SkyEye Internal

Skyeye-1.3.2

Blackfin.kang@gmail.com

Table of Contents

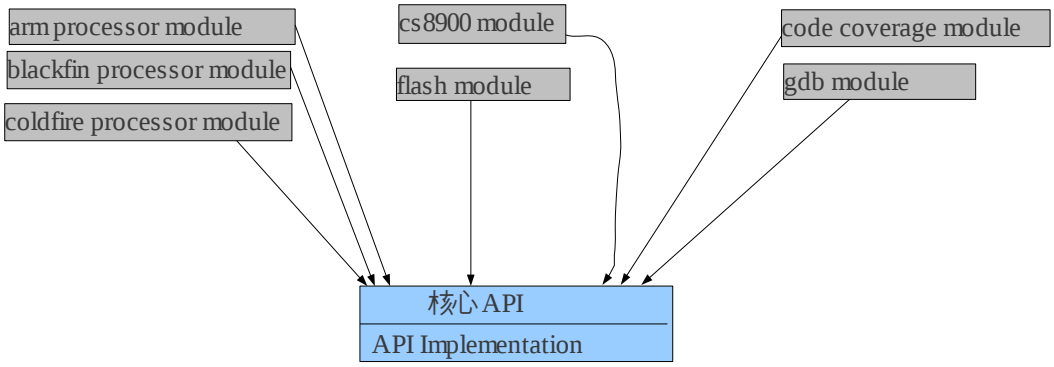
第一章 SkyEye 整体架构和代码结构

1.1 架构介绍

SkyEye 是一个仿真单板的开发平台，提供了丰富的 API 函数来基于已有的仿真模块进行二次开发。我们可以把整个 SkyEye 仿真平台分为两大部分，核心库和各种其他外围动态模块。其中外围动态模块大体分类如下：
处理器核的仿真模块：主要是仿真外设的指令集，中断等。目前可以仿真六个体系结构：arm, mips ,powerpc, blackfin, coldfire, sparc。
外设仿真模块：如网卡，LCD, Flash 等外设控制器的仿真模块
统计分析模块：有代码覆盖率分析模块，函数流跟踪模块等。

最新的 SkyEye 架构主要着眼于模块化和可扩展性，描述如下：
1、模块化
* 每一个模块可以以 so 或者 DLL 的形式存在，并且可以独立开发，独立编译。在 SkyEye 启动的时候会被 SkyEye 的核心库进行动态加载。
* 模块之间不存在任何依赖关系，相互独立，不存在相互调用关系。所有的模块都是调用 SkyEye 核心库提供的标准接口进行操作。
2、可扩展性
* 设计一组核心的 API 及其实现。把核心的 API 和实现放在 SkyEye 的核心库中，并提供给其他外围模块。
* 核心模块负责查找系统中可得到的其他外围模块，并进行动态加载。

其整体结构图如下：



1.2 目录结构

1.2.1 核心库

核心库位于 SkyEye 源代码的 common 目录，主要提供各种数据结构的注册管理，模块管理，命令行界面等等。其中功能在源码目录的分布如下：

- breakpoint 目录：断点管理模块，实现了断点的插入，删除等。

- Bus 目录：IO 读写接口的实现，包括 ram 的实现
- callback: callback 函数的管理，实现了 callback 的插入，删除等。
- cli: 调用了 readline 的库 实现了命令行接口。
- conf_parser: 配置文件的解析函数
- core: 对处理器核仿真模块的管理，包含模块的注册，查询等。
- ctrl: 仿真平台的控制模块，提供了初始化，启动，停止等函数的实现。
- device: 外设仿真模块的管理，包含外设仿真模块的注册，查询等。
- loader: 实现把各种文件加载到指定地址空间的函数。
- log: 实现了日志功能，来记录仿真平台运行过程中的调试信息，错误信息等。
- mach: 实现了仿真单板模块的管理。
- module : 实现了对 SkyEye 动态模块的管理。
- preference: 实现了对预置选项的管理。

1.2.2 其他目录

arch 目录：所有体系结构和处理器相关的模拟代码

device 目录：外设相关的模拟器代码

utils 目录：各种功能模块，反汇编，gdb 调试等。其中在 main 目录存放了 skyeye 的主函数。

1.3 核心库的介绍

第二章　总体流程和关键数据结构

2.1 skyeye 的配置文件模块

2.1.1 介绍

当前 skyeye 使用了一个文本配置文件来描述仿真的目标平台和一些其他的特性。关于当前 SkyEye 支持的各种配置选项，可以参考 SkyEye 的用户手册。我们也可以对配置文件进行扩展来添加自己的选项。

2.1.2 数据结构

我们用一个数据结构来代表配置文件的中的一个选项：

```
 6 typedef struct skyeye_option_s
 7 {
 8        char *option_name;
 9        int (*do_option) (struct skyeye_option_s * this_opion,
10                          int num_params, const char *params[]);
11        char* helper;
17        struct skyeye_option_t *next;
18 } skyeye_option_t;
```

其中， option_name 是一个用来标志这个选项的字符串，do_option 成员用来解析这个选项的函数，helper 成员也是字符串变量用来对选项进行描述。最后 next 成员是一个指向下一个 option 元素的指针。

我们用一个链表来把所有的 option 管理起来。链表的头为 skyeye_option_list 变量，定义在文件 common/conf_parser/skyeye_options.c 下，代码如下：

```
71 static skyeye_option_t* skyeye_option_list;
```

在 common/conf_parser/skyeye_options.c 文件中，我们还实现了对配置选项的链表进行其他的一些操作。register_option 函数用来在链表里面添加一个配置选项，实现代码如下。

```
93 exception_t register_option(char* option_name, do_option_t do_option_func, char* helper){
94        if(option_name == NULL || !do_option_func)
95                return Invarg_exp;
96        skyeye_option_t* node = malloc(sizeof(skyeye_option_t));
97        if(node == NULL)
98                return Malloc_exp;
99        node->option_name = skyeye_strdup(option_name);
```

```
100         if(node->option_name == NULL){
101                 skyeye_free(node);
102                 return Malloc_exp;
103         }
104         node->do_option = do_option_func;
105         /* maybe we should use skyeye_mm to replace all the strdup */
106         node->helper = skyeye_strdup(helper);
107         if(node->option_name == NULL){
108                 skyeye_free(node->option_name);
109                 skyeye_free(node);
110                 return Malloc_exp;
111         }
112         node->next = skyeye_option_list;
113         skyeye_option_list = node;
114         //skyeye_log(Info_log, __FUNCTION__, "register option %s successfully.",
option_name);
115         return No_exp;
116 }
```

2.1.3 运行流程

在每一个模块初始化函数中，可以上面的 register_option 函数来注册自己的配置流程选项。在所有的配置选项注册完毕之后，我们通过输入"start”命令或者函数调用的方式去运行 SIM_start 函数，这时 skyeye_read_config 函数被调用来解析指定的配置文件，如下：

```
106         sky_pref_t *pref;
107         /* get the current preference for simulator */
108         pref = get_skyeye_pref();
109         skyeye_config_t* config = get_current_config();
110         if(pref->conf_filename)
111                 skyeye_read_config(pref->conf_filename);
```

skyeye_read_config 会调用我们预先注册好的钩子函数对相应的选项进行解析每一个配置选项。

```
34 static skyeye_config_t skyeye_config;
35
```

get_current_config 用来获得当前配置数据结构体的指针。我们在编写其他模块的时候，有时候需要获得配置数据结构体的指针。

```
36 skyeye_config_t* get_current_config(){
37         return &skyeye_config;
38 }
```

2.2 SkyEye 的命令行接口模块

2.2.1 介绍

skyeye 的命令行接口是调用 readline 的库来实现的命令行解析和管理工作。Readline 是一个功能强大的命令行接口库。

2.2.2 数据结构

SkyEye 命令行接口中的每一条命令我们都用了一个 COMMAND 的数据结构进行描述，其代码如下：

```
41 /* A structure which contains information on the commands this program
42    can understand. */
43 struct command_s{
44    char *name;                 /* User printable name of the function. */
45    rl_icpfunc_t *func;          /* Function to call to do the job. */
46    char *doc;                  /* Documentation for this function.  */
47    struct command_s *next;
48 };
49 typedef struct command_s  COMMAND;
```

name 变量是我们敲入命令的字符串，func 是执行命令的函数，doc 是这条命令的帮助信息。next 是指向下一条命令数据结构的指针。我们把所有命令的数据结构用一个链表管理起来。

```
73 static COMMAND *command_list;
```

2.2.3 运行流程

2.3 模块动态加载部分

SkyEye 启动的时候会调用 SkyEye 模块加载函数，从系统安装 SkyEye 默认的目录去查找符合 SkyEye 规范的动态链接库。

实现代码位于 common/module/skyeye_module.c

```
/* on *nix platform, the suffix of shared library is so. */
const char* Default_libsuffix = ".so";
/* we will not load the prefix with the following string */
const char* Reserved_libprefix = "libcommon";

const char Dir_splitter = '/';

typedef struct skyeye_modules_s{
	skyeye_module_t* list;
	int total;
}skyeye_modules_t;

static skyeye_modules_t* skyeye_modules;

static void set_module_list(skyeye_module_t *node){
	skyeye_modules->list = node;
}
exception_t init_module_list(){
	skyeye_modules = skyeye_mm(sizeof(skyeye_modules_t));
	if(skyeye_modules == NULL)
		return Malloc_exp;
	return No_exp;
}
skyeye_module_t* get_module_list(){
	return skyeye_modules->list;
}

static exception_t register_skyeye_module(char* module_name, char* filename, void*
handler){
	exception_t ret;
	skyeye_module_t* node;
```

```
    skyeye_module_t* list;
    list = get_module_list();
    if(module_name == NULL|| filename == NULL)
        return Invarg_exp;

    node = malloc(sizeof(skyeye_module_t));
    if(node == NULL){
        skyeye_log(Error_log, __FUNCTION__, get_exp_str(Malloc_exp));
        return Malloc_exp;
    }

    node->module_name = strdup(module_name);
    if(node->module_name == NULL){
        free(node);
        return Malloc_exp;
    }

    node->filename = strdup(filename);
    if(node->filename == NULL){
        free(node->module_name);
        free(node);
        return Malloc_exp;
    }

    node->handler = handler;

    node->next = list;;
    set_module_list(node);
    return No_exp;
}

exception_t SKY_load_module(const char* module_filename){
    exception_t ret;
```

```
    char **module_name;
    void * handler;
    char* err_str;
    //skyeye_log(Debug_log, __FUNCTION__, "module_filename = %s\n",
module_filename);
    handler = dlopen(module_filename, RTLD_LAZY);
    if (handler == NULL)
    {
        err_str = dlerror();
        skyeye_log(Warnning_log, __FUNCTION__, "%s\n", err_str);
        return Dll_open_exp;
    }

    module_name = dlsym(handler, "skyeye_module");
    if((err_str = dlerror()) != NULL){
        skyeye_log(Warnning_log, __FUNCTION__, "dll error %s\n", err_str);
        skyeye_log(Warnning_log, __FUNCTION__, "Invalid module in file %s\n",
module_filename);
        dlclose(handler);
        return Invmod_exp;
    }
    //skyeye_log(Debug_log, __FUNCTION__, "Load module %s\n", *module_name);

    ret = register_skyeye_module(*module_name, module_filename, handler);
    if(ret != No_exp){
        dlclose(handler);
        return ret;
    }
    return No_exp;
}

void SKY_load_all_modules(char* lib_dir, char* suffix){
    /* we assume the length of dirname + filename does not over 1024 */
```

```
char* full_filename[1024];
char* lib_suffix;
/* Find all the module under lib_dir */
DIR *module_dir = opendir(lib_dir);
exception_t exp;
/*FIXME we should throw some exception. */
if(module_dir == NULL)
	return;
if(suffix == NULL)
	lib_suffix = Default_libsuffix;
else
	lib_suffix = suffix;
struct dirent* dir_ent;
while((dir_ent = readdir(module_dir)) != NULL){
	char* mod_name = dir_ent->d_name;
	/* exclude the library not end with lib_suffix */
	char* suffix = strrchr(mod_name, '.');
	if(suffix == NULL)
		continue;
	else{
		//skyeye_log(Debug_log, __FUNCTION__, "file suffix=%s\n", suffix);
		if(strcmp(suffix, lib_suffix))
			continue;
	}
		/* exclude the reserved library */
	if(!strncmp(mod_name, Reserved_libprefix, strlen(Reserved_libprefix)))
		continue;

	/* contruct the full filename for module */
	int lib_dir_len = strlen(lib_dir);
	memset(&full_filename, '\0', 1024);
	strncpy(&full_filename[0], lib_dir, lib_dir_len);
	full_filename[lib_dir_len] = Dir_splitter;
```

```
            full_filename[lib_dir_len + 1] = '\0';
            //skyeye_log(Debug_log, __FUNCTION__, "1 full_filename=%s\n",
full_filename);
            strncat(full_filename, mod_name, strlen(mod_name) + 1);
            //skyeye_log(Debug_log, __FUNCTION__, "full_filename=%s\n",
full_filename);
            /* Try to load a module */
            exp = SKY_load_module(full_filename);
            if(exp != No_exp)
                skyeye_log(Info_log, __FUNCTION__, "Can not load module from file
%s.\n", dir_ent->d_name);
            //}
        }
        closedir(module_dir);
}
skyeye_module_t * get_module_by_name(const char* module_name){
        skyeye_module_t* list = get_module_list();
        while(list != NULL){
            if(!strncmp(list->module_name, module_name, strlen(module_name)))
                return list;
            list = list->next;
        }
        return NULL;
}
```

提供的 API 定义接口位于 common/include/skyeye_module.h,部分代码如下：

```
/*
 * the contructor for module. All the modules should implement it.
 */
void module_init ()  __attribute__((constructor));

/*
```

```
 * the decontructor for module. All the modules should implement it.
 */
void module_fini () __attribute__((destructor));


typedef struct skyeye_module_s{
	/*
	 * the name for module, should defined in module as an varaible.
	 */
	char* module_name;
	/*
	 * the library name that contains module
	 */
	char* filename;
	/*
	 * the handler for module operation.
	 */
	void* handler;
	/*
	 * next node of module linklist.
	 */
	struct skyeye_module_s *next;
}skyeye_module_t;

/*
 * load all the modules in the specific directory with specific suffix.
 */
void SKY_load_all_module(const char* lib_dir, char* lib_suffix);

/*
 * load one module by its file name.
 */
exception_t SKY_load_module(const char* module_filename);
```

模块加载的命令行演示

ksh@server:/opt/skyeye> bin/skyeye

SkyEye is an Open Source project under GPL. All rights of different parts or modules are reserved by their author. Any modification or redistributions of SkyEye should note remove or modify the annoucement of SkyEye copyright.

Get more information about it, please visit the homepage http://www.skyeye.org.

Type "help" to get command list.

(skyeye)list-modules

Module Name	File Name
sparc	/opt/skyeye/lib/skyeye/libsparc.so
mips	/opt/skyeye/lib/skyeye/libmips.so
flash	/opt/skyeye/lib/skyeye/libflash.so
code_cov	/opt/skyeye/lib/skyeye/libcodecov.so
uart	/opt/skyeye/lib/skyeye/libuart.so
gdbserver	/opt/skyeye/lib/skyeye/libgdbserver.so
coldfire	/opt/skyeye/lib/skyeye/libcoldfire.so
arm	/opt/skyeye/lib/skyeye/libarm.so
touchscreen	/opt/skyeye/lib/skyeye/libts.so
net	/opt/skyeye/lib/skyeye/libnet.so
nandflash	/opt/skyeye/lib/skyeye/libnandflash.so
ppc	/opt/skyeye/lib/skyeye/libppc.so
bfin	/opt/skyeye/lib/skyeye/libbfin.so

(skyeye)

SkyEye 模块规范及编写示例：

每个模块需要实现两个函数 module_init 和 module_fini，其中 module_init 函数会在动态模块加载的时候被自动执行，而 module_fini 函数会在动态模块卸载的时候被自动调用。

为了区分 SkyEye 的动态模块和其他动态链接库，每个 SkyEye 的动态模块需要定义一个全局的字符串变量, skyeye_module, skyeye 会通过判断当前的动态链接库中是否存在变量 skyeye_module 来决定这个动态链接库是否是 SkyEye 的合法模块。

第三章、仿真平台的初始化代码分析

当前 SkyEye 提供了核心库和插件的实现方式，任何功能都可以以插件的形式实现。SkyEye 的核心库只是提供了一组函数，并不包含 SkyEye 的主函数。用户可以自己实现一个主函数，调用 SkyEye 提供的 API 来启动仿真平台。也可以把通过调用 SkyEye 的 API 的方式来把 SkyEye 集成到第三方的硬件平台上。

3.1 main 函数

SkyEye 的主函数 main 位于 utils/main/skyeye.c。

```
477 /**
478  *  The main function of skyeye
479  */
480
481 int
482 main (int argc, char **argv)
483 {
484         int ret;
485
486         sky_pref_t* pref = get_skyeye_pref();
487         assert(pref != NULL);
488         /* initialization of options from command line */
489         ret = init_option(argc, argv, pref);
490         /* set the current preference for skyeye */
491         //update_skyeye_pref(pref);
492
493         SIM_init();
544         return ret;
545 }
```

在代码的 486 到 489 行，我们设置了 SkyEye 需要的运行环境变量，其中可以是 elf 镜像文件的名称，要运行机器的大小端等信息。在这里，我们的这些运行环境变量主要是通过解析 skyeye 的命令行参数和镜像文件获得。

在 493 行中，通过调用 SIM_init 函数，SkyEye 的命令行接口会在 SIM_init 中启动，这时我们就可以输入各种命令来控制 SkyEye 的运行。而在执行完 SIM_init，main 函数的任务就基本完成了，剩下的执行都交给 SkyEye 的 CLI 的界面了。

3.2 SIM_init 函数

SIM_init 的功能主要是动态模块的加载以及各种初始化的工作，相关实现代码位于 common/ctrl/sim_ctrl.c 。

```
27 void SIM_init(){
28      sky_pref_t* pref;
29      char* welcome_str = get_front_message();
30      /*
31       * get the corrent_config_file and do some initialization
32       */
33      skyeye_config_t* config = get_current_config();
34      skyeye_option_init(config);
35      /*
36       * initialization of callback data structure, it needs to
37       * be initialized at very beginning.
38       */
39      init_callback();
40
41      /*
42       * initilize the data structure for command
43       * register some default built-in command
44       */
45      init_command_list();
46
47      init_stepi();
48
49      /*
50       * initialization of module manangement
51       */
52      init_module_list();
53
54
55      /*
56       * initialization of architecture and cores
```

```
57          */
58         init_arch();
59
60         /*
61          * initialization of bus and memory module
62          */
63         init_bus();
64
65
66         /*
67          * initialization of machine module
68          */
69         init_mach();
70
71
72         /*
73          * initialization of breakpoint, that depends on callback module.
74          */
75         init_bp();
75         init_bp();
76
77         /*
78          * get the current preference for simulator
79          */
80         pref = get_skyeye_pref();
81
82         /*
83          * loading all the modules in search directory
84          */
85         if(!pref->module_search_dir)
86                 pref->module_search_dir = skyeye_strdup(default_lib_dir);
87         SKY_load_all_modules(pref->module_search_dir, NULL);
88         //skyeye_config_t *config;
```

```
89        //config = malloc(sizeof(skyeye_config_t));
90        if(try_init() == No_exp){
91                if(pref->autoboot == True){
92                        SIM_run();
93                }
94        }
95        /*
96         * if we run simulator in GUI or external IDE, we do not need to
97         * launch our CLI.
98         */
99        if(pref->interactive_mode == True){
100                SIM_cli();
101        }
102 }
```

在上面代码的 100 行处，SkyEye 调用了 SIM_cli 函数，从而进入了命令行界面，我们可以输入各种命令对仿真平台进行操作。

如代码 99 到 100 行所示，我们也可以通过在 SkyEye 的 main 函数中通过设置 pref->interactive_mode 的参数来对 SkyEye 进行设置，选择是否要启动 SkyEye 的命令行界面。

3.3 SIM_start

接下来，如果我们要启动我们要运行的目标板，我们可以输入 start 命令来初始化仿真目标板，并加载要运行的镜像文件。

下面的代码为 start 命令对应的执行函数。

```
54 /*
55  * start running of  SkyEye
56  */
57
58 com_start (arg)
59      char *arg;
60 {
61        int flag = 0;
62        SIM_start();
63        return flag;
```

```
64 }
```

如 62 行，SIM_start 会被调用。SIM_start 的实现如下：

```
105 void SIM_start(void){
106         sky_pref_t *pref;
107         /* get the current preference for simulator */
108         pref = get_skyeye_pref();
109         skyeye_config_t* config = get_current_config();
110         if(pref->conf_filename)
111                 skyeye_read_config(pref->conf_filename);
112
113         if(config->arch == NULL){
114                 skyeye_log(Error_log, __FUNCTION__, "Should provide valid arch
option in your co    nfig file.\n");
115                 return;
116         }
117         generic_arch_t *arch_instance = get_arch_instance(config->arch->arch_name);
118
119         if(config->mach == NULL){
120                 skyeye_log(Error_log, __FUNCTION__, "Should provide valid mach
option in your co    nfig file.\n");
121                 return;
122         }
123
124         arch_instance->init();
125
126         /* reset all the memory */
127         mem_reset();
128
129         config->mach->mach_init(arch_instance, config->mach);
130         /* reset current arch_instance */
131         arch_instance->reset();
132         /* reset all the values of mach */
133         config->mach->mach_io_reset(arch_instance);
```

```
134
135         if(pref->exec_file){
136                 exception_t ret = load_elf(pref->exec_file);
137         }
138
139         skyeye_log(Info_log, __FUNCTION__, "Set PC to the address 0x%x\n",
config->start_address    );
140         /* set pc from config */
141         arch_instance->set_pc(config->start_address);
142
143         pthread_t id;
144         create_thread(skyeye_loop, arch_instance, &id);
166 }
```

SIM_start 函数主要完成如下功能，在代码 110 和 111 行，读入 skyeye.conf 文件并解析文件中的所有配置选项。

代码 117 到 133 行，根据配置选项，对所选择的 arch，mach, memory 等数据结构进行初始化，为运行做准备。

代码 135 行到 137 行，是根据 pref 文件中的设置来加载一个 elf 镜像文件。

代码 141 行来设置仿真目标板的 PC 地址。

144 行用来创建一个线程并执行 skyeye_loop 函数。

3.4 skyeye_loop 函数

skyeye_loop 是仿真平台用来执行每一条指令的主循环，其代码如下：

```
137 /*
138  * mainloop of simulatior
139  */
140 void skyeye_loop(generic_arch_t *arch_instance){
141         for (;;) {
142                 /* chech if we need to run some callback functions at this time */
143                 exec_callback(Step_callback, arch_instance);
144                 while (!running) {
145                         /*
146                          * spin until it's time to go.  this is useful when
147                          * we're not auto-starting.
```

```
148                     */
149                     sleep(1);
150             }
157             /* run step once */
158             arch_instance->step_once ();
159         }
160 }
```

上面的函数主体是一个无限循环，143 行调用 exec_callback 函数，来检测是否在这个指令执行周期中有需要执行的 callback 函数，如果有，则执行。

144 到 150 行是通过检测 running 变量来判断当前仿真目标板是否处于运行状态，如果 running 为 0, 仿真目标板处于停止状态，则调用 sleep 进行睡眠来释放处理器资源。然后继续进行 while 循环。

如果 running 变量为 1，则整个仿真目标板处于运行态，则在 158 行处调用 arch_instance->step_once 来执行目标单板的一条指令。

第四章、PowerPC 处理器仿真模块分析

4.1 PowerPC 仿真模块的背景介绍

SkyEye 中 PowerPC 仿真模块的部分代码来自于 PearPC 项目的代码。SkyEye 目前主要是仿真 e500 系列的处理器，mpc8560 和 mpc8572。其中我们仿真的 mpc8572 处理器是一款双核的处理器，在 SkyEye 的仿真代码中也实现了对双核启动和通信的一些仿真。关于 mpc8560 和 mpc8572 的相关文档可以在 FreeScale 的官方网站上进行下载。

我们可以运行 linux-2.6.22 和 linux-2.6.23 的内核在 SkyEye 的 PowerPC 仿真模块上。

4.2 和 skyEye 核心模块的接口部分

如我们前面介绍，对于需要仿真的每个体系结构都需要实现 arch_config_t 的接口数据结构。在 PowerPC 仿真模块中，实现 arch_config_t 接口的数据结构的代码位于[arch/ppc/common/ppc_arch_interface.c : init_ppc_arch]中 ，代码如下：

```
350     static arch_config_t ppc_arch;
351
352     ppc_arch.arch_name = "ppc";
353     ppc_arch.init = ppc_init_state;
354     ppc_arch.reset = ppc_reset_state;
355     ppc_arch.set_pc = ppc_set_pc;
356     ppc_arch.get_pc = ppc_get_pc;
357     ppc_arch.get_step = ppc_get_step;
358     ppc_arch.step_once = ppc_step_once;
359     ppc_arch.ICE_write_byte = ppc_ICE_write_byte;
360     ppc_arch.ICE_read_byte = ppc_ICE_read_byte;
361     ppc_arch.parse_cpu = ppc_parse_cpu;
362     ppc_arch.get_regval_by_id = ppc_get_regval_by_id;
363     ppc_arch.get_regname_by_id = ppc_get_regname_by _id;
```

```
364        //ppc_arch.parse_mach = ppc_parse_mach;
365
366        register_arch (&ppc_arch);
```

init_ppc_arch 函数会在模块加载的时候被调用，在上面代码的 366 行 register_arch 完成把 ppc_arch 数据结构注册到 SkyEye 中的过程。后面的过程基本上是 SkyEye 通过操作注册进来的 ppc_arch 的函数指针来获得和设置 PowerPC 仿真模块的各种信息。

描述 PowerPC 处理器状态的数据结构为 PPC_CPU_State，其定义位于 arch/ppc/common/ppc_cpu.h 文件，

```
41 typedef struct PPC_CPU_State_s {
42        e500_core_t * core;
43        uint32_t bptr;
44        uint32_t eebpcr;
45        uint32_t ccsr;
46        uint32_t core_num;
47 }PPC_CPU_State;
```

其中的 core 变量是一个类型为 e500_core_t 的数组，数组中的每一个成员代表了一个 e500 核。数组的大小由 core_num 来确定。在 PowerPC 仿真模块启动的时候，根据 core_num 的数值，对 core 数组进行初始化。

对 core 的初始化在 ppc_cpu_init [arch/ppc/common/ppc_arch_interface.c]函数中，代码如下：

```
73        if(!gCPU.core_num){
74              fprintf(stderr, "ERROR:you need to set numbers of core in mach_init.\n");
75              skyeye_exit(-1);
76        }
77        else
78              gCPU.core = malloc(sizeof(e500_core_t) * gCPU.core_num);
```

4.3 运行流程分析

在 SkyEye 执行 SIM_start 函数之后，skyeye 的配置文件 skyeye.conf 会被解析。SkyEye 会根据配置文件的选项，对各个模块进行了相应的设置，加载我们要运行的 elf 文件或者其他镜像文件，最后设置了我们仿真处理器的 PC 地址，为运

行目标机上的第一条指令做好准备。在 SIM_start 函数的最后会调用 skyeye_loop 函数。

4.4 中断和异常仿真的代码分析

e500 平台的中断和异常的架构是一种向量化的表，一共有 32 个中断向量。其描述位于 e500 的手册

当出现异常的时候，SkyEye 会调用 ppc_exception[位于文件 arch/ppc/common/ppc_e500_exc.c] 函数对各种异常进行模拟，我们以 PowerPC 架构中的时钟模拟为例来介绍异常和中断的触发和模拟的过程。

4.4.1 时钟中断仿真代码分析

PowerPC 的时钟中断涉及到的寄存器为

```
223 exec_npc:
224         if(!ppc_divisor){
225                 dec_io_do_cycle(core);
```

```
226             ppc_divisor = 0;
227         }
228         else
229             ppc_divisor--;
```

在每一个条指令执行的周期中，dec_io_do_cycle 函数被调用，函数的实现如下：

```
28 #define TCR_DIE (1 << 26)
29 #define TSR_DIS (1 << 27)
30 void dec_io_do_cycle(e500_core_t * core){
31         core->tbl++;
32         /**
33          *  test DIE bit of TCR if timer is enabled
34          */
35         if(!(core->tsr & 0x8000000)){
36             if((core->tcr & 0x4000000) && (core->msr & 0x8000)) {
37
38                 if(core->dec > 0)
39                     core->dec--;
40                 /* if decrementer eqauls zero */
41                 if(core->dec == 0){
49                     ppc_exception(core, DEC, 0, core->pc);
50                 }
51             }
52         }
53         return;
54 }
```

第 31 行，对 e500 中的 tbl 寄存器进行累加。然后分别判断 tsr 寄存器的

ppc_exception 中的时钟中断的实现代码如下：

```
78             case DEC:
79                 core->srr[0] = core->npc;
80                 core->srr[1] = core->msr;
81
82                 /* CE,ME and DE bit unchanged, other bit should be clear*/
83                 core->msr &= 0x21200;
```

```
84
85                  /* DIS bit is set */
86                  core->tsr |= 0x8000000;
87                  //printf("In %s, timer interrupt happened.\n", __FUNCTION__);
88                  break;
```

4.4.2 数据异常的仿真代码分析

相比于上面的时钟中断异常的仿真，数据 TLB 异常的仿真复杂很多。一般来说，操作系统会在初始化的时候对 TLB 和 MMU 进行初始化，添加一些 TLB 表项，来建立虚实地址的映射。然后在后续的数据和指令访问过程中，TLB 不断的进行虚实地址翻译。一旦一些虚拟地址无法在 TLB 表项找到对应的表项，这时就会产生一个 TLB 异常。会跳转到操作系统的 TLB 异常处理函数进行中异常处理。

SkyEye 中对 e500 的 MMU 的地址翻译过程的仿真代码位于文件 arch/ppc/common/ppc_mmu.c 文件中。

```
 68 int ppc_effective_to_physical(e500_core_t * core, uint32 addr, int flags, uint32 *result){
 69      int i,j;
 70      uint32 mask;
 71      ppc_tlb_entry_t *entry;
 72      int tlb1_index;
 73      int pid_match = 0;
 74
 75      if((gCPU.bptr & 0x80000000) && (addr >> 12 == 0xFFFFF)){ /* if bootpage translation enabled? */
 76          //printf("do bootpage translation\n");
 77          *result = (addr & 0xFFF) | (gCPU.bptr << 12); /* please refer to P259 of MPC8572UM */
 78          return PPC_MMU_OK;
 79      }
 80      i = 0;
 81      /* walk over tlb0 and tlb1 to find the entry */
 82      while(i++ < (L2_TLB0_SIZE + L2_TLB1_SIZE)){
 83          if(i > (L2_TLB0_SIZE - 1)){
 84              tlb1_index = i - L2_TLB0_SIZE;
 85              entry = &current_core->mmu.l2_tlb1_vsp[tlb1_index];
 86          }
```

```
87              else
88                      entry = &current_core->mmu.l2_tlb0_4k[i];
89              if(!entry->v)
90                      continue;
91              //if(addr == 0xfdff9080)
92              //      printf("In %s,entry=0x%x, i = 0x%x, current_core->pir=0x%x\n",
__FUNCTION__, entry, i, current_c      ore->pir);
93              /* FIXME, not check ts bit now */
94              if(entry->ts & 0x0)
95                      continue;
96              if(entry->tid != 0){
97                  /*
98                      for(j = 0; j < 3; j++){
99                              if(current_core->mmu.pid[j] == entry->tid)
100                                     break;
101                     }*/
102                     //printf("entry->tid=0x%x\n", entry->tid);
103                     /* FIXME, we should check all the pid register */
104                     if(current_core->mmu.pid[0] != entry->tid)
105                             continue;
106
107             }
108             if(i > (L2_TLB0_SIZE - 1)){
109                     int k,s = 1;
110                     for(k = 0; k < entry->size; k++)
111                             s = s * 4;
112                     mask = ~((1024 * (s - 1) - 0x1) + 1024);
113             }
114             else
115                     mask = ~(1024 * 4 - 0x1);
116             if(entry->size != 0xb){
117                     if((addr & mask) != ((entry->epn << 12) & mask))
118                             continue;
```

```
119                     /* check rwx bit */
120                     if(flags == PPC_MMU_WRITE){
121                         if(current_core->msr & 0x4000){ /* Pr =1 , we are in user mode
*/
122                             if(!(entry->usxrw & 0x8)){
123                                 //printf("In %s,usermode,offset=0x%x, entry-
>usxrw=0x%x,pc=0x%x\n", __FU     NCTION__, i, entry->usxrw, current_core->pc);
124                                 ppc_exception(core, DATA_ST, flags, addr);
125                                 return PPC_MMU_EXC;
126                             }
127                         }
128                         else{/* Or PR is 0,we are in Supervisor mode */
129                             if(!(entry->usxrw & 0x4)){/* we judge SW bit */
130                                 //printf("In %s,Super mode,entry->usxrw=0x
%x,pc=0x%x\n", __FUNCTION__, e     ntry->usxrw, current_core->pc);
131                                 ppc_exception(core, DATA_ST, flags, addr);
132                                 return PPC_MMU_EXC;
133                             }
134                         }
135                     }
136
137                     *result = (entry->rpn << 12) | (addr & ~mask); // get real address
138                 }
139                 else {/*if 4G size is mapped, we will not do address check */
140                     //fprintf(stderr,"warning:4G address is used.\n");
141                     if(addr < (entry->epn << 12))
142                         continue;
143                      *result = (entry->rpn << 12) | (addr - (entry->epn << 12)); // get real
address
144
145                 }
146                 return PPC_MMU_OK;
147         }
```

最后，如果没有找到合适的 TLB 选项，则会触发 TLB 异常。

```
149        if(flags == PPC_MMU_CODE){
150                ppc_exception(core, INSN_TLB, flags, addr);
151                return PPC_MMU_EXC;
152        }
153        else{
154                if(ppc_exception(core, DATA_TLB, flags, addr))
155                        return PPC_MMU_EXC;
156        }
157        return PPC_MMU_FATAL;
```

数据 TLB 异常的代码位于 ppc_exception 函数中，主要做了如下动作

第一步、设置处理器核相应的寄存器，如代码 94 行到 101 行：

```
94              case DATA_TLB:
95                      //printf(" In %s, DATA_TLB exp happened, pc=0x%x,addr=0x%x, pir=0x%x\n", __FUNCTION__, core->pc,    a, core->pir);
96                      core->srr[0] = core->pc;
97                      core->srr[1] = core->msr;
98                      //core->esr |= ST;
99                      core->dear = a; /* save the data address accessed by exception instruction */
100
101                      core->msr &= 0x21200;
```

其中 96 行设置 srr[0]为发生异常的 pc，97 行用来保存发生异常的 MSR 到 srr[1]寄存器中。99 行保存异常访问的数据的地址到处理器核的 dear 寄存器中。最后 101 行根据数据 TLB 异常的定义，设置当前的 MSR 的值。

第二步、更新 MMU 中的相关寄存器，如下代码

```
102                      /* Update TLB */
103              /**
104                * if TLBSELD = 00, MAS0[ESEL] is updated with the next victim information for TLB0.Finially,          * the MAS[0] field is updated with the incremented value of TLB0[NV].Thus, ESEL points to
105                * the current victim
106                * (the entry to be replaced), while MAS0[NV] points to the next victim to be used if a TLB0          * entry is replaced
107                */
```

```
108
109             /**
110              * update TLBSEL with TLBSELD
111              */
112                     core->mmu.mas[0] = (core->mmu.mas[4] & 0x10000000) | (core-
>mmu.mas[0] & (~0x10000000));
113                     /* if TLBSELD == 0, update ESEL and NV bit in MAS Register*/
114                     if(!TLBSELD(core->mmu.mas[4])){
115                             /* if TLBSELD == 0, ESEL = TLB[0].NV */
core->mmu.mas[0] = (core->mmu.tlb0_nv << 18) | (core->mmu.mas[0] & 0xFFF0FFFF)
;
126                             /* update NV of MAS0 , NV = ~TLB[0].NV */
127                             core->mmu.mas[0] = (~core->mmu.tlb0_nv & 0x3) | (core-
>mmu.mas[0] & 0xFFFFFFFC);
128                             //printf("In %s,core->mmu.mas[0]=0x%x\n", __FUNCTION__,
core->mmu.mas[0]);
129                     }
130             /**
131              *  set zeros of permis and U0 - U3
132              */
133                     core->mmu.mas[3] &= 0xFFFFFC00;
134             /**
135              *  set zeros of RPN
136              */
137                     core->mmu.mas[3] &= 0xFFF;
138
139             /**
140              * Set EPN to EPN of access
141              */
142                     core->mmu.mas[2] = (a & 0xFFFFF000) | (core->mmu.mas[2]
&0xFFF);
143             /**
144              * Set TSIZE[0 - 3] to TSIZED
145              */
```

```
146                 core->mmu.mas[1] = (core->mmu.mas[4] & 0xF00)|(core->mmu.mas[1] & 0xFFFFF0FF);
147             /**
148              * Set TID
149              */
150                 core->mmu.mas[1] = (core->mmu.mas[1] & 0xFF00FFFF)|((core->mmu.pid[0] & 0xFF) << 16);
151
152             /**
153              * set Valid bit
154              */
155                 core->mmu.mas[1] = current_core->mmu.mas[1] | 0x80000000;
156             /* update SPID with PID */
157                 core->mmu.mas[6] = (core->mmu.mas[6] & 0xFF00FFFF) | ((core->mmu.pid[0] & 0xFF) << 16);
158                 if(flags == PPC_MMU_WRITE)
159                     core->esr = 0x00800000;
160                 else
161                     core->esr = 0x0;
162                 break;
```

4.5 多核仿真的代码分析

SkyEye 当前实现了 mpc8572 双核处理器的模拟，可以运行支持 SMP 的 Linux 内核。

4.5.1 多核启动分析

多核仿真是通过在一个循环中每个处理器轮流运行一条执行实现的，虽然在 SkyEye 的内部实现中，两个处理器核运行指令是顺序执行，但是对于运行其上的系统软件，它“意识”不到这种顺序执行，它会认为 SkyEye 仿真的两个核是并行的。实现代码如下：

```
246             /* if CPU1_EN is set? */
247             if(!i || gCPU.eebpcr & 0x2000000)
248                 per_cpu_step(current_core);
```

其中 247 行通过判断 eebpcr 寄存器的相应位，判断是否第二个处理器核已经启动。在这个 for 循环中，SkyEye 仿真了两个核的指令执行。

PowerPC 的 e500 系列为了支持第二个核的启动，还添加了 bootpage 的特性。当 bptr 的在开始执行的时候，判断 bptr 寄存器的

```
 75        if((gCPU.bptr & 0x80000000) && (addr >> 12 == 0xFFFFF)){ /* if bootpage translation enabled? */
 76                //printf("do bootpage translation\n");
 77                *result = (addr & 0xFFF) | (gCPU.bptr << 12); /* please refer to P259 of MPC8572UM */
 78                return PPC_MMU_OK;
 79        }
```

下面的代码是对处理器核的初始化，每一个处理器核都有一个单独的寄存器 pir 用来标志自己的 ID。在这个函数中，如代码 77 行，处理器核会根据参数 core_id 来对自己的 pir 寄存器进行初始化。

```
66 /*
67  * Initialization for e500 core
68  */
69 void ppc_core_init(e500_core_t * core, int core_id){
70        // initialize srs (mostly for prom)
71        int j;
72        for (j = 0; j < 16; j++) {
73                core->sr[j] = 0x2aa*j;
74        }
75        //core->pvr = 0x8020000; /* PVR for mpc8560 */
76        core->pvr = 0x80210030; /* PVR for mpc8572 */
77        core->pir = core_id;
78
79        e500_mmu_init(&core->mmu);
80 }
```

4.5.2 多核同步

PowerPC 通过发送 IPI 中断实现了多核之间的同步。实现代码如下：

```
965                     case 0x60040:
 966                             io->mpic.ipidr[0] = data;
 967                             int core_id = -1;
 968                             if (data & 0x1) /* dispatch the interrupt to core 0 */
 969                                     core_id = 0;
 970                             if (data & 0x2) /* dispatch the interrupt to core 1 */
 971                                     core_id = 1;
 972                             if(data & 0x3){
 973                                     /* trigger an interrupt to dedicated core */
 974                                     e500_core_t* core = &cpu->core[core_id];
 975                                     core->ipr |= IPI0;
 976                                     io->mpic.ipivpr[0] |= 0x40000000; /* set activity bit in vpr
*/
 978                                     io->pic_percpu.iack[core_id] = (io-
>pic_percpu.iack[core_id] & 0xFFFF0000) | (io->mpic.ipivpr[0] & 0xFFFF);
 980                                     ppc_exception(core, EXT_INT, 0, 0);
 981                                     core->ipi_flag = 1; /* we need to inform the core that
npc is changed to exception vector */
 983                             }
 984                             return;
```

我们当前只判断了当写入数据等于 3 的时候，我们需要设置处理器核的 ipr 寄存器为相应的中断位。

4.6 在 *SkyEye* 上运行和调试 *PowerPC* 平台的 *Linux*

第五章、ARM 处理器仿真模块分析

5.1 介绍

SkyEye 的 ARM 仿真代码最初的版本来自于 gdb 的 ARMulator 模块。

5.2 与 *SkyEye* 核心模块的接口

每个处理器架构需要实现

```
void
init_arm_arch ()
{
	static arch_config_t arm_arch;

	arm_arch.arch_name = "arm";
	arm_arch.init = arm_init_state;
	arm_arch.reset = arm_reset_state;
	arm_arch.set_pc = arm_set_pc;
	arm_arch.get_pc = arm_get_pc;
	arm_arch.get_step = arm_get_step;
	arm_arch.step_once = arm_step_once;
	arm_arch.ICE_write_byte = arm_ICE_write_byte;
	arm_arch.ICE_read_byte = arm_ICE_read_byte;
	arm_arch.parse_cpu = arm_parse_cpu;
	//arm_arch.parse_mach = arm_parse_mach;
	//arm_arch.parse_mem = arm_parse_mem;
	arm_arch.parse_regfile = arm_parse_regfile;
	arm_arch.get_regval_by_id = arm_get_regval_by_id;
	arm_arch.get_regname_by_id = arm_get_regname_by_id;

	register_arch (&arm_arch);
}
```

5.3 内部运行流程

在这里我们介绍对于 arm 处理器仿真的一些细节。每一个体系结构的仿真需要实现单步执行的操作。arm 体系结构单步执行的函数实现在文件 arch/arm/common/arm_arch_interface.c 中。实现如下：

```
static void

arm_step_once ()

{

	//ARMul_DoInstr(state);
```

```
        step++;

        cycle++;

        state->EndCondition = 0;

        stop_simulator = 0;

        state->NextInstr = RESUME;      /* treat as PC change */

        state->Reg[15] = ARMul_DoProg(state);

        //state->Reg[15] = ARMul_DoInstr(state);

        FLUSHPIPE;

}
```

5.4 MMU 相关接口和实现

ARM 体系结构的不同处理器核在协处理器 15，也就是内存管理单元的差别也比较大，所以我们设计了一个 MMU 的抽象接口，来屏蔽不同的 MMU 硬件细节。实现代码如下：

```
typedef struct mmu_state_t
{
        ARMword control;
        ARMword translation_table_base;
        ARMword domain_access_control;
        ARMword fault_status;
        ARMword fault_address;
        ARMword last_domain;
        ARMword process_id;
        ARMword cache_locked_down;
        ARMword tlb_locked_down;
//chy 2003-08-24 for xscale
        ARMword cache_type;     // 0
        ARMword aux_control;    // 1
        ARMword copro_access;   // 15

        mmu_ops_t ops;
        union
        {
                sa_mmu_t sa_mmu;
                arm7100_mmu_t arm7100_mmu;
                arm920t_mmu_t arm920t_mmu;
                arm926ejs_mmu_t arm926ejs_mmu;
        } u;
} mmu_state_t;
```

5.5 添加 *skyeye s3c6410* 时所作的工作

5.51.添加 skyeye 中对 armv6 指令集和 MMU 的选择部分

1)在 arm_cpus 数组中添加
{"armv6", "arm11", 0x0007b000, 0x0007f000, NONCACHE}
其中 arm11 是 cpu 选项,
0x0007b000 是 smdk6410 的 ID
0x0007f000 是 machine ID 的屏蔽码
NONCACHE 不支持 cache (暂时)
这个 ID 在 linux 启动时会去探测。 (arm 通过访问 MMU_ID 寄存器获得)
2) 在 machine 代码初始化中增加
```
if (!strcmp(p_arm_cpu->cpu_arch_name, "armv6"))
    ARMul_SelectProcessor (state, ARM_v6_Prop);
if (!strcmp(p_arm_cpu->cpu_name, "arm11"))
            state->lateabtSig = LOW;
```
在 arch/arm/common/arminit.c skyeye_mach_init 中添加
```
state->is_v6 = (properties & ARM_v6_Prop) ? HIGH : LOW;
```
这些代码的作用是添加选择 armv6 指令集,在初始化时被调用,选择 armv6 是会同时打开 v4,v5 的选项。
3)在 mmu_init 中添加
```
case 0x0007b000:
        fprintf (stderr, "SKYEYE: use arm11jzf-s mmu ops\n");
    state->mmu.ops = arm1176jzf_s_mmu_ops;
    break;
```
作用是添加 arm11 的 mmu 操作函数选项。在启动时 MMU 的接口会根据 CPUID 选择 MMU 的操作函数。
4)在 mmu_state_t 中添加
```
ARMword translation_table_base0;
ARMword translation_table_base1;
ARMword translation_table_ctrl;
ARMword fault_statusi;  /* prefetch fault status */
```
添加了 armv6 页基址寄存器的定义和对取值异常原因的记录。
4.1)下面的三个寄存器用来保存页描述符的基地址
mmu.translation_table_base0
mmu.translation_table_base1
mmu.translation_table_ctrl
通过写入 mmu.translation_table_ctrl 基本器的值,来选择 base0 或者 base1 寄存器的使用。详见下面 mmu 翻译部分的介绍
4.2)因为没有加入 cache, armv6 在访问指令时会产生由缺页导致的取值异常,linux 会查询异常的原因。过去的实现中没有给取值异常记录异常原因和异常发生的地址,在异

常处理部分添加了这一部分。

```
if (!(skyeye_cachetype == INSTCACHE)) {
    /* set translation fault on prefetch abort */
      state->mmu.fault_statusi = fault              & 0xFF;
      state->mmu.fault_address = address;
}
```

5.5.2.已添加的 armv6 指令

在调试 linux kernel 的启动过程中,逐步添加了所需要的指令。

指令的实现添加到了 ARMul_Emulator32 函数中。

指令列表如下:

下面会使用的标注

SBO: should be one

SBZ :should be zero

2.1)strex

bit: 20-27: 0x18　　4,7:0x9　　8,11 SBO

寄存器 bit: 16-19: Rn　　12-15:Rd　　0-3:Rm

STREX{<cond>} <Rd>, <Rm>, [<Rn>]

STREX (Store Register Exclusive) performs a conditional storetomemory. Thestore onlyoccurs if the

executing processor has exclusive access to the memory addressed.

operation:

```
MemoryAccess(B-bit, E-bit)
ifConditionPassed(cond) then
   processor_id = ExecutingProcessor()
   physical_address = TLB(Rn)
   if IsExclusiveLocal(physical_address, processor_id, 4)then
        if Shared(Rn) == 1 then
            if IsExclusiveGlobal(physical_address, processor_id, 4)then
                Memory[Rn,4] = Rm
                Rd= 0
                ClearExclusiveByAddress(physical_address,processor_id,4)
            else
                Rd= 1
       else
            Memory[Rn,4] = Rm
            Rd= 0
   else
       Rd= 1
   ClearExclusiveLocal(processor_id)
   /*SeeSummary of operation on page A2-49*/
   /*Thenotes take precedence over any implied atomicity or
     order of events indicated in thepseudo-code */
```

2.2)ldrex

bit:20-27: 0x19　　4,7:0x9　　8,11: SBO 0,3:SBO

寄存器 bit: 16-19: Rn　　12-15:Rd

LDREX{<cond>} <Rd>, [<Rn>]
LDREX (LoadRegister Exclusive) loads a register from memory, and: ifthe address has theSharedmemory
attribute, marks the physical address as exclusive access for the executing processor in a shared monitor
causes the executing processor to indicate an activeinclusive access in the local monitor.
operation
MemoryAccess(B-bit, E-bit)
ifConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    Rd= Memory[Rn,4]
    physical_address = TLB(Rn)
    ifShared(Rn)== 1 then
        MarkExclusiveGlobal(physical_address,processor_id,4)
    MarkExclusiveLocal(physical_address,processor_id,4)
    /*SeeSummary of operation on page A2-49*/
2.3)strexb
bits:20,27: 0x1d 4,7:0x9 8,11:SBO
寄存器 bits 16,19: Rn 12,15 Rd 0,3 Rm
STREXB{<cond>} <Rd>, <Rm>, [<Rn>]]
Operation:
ifConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    if IsExclusiveLocal(processor_id)then
        if Shared(Rn)==1 then
            physical_address=TLB(Rn)
            if IsExclusiveGlobal(physical_address,processor_id,1)then
                Memory[Rn,1] = Rm
                Rd= 0
                ClearByAddress(physical_address,1)
            else
                Rd=1
        else
            Memory[Rn,1] = Rm
            Rd= 0
    else
        Rd= 1
    ClearExclusiveLocal(processor_id)
2.4)ldrexb
bits:20,27: 0x1d 4,7:0x9 8,11:SBO 0,3:SBO
寄存器 16,19: Rn 12:15 Rd
LDREXB{<cond>} <Rxf>, [<Rbase>]
operation
ifConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    Rd= Memory[Rn,1]

```
    ifShared(Rn)==1then
        physical_address=TLB(Rn)
        MarkExclusiveGlobal(physical_address,processor_id,1)
    MarkExclusiveLocal(processor_id)
```

2.5)SXTB

bits: 20,27:0x6a 16,19:0xf 8,9:SBZ 4,7:0x7

寄存器 bits:12,15:Rd 10,11:rotate 0,3 Rm

SXTB extracts an 8-bit value from a register and sign extends it to 32bits. You can specify a rotationby 0, 8,
16, or 24bits before extracting the8-bit value.

SXTB{<cond>} <Rd>, <Rm>{, <rotation>}

operation

```
ifConditionPassed(cond) then
    operand2= Rm Rotate_Right(8 * rotate)
    Rd[31:0] = SignExtend(operand2[7:0])
```

2.6)SXTAB

bits: 20,29:0x6a 8,9:SBZ 4,7:0x7

寄存器和功能 16,19:Rn 12,15:Rd 0,3:Rm

SXT AB{<cond>} <Rd>, <Rn>, <Rm>{, <rotation>}

SXT extracts an 8-bit value from a register, sign extends it to 32 bits, and adds the result to the value in
AB
another register. Y canspecify a rotation by0, 8, 16, or 24 bits before extracting the 8-bit value.

ou

Operation

```
ifConditionPassed(cond) then
    operand2= Rm Rotate_Right(8 * rotate)
    Rd= Rn+ SignExtend(operand2[7:0])
```

2.7)SXTH

bits: 20,27:0X6b 16,19:0xf 4,7:0x7 8,9:SBZ

寄存器 bits: 12,15: Rd 0,3:Rm 10:11 rotate

SXTH{<cond>} <Rd>, <Rm>{, <rotation>}

SXTH extracts a 16-bit value from a register and sign extends it to 32 bits. You can specify a rotation by 0,
8,16, or 24 bits before extracting the 16-bit value.

operation

```
ifConditionPassed(cond) then
    operand2= Rm Rotate_Right(8 * rotate)
    Rd[31:0] = SignExtend(operand2[15:0])
```

2.8)SXTAH

bits:20,27:0x6b 4,7:0x7 8,9:SBZ

寄存器 bits:16,19:Rn 12,15:Rd 0,3:Rm 10,11:rotate

SXT AH{<cond>} <Rd>, <Rn>, <Rm>{, <rotation>}

SXT extracts a 16-bit value from a register, sign extends it to 32 bits, and adds the result to a value in

AH
another register. Y canspecify a rotation by0, 8, 16, or 24 bits before extracting the 16-bit value.
ou
Operation
ifConditionPassed(cond) then
operand2= Rm Rotate_Right(8 * rotate)
Rd = Rn + SignExtend(operand2[15:0])
2.9)UXTB
bits: 20,27:0x6e 16,19:0xf 4,7:0x7 8,9:SBZ
寄存器 bits:12,15:Rd 0,3:Rm 10,11:rotate
UXTB{<cond>} <Rd>, <Rm>{, <rotation>}
UXTB extracts an 8-bit value from a register and zero extends it to 32 bits. Y can specify a rotation by 0,
ou
8,16, or 24 bits before extracting the 8-bit value.
Operation:
ifConditionPassed(cond) then
Rd[31:0] = (Rm Rotate_Right(8 * rotate)) AND0x000000ff
2.10)UXTAB
bits:20,27:0x6e 4,7:0x7 8,9:SBZ
寄存器 bits:16,19:Rn 12,15:Rd 0,3:Rm 10,11:rotate
UXTAB{<cond>} <Rd>, <Rn>, <Rm>{, <rotation>}
UXTAB extracts an 8-bit value from a register, zero extends it to 32 bits, and adds the result to the value in
another register. Y canspecify a rotation by0, 8, 16, or 24 bits before extracting the 8-bit value.
ou
Operation:
ifConditionPassed(cond) then
operand2= (Rm Rotate_Right(8 * rotate)) AND0x000000ff
Rd= Rn+ operand2
2.11)UXTAH
bits: 20,27:0x6f 4,7:0x7 8,9:SBZ
寄存器 bits: 16,19:Rn 12,15:Rd 0,3:Rm 10,11:rotate
UXTAH{<cond>} <Rd>, <Rn>, <Rm>{, <rotation>}
UXTAH extracts a 16-bit value from a register, zero extends it to32 bits, and adds the result to a value in
another register. Y canspecify a rotation by0, 8, 16, or 24 bits before extracting the 16-bit value.
ou
Operation:
ifConditionPassed(cond) then
operand2= (Rm Rotate_Right(8 * rotate)) AND0x0000ffff
Rd = Rn + operand2
2.12)CPS

bits:28,31:0xf 20,27:0x10 9,16:0 15:SBZ 0,4:0
数据 bits: 18,19 :imod 17:mmod 0,4:mod 6:F 7:I 8:A
CPS<effect> <iflags> {, #<mode>}
Specifies what effect is wanted on the interrupt disable bits A, I, and F in theCPSR. This is
Operation:
if InAPrivilegedMode() then
if imod[1] == 1 then
if A== 1 thenCPSR[8] = imod[0]
if I == 1 thenCPSR[7] = imod[0]
ifF == 1 thenCPSR[6] = imod[0]
/* elseno changeto themask */
ifmmod == 1 then
CPSR[4:0] = mode

5.5.3.增加了 mmu 部分的代码文件 arm1176jzf_s_mmu.c

mmu 部分基于 arm920t MMU 的实现,去掉了 tlb 和 cache 部分。 地址翻译部分有些不同,armv6 只支持 4k,64K page 和 1M, 16M 的 Section。页基址的寄存器的使用上也有些不同。

3.1 页基地址寄存器的使用

s3c6410 中有关页基址寄存器有下面三个:
Translation Table Base 0 (TTBR0)
Translation Table Base 1(TTBR1)
Translation Table Base Control。
称 Table Base Control 中的值为 N,则 N 的取值有 0<=N<= 7,跟据 N 的值对下面两个 base 寄存器的选择方法如下:
if , N == 0 , Use base0
others, if Vaddr bit[31:32-N] == 0 Use base0
if Vaddr bit[31:32-N] >0 Use base1
举例:
N=1 时,bit[31] == 0 Use base0 翻译的地址范围 0G-2G
翻译的地址范围 2G-4G
other Use base1
N=2 时,bit[31,30] == 0b00 Use base0 翻译的地址范围
0G-1G
Use base1 翻译的地址范围 1G-4G
other
其它 N 值.....

Base1 和 Base0 的使用:

使用 Base1 时:
一级描述符的地址由 BASE 中的前 18 位,虚拟地址的前 12 位和最后的两个 0 组成
(BASE1 & 0xFFFFC000)| vaddr >> 18 << 2 & ~3

使用 Base0 时:
一级描述符的地址由 Base0 中的 18+N 位和虚拟地址的前 14-N 位组成,N 为 ctrl 寄存器中的值。

(BASE1 >> (14-n) << (14-n)) | vaddr >> (18+N) & ~3

3.2.翻译过程

3.2.1 根据获取的一级描述符, 根据后两个 bit 的值进行判断, 进行段映射的翻译或是获取二级描述符的地址。

level1[1,0]:

0,3: 表示有错误

1:表示使用页映射(page table)

二级描述符的地址由一级描述符的 31-10 位, 虚拟地址的 19-12 位和最后的 2 位 0 组成

((Levfel1 & 0xFFFFFC00) | (Vaddr &0x000FF000) >> 10) & ~3

获的二级描述的地址。在 3.2.2 中继续翻译工作。

2: 表示使用段映射(section)

根据取得一级描述符的第 19 (bit18)位判定

1:16M 翻译得到的地址为

(level1 & 0xFF000000) | (vaddr & 0x00FFFFFF)

0: 1M 翻译得到的地址为:

(level1 & 0xFFF00000) |(vaddr & 0x000FFFFF)

结束地址翻译。

3.2.2 根据页映射二级描述符获得翻译地址

如果为页映射则取二级描述符

根据二级描述符的后两位选择:

0:错误

1: 64K 页,翻译的地址为有二级描述符的 16-31 位和虚拟地址 0-15 位组成.

(level2 & 0xFFFF0000) | (Vaddr & 0x0000FFFF)

2,3: 4k 页, 翻译的地址为有二级描述符的 12-31 位和虚拟地址 0-11 位组成

(level2 & 0xFFFFF000) | (Vaddr & 0x00000FFF)

5.5.4.添加 6410 machine 的外设部分

使用了 2410machine 部分的模板,重新定义了控制器地址(详见 s3c6410.h),仿照 2140 的 machine 部分实现了 timer、uart 和中断部分的控制器。

4.1)timer 的实现:跟据设定,在 io_do_cycle 函数中实现时钟计数的累加,在时间到达时发生中断,并重新设定。与 2410 中的实现相同。

4.2) uart 的实现:与 2410 中的实现相同。

4.3)中断的实现:

中断部分定义了下面的寄存器:

VIC0RAWINTR //中断记录(被屏蔽的中断发生时也会记录)

VIC0INTSELECT //中断类型(是否使用快速中断)

VIC0INTENABLE //中断始能

VIC0IRQSTATUS //中断状态

VIC0FIQSTATUS

中断处理过程简述:

有中断发生是通过判断中断是否始能, 中断选择的类型和发生的中断编号来设置状态寄存器的中断状态, 根据是否有中断状态的设置来触发对 arm11 核的中断请求, 在读取状态位时清除此中断状态,对 cpu 的中断请求停止。完成本次中断。

4.4)添加开发板的标识:

内核启动时会通过访问已定地址 0x7e00f118 来探测开发版的类型, smdk6410 的标识为 0x36410100, 则在 s3c6410x_io_read_word 函数中加入这个地址,供内核访问。

```
case 0x7e00f118:
  data = 0x36410100;
  break;
```

5.5.5.armv6 的特性列表

1.Unaligned data support for word halfwords
2.SIMD single instruct operating two half or four bytes
3.ARMv6 introduces a set of memory types - Normal, Device, and Strongly
4. CP15 register 1 add S and R option
S (bit[8]) System protection bit, supported for backwards compatibility. The effect of this bit is described in Access permissions on page B4-8. The functionality is deprecated in ARMv6.
R (bit[9]) ROM protection bit, supported for backwards compatibility. The effect of this bit is described in Access permissions on page B4-8. The functionality is deprecated in ARMv6.
5.CP15 registers 12 and 14 UNDEFINED from ARMv6.
6. ARMv6 systems shall include a System Control Coprocessor, with support for automatic interrogation of cache, tightly coupled memory, and coprocessor provision. It also provides the control mechanism for memory management (MMU and MPU support as applicable).
7.VMSAv6 has added definitions for different memory types.
8.PSMA v6
9.Imprecise Abort (external abort) - ARMv6
improve exception handling
10.The Jazelle Extension Armv6
11.From ARMv6, a byte-invariant mixed-endian format is supported, along with alignment checking options。defined by the CPSR E-bit.
12.context id armv6
13.debug provisions armv6
14.debug armv6(12-14 详见 armv6 手册)
15.armv6 扩展的指令列表:
E xtend instr uctions:
XTAB16,XTAB,XTAH,XTB16,XTB,XTH,
UXTAB,UXTAB16,UXTAH,UXTB,UXTB16,UXTH,
L ist of sign/zer o extend and add instructions:
SXTAB16,SXTAB,SXTAH,SXTB16,SXTB,SXTH,UXTAB16,UXT

AB,UXTAH,UXTB16,UXTB,,
CPSR oper ation instr uctions:
CPSIE,CPSID,CPS
Synchr onization instructions:
SWAP,SWAPB,
LDREX,STREX,CLREX,LDREXB,LDREXH,STREXB,STREXH,
ARM v6 media data-pr ocessing instr uctions :
QADD16 ,QADD8 ,QADDSUBX,QSUB16 ,QSUBADDX
UQADD16,UQADD8,UQADDSUBX,UQSUB16,UQSUB8
AQSUBADDX
SADD16,SADD,SADDADDX,SETEND,SHADD16,SHADD8,SHA
DDSUBX,SHSUB16,SHSUB8,SHSUBADDX,SSUB16,SSUB8,SSU
BADDX,
UADD16,UADD8,UADDSUBX,UHADD16,UHADD8,UHADDSU
BX,UHSUB16,UHSUB8,UHSUBADDX,USUB8,USUBADDX,USU
B16,USAD8,USADA8,
Other miscellaneous instr uctions:
PKHBT,PKHTB,REV,REV16,REVSH,SEL,SSAT,SSAT16,USAT,U
SAT16
Multiply instr uction:
SMULL,SMULLS,SMLAL,SMLALS,
SMLAD,SMLAL,SMLALD,SMLSD,SMLSLD,SMMLA,SMMUL,S
MUAD,SMULL,SMUSD,SRS,,UMAAL
Unconditional instruction :
CPS/SETEND,PLD,RFE,SRS,BLX,MCRR2,MRRC2,STC2,LDC2,
CDP2,MCR2,MRC2
(在 skyeye 中并没有实现的扩展,只加入运行 linux 内核时所需要的指令)

第六章、MIPS 处理器仿真模块的代码分析

6.1 背景介绍

MIPS 仿真实现了对 MIPS32 平台的仿真，支持 au1100, godson 处理器的仿真。

6.2 与核心模块的接口部分

```
603     static arch_config_t mips_arch;
604     mips_arch.arch_name = arch_name;
605     mips_arch.init = mips_init_state;
606     mips_arch.reset = mips_reset_state;
607     mips_arch.step_once = mips_step_once;
608     mips_arch.set_pc = mips_set_pc;
609     mips_arch.get_pc = mips_get_pc;
610     mips_arch.ICE_read_byte = mips_ICE_read_byte;
611     mips_arch.ICE_write_byte = mips_ICE_write_byte;
612     mips_arch.parse_cpu = mips_parse_cpu;
613     mips_arch.get_step = mips_get_step;
614     //mips_arch.parse_mach = mips_parse_mach;
615     //mips_arch.parse_mem = mips_parse_mem;
616     mips_arch.get_regval_by_id = mips_get_regval_by_id;
617     mips_arch.get_regname_by_id = mips_get_regname_by_id;
618     register_arch (&mips_arch);
```

6.3 运行流程的代码分析

mips 单步执行一条指令的代码在函数 mips_step_once 函数，实现如下：

```
281 static void
282 mips_step_once()
```

```
283 {
284         mstate->gpr[0] = 0;
285
286         /* Check for interrupts. In real hardware, these have a priority lower
287          * than all exceptions, but simulating this effect is too hard to be
288          * worth the effort (interrupts and resets are not meant to be
289          * delivered accurately anyway.)
290          */
291         if(mstate->irq_pending)
292         {
293                 mips_trigger_irq(mstate);
294         }
295
296         /* Look up the ITLB. It's not clear from the manuals whether the ITLB
297          * stores the ASIDs or not. I assume it does. ITLB has the same size
298          * as in the real hardware, mapping two 4KB pages.  Because decoding a
299          * MIPS64 virtual address is far from trivial, ITLB and DTLB actually
300          * improve the simulator's performance: something I cannot say about
301          * caches and JTLB.
302          */
303
304         PA pa; //Shi yang 2006-08-18
305         VA va;
306         Instr instr;
307         int next_state;
```

```
308         va = mstate->pc;
309         mstate->cycle++;
310         if(translate_vaddr(mstate, va, instr_fetch, &pa) == TLB_SUCC){
311                 mips_mem_read(pa, &instr, 4);
312                 next_state = decode(mstate, instr);
313                 //skyeye_exit(-1);
314         }
315         else{
316                 //fprintf(stderr, "Exception when get instruction!\n");
317         }
318
319         /* NOTE: mstate->pipeline is also possibely set in decode function */
332
333         switch (mstate->pipeline) {
334                 case nothing_special:
335                         mstate->pc += 4;
336                         break;
337                 case branch_delay:
338                         mstate->pc = mstate->branch_target;
339                         break;
340                 case instr_addr_error:
341                         process_address_error(mstate, instr_fetch, mstate->branch_target);
342                 case branch_nodelay: /* For syscall and TLB exp, we donot like to add pc
*/
343                         mstate->pipeline = nothing_special;
344                         return; /* do nothing */
```

```
345         }
346         mstate->pipeline = next_state;
```

310 到 312 行实现了地址翻译，取指，执行指令的过程。

6.4 异常和中断的仿真的代码分析

```
57 void
58 process_exception(MIPS_State* mstate, UInt32 cause, int vec)
59 {
61         UInt32 exc_code = cause & 0x7f;
62         mstate->now += 5;
63         /* we need to modify pipeline according to different exception */
64
65         VA epc;
66         if (!branch_delay_slot(mstate))
67                 epc = mstate->pc;
68         else {
69                 epc = mstate->pc - 4;
70                 cause = set_bit(cause, Cause_BD);
71         }
72
73         if((exc_code == EXC_Sys) || (exc_code == EXC_TLBL)
74                 ||(exc_code) == EXC_CpU || (exc_code == EXC_TLBS) || (exc_code ==
EXC_Mod)){
75                 mstate->pipeline = branch_nodelay;
76                //fprintf(stderr, "KSDBG:1 in %s, vec=0x%x, cause=0x%x, ,v0=0x%x,
pc=0x%x\n", __FUNCTION__, vec, exc_code     , mstate->gpr[2], mstate->pc);
77         }
```

```
78        else{
79                mstate->pipeline = nothing_special;
80        }
81
82        /* Set ExcCode to zero in Cause register */
83        mstate->cp0[Cause] &= 0xFFFFFF83;
84        mstate->cp0[Cause] |= cause;
85                mstate->cp0[EPC] = epc;
86        mstate->pc = vec + (bit(mstate->cp0[SR], SR_BEV) ? general_vector_base :
boot_vector_base);
87        /* set EXL to one */
88        mstate->cp0[SR] |= 0x2;
89        /* Set Exl bit to zero, disable interrupt */
90        mstate->cp0[SR] &= 0xFFFFFFFD;
91        enter_kernel_mode(mstate);
94 }
```

6.5 多核仿真的代码分析

无

第七章、X86 处理器仿真模块的代码分析

7.1 背景介绍

X86 仿真实现了对 i386 体系结构的仿真，支持一般的 i386 的处理器。其原始代码来自于 Bochs 项目。

7.2 与核心模块的接口部分

7.3 运行流程的代码分析

7.4 异常和中断的仿真的代码分析

7.5 多核仿真的代码分析（无）

第八章、外设仿真－cs8900网卡仿真

5.1 关键数据结构介绍

每一个外设需实现数据结构 device_desc_t，其描述如下：

```
84 typedef struct device_desc
85 {
86        /* device type name.
87         * if inexistance, can be gotten from "mach name"
88         * e.g. ep7312, at91.
89         * */
90        char type[MAX_STR_NAME];
91
92        /* device instance name.
93         * The same type of device may have two or more instances, but they
94         * have different name. "name" can identify different instances.
95         * e.g. the "s3c4510b" uart has two instances: uart1 and uart2.
96         * */
97        char name[MAX_STR_NAME];
98
99        /*I/O or memory base address and size */
100       uint32 base;
101       uint32 size;
102
103       /* interrupt of device.
104        * */
105       struct device_interrupt intr;
106
107       /* mem operation
108        * */
109       struct device_mem_op mem_op;
110
111       void (*fini) (struct device_desc * dev);        /*finish routine */
112       void (*reset) (struct device_desc * dev);       /*reset device. */
113       void (*update) (struct device_desc * dev);      /*called by io_do_cycle */
114
115       int (*filter_read) (struct device_desc *dev, uint32 addr, uint32 *data, size_t
count);
116       int (*filter_write) (struct device_desc *dev, uint32 addr, uint32 data, size_t
count);
117
118       int (*read_byte) (struct device_desc * dev, uint32 addr, uint8 * result);
119       int (*read_halfword) (struct device_desc * dev, uint32 addr,
120                             uint16 * result);
121       int (*read_word) (struct device_desc * dev, uint32 addr, uint32 * result);
122
```

```
123         int (*write_byte) (struct device_desc * dev, uint32 addr, uint8 data);
124         int (*write_halfword) (struct device_desc * dev, uint32 addr, uint16 data);
125         int (*write_word) (struct device_desc * dev, uint32 addr, uint32 data);
126
127         /* refer the "mach" that the device belongs to.
128          * */
129         void *mach;
130
131         /* specific common data for a type of device
132          * */
133         void *dev;
134
135         /* device specific data
136          * usually be an "io" struct.
137          * */
138         void *data;
139 } device_desc_t;
```

device_desc_t 描述了一个外设需要实现的接口和数据变量的实例。每一个这样的数据结构都代表了系统中的一个外设实例或者表示一个实际的物理外设设备。

然后我们在 device/net/dev_net_cs8900a.c 文件中定义了大小为 MAX_DEVICE_NUM 的 cs8900a_devs 这样一个数组，用来代表们最大可以有 MAX_DEVICE_NUM 个 cs8900a 的网卡同时存在在系统中。

```
41 #define MAX_DEVICE_NUM 10
42 static struct device_desc *cs8900a_devs[MAX_DEVICE_NUM];
```

第九章、 外设仿真 - 触摸屏的仿真设计

9.1 TouchScreen 模拟的设计

TouchScreen 模拟模块的设计思路将，将与 LCD 模拟窗口同样大小的 GTK+组件置于 LCD 组件容器中，并为该组件注册鼠标键按下，释放及移动三种事件，当鼠标在组件窗口有键按下，释放或移动的动作，则在相应的事件回调函数种记录其在窗口上的坐▪及▪的状▪，并▪生修改中断寄存器中的相▪位置 1，在 SkyEye 上运行的嵌入式 OS 检测到中断寄存器的数据变化就产生中断，TouchScreen 驱动程序中注册了该中断的中断服务程序 ISR 则复制所记录的数据供应用程序使用，这一思路简单说来就是，完成 GTK+的鼠标事件到 TouchScreen 事件的映射。

因此 TouchScreen 模拟模块只需要关注 GTK+鼠▪事件的▪生，▪▪事件数据并在*_io_do_cycle 函数中对 I/O 模拟模块所模拟的中断状态寄存器进行置数操作, 即为嵌入式操作系统内核产生中断信号的条件。

下▪就是 SkyEye 模拟器的 TouchScreen 模拟的流程图（包括与真实硬件的比较）

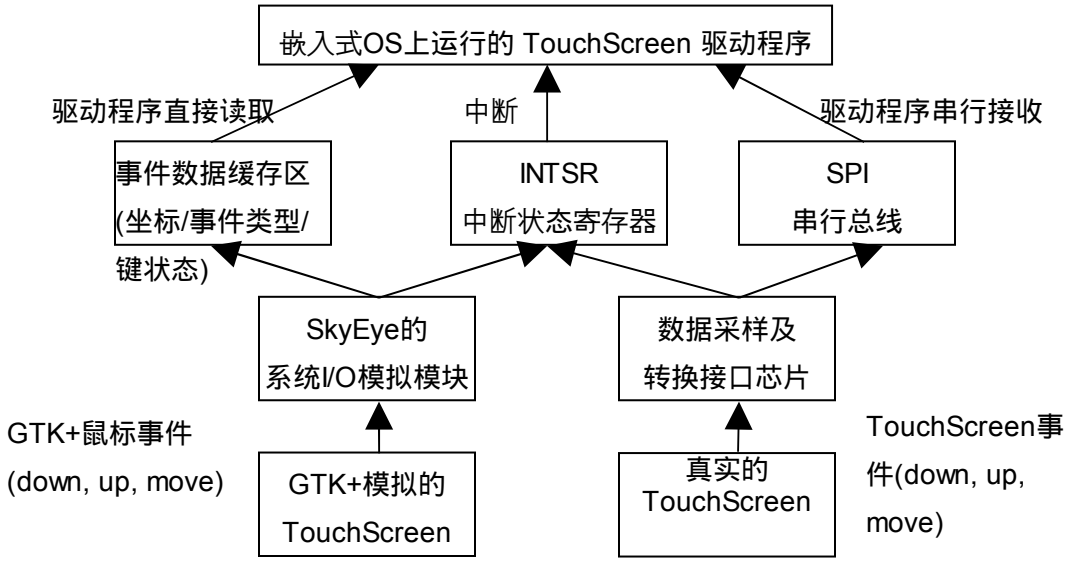


图 2.3 SkyEye 模拟器的 TouchScreen 模拟流程图

9.2 TouchScreen 模拟的实现

TouchScreen 模拟模块的实现采用了与模拟采用 DragonBall 开发板的 Xcopilot 模拟器相类似的简化方式。在实际的 TouchScreen 硬件中，为了定位动作发生的坐标，要先经过一个 12 位的 A/D 转换器分别转换 X，Y 坐标对应 12 位数字量，然后由驱动程序通过 SPI 串行总线串行接收。SkyEye 作为一个指令级的模

拟器，无需保证与真实时钟节拍在时序上的一致，因此允许对 TouchScreen 这样的外设的模拟进行简化。

首先注册 GTK 鼠标事件，包括鼠标的按下事件，鼠标的释放事件和鼠标的移动事件。为获取鼠标事件及其状态坐标信息做准备。

```
// 注册鼠标的按下事件
gtk_signal_connect (GTK_OBJECT(TouchScreen), "button-press-event",
                    GTK_SIGNAL_FUNC (callback_button_press), NULL);
// 注册鼠标的释放事件
gtk_signal_connect (GTK_OBJECT(TouchScreen), "button-release-event",
                    GTK_SIGNAL_FUNC (callback_button_release), NULL);
// 注册鼠标的移动事件
gtk_signal_connect (GTK_OBJECT(TouchScreen), "motion-notify-event",
                    GTK_SIGNAL_FUNC (callback_motion_notify), NULL);
```

由于鼠标的移动事件只是简单的移动事件，而真实的触摸屏的移动事件是笔触杆接触面板按下且移动，是两个事件组成的复合事件，为了映射这两种事件，需要对鼠标移动事件做特殊处理，在鼠标移动的回调处理函数中判断鼠标移动时的按键状态，只处理鼠标按下且移动事件，抛弃鼠标移动但鼠标未按下的事件。

```
// 鼠标按下的回调处理函数
void callback_button_press(GtkWidget *w, GdkEventButton *event)
{
        skPenEvent(Pen_buffer,0,1,event->x,event->y); // 记录鼠标事件类型，状态及坐标
}// 鼠标释放的回调处理函数
void callback_button_release(GtkWidget *w, GdkEventButton *event)
{
        skPenEvent(Pen_buffer,1,0,event->x,event->y); // 记录鼠标事件类型，状态及坐标
}
// 鼠标移动的回调处理函数
void callback_motion_notify(GtkWidget *w, GdkEventMotion *event)
{
        if(Pen_buffer[5]==1){
                skPenEvent(Pen_buffer,2,1,event->x,event->y); // 记录鼠标事件类型，状
态及坐标
   }
}
```

在*_io_do_cycle 中判断触摸屏的中断位是否为 0，如果为 0，则查询是否有新的鼠标事件的数据产生将鼠标事件数据映射为触摸屏事件数据，并对触摸屏中断

位置 1，等待触摸屏 ISR(中断服务程序)读取这些数据，填充到触摸屏事件数据结构体中，供上层应用程序使用。

```
if(!(io.intsr&io.ts_int)){
        if(Pen_buffer[6]==1){    // 判断缓冲区中是否有未处理的鼠标事件数据
                // 复制鼠标事件的数据到缓冲区，驱动程序将读取这些数据
                *(state->mach_io.ts_buffer+0)=Pen_buffer[0];
                *(state->mach_io.ts_buffer+1)=Pen_buffer[1];
                *(state->mach_io.ts_buffer+4)=Pen_buffer[4];
                *(state->mach_io.ts_buffer+6)=Pen_buffer[6];
                io.intsr|= io.ts_int;          // 中断置位，触发中断服务程序
      Pen_buffer[6]=0;
        }
}
```

第十章、外设仿真 - LCD 的仿真设计和实现

10.1 LCD 模拟的设计

LCD 模拟模块的设计思路是，使用 GTK+图形系统在 X Window 系统和 Win32 系统上实现一个 LCD 屏幕模拟，在 SkyEye 上运行的嵌入式操作系统中的 LCD 驱动程序象驱动真正的 LCD 控制器一样发送控制命令或对 LCD 显示内存进行访问操作，而 SkyEye 解释这些控制命令，并根据这些命令对 LCD 屏幕窗口进行相应的 GTK+图形操作，完成对不同灰度或颜色图形的绘制。

在 SkyEye 模拟器中，如果嵌入式操作系统要执行 I/O 地址访问，具体的处理过程由特定 CPU 和开发板 I/O 模拟模块中的 read/write_byte/halfword/word 函数处理。所以 LCD 模拟模块关注的主要是内存模拟模块模拟出来的 LCD 显示内存中存储的数据。

LCD 的显示内存映射到内存 RAM 中，代表了要在 LCD 屏幕上显示的图像。显示内存必须足够大，以处理显示屏幕上所有的象素。应用程序通过直接或间接地存取显示内存中的数据来进行图形操作，改变屏幕显示的内容。

LCD 模拟模块对 GTK+的使用目前仅限于根据分辨率(例如 320x240，640x480)创建相应大小的窗口以及根据显示内存中的数据逐点在该窗口进行绘制，因为画点是 LCD 屏幕最基本的动作，所有其它的相对复杂工作如图形绘制，嵌入式 GUI 系统的实现都应该由基于 LCD 驱动程序的应用程序（包括基于 FrameBuffer 驱动程序的嵌入式 GUI 系统，例如 MiniGUI）通过对 LCD 显示内存的读写操作来实现，SkyEye“看到”的只是显存中对应于屏幕上各个点的像素值，而不关心这些像素值组成的是什么样的图像。

下图就是 SkyEye 模拟器的 LCD 模拟的流程图（包括与真实硬件的比较）。

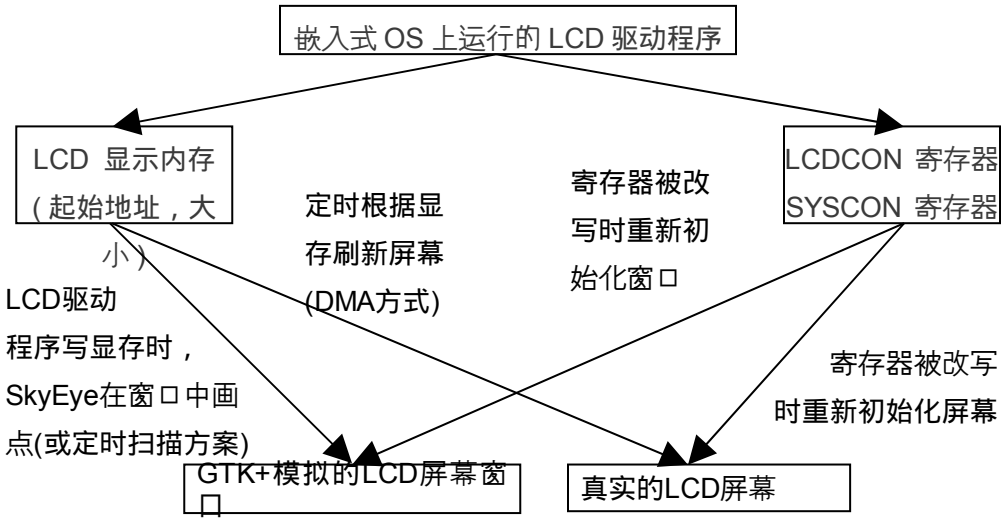


图 2.1 SkyEye 模拟器的 LCD 模拟流程图

第十一章、外设仿真 - UART的仿真设计和实现

第十二章、外设仿真 - Flash 的仿真设计和实现

12.1 Flash 模拟的设计

Flash 模拟的设计思路是，在 SkyEye 的存储器模拟框架下，以 SkyEye 模拟的 RAM，即一段内存空间，作为 Flash 的“物理”存储空间，实现该类型存储空间之上特有的读写函数，这些读写函数属于 Flash 空间的属性，与 Flash 空间“绑定”在一起，SkyEye 会自动解析并判断读写操作的目标地址是否属于 Flash 空间，从而正确调用相应的 Flash 读写函数。

Flash 的模拟原则就是，使符合 CFI 接口标准 Flash 驱动程序不经任何修改就能按照标准的 Flash 读/写/擦除流程操作这块模拟的 Flash，从而支持 bootloader 的 Flash 操作或更高层的 MTD 驱动和 JFFS2 文件系统。

由于 SkyEye 模拟的时钟目前还不完善，因此需要系统定时器支持的 Flash 写/擦除操作的暂停和恢复无法起作用，这使 Flash 模拟的真实性受到一定的影响，但是不影响 Flash 驱动程序在模拟的 Flash 上的正确运行。

下图就是 SkyEye 模拟器的 Flash 模拟的流程图：

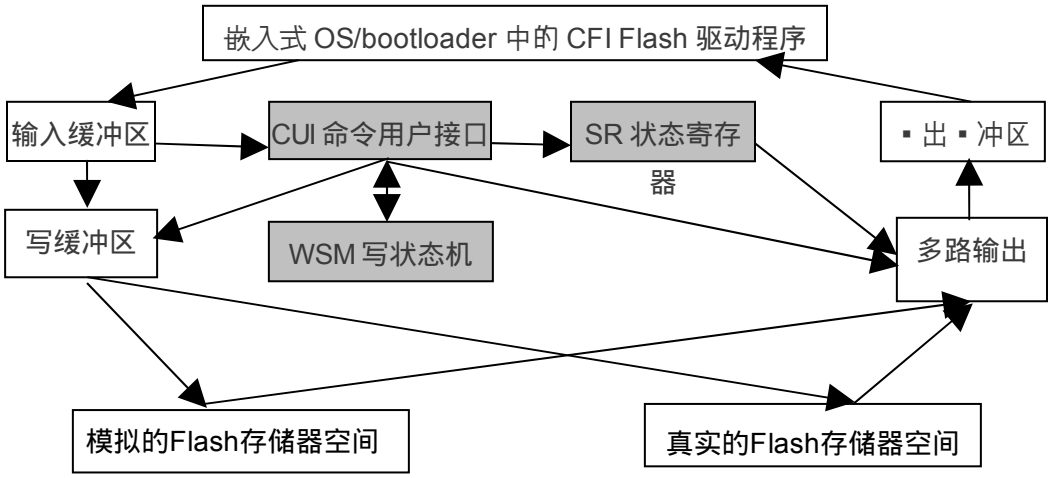


图 3.2 SkyEye 模拟器的 Flash 模拟流程图

12.2 Flash 模拟的实现

首先在 skyeye/sim/arm/skyeye_options.c 中增加对 skyeye.conf 中 Flash 型存储器的支持。Flash 空间将拥有不同于 ROM/RAM 空间和 IO 空间的读写操作函数 flash_read/write_byte/halfword/word()。Flash 模拟的核心也就是实现这六个 Flash 读写函数，模拟真实的 Flash 的读/写/擦除操作及其流程，并使 Flash 状态寄存器产生相应的状态值，供 Flash 驱动程序读取并验证读/写/擦除操作是否成功。

由于 Flash 模拟首先是在 SkyEye 模拟的 Lubbock 开发板上实现，而 Lubbock 开发板采用了两组各两片 16 位宽度的 28F640，所以最大的读写宽度是 16x2=32 位 Error: Reference source not found。所以 flash_read/write_word()是实现的主体，flash_read/write_byte/halfword ()这四个函数可以对数据宽度进行处理后调用 flash_read/write_word()。

```
if (!strncmp("F", value, strlen(value))){                    //Flash 类型的存储空间
        mb[num].read_byte = flash_read_byte;                   // 按字节读 Flash
        mb[num].write_byte = flash _write_byte;                // 按字节写 Flash
        mb[num].read_halfword = flash _read_halfword;// 按半字读 Flash
        mb[num].write_halfword = flash _write_halfword;        // 按半字写 Flash
        mb[num].read_word = flash _read_word;                  // 按字读 Flash
        mb[num].write_word = flash _write_word;                // 按字写 Flash
        mb[num].type=MEMTYPE_FLASH;
}
```

12.2.1 Byte/word 编程的实现

图 3.3 Byte/word 编程流程图

Byte/Word 编程由一个双周期命令序列执行。先写入Byte/Word 编程开始命令（标准的是0x40，也可以用0x10），紧接着写入目标地址和数据。CUI启动一个写时序后，将具体的写入操作交由WSM处理。WSM自动执行预置的编程和验证算法。Byte/Word 编程操作的双周期命令序列被写入后，读操作将触发设备自动产生状态寄存器数据。

```
case WSM_PROGRAM:
        program_latch_addr = addr;                                  // 目 ▪ 地址
        program_latch_data = data;                                  // 数据
        read_mode = WSM_READ_STATUS;
        if (ISLOCKED(program_latch_addr)) {          // 判断目标地址所在块是否被锁位
                program_setlb_error = protection_error = 1;
        }else{ // 将数据写入与目标地址对应的 Flash 存储空间

real_write_word(state,program_latch_addr,real_read_word(state,program_latch_addr)&data);
        }
        wsm_mode = WSM_READY;
        break;
```

12.2.2 缓冲区写入的实现

图 3.4 缓冲区写入流程图

Byte/Word 编程操作是按字节/字对Flash存储空间进行写操作，为了提高写入效率，缓冲区写入操作则可以将写缓冲区的数据一次性写入Flash存储空间。

缓冲区写入操作流程由缓冲区写入设备命令启动。不超过写缓冲区大小(32字节)的数据块装入写缓冲区后，由WSM一次性将其复制到Flash存储空间。首先向Input Buffer 输入缓冲区写入设备命令(0xE8)，然后需要先检查写缓冲区是否可用。如果XSR（扩展状态寄存器）的最高位为1，表明写缓冲区可用，就可以继续缓冲区写入操作流程，如果XSRD7为0且没有超时，可以再次查询状态，直到写缓冲区可用。

接下来就是要获取数据块的长度（字节或字数），该长度不能超过写缓冲区的长度，然后开始向写缓冲区逐一装入数据，记录第一个数据的目标地址为数据块

的目标首地址，另外还有一个计数器记录以装入的数据的长度。该技术器的值不能超过先前确定的数据块的长度。

当输入最后一个数据后，写入操作确认命令(0xD0)，触发WSM将写缓冲区内的数据复制到Flash存储空间中。如果写入的不是操作确认命令(0xD0)，将会产生“非法命令或命令序列”的错误，状态寄存器的SR5和SR4会被置位，Flash驱动程序读取SR的值经过判断后将调用相应的出错处理程序。

```
case WSM_WRITE_BUFFER:
        if (pb_count == 0) {                                    // 获取数据块的字节数
                pb_count = (data & 0xff) + 1;
                progbuf_latch_addr = addr;
        }
        if (pb_loaded < pb_count) {                             // 将 Input Buffer 中的数据装入写缓冲区中
                if (pb_loaded == 0){
                        pb_start = addr;                        // 获取数据块的的目标首地址
                }
                pb_buf[WORD_ADDR(addr) & INTEL_WRITEBUFFER_MASK] = data;
                pb_loaded ++;
                break;
        }
        if ((data & 0xff) == WSM_CONFIRM) { // 操作确认
                progbuf_busy = 1;
                read_mode = WSM_READ_STATUS;
                if (ISLOCKED(progbuf_latch_addr)) {     // 判断目标地址所在块是否被锁位
                        program_setlb_error = protection_error = 1;
                }
```

```
                else {    // 如果该块没有被锁位，则将 buffer 中的数据复制到 Flash 存储
空间
                        // 也可以采用 memcpy 实现复制
                    for (j = WORD_ADDR(pb_start); j < WORD_ADDR(pb_start) +

INTEL_WRITEBUFFER_SIZE; j ++){
        real_write_word(state,j,real_read_word(state,j)
                                                    &pb_buf[j &
INTEL_WRITEBUFFER_MASK]);
                        }
                }
                wsm_mode = WSM_READY;
                progbuf_busy = 0;
                break;
        }
        break;
```

12.2.3 块擦除的实现

图 3.5 块擦除流程图

擦除操作以块为单位，由一个双周期命令序列触发，第一个是擦除操作命令0x20，第二个是确认命令。块擦除操作将目标地址所在块擦除为全“1”。写入块擦除操作的双周期命令序列后，读操作将触发设备自动产生状态寄存器数据。

```
case WSM_BLOCK_ERASE:
        if ((data & 0xff) == WSM_CONFIRM) {
                if (ISLOCKED(erase_latch_addr)) {                    // 判断目标地址所在块
是否被锁位
                        erase_clearlb_error = protection_error = 1;
                }else {
                        for (i = 0; i < WORD_ADDR(FLASH_SECTOR_SIZE); i ++){
                                real_write_word(state,erase_latch_addr+i*4,0xffffffff);  //
全“ 1” 擦除
                        }
                }
        }
        break;
```

12.2.4 设置块锁位的实现

图 3.6 设置块锁位(写保护)流程图

块写保护通过设置对应块锁位实现。出于写保护状态的块不能进行编程和擦除操作。通过块锁位设置命令(0x60)可以进行单独的块锁位设置。当 WSM 正在运行或设备出于暂停状态时，不能进行块锁位设置操作。块锁位设置由一个双周期命令序列执行。首先向块内地址写入块锁位设置命令，紧接着写入块锁位设置确认命令(0x01)。CUI 启动一个块锁位设置时序后，将具体的块锁位设置操作交由 WSM 处理。WSM 自动执行块锁位设置算法。块锁位设置操作的双周期命令序列被写入后，读操作将触发设备自动产生状态寄存器数据。

```
case WSM_LOCK_ACCESS:
        if ((data & 0xff) == 0x01) {                    // 确认是否进行块锁
位设置操作
                lock_busy = 1;
                read_mode = WSM_READ_STATUS;

                lock[SECTOR_ADDR(addr)] = 1;            // 对块设置块锁位

                lock_busy = 0;
                wsm_mode = WSM_READY;
                break;
                }
        break;
```

第十三章、代码覆盖率模块的实现代码和分析(暂无)

第十四章、gdb 远程调试代理模块的实现代码和分析

14.1 介绍

gdb 可以调试本地的程序，也可以通过网络协议对其他机器的程序进行调试。gdb 和远端的客户端使用 RDI(remote debug interface)进行通信，我们一般把远端的客户端称为 gdbserver，gdbserver 来提供被调试程序的内存，寄存器等信息。gdb 获得这些信息后，然后进行分析并给出输出。我们一般都是在本地运行 gdb，而在目标机或者另外一台被调试机器上运行 gdbserver。
SkyEye 为了使用 gdb 来调试运行在 SkyEye 上的程序，在内部实现了 gdb 的 RDI 协议，源码位于 utils/debugger/目录。

14.2 代码分析

SkyEye

第十五章、Sparc 处理器的模拟实现

15.1 介绍

sparc 处理器模拟的相关代码位于 arch/sparc 目录下：

- common 目录下的代码为 sparc 处理器核的代码，包括中断，和以及实现 skyeye 的相关接口。
- mach 目录下的代码为外设相关的代码，只和 leon2 相关，而和 sparc 体系结构无关。
- Instructions 目录下的代码是专门解析 sparc 指令集的代码。

15.2 与 *skyEye* 核心模块的接口

在 skyEye 中，实现每一种体系结构的仿真都需要实现 arch_config_t 的数据结构，并在模块初始化的时候注册自己的 arch_config_t 的变量。对于我们的 sparc 仿真模块，其注册函数在 common/sparc_arch_interface.c 文件中的 init_sparc_arch 函数。其代码如下：

```
399 void init_sparc_arch()
400 {
401     static arch_config_t sparc_arch;
402
403         sparc_arch.arch_name = "sparc";
404         sparc_arch.init = sparc_init_state;
405         sparc_arch.reset = sparc_reset_state;
406         sparc_arch.set_pc = sparc_set_pc;
407         sparc_arch.get_pc = sparc_get_pc;
408         sparc_arch.step_once = sparc_step_once;
409         sparc_arch.ICE_write_byte = sparc_ICE_write_byte;
410         sparc_arch.ICE_read_byte = sparc_ICE_read_byte;
411         sparc_arch.parse_cpu = sparc_parse_cpu;
412         sparc_arch.parse_mach = sparc_parse_mach;
413     //sparc_arch.parse_mem = sparc_parse_mem;
414         sparc_arch.get_regval_by_id = sparc_get_regval_by_id;
415         sparc_arch.get_regname_by_id = sparc_get_regname_by_id;
416         sparc_arch.get_step = sparc_get_step;
417
418         register_arch (&sparc_arch);
419 }
```

在 401 行，我们定义了一个静态变量 sparc_arch 做为我们的 sparc 仿真实现的 arch_config_t 的结构体。然后在 403 到 416 行进行填充。并在 418 行注册到 SkyEye 的核心模块中。

15.3 内部运行流程

15.3.1 处理器状态初始化

在 skyEye 运行 start 命令的时候，在前面 init_sparc_arch 的函数中注册的 sparc_init_state 函数会被调用，来初始化处理器的状态，其代码如下：

```
75        status = init_sparc_iu();
76
77        /*  RESET the statistics    */
78        STAT_reset();
```

在 75 行调用 init_sparc_iu 函数，然后调用 sparc_register_iu，最后调用 iu_init_state 函数。在 iu_init_state 里面做了大部分的初始化工作，代码如下：

```
207     state->irq_pending = 0;
208     state->cycle_counter = 0;
209     state->regwptr[0] = &state->global[0];
210     state->regwptr[1] = state->regbase + (0 * 16);  /*  we start in the cwp = 0 */
211     PSRREG = 0x0;
212     WIMREG = 0x2;
213
214     /*
215      *  Put the processor in supervisor mode, S=1 PS=1
216      *  Enable the traps
217      *  Enable the floating point unit
218      */
219     set_bit(PSRREG, PSR_S);
220     set_bit(PSRREG, PSR_PS);
221     set_bit(PSRREG, PSR_ET);
222     set_bit(PSRREG, PSR_EF);
223     for( i = 0; i < FORMAT_TYPES; ++i)
224         i_count[i] = 0;
```

207 到 224 行都是根据 sparc 处理器的硬件规范来对相应的寄存器做初始化。

```
250     /*  Register the trap handling  */
251     traps = &trap_handle;
252     /*  initialize the trap handing. The processor state is needed  */
253     traps->init(state);
254
255     iu_isa_register();
```

250 行到 253 行是对中断陷阱的初始化。而 iu_isa_register 是对指令集进行初始化，为后面的指令解码做准备。

15.3.2 指令执行

其中 sparc_step_once 为单步执行函数，会被 SkyEye 循环调用来执行每一条指令。代码如下：

```
130 void sparc_step_once ()
131 {
132     int n_cycles;
133
134     /*  Execute the next instruction    */
135     /*  FIXME:  n_cycles must be used to update the system clock    */
136     n_cycles = iu->iu_cycle_step();
137
138     /*  Check for insterrupts   */
139     iu->iu_trap_cycle();
140         steps++;
141     /*  Execute the I/O cycle   */
142     //skyeye_config.mach->mach_io_do_cycle ((void*)&sparc_state);
143         skyeye_config_t* config = get_current_config();
144         config->mach->mach_io_do_cycle((void*)&sparc_state);
145
146 }
```

其中 136 行的 iu 为整数单元来执行整数相关的指令。它的实现是在 iu.c 文件中的 iu_cycle_step 函数。代码片段如下：

```
297     pc = iu_get_pc();
298
299     /*  Read the memory where the next instruction is   */
300     sparc_memory_read_word32(&instr, pc);
310
311     if( (pi = iu_get_instr(instr)) == NULL )
```

```
312     {
313         DBG("Instruction not implemented at PC=0x%x\n", pc);
314         skyeye_exit(1);
315     }
316
317     cycles = (*pi->execute)(&sparc_state);
```

其中 297 到 300 行为取指，从内存中读取地址为 pc 值的 sparc 指令。311 行为获得对应指令的结构体，这个结构体包含了指令的一些信息，这个步骤我们可以理解为解码。317 行为指令执行。

15.3.3 中断检测

外设的中断检测在每条指令执行的时候都会被调用，在 sparc_step_once 的相关代码，如下所示：

```
143         skyeye_config_t* config = get_current_config();
144         config->mach->mach_io_do_cycle((void*)&sparc_state);
```

在 144 行调用相应模拟机器的 mach_io_do_cycle 来进行中断检测。其相应实现在 arch/sparc/mach/mach_leon2_io.c 文件中，摘出相关代码如下

```
275 void leon2_io_do_cycle(void * state)
276 {
277     /*  FIXME: it is not used so far    */
278     sparc_state_t *pstate = state;
279
280     //  UART cycle
281     leon2_uart_cycle(state);
282
283      // TIMER cycling
284     leon2_timer_core_cycle(state);
```

```
285
286     /*---------------------------------------------------------------------------
287      *  Handle all the possible IRQs during the I/Os cycles.
288      *---------------------------------------------------------------------------*/
289     handle_irq ();
290 }
```

上面代码的 281 行为串口中断检测，284 行为时钟中断检测。然后在 289 行的 handle_irq 中对中断控制器进行操作，对中断状态进行更新。