kendryte K210 startup_xxx.S文件分析

源码

```
//Part1
//初始化mstatus状态寄存器使用，设置浮点控制状态，该条指令指示当前状态处于关闭FS功能。
#define MSTATUS_FS      0x00006000U /* initial state of FPU */

//导入cpuport.h头文件，改文件中对部分汇编指令重命名
#include <cpuport.h>

//Part2
  .global   _start
  .section ".start", "ax"

//Part3
_start:
  j 1f
  .word 0xdeadbeef
  .align 3
  .global g_wake_up
  g_wake_up:
      .dword 1
      .dword 0

//Part4
1:
  csrw mideleg, 0
  csrw medeleg, 0
  csrw mie, 0
  csrw mip, 0
  la t0, trap_entry
  csrw mtvec, t0

  li x1, 0
  li x2, 0
  li x3, 0
  li x4, 0
  li x5, 0
  li x6, 0
  li x7, 0
  li x8, 0
  li x9, 0
  li x10,0
  li x11,0
  li x12,0
  li x13,0
  li x14,0
  li x15,0
  li x16,0
  li x17,0
  li x18,0
```

```
  li x19,0
  li x20,0
  li x21,0
  li x22,0
  li x23,0
  li x24,0
  li x25,0
  li x26,0
  li x27,0
  li x28,0
  li x29,0
  li x30,0
  li x31,0

  /* set to initial state of FPU and disable interrupt */
  li t0, MSTATUS_FS
  csrs mstatus, t0

  fssr    x0
  fmv.w.x f0, x0
  fmv.w.x f1, x0
  fmv.w.x f2, x0
  fmv.w.x f3, x0
  fmv.w.x f4, x0
  fmv.w.x f5, x0
  fmv.w.x f6, x0
  fmv.w.x f7, x0
  fmv.w.x f8, x0
  fmv.w.x f9, x0
  fmv.w.x f10,x0
  fmv.w.x f11,x0
  fmv.w.x f12,x0
  fmv.w.x f13,x0
  fmv.w.x f14,x0
  fmv.w.x f15,x0
  fmv.w.x f16,x0
  fmv.w.x f17,x0
  fmv.w.x f18,x0
  fmv.w.x f19,x0
  fmv.w.x f20,x0
  fmv.w.x f21,x0
  fmv.w.x f22,x0
  fmv.w.x f23,x0
  fmv.w.x f24,x0
  fmv.w.x f25,x0
  fmv.w.x f26,x0
  fmv.w.x f27,x0
  fmv.w.x f28,x0
  fmv.w.x f29,x0
  fmv.w.x f30,x0
  fmv.w.x f31,x0


//Part5
.option push
```

```
.option norelax
  la gp, __global_pointer$
.option pop

//Part6
  /* get cpu id */
  csrr a0, mhartid

  la   sp, __stack_start__
  addi t1, a0, 1
  li   t2, __STACKSIZE__
  mul  t1, t1, t2
  add  sp, sp, t1 /* sp = (cpuid + 1) * __STACKSIZE__ + __stack_start__ */

  /* other cpu core, jump to cpu entry directly */
  bnez a0, secondary_cpu_entry
  tail primary_cpu_entry


//Part7
secondary_cpu_entry:
#ifdef RT_USING_SMP
  la a0, secondary_boot_flag
  ld a0, 0(a0)
  li a1, 0xa55a
  beq a0, a1, 1f
#endif
  j secondary_cpu_entry

#ifdef RT_USING_SMP
1:
  tail secondary_cpu_c_start

.data
.global secondary_boot_flag
.align 3
secondary_boot_flag:
    .dword 0
#endif
```

我们现在开始逐行分析整个代码。

Part1

```
//Part1
//初始化mstatus状态寄存器使用，设置浮点控制状态，该条指令指示当前状态处于关闭FS功能。
#define MSTATUS_FS      0x00006000U /* initial state of FPU */

//导入cpuport.h头文件，改文件中对部分汇编指令重命名
#include <cpuport.h>
```

1.首先初始化了mstatus寄存器，该寄存器是RISC-V机器模式状态寄存器(mstatus)是一个MXLEN位宽的可读写寄存器，该寄存器主要用于跟踪和控制Hart电路状态。RV64处理器mstatus结构参见下图，在S模式下的状态寄存器被命名为sstatus。[1]

Machine Status Registers: mstatus寄存器				
作用：该寄存器主要用于跟踪和控制Hart电路状态				
寄存器位号	名称	作用	位值	指令
63	SD	浮点、矢量和扩展单元 dirty 状态总和位		MSTATUS_FS=0x00006000U
62	WPRI			
61				
60				
59				
58				
57				
56				
55				
54				
53				
52				
51				
50				
49				
48				
47				
46				
45				
44				
43				
42				
41				
40				
39				
38				
37	MBE	机器模式端序控制位		
36	SBE	超级用户模式端序控制位		
35	SXL	寄存器位宽		
34				
33	UXL	寄存器位宽		
32				
31	WPRI			
30				
29				
28				
27				
26				
25				
24				
23				
22	TSR	陷阱 sret		
21	TW	超时等待		
20	TVM	陷阱虚拟内存		
19	MXR	允许加载请求访问标记为可执行的内存空间		
18	SUM	允许超级用户模式下访问 U 态虚拟内存空间		
17	MPRV	修改特权模式		
16	XS	扩展单元状态位		
15			0	
14	FS	浮点单元状态位	0	
13			0	
12	MPP	机器模式保留特权状态位	1	
11			0	
10	VS	矢量单元状态位	1	
9			1	
8	SPP	超级用户模式保留特权状态位	1	
7	MPIE	机器模式保留中断使能位	0	
6	UBE	端序控制位	1	
5	SPIE	超级用户模式保留中断使能位	1	
4	WPRI		1	
3	MIE	MIE-机器模式中断使能位：	0	
2	WPRI		0	
1	SIE	超级用户模式中断使能位	0	
0	WPRI		0	

我们看到起始状态我们没有启动FS浮点单元状态位。

2.导入cpuport.h头文件

```
#ifndef CPUPORT_H__
#define CPUPORT_H__

#include <rtconfig.h>              //导入rtconfig.文件，配置
ARCH_CPU_64BIT/ARCH_RISCV_FPU/ARCH_RISCV_FPU_D/ARCH_RISCV_FPU_S等参数

/* bytes of register width  */
#ifdef ARCH_CPU_64BIT
#define STORE                   sd              //替换STORE指令
#define LOAD                    ld              //替换LOAD指令
#define REGBYTES                8               //替换REGBYTES
#else
#define STORE                   sw
#define LOAD                    lw
#define REGBYTES                4
#endif

#ifdef ARCH_RISCV_FPU
#ifdef ARCH_RISCV_FPU_D
#define FSTORE                  fsd
#define FLOAD                   fld
#define FREGBYTES               8
#define rv_floatreg_t           rt_int64_t
#endif
#ifdef ARCH_RISCV_FPU_S
#define FSTORE                  fsw
#define FLOAD                   flw
#define FREGBYTES               4
#define rv_floatreg_t           rt_int32_t
#endif
#endif

#endif
```

仅使用了我们注释的几行。

Part2

```
//Part2
  .global   _start
  .section ".start", "ax"
```

`.global`关键字用来让一个符号对链接器可见，可以供其他链接对象模块使用。让`_start`符号成为可见的标示符，这样链接器就知道跳转到程序中的什么地方并开始执行，这里`_start`标签作为程序的默认进入点。[2]

汇编程序中以`.`开头的名称并不是指令的助记符，不会被翻译成机器指令，而是给汇编器一些特殊指示，称为汇编指示（Assembler Directive）或伪操作（Pseudo-operation）。`.section`指示把代码划分成若干个段（`section`），这样程序加载执行时，每个`section`会被加载到不同的内存地址，编译器设置不同`section`的读、写、执行权限。[2]

`ax`表示该节区可分配并且可执行，ax是 allocation execute的缩写。[3]

Part3

```
//Part3
_start:
  j 1f
  .word 0xdeadbeef   //用一个32位的立即数装入寄存器
  .align 3           //2^3对齐
  .global g_wake_up  //符号对编译器可见
  g_wake_up:
      .dword 1
      .dword 0
```

`_start`符号是整个程序的起始位置

`j 1f`意思是jump symbol 1 forward的意思，就是跳转到符号1位置继续执行，这里涉及到了一个汇编指令`j`。[4]

`j offset`是一条伪指令，把pc寄存器的值设置为当前值加上符号位扩展的offset，等同于`jal x0 offset`。`jal`即jump and link。

代码已经跳走了，后面代码暂时用不到，先不用去了解。

Part4

```
1:
  csrw mideleg, 0
  csrw medeleg, 0
  csrw mie, 0
  csrw mip, 0
  la t0, trap_entry
  csrw mtvec, t0

  li x1, 0
  li x2, 0
  li x3, 0
  li x4, 0
  li x5, 0
  li x6, 0
  li x7, 0
  li x8, 0
  li x9, 0
  li x10,0
  li x11,0
  li x12,0
  li x13,0
  li x14,0
  li x15,0
  li x16,0
  li x17,0
  li x18,0
  li x19,0
  li x20,0
```

```
  li x21,0
  li x22,0
  li x23,0
  li x24,0
  li x25,0
  li x26,0
  li x27,0
  li x28,0
  li x29,0
  li x30,0
  li x31,0

//FPU没开，后续不用管
  /* set to initial state of FPU and disable interrupt */
  li t0, MSTATUS_FS
  csrs mstatus, t0

  fssr    x0
  fmv.w.x f0, x0
  fmv.w.x f1, x0
  fmv.w.x f2, x0
  fmv.w.x f3, x0
  fmv.w.x f4, x0
  fmv.w.x f5, x0
  fmv.w.x f6, x0
  fmv.w.x f7, x0
  fmv.w.x f8, x0
  fmv.w.x f9, x0
  fmv.w.x f10,x0
  fmv.w.x f11,x0
  fmv.w.x f12,x0
  fmv.w.x f13,x0
  fmv.w.x f14,x0
  fmv.w.x f15,x0
  fmv.w.x f16,x0
  fmv.w.x f17,x0
  fmv.w.x f18,x0
  fmv.w.x f19,x0
  fmv.w.x f20,x0
  fmv.w.x f21,x0
  fmv.w.x f22,x0
  fmv.w.x f23,x0
  fmv.w.x f24,x0
  fmv.w.x f25,x0
  fmv.w.x f26,x0
  fmv.w.x f27,x0
  fmv.w.x f28,x0
  fmv.w.x f29,x0
  fmv.w.x f30,x0
  fmv.w.x f31,x0
```

`mideleg`和`medeleg`

默认情况下，所有的异常和中断都在机器模式下处理。但是通过这两个寄存器的设置，可以把异常和中断授权给其他特权等级处理。此处两个寄存器值都设置为0，表示不进行授权，异常和中断都在机器模式下处理。

`mie`和`mip`

软件中断、定时器中断和外部中断的使能位及标志位。代码里先关闭所有中断，然后清除所有中断标志。

`mtvec`

包含了所有异常和中断的入口地址。代码里设置为`trap_entry`，该标号在interrupt_gcc.S中定义，也就是`trap_entry`是所有异常和中断的入口。[6]

`la`即load address。

`li`即load immediate。

`csrw`即control and status register。

最后，我们想`x1`到`x31`写入立即数。

这里面没有`x0`，因为riscv的`x0`寄存器在硬件上设置常为0。

最后，我们给出一个内核控制相关寄存器的列表。

寄存器名称	别称	全称	寄存器号	用途	调用时是否保存
x0			0	常数0	不适用
x1	ra	return address	1	返回赋值（链接寄存器）	是
x2	sp	stack pointer	2	栈指针	是
x3	gp	global pointer	3	全局指针	是
x4	tp	thread pointer	4	线程指针	是
x5	t0	Temporary	5	临时	否
x6	t1	Temporary	6		否
x7	t2	Temporary	7		否
x8	s0/fp	Save register	8	保存	是
x9	s1	Save register	9		是
x10	a0	Function argument,return value	10	参数/结果	否
x11	a1	Function argument,return value	11		否
x12	a2	Function argument	12		否
x13	a3	Function argument	13		否
x14	a4	Function argument	14		否
x15	a5	Function argument	15		否
x16	a6	Function argument	16		否
x17	a7	Function argument	17		否
x18	s2	Save register	18	保存	是
x19	s3	Save register	19		是
x20	s4	Save register	20		是
x21	s5	Save register	21		是
x22	s6	Save register	22		是
x23	s7	Save register	23		是
x24	s8	Save register	24		是
x25	s9	Save register	25		是
x26	s10	Save register	26		是
x27	s11	Save register	27		是
x28	t3	Temporary	28	临时	否
x29	t4	Temporary	29		否
x30	t5	Temporary	30		否
x31	t6	Temporary	31		否

Part5

```
//Part5
.option push
.option norelax
  la gp, __global_pointer$          /* 将ld文件中的标签__global_pointer所处的地址值赋给gp寄存器，__global_pointer$ = . + 0x800 */
.option pop
```

`.option` 部分为汇编伪指令，用于限定 `.option push` 和 `.option pop` 之间的链接时代码优化。其中 `.option norvc` 禁用压缩指令（“C”扩展指令集，16位编码，占2字节），确保动态启停时刚好有 4 字节的 `nop` 和 `JAL offset` 写入空间。[7] `.option norelax` 是阻止 Linker Relaxation，从 RISC-V Assembly Programmer's Manual: .option 的描述和 All Aboard, Part 3: Linker Relaxation in the RISC-V Toolchain 举的例子来看，主要是防止链接时编译器做进一步的指令精简，比如长跳转变为短跳转，虽然看上去这里的 `nop` 和 `jal zero,%l[label]` 都已经没有优化空间。

Part6

```
//Part6
  //获取core的ID
  /* get cpu id */
  csrr a0, mhartid

  //设置栈空间
  la   sp, __stack_start__      //栈起始地址存到sp寄存器
  addi t1, a0, 1                //a+1指示有几个core，保存到t1
  li   t2, __STACKSIZE__        //栈大小，4096
  mul  t1, t1, t2               //t1=t1*t2
  add  sp, sp, t1 /* sp = (cpuid + 1) * __STACKSIZE__ + __stack_start__ */

  //跳转到C语言
  /* other cpu core, jump to cpu entry directly */
  bnez a0, secondary_cpu_entry
  tail primary_cpu_entry
```

使用 mhartid 寄存器获取 hart 的 ID， `csrr` 是一个伪指令，它读取一个 CSR 寄存器。[8]

读取mhartid的值读到a0中，保存到后面用。

`a0` 中保存的仍然是上面读到的mhartid。如果mhartid=0，即是core0，则会跳到 `primary_cpu_entry`，否则就跳到 `secondary_cpu_entry`。

后续工作

- [x] 绘制K210所有和任务切换相关寄存器
- [x] 分析startup_XXX.S文件
- [ ] 分析context_gcc.S文件
- [ ] 分析interrupt.c文件
- [ ] 运行一个最简单的任务切换demo

参考文献

[1]. 04-机器模式状态寄存器详解

[2]. ARM汇编.global .extern 和.text

[3]. .section 后面跟着的“ax”是什么意思

[4]. RISC-V jump_label 详解，第 2 部分：指令编码

[5]. 伪指令 .align 的含义

[6]. RT-Thread在Kendryte K210上的启动过程解析

[7]. RISC-V jump_label 详解，第 3 部分：核心实现

[8]. RISC-V from scratch 5：机器模式