lr的值，pop出来给赋给PC，因为实现了函数的返回。

另外，我们注意到，在CopyCode2Ram的倒数第二行是：

```
33d009a8: e3a00000 mov r0, #0 ; 0x0
```

是把0赋值给r0寄存器，这个就是我们所谓返回值的传递，是通过r0寄存器的。

此处的返回值是0，也对应着C语言的源码中的“return 0”。

对于使用哪个寄存器来传递返回值：

当然你也可以用其他暂时空闲没有用到的寄存器来传递返回值，但是这些处理方式，本身是根据ARM的APCS的寄存器的使用的约定而设计的，你最好不要随便改变使用方式，最好还是按照其约定的来处理，这样程序更加符合规范。

3.6. 关于为何不直接用 mov 指令，而非要用 adr 伪指令

在分析uboot的start.S中，看到一些指令，比如：

```
adr r0, _start
```

觉得好像可以直接用mov指令实现即可，为啥还要这么麻烦地，去用ldr去实现？

关于此处的代码，为何要用adr指令：

```
adr r0, _start
```

```
(
```

注：其被编译器编译后，会被翻译成：

```
sub r0, pc, #172
```

```
)
```

而不直接用mov指令直接将_start的值赋值给r0，类似于这样：

```
mov r0, _start
```

呢？

其原因主要是，

```
sub r0, pc, #172
```

这样的代码，所处理的值，都是相对于PC的偏移量来说的，这样的代码中，没有绝对的物理地址值，都是相对的值，利用产生位置无关代码。因为如果用mov指令：

```
mov r0, _start
```

那么就会被编译成这样的代码：

```
mov r0, 0x33d00000
```

如果用了上面这样的代码：

```
mov r0, 0x33d00000
```

那么，如果整个代码，即要执行的程序的指令，被移动到其他位置，那么

```
mov r0, 0x33d00000
```

这行指令，执行的功能，就是跳转到绝对的物理地址，而不是跳转到相对的_start的位置了，就不能实现我们想要的功能了，这样包含了绝对物理地址的代码，也就不是位置无关的代码了。

于此相对，这行指令：

```
sub r0, pc, #172
```

即使程序被移动到其他位置，那么该行指令还是可以跳转到相对PC往前172字节的地方，也还是我们想要的_start的位置，这样包含的都是相对的偏移位置的代码，就叫做位置无关代码。其优点就是不用担心你的代码被移动，即使程序的基地址变了，所有的代码的相对位置还是固定的，程序还是可以正常运行的。

关于，之所以不用上面的：

```
mov r0, 0x33d00000
```

类似的代码，除了上面说的，不是位置无关的代码之外，其还有个潜在的问题，那就是，关于mov指令的源操作数，此处即为0x33d00000，不一定是合法的mov指令所允许的值，这也正是下面要详细解释的内容“mov指令的操作数的取值范围到底是多少”。

【总结】

之所以用adr而不用mov，主要是为了生成地址无关代码，以及由于不方便判断一个数，是否是有效的mov的操作数。

3.7.mov 指令的操作数的取值范围到底是多少

关于mov指令操作数的取值范围，网上看到一些人说是0x00-0xFF，也有人说是其他的值的，但是经过一番求证，发现这些说法都不对。下面就是来详细解释，mov指令的操作数的取值范围，到底是多少。

在看了我说的，关于这行代码：

```
mov r0, 0x33d00000
```

的源操作数0x33d0000，可能是mov指令所不允许的，这句话后，可能有人会说，我知道，那是因为mov的操作数的值，不允许大于255，至少网上很多人的资料介绍中，都是这么说的。

对此，要说的是，你的回答是错误的。

关于mov操作数的真正的允许的取值范围，还真的不是那么容易就能搞懂的，下面就来详细解释解释。

总的来说，我是从这个帖子：

<http://blog.chinaunix.net/space.php?uid=20799298&do=blog&cuid=2055392>

里面，才算清楚mov的取值范围，以及找了相应的datasheet，才最终看懂整个事情的来龙去脉的。

首先，mov的指令，是属于ARM指令集中，数据处理（Data Process）分类中的其中一个指令，而数据处理指令的具体格式是：

ARM Processor Instruction Set

http://netwinder.osuosl.org/pub/netwinder/docs/arm/ARM7500FEvB_3.pdf

图表 31 数据处理指令的指令格式

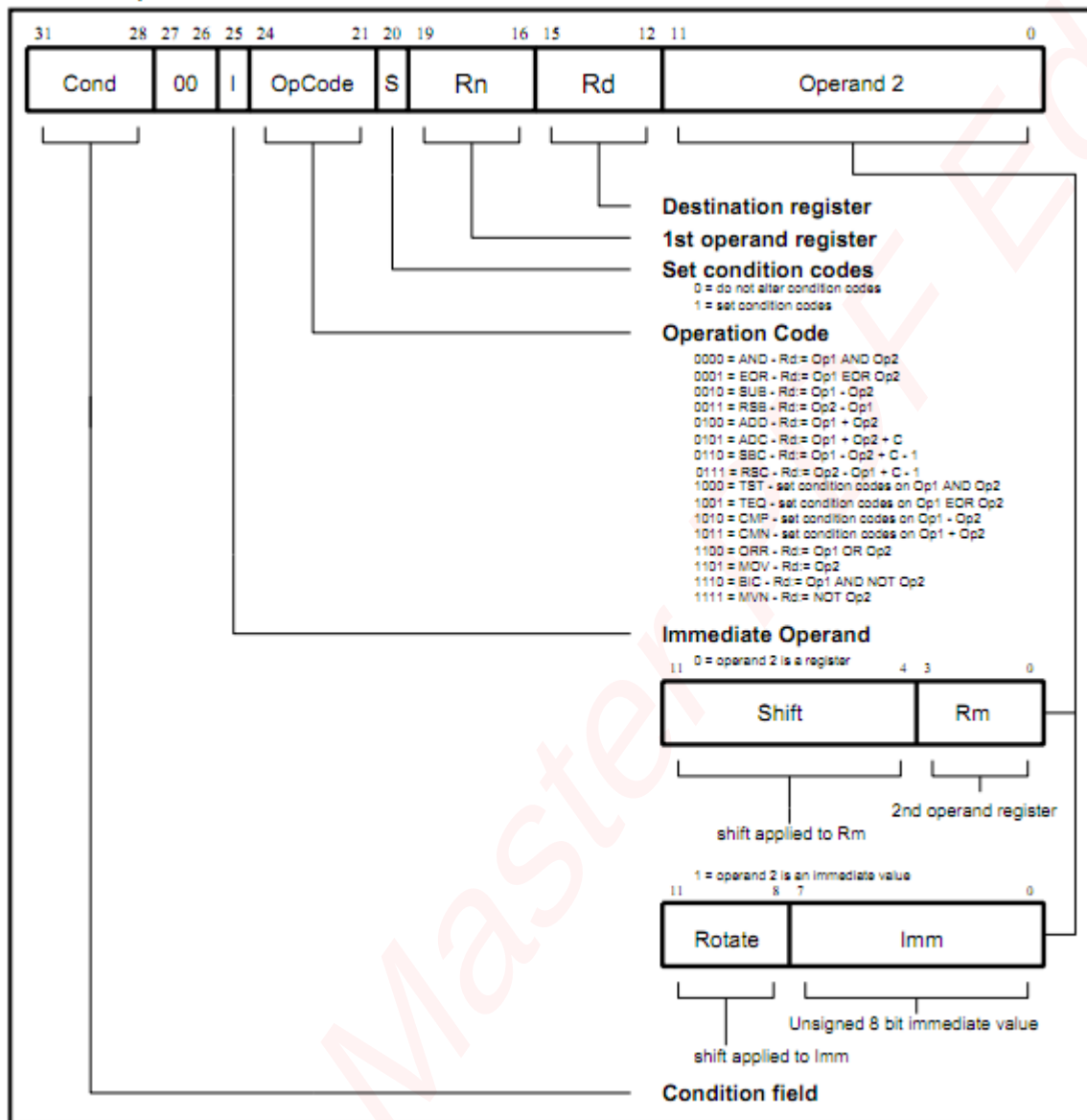


Figure 5-4: Data processing instructions

对于此格式，我们可以拿：

```
arm-linux-objdump -d u-boot > dump_u-boot.txt
```

中得到的汇编代码中关于：

```
ldr    r0, =0x53000000
```

所对应的，真正的汇编代码：

```
33d00068: e3a00453  mov r0, #1392508928 ; 0x53000000
```

来分析，就容易看懂了：

```
mov r0, #1392508928
```

```
= mov r0, #0x53000000
```

的作用就是，把0x53000000移动到r0中去。

其对应的二进制指令是上面的：

```
0xe3a00453 = 1110 0011 1010 0000 0000 0100 0101 0011 b
```

下面对照mov指令的格式，来分析这些位所对应的含义：

图表 32 mov 指令 0xe3a00453 的位域含义解析

31-28	27-26	25	24-21	20	19-16	15-12	11-0	
Condition Field	00	I (Immediate Operand)	OpCode (Operation Code)	S (Set Condition Code)	Rn (1st Operand Register)	Rd (Destination Register)	Operand 2 (1 = operand 2 is an immediate value)	
							11-8 Rotate	7-0 Imm
1110	00	1	1101	0	0000	0000	0100	0101 0011
		表明是立即数	1101对应的是MOV指令		MOV指令做的事情是: Rd:= Op2, 和Rn无关, 所以忽略这个Rn	表示0000号寄存器, 即r0	0100=4, 含义参见注释1	0x53

注释1:

上述datasheet中写到:

“5.4.3 Immediate operand rotates

The immediate operand rotate field is a 4 bit unsigned integer which specifies a shift operation on the 8 bit immediate value. This value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2.”

意思是, 对于bit[11:8]的值, 是个4位, 无符号的整型, 其指定了bit[7:0]的8bit立即数值的位移操作。具体如何指定呢, 那就是将bit[7:0]的值, 循环右移2x bit[11:8]位。对于我们的例子, 就是, 将bit[7:0]的值0x53, 循环右移 2xbit[11:8]= 2 x 4 = 8位, 而0x53循环右移8位, 就得到了0x53000000, 就是我们要mov值, mov到目的寄存器rd, 此处为r0中。

而上面英文最后一句说的是, 通过将bit[7:0]的值, 循环右移 2xbit[11:8]的方式, 就可以产生出很多个数值了, 即mov的操作数中, 其中符合可以通过0x00-0xFF循环右移偶数位而产生的数值, 都是合法的mov的操作数, 而这样的数, 其实是很多的。

【总结】

所以, mov指令的操作数的真正的取指范围, 即不是0-0xFF (0-255), 也不是只有2的倍数, 而是:

只要该数, 可以通过0x00-0xFF中某个数, 循环右移偶数位而产生, 就是合法的mov的操作数, 否则就是非法的mov的操作数。

嵌入式资源免费下载

总线协议:

1. [基于 PCIe 驱动程序的数据传输卡 DMA 传输](#)
2. [基于 PCIe 总线协议的设备驱动开发](#)
3. [CANopen 协议介绍](#)
4. [基于 PXI 总线 RS422 数据通信卡 WDM 驱动程序设计](#)
5. [FPGA 实现 PCIe 总线 DMA 设计](#)
6. [PCI Express 协议实现与验证](#)
7. [VPX 总线技术及其实现](#)
8. [基于 Xilinx FPGA 的 PCIE 接口实现](#)
9. [基于 PCI 总线的 GPS 授时卡设计](#)
10. [基于 CPCI 标准的 6U 信号处理平台的设计](#)
11. [USB30 电路保护](#)
12. [USB30 协议分析与框架设计](#)
13. [USB 30 中的 CRC 校验原理及实现](#)
14. [基于 CPLD 的 UART 设计](#)
15. [IPMI 在 VPX 系统中的应用与设计](#)
16. [基于 CPCI 总线的 PMC 载板设计](#)
17. [基于 VPX 总线的工件台运动控制系统研究与开发](#)

VxWorks:

1. [基于 VxWorks 的多任务程序设计](#)
2. [基于 VxWorks 的数据采集存储装置设计](#)
3. [Flash 文件系统分析及其在 VxWorks 中的实现](#)
4. [VxWorks 多任务编程中的异常研究](#)
5. [VxWorks 应用技巧两例](#)
6. [一种基于 VxWorks 的飞行仿真实时管理系统](#)
7. [在 VxWorks 系统中使用 TrueType 字库](#)
8. [基于 FreeType 的 VxWorks 中文显示方案](#)
9. [基于 Tilcon 的 VxWorks 简单动画开发](#)
10. [基于 Tilcon 的某武器显控系统界面设计](#)
11. [基于 Tilcon 的综合导航信息处理装置界面设计](#)

12. [VxWorks 的内存配置和管理](#)
13. [基于 VxWorks 系统的 PCI 配置与应用](#)
14. [基于 MPC8270 的 VxWorks BSP 的移植](#)
15. [Bootrom 功能改进经验谈](#)
16. [基于 VxWorks 嵌入式系统的中文平台研究与实现](#)

Linux:

1. [Linux 程序设计第三版及源代码](#)
2. [NAND FLASH 文件系统的设计与实现](#)
3. [多通道串行通信设备的 Linux 驱动程序实现](#)
4. [Zsh 开发指南-数组](#)
5. [常用 GDB 命令中文速览](#)
6. [嵌入式 C 进阶之道](#)
7. [Linux 串口编程实例](#)
8. [基于 Yocto Project 的嵌入式应用设计](#)
9. [Android 应用的反编译](#)
10. [基于 Android 行为的加密应用系统研究](#)
11. [嵌入式 Linux 系统移植步步通](#)
12. [嵌入式 C++ 语言精华文章集锦](#)
13. [基于 Linux 的高性能服务器端的设计与研究](#)
14. [S3C6410 移植 Android 内核](#)
15. [Android 开发指南中文版](#)
16. [图解 Linux 操作系统架构设计与实现原理（第二版）](#)

Windows CE:

1. [Windows CE.NET 下 YAFFS 文件系统 NAND Flash 驱动程序设计](#)
2. [Windows CE 的 CAN 总线驱动程序设计](#)
3. [基于 Windows CE.NET 的 ADC 驱动程序实现与应用的研究](#)
4. [基于 Windows CE.NET 平台的串行通信实现](#)
5. [基于 Windows CE.NET 下的 GPRS 模块的研究与开发](#)
6. [win2k 下 NTFS 分区用 ntldr 加载进 dos 源代码](#)
7. [Windows 下的 USB 设备驱动程序开发](#)
8. [WinCE 的大容量程控数据传输解决方案设计](#)
9. [WinCE6.0 安装开发详解](#)
10. [DOS 下仿 Windows 的自带计算器程序 C 源码](#)

11. [G726 局域网语音通话程序和源代码](#)
12. [WinCE 主板加载第三方驱动程序的方法](#)
13. [WinCE 下的注册表编辑程序和源代码](#)
14. [WinCE 串口通信源代码](#)
15. [WINCE 的 SD 卡程序\[可实现读写的源码\]](#)
16. [基于 WinCE 的 BootLoader 研究](#)

PowerPC:

1. [Freescale MPC8536 开发板原理图](#)
2. [基于 MPC8548E 的固件设计](#)
3. [基于 MPC8548E 的嵌入式数据处理系统设计](#)
4. [基于 PowerPC 嵌入式网络通信平台的实现](#)
5. [PowerPC 在车辆显控系统中的应用](#)
6. [基于 PowerPC 的单板计算机的设计](#)
7. [用 PowerPC860 实现 FPGA 配置](#)

ARM:

1. [基于 DiskOnChip 2000 的驱动程序设计及应用](#)
2. [基于 ARM 体系的 PC-104 总线设计](#)
3. [基于 ARM 的嵌入式系统中断处理机制研究](#)
4. [设计 ARM 的中断处理](#)
5. [基于 ARM 的数据采集系统并行总线的驱动设计](#)
6. [S3C2410 下的 TFT LCD 驱动源码](#)
7. [STM32 SD 卡移植 FATFS 文件系统源码](#)
8. [STM32 ADC 多通道源码](#)
9. [ARM Linux 在 EP7312 上的移植](#)
10. [ARM 经典 300 问](#)
11. [基于 S5PV210 的频谱监测设备嵌入式系统设计与实现](#)

Hardware:

1. [DSP 电源的典型设计](#)
2. [高频脉冲电源设计](#)
3. [电源的综合保护设计](#)
4. [任意波形电源的设计](#)
5. [高速 PCB 信号完整性分析及应用](#)
6. [DM642 高速图像采集系统的电磁干扰设计](#)
7. [使用 COMExpress Nano 工控板实现 IP 调度设备](#)