romInit()函数执行过程研究

SWD 聂勇

版 本 历 史

版本/状态	责任人	起止日期	备注
V1.0/正式	聂勇	26Oct2010	romInit()函数执行过程研究
V1.1/正式	聂勇	12Nov2010	细节修改

目　录

1. 说明

下面没有特别说明，提到的地址都是指的虚拟地址（Virtual Address）。

2. 硬件初始化

上电之后开始执行的地址是 0xbfc0.0000 处。从 0xbfc0.0000 开始的 0x400 字节（1K）的空间被作为默认的异常向量表。在 0xbfc0.0000 处的指令就是执行一个跳转，跳转到 0xbfc0.0400 处，这里是硬件初始化的地方。

下面按照执行的顺序，一步步来分析这个启动的过程。

romInit.s

```
……
    li      v0,SR_BEV
    ……
    mtc0    v0,C0_SR
    mtc0    zero,C0_CAUSE
    mtc0    zero,$18        # C0_WATCHLO
    mtc0    zero,$19        # C0_WATCHHI
……
```

SR_BEV 表示 CP0 的 Status Register 的 BEV 位被设置为 1，此时 v0 中的值为 0x4000.0000。Status Register 的 BEV 位是用来控制异常向量的位置（control the location of exception vector），1: 0xbfc00380, 0: 0x80000180。显然，这里被设置为 1，则异常向量位于 0xbfc00380 处。

查看 0xbfc00380 处的代码，可以看到

romInit.s

```
……
    XVECENT(romExcHandle,0x380)    /* bfc00380: RC32364 general vector */
……
```

同时可以看到，CP0 的 Status Register 的 IE 位被设置为 0，所有终端都被关闭。

romInit.s

```
……
    li      t0, 2
    mtc0    t0, $16
    nop
```

```
……
```

这里配置 CP0 的 config0 寄存器的 k0 field 为 010。意思是 kseg0 被设置为 nocacheable。

CP0 的 config0 寄存器的 k0 field

K0 : cache type setting for memory access in Kseg0,

000- cacheable, write-through, not allocated on write misses

001- cacheable, write-through, allocated on write misses

010- noncacheable

011- cacheable, write-back

接下来就是扩展总线（external bus interface EBI）接口模块寄存器的设置以及时钟的设置。

romInit.s

```
……
        /* Enable M_CS1 CS01 Enable*/
        li     a3, 1
        sw   a3, 0x100(a2)     /* Enable EBI for Parallel/Async mode*/

        /* Enable M_CS2 */
        sw   a3, 0x110(a2)     /* CS23Config, Enable */

        /* Enable MCS3 */
        sw   a3, 0x120(a2)     /* CS4Config, Enable Expansion bus */
        li     a3, 0x01020a0c   /* CS4 Timing parameters */
        sw   a3, 0x124(a2)      /* Set timing for Local bus */

        /* Set timing for CS01 CS23 Flashwait count */
        li    a3, 0x100010a
        sw   a3, 0x104(a2)      /* Set timing for Local bus */
        sw   a3, 0x114(a2)      /* Set timing for Local bus */
        sw   a3, 0x124(a2)      /* Set timing for Local bus */
        sw   a3, 0x12c(a2)      /* Set timing for Local bus */

        /* Change clock div on 4704 to 0x36 (54) */
        li     a2, CHIPC_BASE
```

```
        li    a3, 0x00000036 /* @ 100 Mhz backplane clock UART Freq = 1.85 Mhz */
        sw    a3, 0xa4(a2)
        lw    t5, 0(a2)          /* Read Chip ID */
……
```

下面，就是最重要的内存SDRAM的配置。

romInit.s

```
……
bal board_draminit
……
```

3. 内存 SDRAM 配置

内存控制器的配置是 romInit()函数的重点。有关 Memory Controller 的知识，请参考《BCM5836 vxWorks BSP guide for TP-LINK》和《BCM5836 VxWorks BSP User's Guide》。下面的分析将按照功能划分，采取逐句，逐段分析的方法。

函数（或者说是汇编代码中的标签）调用的整个过程参考下图：

3.1 查找 SDRAM Memory Controller 的位置

romInit.s Line 1064

```
……
    /* Scan for an SDRAM controller (a0) */
    li    a0, _KSEG1ADDR(SB_ENUM_BASE)
1:  lw    a1, (SBCONFIGOFF + SBIDHIGH)(a0)
    and a1, a1, SBIDH_CC_MASK
    srl   a1, a1, SBIDH_CC_SHIFT
    beq a1,SB_MEMC,read_nvram
    nop
    beq a1,SB_SDRAM,read_nvram
    nop
    addu      a0, SB_CORE_SIZE
    bne a1, (SBIDH_CC_MASK >> SBIDH_CC_SHIFT), 1b
    nop
……
```

分析：li a0, _KSEG1ADDR(SB_ENUM_BASE)

SB_ENUM_BASE 在 sbconfig.h 中定义，为 bcm5836 片上资源的中的 core 的起始地址，具体分布参考《启动过程研究_MIPS_NY.docx》的 Physics Address 一节。很显然，这里给出的 0x18000000 是一个物理地址，所以通过宏_KSEG1ADDR 映射到虚拟地址。

sbconfig.h Line 36

```
……
#define SB_ENUM_BASE      0x18000000   /* Enumeration space base */
#define  SB_ENUM_LIM   0x18010000  /* Enumeration space limit */
……
```

分析：

```
1:   lw    a1, (SBCONFIGOFF + SBIDHIGH)(a0)
     and a1, a1, SBIDH_CC_MASK
     srl   a1, a1, SBIDH_CC_SHIFT
     beq a1,SB_MEMC,read_nvram
     nop
     beq a1,SB_SDRAM,read_nvram
```

获取从 a0 开始的 SBCONFIGOFF（0xf00）+ SBIDHIGH（0xfc）地址处寄存器中的值（是 SOC 片上资源的一个配置寄存器）到 a1 寄存器，然后取出其中 SBIDH_CC_MASK，然后与宏 SB_MEMC，SB_SDRAM 比较，如果相等，就说明在这片 SOC 上找到了 Memory Control Core ，然后就可以跳转到 Memory Control Core 配置代码处。

以上过程并不是必须得，因为 Memory Control Core 在我们的芯片上已经存在，参考芯片的说明文档，我们可以知道，Memory Control Core 的位置是 0x18008000 处。

3.2 初始化 Memory Control Core

在文档中，read_nvram 标签处没有做任何事情，只是跳转到了 init 标签处。在开发板上，我们使用的是 SDR SDRAM 芯片，所以在 config.h 中定义了宏 SDRAM_MODE。下面分析针对 SDR SDRAM 芯片，Memory Control Core 的初始化。

3.2.1 NCDL 测试

NCDL（Numerical Controlled Delay Lines）测试是用来调整输出，输入延时。整个过程比较复杂，在 CFE 和 vxWorks BSP 中，NCDL 过程自动配置被实现了。下面的代码就是 init 标签处跳转到 sdr_find_ncdl 处开始执行。

sbconfig.h Line 36

```
……
memc_sdr_init:
    li    t0,MEMC_SDR_INIT
    li    t1,MEMC_SDR_MODE
    li    t3,MEMC_SDR_NCDL
    beqz      t3,sdr_find_ncdl   # Do we have ncdl values?
    nop

    li    t4,-1
    bne t3,t4,break_sdr_ncdl
    nop
……
```

3.2.2 初始化 SDRAM

sdr_do_init 标签处开始是对 SDRAM 的初始化。其中，SDR SDRAM 和 DDR SDRAM 使用到了很多相同的代码段，这些代码段放置在从标签 szmem 到标签 memdone 位置的所有代码。

3.3 DDR SDRAM 与 SDR SDRAM 使用相同的代码段

从标签 szmem 到标签 memdone 位置的所有代码。

对于其中的每一个代码段，都需要认真分析。

4. romReboot、romWarm、copyLoop

这三项是在内存配置结束，跳转到 C 函数 romStart()函数之前还需要做的三件事情。有关其中的实现细节，下面给予分析。

有关热启动的顺序问题一直没有明白。还有就是这里为什么要命名为 romReboot 和 romWarm 呢？在冷启动的过程中，这两段代码也是一定需要去执行的，而且完成了 cache 初始化等工作。

copyLoop 是蔡培丰添加的代码，对于数量为 M 级别的代码段的拷贝。但是其实现的动机和操作方式还有待研究。

有关从整体上把握 cache，请参考《usrInit()函数.docx》的 cache 一节。

4.1 romReboot 的作用

什么时候，这段代码会执行呢？

在 cold start 的时候，这段代码会执行。

在 warm start 的时候，上面的硬件初始化，内存配置代码段都不会执行，而是直接跳转到此处开始执行。可以看到位于 0xbfc0.0000 开始的 0x400 字节（1K）的空间的异常向量，第一条是正常启动跳转到 0xbfc0.0400 处 romInit()的入口，而第二条就是跳转到 romReboot 函数。

romInit.s

```
……
romInit:
_romInit:
    .set noreorder
    RVECENT(__romInit,0)        /* PROM entry point */
    RVECENT(romReboot,1)        /* software reboot */
……
```

那么，什么时候会跳转到这第二条异常向量处呢？

那就是在 sysLib.c 或者 sysLib.s 文件中定义的 sysToMonitor()函数。有关热启动的相关内容，可以仔细分析该函数。

4.2 Cache 的初始化

Cache 的初始化在 romReboot 这个段中调用 romCacheReset 段完成。下面就分析者一部分代码。

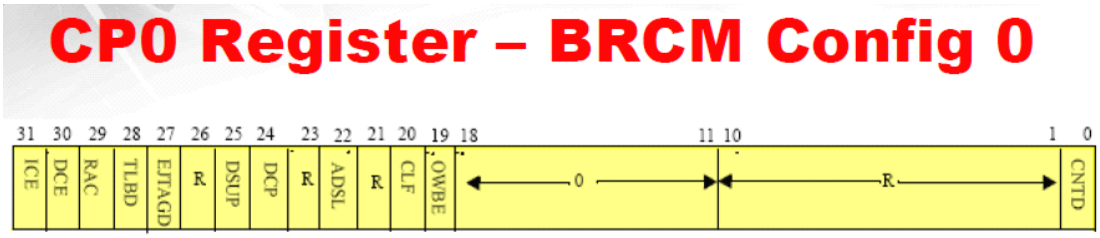
4.2.1 开启 I-cache 和 D-cache

romReboot 首先做的几件事情：

- 关闭所有中断
- 清除所有软件中断
- 使能 I-Cache 和 D-Cache

配置 CPU 控制寄存器（CPU control registers 或者叫做 coprocessor registers）的$22。$22 号寄存器有 BRCM Config 0、BRCM Config 3、BRCM Config 4、BRCM Config 5，这里配置的是 BRCM Config 0。

有关 BRCM Config 0 寄存器每一位的描述，请看下图：



可以看到，注意第 30,31 位。

- Bit 30: set value 1 to enable data cache
- Bit 31: set value 1 to enable instruction cache

这一段代码就是将 30,31 这两位设置为 1，开启 I-Cache 和 D-Cache。

sbconfig.h Line 36

```
……
    mfc0    v0,$22
    nop
    nop
    nop
    or      v0,0xc0000000       /* Enable both I$ and D$ */
    mtc0    v0,$22
    nop
    nop
……
```

- 关闭 prefetch cache

prefetch cache（PFC）控制寄存器的位置是在 CPU Register Space，开始位置是 0xff40.0000。

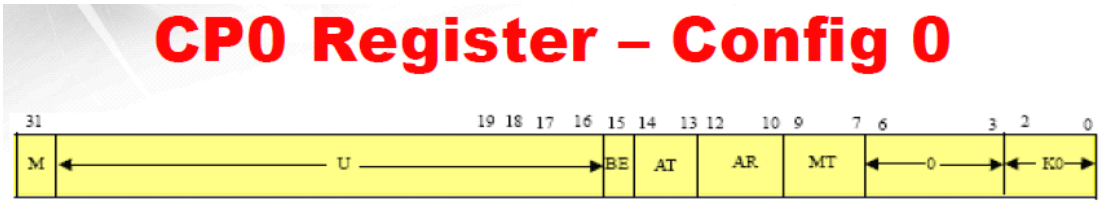
romInit.s

```
……
  /* Enable Prefetch Cache (RAC) */
    lui     v0, 0xff40
    lui     t1, 0x0400
    sw      t1, 4(v0)
    li      t1, 0x0000              /* Disable Prefectch for I$ & D$ */
    sw      t1, 0(v0)
……
```

- 设置 kseg0 这个地址区域的 cache 模式

配置 CPU 控制寄存器（CPU control registers 或者叫做 coprocessor registers）的$16，配置为 cache write back 模式。

$16 号寄存器有 Config 0、Config 1，这里配置的是 Config 0。



- K0 : cache type setting for memory access in Kseg0,

000- cacheable, write-through, not allocated on write misses

001- cacheable, write-through, allocated on write misses

010- noncacheable

011- cacheable, write-back

romInit.s

```
……
    #define CACHE_MODE_BITS 3
……
    /* set k0 cache mode*/
    mfc0    t2, $16
    nop

    and t2, ~(CACHE_MODE_BITS)
    or  t2, CACHE_WRITE_BACK
    mtc0    t2, $16
    nop
……
```

显然，这里将$16 的 K0 设置成了 011，也就是 cacheable，write-back 模式。

4.2.2 初始化 cache 寄存器

romInit.s　　romCacheReset 函数

```
……
.set mips3
.set noreorder
.ent romCacheReset
romCacheReset:

    /*   a0 = cache size
     *   a1 = line   size
     */
    li          a0, ICACHE_SIZE
    li          a1, CACHELINE_SIZE

    /* CFLUSH */

    mtc0        zero, C0_TAGLO
    mtc0        zero, C0_TAGHI              /* TagHi is not really used */

    /* Calc an address that will correspond to the first cache line */
    li      a2, KSEG0BASE

    /* Calc an address that will correspond to the last cache line   */
    addu        a3, a2, a0
    subu        a3, a1

    /* Loop through all lines, invalidating each of them */
1:
    cache     ICACHE_INDEX_STORE_TAG, 0(a2)    /* clear tag */
    bne a2, a3, 1b
    addu        a2, a1

    /* DFLUSH INV */

    /*   a0 = cache size
     *   a1 = line   size
```

```
        */
        li          a0, DCACHE_SIZE
        li          a1, CACHELINE_SIZE

        mtc0        zero, C0_TAGLO
        mtc0        zero, C0_TAGHI              /* TagHi is not really used */

        /* Calc an address that will correspond to the first cache line */
        li      a2, KSEG0BASE

        /* Calc an address that will correspond to the last cache line    */
        addu        a3, a2, a0
        subu        a3, a1

        /* Loop through all lines, invalidating each of them */
1:
        cache     DCACHE_INDEX_STORE_TAG, 0(a2)  /* clear tag */
        bne a2, a3, 1b
        addu        a2, a1

        j       ra
.end        romCacheReset
.set        mips0
.set        reorder
……
```

5. 跳转到 C 函数 romStart()

5.1 地址重定位 RELOC

- 首先明确几个概念物理地址、虚拟地址、链接地址、指令所在地址（程序地址）
- 然后了解链接的过程，最重要的是链接器 ldmips 的选项-Ttext。这样有助于理解什么是地址无关的代码（PIC）
- 第三点需要明确 MIPS CPU 的跳转指令，尤其是其中的 PC-relative 跳转和绝对地址跳转

当上面三点有了一定的概念之后，就可以分析下面的从汇编代码 romInit()跳转到 C 代码 romStart()的整个过程。

为了代码阅读的方便，跳转代码中一些宏定义被删除。

romInit.s Line 566

```
……
    /* set stack to grow down from beginning of data and call init */
    la    gp, _gp              # set global ptr from compiler
    la    sp, STACK_ADRS       # set stack to begin of data
    /*    la    sp, RAM_LOW_ADRS          # set stack to begin of data */
    move      a0, s0                      # push arg = start type
    sync                          /* flush any last-minute writes */
    RELOC(t0, romStart)
    jal   t0              # never returns - starts up kernel
……
```

以上代码，需要重点分析的就是 RELOC(to,romStart)这个宏。该宏的定义如下：

romInit.s Line 1064

```
……
#define RELOC(toreg,address)          \
            bal         9f                    ; \
9:                                            ; \
            la          toreg,address         ; \
            addu        toreg,ra              ; \
            la          ra,9b                 ; \
            subu        toreg,ra
……
```

RELOC(toreg,address)宏就是 vxworks bootrom 位置无关代码的处理手法。 romStart 的链接地址是在 ram 所在的地址，而实际运行是在 flash 上，我们需要将 romStart 的地址换成在 flash 中的位置。首先，将链接时生成 romStart 的地址放入 to 寄存器中，然后加上当前运行位置的地址（addu toreg,ra ;的 ra 中），然后减去链接时生成的当前位置的地址(la ra,9b;的 ra 中)，这样就得到实际运行时 romStart 的相对地址（最后存放在 t0 中），这样就可以直接跳转过去了。

将上面整个过程分解到每一行汇编代码中，将 RELOC(to,romStart)展开，如下所示。注意，指令前面的地址（例如 0Xbfc01004 等）为当前运行位置的地址，并不是链接是生成

的地址。那么，链接生成的可执行文件中是如何呢？我们将可执行文件 bootrom 文件反汇编，可以看到其中的真实面目。

romInit.s Line 1064

```
……
RELOC(t0,romStart)   \
0Xbfc01004: bal    9f; \                    /* ra 寄存器存放 addu 指令的地址  */
9:   ;\
0Xbfc01008:la     t0, romStart;   \     /*  将 romStart 的链接地址放入 t0 寄存器  */
0Xbfc01010:addu t0, ra;            \      /*  将 ra 寄存器的值加 t0 的之后存入 t0 */
0Xbfc01014:la ra, 9b; \              /*  将标号 9 所在的地址放入 t0，该地址是链接后的地址  */
0Xbfc0101c:subu t0, ra     \              /* ra 寄存器的之减 t0 寄存器的值存入 t0 */
 jal          t0                                 /*  跳转到 t0 并且 ra 寄存器存放 nop 后的指令地址  */
 nop
……
```

执行 bal 之后，ra 寄存器的值为 0Xbfc0100c；

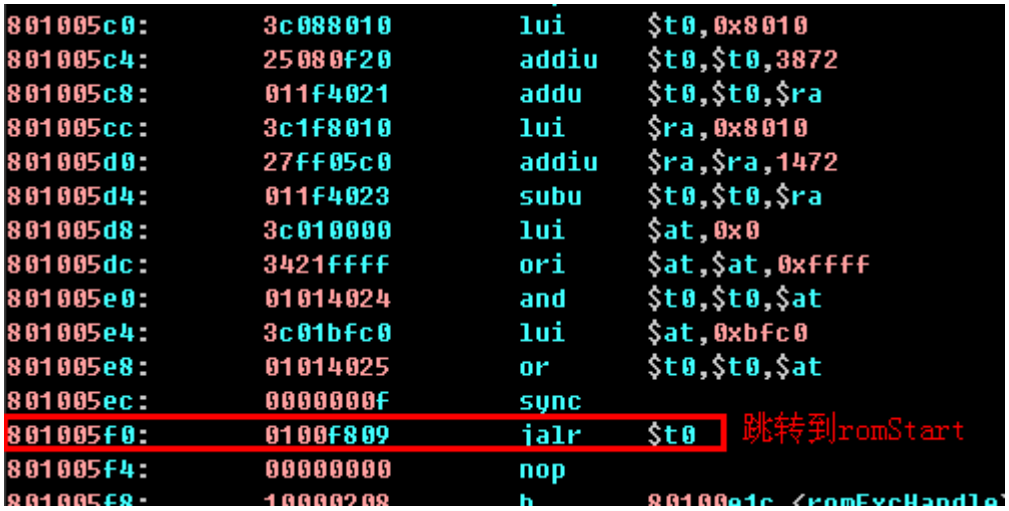
执行 la 之后，t0 寄存器存放的是 romStart 的链接地址。

执行 addu 之后，t0 寄存器存放的是 0Xbfc0100c+romStart 的链接地址。

执行 la 之后，ra 寄存器存放的是 0xbfc01008 的地址对应的链接地址。

执行 subu 之后，t0 寄存器存放的是 0xbfc01008 的地址对应的链接地址-（0Xbfc0100c+romStart），此时 t0 寄存器存放的即是 romStart 对应的 ROM 地址。

MIPS 的函数调用使用 jal 指令，返回地址位于$ra($31)中，返回地址是下下一条指令。紧跟分支指令后的下一条指令是分支延时槽，这条指令总是在函数调用前之前。因此所有 jal 指令后的下一条指令都是 nop 指令。MIPS 函数返回时使用 jr ra



在跳转到 C 函数 romStart()之前，还有 gp、sp 两个寄存器的设置。gp 的初始化除了在此处之外，还在跳转到 RAM_HIGH_ADRS 的 usrStart()函数的时候有一次初始化。具体这两次的区别。有关 gp 的相关问题，可以参考阅读《MIPS 汇编_NY.docx》。