# RT-Thread for Nios II 移植手记

目录



Author: aozima
2011-3-4
http://www.rt-thread.org

# 1. Nios II 介绍

Nios II 小册子: http://www.altera.com.cn/literature/br/br_NiosII_SC.pdf
Nios II 处理器参考手册: www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

# 2. Nios II 寄存器介绍

| 寄存器 | 助记符 | 功能 | 说明 |
|---|---|---|---|
| R0 | zero | 清零 | 总是存放 0 值，对它读写无效。<br>Nios II 没有专门的清零指令，所以常用它来对寄存器清零 |
| R1 | at | | 汇编中的临时变量 |
| R2 | | 返回值(低 32 位) | 用于返回一个值给调用者，r3 存放返回值的高 32 位. 如果这两个寄存器不够存放需要返回的值，编译器将通过堆栈来传递. |
| R3 | | 返回值(高 32 位) | |
| R4 | | 第一个参数 | 用来传递 4 个非浮点参数给一个子程序，r4 传递第一个参数，r5 传递第二个参数，以此类推，如果这四个寄存器不够传递参数，编译器将通过堆栈来传递 |
| R5 | | 第二个参数 | |
| R6 | | 第三个参数 | |
| R7 | | 第四个参数 | |
| R8-R15 | | | 调用者要保存的寄存器 |
| R16-R23 | | | 子程序要保存的寄存器 |
| R24 | et | 为异常处理保留 | 在程序异常或断点处理时使用. 使用时，可以不恢复原来的值. 该寄存器很少作其它用途. |
| R25 | bt | 为程序断点保留 | |
| R26 | gp | 全局指针 | 它指向静态数据区中的一个运行时临时决定的地址. 这意味着在存取位于 gp 值上下 32KB 范围内的数据时，只需要一条以 gp 作为基指针的指令即可完成. |
| R27 | sp | 栈指针 | 堆栈指针. Nios II 没有专门的出栈(POP)入栈(PUSH)指令，在程序中，以 sp 为基址，用寄存器基址 + 偏移地址的方式来访问处理栈. |
| R28 | fp | 帧指针 | 帧指针. 习惯上用于跟踪栈的变化和维护运行时环境. |
| R29 | ea | 异常返回地址 | 保存异常返回地址 |
| R30 | ba | 断点返回地址 | 保存断点返回地址 |
| R31 | ra | 函数返回地址 | 保存函数返回地址 |

# 3. Nios II 指令集介绍

Nios II 使用类 MIPS 体系，指令与 MIPS 基本一致. 但比较简单.
Instruction Set Reference, Nios II Processor Reference Handbook:
http://www.altera.com/literature/hb/nios2/n2cpu_nii51017.pdf

# 4. Nios II 编译器介绍

Nios II 所使用的编译器从 GCC 移植，兼容大部分 GCC 所支持的特性.

* 传递非浮点参数给子程序: r4-r7.
* 传递函数的返回值: R2-R3.
* 栈生长方向: 由上向下生长.
* 大小端模式: 默认小端模式.

# 5. 需要实现的函数

* `rt_base_t rt_hw_interrupt_disable(void);`
  关中断，并返回之前的中断状态
* `void rt_hw_interrupt_enable(rt_base_t level);`
  开中断，参数为中断状态，一般为 rt_hw_interrupt_disable 的返回值
* `void rt_hw_context_switch_to(rt_uint32_t to);`
  上下文切换: 切换至 to，用于系统启动时第一个线程的启动
* `void rt_hw_context_switch(rt_uint32_t from, rt_uint32_t to);`
  上下文切换: 由 from 切换到 to，用于线程正常执行时被自己把自己挂起时切换

Author: aozima
2011-3-4
http://www.rt-thread.org

* **void rt_hw_context_switch_interrupt**(rt_uint32_t from, rt_uint32_t to);
  上下文切换: 在中断状态下由 from 切换到 to. 一般在本函数中并不切换,
  只记录需要切 换的 from, to 两个线程的信息, 待中断退出后再执行真正的切换.

## 6. 需要系统提示的服务

* tick
* rt_kprintf 打印(非必要, 用于方便调试)

## 7. 线程切换移植过程笔记

Nios II 使用类 MIPS 体系, 寄存器结构与指令与 MIPS 基本兼容. 但比较简单.

* 首先需要实现的 rt_hw_interrupt_disable 函数. 功能是关中断并返回执行前的中断状态. Nios II 的总中断通过 status 寄存器的 PIE 位([0])进行控制, 为 1 是打开中断, 为 0 时关闭中断. 因此, 我们先把 status 保存在 r2(返回值), 然后再判断中断即可.

```
/*
 * rt_base_t rt_hw_interrupt_disable();
 */
.global rt_hw_interrupt_disable
.type rt_hw_interrupt_disable, %function
rt_hw_interrupt_disable:
    rdctl r2, status      /* return status */
    wrctl status, zero    /* disable interrupt */
    ret
```

* 对应地需要实现 rt_hw_interrupt_enable 函数. 功能是按参数打开中断. Nios II 使用 r4 来传递第一个参数, 因此, 我们只需要把 r4 写入到 status 即可.

```
/*
 * void rt_hw_interrupt_enable(rt_base_t level);
 */
.global rt_hw_interrupt_enable
.type rt_hw_interrupt_enable, %function
rt_hw_interrupt_enable:
    wrctl status, r4      /* enable interrupt by argument */
    ret
```

* 上下文切换第一个要实现的就是 rt_hw_context_switch_to, 用于系统启动时切换到第一个线程. 在做系统上下文切换时, 先要确定是线程切换时需要保存的寄存器, 以及保存的顺序, 用于保存和恢复时匹配. 在线程做初始化时, rt_hw_stack_init()会进行线程运行前的栈的初始化, 并把线程入口, 参数, 及线程退出时调用的函数设置好. rt_hw_stack_init()代码如下:

```
/**
 * This function will initialize thread stack
 *
 * @param tentry the entry of thread
 * @param parameter the parameter of entry
 * @param stack_addr the beginning stack address
```

```
 * @param texit the function will be called when thread exit
 *
 * @return stack address
 */
rt_uint8_t *rt_hw_stack_init(void *tentry, void *parameter,
    rt_uint8_t *stack_addr, void *texit)
{
    unsigned long *stk;

    stk  = (unsigned long *)stack_addr;
    *(stk)   = 0x01;                        /* status  */
    *(--stk) = (unsigned long)texit;        /* ra  */
    *(--stk) = 0xdeadbeef;                  /* fp  */
    *(--stk) = 0xdeadbeef;                  /* r23 */
    *(--stk) = 0xdeadbeef;                  /* r22 */
    *(--stk) = 0xdeadbeef;                  /* r21 */
    *(--stk) = 0xdeadbeef;                  /* r20 */
    *(--stk) = 0xdeadbeef;                  /* r19 */
    *(--stk) = 0xdeadbeef;                  /* r18 */
    *(--stk) = 0xdeadbeef;                  /* r17 */
    *(--stk) = 0xdeadbeef;                  /* r16 */
//  *(--stk) = 0xdeadbeef;                  /* r15 */
//  *(--stk) = 0xdeadbeef;                  /* r14 */
//  *(--stk) = 0xdeadbeef;                  /* r13 */
//  *(--stk) = 0xdeadbeef;                  /* r12 */
//  *(--stk) = 0xdeadbeef;                  /* r11 */
//  *(--stk) = 0xdeadbeef;                  /* r10 */
//  *(--stk) = 0xdeadbeef;                  /* r9  */
//  *(--stk) = 0xdeadbeef;                  /* r8  */
    *(--stk) = 0xdeadbeef;                  /* r7  */
    *(--stk) = 0xdeadbeef;                  /* r6  */
    *(--stk) = 0xdeadbeef;                  /* r5  */
    *(--stk) = (unsigned long)parameter;    /* r4 argument */
    *(--stk) = 0xdeadbeef;                  /* r3  */
    *(--stk) = 0xdeadbeef;                  /* r2  */
    *(--stk) = (unsigned long)tentry;       /* pc  */

    /* return task's current stack address */
    return (rt_uint8_t *)stk;
}
```

在执行rt_hw_context_switch_to时，线程还没有开始运行，所有状态都是被rt_hw_stack_init()初始化。因此，只需要把rt_hw_stack_init()中保存在栈中的状态全部恢复，然后再跳转到原来所保存的PC处运行即可。（注：此时线程还没有开始运行，线程函数

的入口为线程的入口)

因Nios II的PC不能直接被改变,也没有自动从栈中恢复PC的机制,只能通过ret(返回),eret(异常或中断返回),jmp(跳转),call(调用)来改变.又因Nios II置位status的PIE位后,即刻打开中断,如果此时有中断发生,那么现场即会遭到破坏.综合考虑下,决定使用eret来实现恢复后的跳转.

因为虽然eret是从异常返回,但Nios II并没明确的异常状态,eret仅是把estatus复制到status,然后跳转到ea处运行,然后estatus和ea就失效了.因此,我们把线程的status放入estatus,把线程的PC放入ea,然后执行eret,即可恢复status并跳转到线程继续执行.程序流程如下,具体代码请见发行包或SVN.

```
/*
 * void rt_hw_context_switch_to(rt_uint32 to);
 * r4: to
 */
.global rt_hw_context_switch_to
.type rt_hw_context_switch_to, %function
rt_hw_context_switch_to:
    /* save to thread */
    /* 把目标线程的SP指针保存到rt_interrupt_to_thread */
    stw r4,%gprel(rt_interrupt_to_thread)(gp)

    /* get sp */
    /* 更新当前的栈为线程的栈 */
    ldw sp, (r4)

    /* 先从栈中恢复 status到r2 */
    ldw r2,  68(sp)  /* status */
    /* 并保存到estatus */
    wrctl estatus, r2

    /* 恢复线程的PC */
    ldw ea, 0(sp)    /* thread entry */

    ldw r2,  4(sp)
    ldw r3,  8(sp)
......恢复其它寄存器......
    ldw fp,  60(sp)
    ldw ra,  64(sp)

    /* 因从栈中恢复了18个寄存器,因此栈要释放72字节 */
    addi sp, sp, 72

    /* estatus --> status,ea --> pc */
    /* 开始执行新线程 */
    eret
```

* 线程切换函数 rt_hw_context_switch.

当线程中执行会有阻塞的操作后,通过 rt_hw_context_switch 切换到最高优先级就绪线程. rt_hw_context_switch 在 rt_schedule()中被调用:

```
void rt_schedule()
{
    ..............
          if (rt_interrupt_nest == 0)
          { // call
             rt_hw_context_switch((rt_uint32_t)&from_thread->sp,
(rt_uint32_t)&to_thread->sp);
          } // retun
          else
          {
          }

   /* enable interrupt */
   rt_hw_interrupt_enable(level);
}
```

我们所希望的是在 rt_schedule 中,在 call 处调用 rt_hw_context_switch 切换到新线程去运行,待条件满足,再次切换本线程时,把栈恢复成 call 的内容,并运行到 return 处.

因此,我们在 rt_hw_context_switch 中,先保存当前线程的的状态,然后把 ra 保存为当前的 PC,然后更新 SP. 当前线程保存完毕. 下一步,就开始恢复新线程,恢复的过程与 rt_hw_context_switch_to 一样,程序流程如下,具体代码请见发行包或 SVN.

```
/*
 * void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
 * r4: from
 * r5: to
 */
.global rt_hw_context_switch
.type rt_hw_context_switch, %function
rt_hw_context_switch:
   /* save from thread */
   addi sp,sp,-72

   /* frist save r2,so that save status */
    /* 先保存r2,然后留出r2用于保存status */
   stw r2,  4(sp)

   /* save status */
   rdctl r2, status
   stw r2,  68(sp)  /* status */

   stw ra,  0(sp)  /* return from rt_hw_context_switch */
   stw r3,  8(sp)
```

```
 ....保存各寄存器....
stw fp,  60(sp)
stw ra,  64(sp)

/* save form thread sp */
/* from_thread->sp(r4) = sp */
stw sp, (r4)

/* update rt_interrupt_from_thread */
/* rt_interrupt_from_thread = r4(from_thread->sp) */
stw r4,%gprel(rt_interrupt_from_thread)(gp)

 /* 当前线程状态保存完毕,开始恢复新线程 */
/* update rt_interrupt_to_thread */
/* rt_interrupt_to_thread = r5 */
stw r5,%gprel(rt_interrupt_to_thread)(gp)

/* get to thread sp */
/* sp = rt_interrupt_to_thread(r5:to_thread->sp) */
ldw sp, (r5)

ldw r2,  68(sp)  /* status */
wrctl estatus, r2

ldw ea, 0(sp)    /* thread pc */
 ....与rt_hw_context_switch_to中恢复过程一样....
/* estatus --> status,ea --> pc */
 /* 执行到新线程 */
eret
```

* 线程切换函数 rt_hw_context_switch_interrupt.

这是最重要的一个切换,在发生中断后,如果有新的更高优先级别的线程就绪,将执行切换.比如释放信号量,有使用rt_thread_delay()挂起的线程定时时间到等.

在rt_hw_context_switch_interrupt的实现中,一般的作法都是不在其中进行切换,而是只在其中记录参与切换的两个线程的栈.待中断退出后再执行真正的线程切换.

Nios II内核在发生中断时,将自动把当前PC保存在ea寄存器中,并把status备份在estatus中,然后转到异常入口执行异常处理程序.异常处理程序首先会保存所有寄存器到栈中(包括ea),如果异常处理程序检测到异常是硬件中断引发的,将自动轮询中断标致位执行对应的中断服务函数,在中断服务返回后,将从栈中恢复所有寄存器,如果异常是硬件中断引发的,将把ea进行减4处理(如果是硬件中断,在中断发现时,PC所指的指令并没有得到执行),然后执行eret返回中断前的状态继续运行.因Nios II处理器没有提供pend服务,因此只能在中断退出时,检查是否需要执行线程切换,然后跳转到真正的线程切换程序,因此,我们在rt_hw_context_switch_interrupt()保存切换标致和原线程和新线程的栈:

```
/*
```

```
 * void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);
 * r4: from
 * r5: to
 */
.global rt_hw_context_switch_interrupt
.type rt_hw_context_switch_interrupt, %function
rt_hw_context_switch_interrupt:
    /* save ea -> rt_current_thread_entry */
     /* 中断发生时,自动保存pc到ea,因此,ea即为中断发生时的from线程的PC,如果是硬件中断,将自动减4 */
    addi ea,ea,-4
     /* 把原线程的pc保存在rt_current_thread_entry变量中 */
    stw ea,%gprel(rt_current_thread_entry)(gp)

    /* set rt_thread_switch_interrput_flag to 1 */
     /* 设置切换标致,以通知切换程序需要切换 */
    movi r2, 1
    stw r2,%gprel(rt_thread_switch_interrput_flag)(gp)

    /* update rt_interrupt_from_thread */
     /* 更新from线程线指针地址 */
    stw r4,%gprel(rt_interrupt_from_thread)(gp)

    /* update rt_interrupt_to_thread */
     /* 更新to线程栈指针地址 */
    stw r5,%gprel(rt_interrupt_to_thread)(gp)

    /* 从rt_hw_context_switch_interrupt返回 */
     ret
```

因中断的退出,被Nios II 的HAL驱动给接管了,因此,我们需要实现自定义的退出程序.

```
.globl .Lexception_exit

/* 根据HAL中的名称定义一个section,在链接时,自定义的优先级高于库中所带的 */
.section .exceptions.exit.label
.Lexception_exit:
.section .exceptions.exit, "xa"
       /* 以下过程为异常入口的反向过程,我们根据HAL提供的小做修改以达到目标 */
        /* 恢复estatus到r5,estatus即为中断发生时status的状态 */
        ldw r5, 68(sp)

       /* get exception back */
        /* 恢复 ea,此值为异常发生时的PC值  */
       ldw ea, 72(sp)
```

```
        /* if(rt_thread_switch_interrput_flag == 0) goto no_need_context */
         /* 读取rt_thread_switch_interrput_flag的值到r4 */
        ldw r4,%gprel(rt_thread_switch_interrput_flag)(gp)
         /* 判断r4是否为0,如果为0,则不需要进行切换,直接跳到no_need_context */
        beq r4,zero,no_need_context

need_context:
        /* 如果需要切换,则更新ea的值为rt_hw_context_switch_interrupt_do, */
         /* 则更新ea的值为rt_hw_context_switch_interrupt_do为我们实现的线程切换函数
*/
         movia ea, rt_hw_context_switch_interrupt_do
        /* disable interrput */
         /* 因前面我们恢复estatus到r5,因此我们这里清零r5以关闭中断 */
         /* 确保rt_hw_context_switch_interrupt_do执行切换时不会被打断 */
        mov r5, zero

no_need_context:
        /* 回写estatus */
        wrctl estatus, r5

        /*
         * Leave a gap in the stack frame at 4(sp) for the muldiv handler to
         * store zero into.
         */
         ldw   ra,   0(sp)
         /** 恢复其它寄存器 **/
        ldw   r14, 60(sp)
        ldw   r15, 64(sp)

        /* 恢复异常发生时的sp值 */
         addi  sp, sp, 76

        /* 从异常返回 estatus --> status ea --> pc */
         /* 如果需要执行切换,则自动执行rt_hw_context_switch_interrupt_do */
         /* 如果不需要执行切换,则返回到异常发生时的状态 */
         eret
```

接下来我们需要实现rt_hw_context_switch_interrupt_do,的实现与基本相似.主要不同的是的原线程的sp和pc直接从寄存器中取,而得使用我们预先保存的.

```
/* void rt_hw_context_switch_interrupt_do(void) */
.global rt_hw_context_switch_interrupt_do
.type rt_hw_context_switch_interrupt_do, %function
rt_hw_context_switch_interrupt_do:
```

```
/* save from thread */
addi sp,sp,-72

/* frist save r2,so that save status */
/* 先保存r2,然后留出r2用于保存status */
stw r2,  4(sp)

/* save status */
/* when the interrupt happen,the interrupt is enable */
/* 这是rt_hw_context_switch_interrupt_do,因此,线程当时肯定处于中断打开状态 */
/* 因此,我们不必再在中断中保存estatus了,直接设置为打开即可 */
movi r2, 1
stw  r2, 68(sp)  /* status */

stw r3,  8(sp)
stw r4,  12(sp)

/* get & save from thread pc */
 /* 保存原线程的pc,在中断中被保存在rt_current_thread_entry变量里 */
ldw r4,%gprel(rt_current_thread_entry)(gp)
stw r4,  0(sp)  /* thread pc */

stw r5,  16(sp)
stw r6,  20(sp)
/* 保存其它寄存器 */
stw fp,  60(sp)
stw ra,  64(sp)

/* save from thread sp */
/* rt_interrupt_from_thread = &from_thread->sp */
ldw r4, %gprel(rt_interrupt_from_thread)(gp)
/* *r4(from_thread->sp) = sp */
stw sp, (r4)

/* clear rt_thread_switch_interrput_flag */
/* rt_thread_switch_interrput_flag = 0 */
stw zero,%gprel(rt_thread_switch_interrput_flag)(gp)

/* 原线程状态保存完毕,开始恢复新线程 */
 /* load to thread sp */
/* r4 = rt_interrupt_to_thread(&to_thread->sp) */
ldw r4, %gprel(rt_interrupt_to_thread)(gp)
/* sp = to_thread->sp */
ldw sp, (r4)
```

```
    ldw r2,  68(sp)  /* status */
    wrctl estatus, r2

    ldw ea,   0(sp)  /* thread pc */
    /* 恢复其它寄存器 */
    ldw ra,  64(sp)

    /* 从栈中弹出了寄存器,然后把sp前移 */
    addi sp, sp, 72

    /* estatus --> status,ea --> pc */
    /* 执行到新线程 */
    eret
```

## 8. finsh shell 移植过程笔记

finsh使用device方式来进行输入输出,接口方面并没有什么特别注意的地方.但finsh shell 需要保存函数列表和变量列表,因此在系统中添加两个 section: FSymTab, VSymTab.

在一般的系统中,如果使用 GCC 编译器的话我们一般是自己手动编写链接脚本,并在里面添加这两个 section.但 Nios II IDE 把这些都封装好了,不好进入自定义修改,为此,从 9.1 版开始,Nios II IDE 添加了一个 BSP Editor 的工具,用来设置相关自定义参数.我们可以在这里添加这两个 section,但无法为这两个 section 添加 start 和 end 变量.阅读自动生成的 linker.x 文件发现有自动添加_alt_partition 开头的 start/end 变量:

```
    FSymTab :
    {
        PROVIDE (_alt_partition_FSymTab_start = ABSOLUTE(.));
        *(FSymTab FSymTab.*)
        . = ALIGN(4);
        PROVIDE (_alt_partition_FSymTab_end = ABSOLUTE(.));
    } > sdram

    VSymTab :
    {
        PROVIDE (_alt_partition_VSymTab_start = ABSOLUTE(.));
        *(VSymTab VSymTab.*)
        . = ALIGN(4);
        PROVIDE (_alt_partition_VSymTab_end = ABSOLUTE(.));
    } > sdram
```

因此,我们在rtconfig.h的finsh定义下面添加如下4行即可完美解决这个问题.

```
/* SECTION: finsh, a C-Express shell */
#define RT_USING_FINSH
/* Using symbol table */
#define FINSH_USING_SYMTAB
#define FINSH_USING_DESCRIPTION
#define __fsymtab_start _alt_partition_FSymTab_start
#define __fsymtab_end   _alt_partition_FSymTab_end
#define __vsymtab_start _alt_partition_VSymTab_start
```

```
#define __vsymtab_end   _alt_partition_VSymTab_end
```