sim-profile 分析调研报告

2014211304 班

D 组

史文翰 2014211218

林宇辰 2014211223

王剑督 2014211228

崔嘉伟 2014211233

杨　莹 2014211238

徐丹雅 2014211243

郝绍明 2014210123

分工情况

史文翰 2014211218 :负责解析 sim-profile.c 638-end 行, 画流程图, 整合文档

林宇辰 2014211223：负责解析 sim-profile.c 50-154 行

王剑督 2014211228：负责解析 sim.h 行

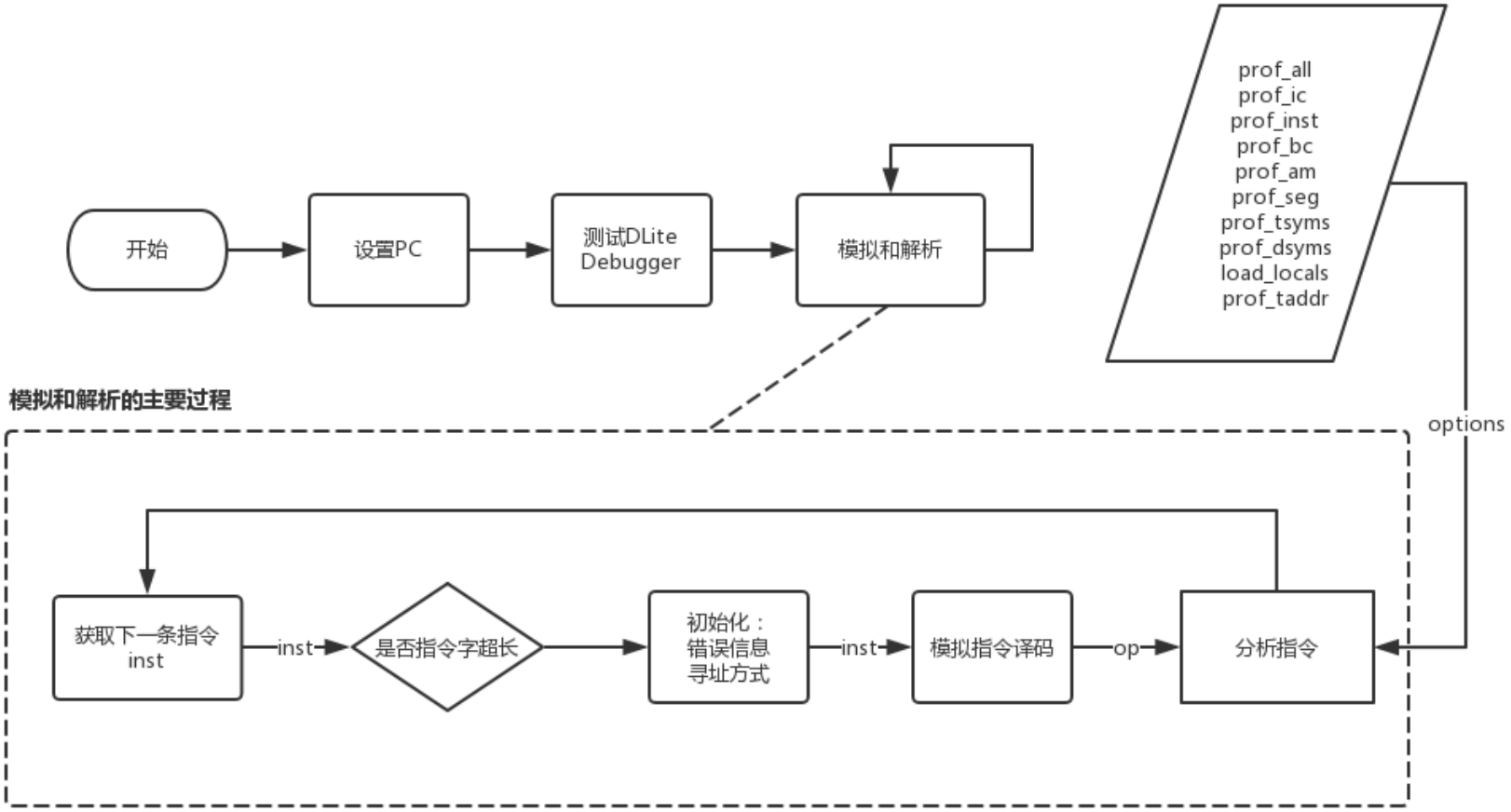
崔嘉伟 2014211233：负责解析 sim-profile.c 248-294 行

杨　莹 2014211238：负责解析 sim-profile.c 154-248 行

徐丹雅 2014211243：负责解析 sim-profile.c 491-638 行

郝绍明 2014210123：负责解析 sim-profile.c 294-491 行

一、 sim-profile 流程图



二、 主函数 main（638-end 行）

/* start simulation, program loaded, processor precise state initialized */

此函数用于开始模拟，载入程序，包括初始化处理程序的状态。
此函数是 simprofile 模拟过程的核心体现，主要是面向过程的对解析指令的需求的判断，并执行一些高度封装的过程和代码来记录信息并控制程序流。
简单来说，simprofile 不断作如下的循环：
读指令》解析指令（按照需求)》统计解析信息》读下一条指令

```
void
sim_main(void)
{
  int i;
  md_inst_t inst;
  register md_addr_t addr;
  register int is_write;
  enum md_opcode op;
  unsigned int flags;
  enum md_fault_type fault;

  fprintf(stderr, "sim: ** starting functional simulation **\n");

  /* set up initial default next PC */
```

设置下一个 PC 值的默认初始值，即 regs_NPC 的值，由 regs_PC 与 md_inst_t 值相加得到。

```
  regs.regs_NPC = regs.regs_PC + sizeof(md_inst_t);

  /* check for DLite debugger entry condition */
```

检测 Dlite 调试器的启动条件。

```
  if (dlite_check_break(regs.regs_PC, /* no access */0, /* addr */0, 0, 0))
    dlite_main(regs.regs_PC - sizeof(md_inst_t), regs.regs_PC,
               sim_num_insn, &regs, mem);
```

开始 sim-profile 模拟过程，是一个无限循环结构，在没有完成模拟或特殊指令的情况下，sim-profile 将一直运行。

```
  while (TRUE)
```

```
    {
      /* maintain $r0 semantics */
      regs.regs_R[MD_REG_ZERO] = 0;
#ifdef TARGET_ALPHA
      regs.regs_F.d[MD_REG_ZERO] = 0.0;
#endif /* TARGET_ALPHA */

      /* get the next instruction to execute */
```

下一条指令以执行，其中 MD_FETCH_INST 是一个宏，其作用是从 mem 中读取同一个 PC 所指向的指令字。

```
      /* fetch an instruction */
      #define MD_FETCH_INST(INST, MEM, PC)                    \
      { (INST) = MEM_READ_WORD((MEM), (PC)); }
```

此时，inst 变量存贮了一条指令，观察 inst 变量的定义

```
      /* instruction formats */
      typedef word_t md_inst_t;
```

可以看出字长变量 md_inst_t 代表了一条指令的格式。

```
      MD_FETCH_INST(inst, mem, regs.regs_PC);
```

此处 verbose 是输入的字节超出范围的一个 flag 标志，下面程序段做 verbose 判断。

如果出现超字长的问题，则将错误信息通过错误流 stderr 输出，并打印这个错误信息。

错误信息包括：

sim_num_insn：程序执行指针

regs_PC：当前的 PC 值

```
      if (verbose)
    {
      myfprintf(stderr, "%10n @ 0x%08p: ", sim_num_insn, regs.regs_PC);
      md_print_insn(inst, regs.regs_PC, stderr);
      fprintf(stderr, "\n");
      /* fflush(stderr); */
    }

      /* keep an instruction count */
```

如果没有出现 verbose 错误，则继续维护程序执行指针这一变量，显然应该做自增操作。

```
      sim_num_insn++;

      /* set default reference address and access mode */
```

设置默认的参考地址为 0，并设置寻址方式。

```
addr = 0; is_write = FALSE;

/* set default fault - none */
```

设置错误信息，fault 是 md_fault_type 变量，而 md_fault_type 是错误信息的种类，是一个枚举变量。

md_fault_none 是枚举变量的中的一个成员，表示为没有错误。

即当前程序未发现错误。

```
fault = md_fault_none;

/* decode the instruction */
```

进行指令解码：

进行 inst 即指令向 md_opcode 的映射，即提取指令的操作码，利用如下的宏

```
#define MD_SET_OPCODE(OP, INST)                                 \
{ OP = md_mask2op[MD_TOP_OP(INST)];                             \
while (md_opmask[OP])                                   \
OP = md_mask2op[((INST >> md_opshift[OP]) & md_opmask[OP])  \
        + md_opoffset[OP]]; }
```

这段宏表示，利用 mask 掩码的过程，将 inst 中的操作码字段提取出来放入变量 op 中。

接下来是一段针对 op 的 switch 过程，即根据提取出来的 op 字段来决定究竟应该如何执行这条指令。

```
MD_SET_OPCODE(op, inst);

/* execute the instruction */
```

指令的执行过程

```
switch (op)
  {
#define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3)        \
  case OP:                                      \
    SYMCAT(OP,_IMPL);                           \
    break;
#define DEFLINK(OP,MSK,NAME,MASK,SHIFT)                     \
    case OP:                                    \
    panic("attempted to execute a linking opcode");
#define CONNECT(OP)
```

```
#define DECLARE_FAULT(FAULT)                                    \
        { fault = (FAULT); break; }
#include "machine.def"
      default:
        panic("attempted to execute a bogus opcode");
```

panic 表示程序进入了一个未知的危险状态，即 switch 的 default 分支，模拟器发出“尝试执行一个不存在的操作码”

```
        }

        if (MD_OP_FLAGS(op) & F_MEM)
      {
        sim_num_refs++;
        if (MD_OP_FLAGS(op) & F_STORE)
          is_write = TRUE;
      }

        /*
         * profile this instruction
         */
```

下面进行指令的解析，首先进行 md_opcode -> opcode 的映射。

```
        flags = MD_OP_FLAGS(op);
```

如果设置了 prof_ic（即"enable instruction class profiling"，对应于允许指令种类解析），则进入下面的环节

```
        if (prof_ic)
      {
        enum inst_class_t ic;

        /* compute instruction class */
```

计算指令的种类，如 LOAD，STORE，COND 类指令。

```
        if (flags & F_LOAD)
          ic = ic_load;
        else if (flags & F_STORE)
          ic = ic_store;
        else if (flags & F_UNCOND)
          ic = ic_uncond;
        else if (flags & F_COND)
```

```
      ic = ic_cond;
    else if (flags & F_ICOMP)
      ic = ic_icomp;
    else if (flags & F_FCOMP)
      ic = ic_fcomp;
    else if (flags & F_TRAP)
      ic = ic_trap;
    else
      panic("instruction has no class");

    /* update instruction class profile */
```

更新指令种类解析信息，统计这个信息之后加入到之前的统计信息中

```
    stat_add_sample(ic_prof, (int)ic);
}
```

同理，判断是否开启 prof_inst 功能，指令解析

```
    if (prof_inst)
{
    /* update instruction profile */
    stat_add_sample(inst_prof, (int)op - /* skip NA */1);
}
```

同理，判断是否开启 prof_bc 功能，指令的分支解析

```
    if (prof_bc)
{
    enum branch_class_t bc;

    /* compute instruction class */
```

计算分支指令的种类，与上面的计算指令本身种类过程类似
分支指令包括如 call 调用，uncond 无条件转移，cond 条件转移等

```
    if (flags & F_CTRL)
      {
        if ((flags & (F_CALL|F_DIRJMP)) == (F_CALL|F_DIRJMP))
      bc = bc_call_dir;
        else if ((flags & (F_CALL|F_INDIRJMP)) == (F_CALL|F_INDIRJMP))
      bc = bc_call_indir;
        else if ((flags & (F_UNCOND|F_DIRJMP)) == (F_UNCOND|F_DIRJMP))
      bc = bc_uncond_dir;
```

```
        else if ((flags & (F_UNCOND|F_INDIRJMP))== (F_UNCOND|F_INDIRJMP))
      bc = bc_uncond_indir;
        else if ((flags & (F_COND|F_DIRJMP)) == (F_COND|F_DIRJMP))
      bc = bc_cond_dir;
        else if ((flags & (F_COND|F_INDIRJMP)) == (F_COND|F_INDIRJMP))
      bc = bc_cond_indir;
        else
      panic("branch has no class");

        /* update instruction class profile */
```

更新统计信息，即对上述分析结果做总结并记录

```
        stat_add_sample(bc_prof, (int)bc);
      }
}
```

am 代表 address mode 即寻址方式的解析，如下过程是解析指令的寻址方式

```
  if (prof_am)
{
  enum md_amode_type am;

  /* update addressing mode pre-probe FSM */
  MD_AMODE_PREPROBE(op, fsm);

  /* compute addressing mode */
```

计算指令的寻址模式

```
  if (flags & F_MEM)
    {
      /* compute addressing mode */
      MD_AMODE_PROBE(am, op, fsm);

      /* update the addressing mode profile */
```

将计算出来的指令寻址模式载入变量 am 中，再更新指令解析的信息
这条信息加入到统计信息中

```
      stat_add_sample(am_prof, (int)am);

      /* addressing mode pre-probe FSM, after all loads and stores */
      MD_AMODE_POSTPROBE(fsm);
```

```
    }
}
```

prof_seg 代表解析 load 和 write 指令的寻址段

```
  if (prof_seg)
{
  if (flags & F_MEM)
    {
      /* update instruction profile */
      stat_add_sample(seg_prof, (int)bind_to_seg(addr));
    }
}
```

接下里是文本符号信息的解析，同样式一个可选的过程

```
  if (prof_tsyms)
{
  int tindex;

  /* attempt to bind inst address to a text segment symbol */
```

下面程序做了这样的尝试：
将指令的物理地址与符号地址绑定
sym_bind_addr 完成了这一过程，将绑定结果载入到 tindex 临时变量中
显然，这个值应该为正，但不能超过界限 sym_ntextsyms
因此在下面的程序段中判断是否满足这个条件

```
  sym_bind_addr(regs.regs_PC, &tindex, /* !exact */FALSE, sdb_text);

  if (tindex >= 0)
    {
      if (tindex > sym_ntextsyms)
    panic("bogus text symbol index");

      stat_add_sample(tsym_prof, tindex);
    }
  /* else, could not bind to a symbol */
```

如果不满足最基本的条件，则不能解析这个符号地址

```
}
```

接下来是可选的数据符号的解析过程，与文本符号的解析过程极其类似

```
    if (prof_dsyms)
  {
    int dindex;

    if (flags & F_MEM)
      {
        /* attempt to bind inst address to a text segment symbol */
```

下面过程做了这样的尝试：

将指令物理地址绑定到符号上，只不过此时不再是 text symbol，而是 data symbol

```
        sym_bind_addr(addr, &dindex, /* !exact */FALSE, sdb_data);
        if (dindex >= 0)
      {
        if (dindex > sym_ndatasyms)
          panic("bogus data symbol index");

        stat_add_sample(dsym_prof, dindex);
      }
        /* else, could not bind to a symbol */
```

如果不满足上述条件，则无法完成地址绑定

```
      }
  }
```

接下来是可选的，对文本地址的解析

```
    if (prof_taddr)
  {
    /* add regs_PC exec event to text address profile */
```

只需将当前的 PC 值加入到文本地址解析结果中

```
    stat_add_sample(taddr_prof, regs.regs_PC);
  }

    /* update any stats tracked by PC */
```

至此，所有的可选解析过程已经结束，追踪 PC 值以更新所有的统计状态信息

```
    for (i=0; i<pcstat_nelt; i++)
```

```
{
  counter_t newval;
  int delta;

  /* check if any tracked stats changed */
```

检查是否有跟踪的状态信息改变，STATVAL 是一个更新状态栈的宏
返回的 newval 是更新之后的状态栈的栈顶
delta 反映了状态栈的变化

```
  newval = STATVAL(pcstat_stats[i]);
  delta = newval - pcstat_lastvals[i];
```

此处，如果 delta 值非 0 表示有状态发生了改变，要做相应的信息记录处理

```
  if (delta != 0)
    {
      stat_add_samples(pcstat_sdists[i], regs.regs_PC, delta);
      pcstat_lastvals[i] = newval;
    }

}

  /* check for DLite debugger entry condition */
```

检查 DLite 调试器的入口条件

```
  if (dlite_check_break(regs.regs_NPC,
                        is_write ? ACCESS_WRITE : ACCESS_READ,
                        addr, sim_num_insn, sim_num_insn))
dlite_main(regs.regs_PC, regs.regs_NPC, sim_num_insn, &regs, mem);
```

此时对这条指令 inst 的解析过程（profile 过程）已全部执行完毕
控制流读入下一条指令，并重复上述的过程

```
  /* go to the next instruction */
```

读下一条指令

```
  regs.regs_PC = regs.regs_NPC;
  regs.regs_NPC += sizeof(md_inst_t);

  /* finish early? */
```

如果超过了最大的指令限制，则解析过程提前结束

```
        if (max_insts && sim_num_insn >= max_insts)
      return;
      }
}
```

三、 50-154 行

```
#include <stdio.h>
```

标准化输入输出的引用

```
#include <stdlib.h>
```

常用系统函数的引用

```
#include <math.h>
```

数学库函数的引用

```
#include "host.h"
```

包含了依赖于本地的所有定义和接口设置

```
#include "misc.h"
```

包含了(miscellaneous)混杂的，其他的接口设置

```
#include "machine.h"
```

包含了有关 alpha ISA 的定义

```
#include "regs.h"
```

包含了已架构的寄存器状态接口

```
#include "memory.h"
```

包含了平面内存模式空间接口

```
#include "loader.h"
```

载入专用配置文件

#include "syscall.h"

包含了有关系统调用的接口

#include "stats.h"

包含了封装好的程序包接口

#include "sim.h"

包含了模拟器总线接口

#include "dlite.h"

包含了有关 Dlite 调试器的接口

#include "symbol.h"

包含了所有标记的接口

#include "options.h"

对其他自定义配置文件的引用。

/*
This file implements a functional simulator with profiling support. Run
with the `-h' flag to see profiling options available.
*/

在各种配置文件的支持下，本.c 文件实现了一个多功能的模拟器。

以-h 方式运行本文件便可以查看配置文件中的各种设置。

-h 方式通常用于修改结果的表现形式，各个配置文件中的设置不需要转化为默认的 byte 为单位，保持原来的单位展现给用户，使得每个配置文件的设置可见。

/* simulated registers */

声明虚拟寄存器结构。

static struct regs_t regs;

在 regs.h 配置文件中有该变量的声明。

该虚拟寄存器中包括整形寄存器、浮点型寄存器、控制寄存器、程序计数器、以及指向下一个 PC 的指针。

/* simulated memory */

虚拟内存的声明。

static struct mem_t *mem = NULL;

在 memory.h 配置文件中有该变量的声明。

该结构中包含了内存空间的名称、反置页表、已分配的页数、缺页数等。
初始化该指针为空。

/* track number of refs */

设置全局变量 sim_num_refs，用来为 refs 计数。

static counter_t sim_num_refs = 0;

该数据类型 counter_t 为 signed long long，在 host.h 中有相关声明。

/* maximum number of inst's to execute */

声明 inst 初始化操作数的上限。

static unsigned int max_insts;

操作数为整形静态全局变量。

/* profiling options */

指令后缀配置文件的初始化设置。

```
static int prof_all /* = FALSE */;
static int prof_ic /* = FALSE */;
static int prof_inst /* = FALSE */;
static int prof_bc /* = FALSE */;
static int prof_am /* = FALSE */;
static int prof_seg /* = FALSE */;
```

```
static int prof_tsyms /* = FALSE */;
static int prof_dsyms /* = FALSE */;
static int load_locals /* = FALSE */;
static int prof_taddr /* = FALSE */;
```

用户尚未设置的项被全部置为 FALSE。

```
/* text-based stat profiles */
```

声明基于文本的静态变量配置。

```
#define MAX_PCSTAT_VARS 8
```

为 pcstat_var 设置最大值为 8。

它为紧接下来要进行的 string_list 的初始化作铺垫。

```
static int pcstat_nelt = 0;
```

pcstat_nelt 记录了程序计数器的状态数，并在后续代码中参与 for 循环，用来遍历每一个程序计数器的状态。

```
static char *pcstat_vars[MAX_PCSTAT_VARS];
```

程序计数器静态变量的域的大小也设置为 8。

```
/* register simulator-specific options */
```

寄存器模拟器环境配置。

```
void
sim_reg_options(struct opt_odb_t *odb)
{
```

在 sim.h 中定义了 struct opt_odb_t，结构中包含了一个指针*odb，代表了 option database，即数据库的环境设定。

```
  opt_reg_header(odb,
"sim-profile: This simulator implements a functional simulator with\n"
"profiling support.   Run with the `-h' flag to see profiling options\n"
"available.\n");
```

同.c 文件中原有的注释意义相同，上文的注释面向程序员，此处面向用户。

它将两个对用户的友好提示初始化放入寄存器的首部，从而用户可以得到如何使用本产品的提示。

```
/* instruction limit */
```

对指令的限制做了具体的规定。

设定了如下十一个后缀：

```
opt_reg_uint(odb, "-max:inst", "maximum number of inst's to execute",
        &max_insts, /* default */0,
        /* print */TRUE, /* format */NULL);
```

该语句设定初始化操作的数的最大值。

```
opt_reg_flag(odb, "-all", "enable all profile options",
        &prof_all, /* default */FALSE, /* print */TRUE, NULL);
```

该语句设定允许所有配置文件的设置。

```
opt_reg_flag(odb, "-iclass", "enable instruction class profiling",
        &prof_ic, /* default */FALSE, /* print */TRUE, NULL);
```

该语句允许指令类的配置文件生效。

```
opt_reg_flag(odb, "-iprof", "enable instruction profiling",
        &prof_inst, /* default */FALSE, /* print */TRUE, NULL);
```

该语句允许指令的配置文件生效。

```
opt_reg_flag(odb, "-brprof", "enable branch instruction profiling",
        &prof_bc, /* default */FALSE, /* print */TRUE, NULL);
```

该语句允许分支指令的配置文件生效。

```
opt_reg_flag(odb, "-amprof", "enable address mode profiling",
        &prof_am, /* default */FALSE, /* print */TRUE, NULL);
```

该语句允许寻址方式的配置文件生效。

```
opt_reg_flag(odb, "-segprof", "enable load/store address segment profiling",
        &prof_seg, /* default */FALSE, /* print */TRUE, NULL);
```

该语句允许存取地址段的配置文件生效。

```
  opt_reg_flag(odb, "-tsymprof", "enable text symbol profiling",
               &prof_tsyms, /* default */FALSE, /* print */TRUE, NULL);
```

该语句允许文本标志的配置文件生效。

```
  opt_reg_flag(odb, "-taddrprof", "enable text address profiling",
               &prof_taddr, /* default */FALSE, /* print */TRUE, NULL);
```

该语句允许文本地址的配置文件生效。

```
  opt_reg_flag(odb, "-dsymprof", "enable data symbol profiling",
               &prof_dsyms, /* default */FALSE, /* print */TRUE, NULL);
```

该语句允许数据标志的配置文件生效。

```
  opt_reg_flag(odb, "-internal",
               "include compiler-internal symbols during symbol profiling",
               &load_locals, /* default */FALSE, /* print */TRUE, NULL);
```

该语句使标志配置文件编译时涵盖了文件内部标志。

```
  opt_reg_string_list(odb, "-pcstat",
                      "profile stat(s) against text addr's (mult uses ok)",
                      pcstat_vars, MAX_PCSTAT_VARS, &pcstat_nelt, NULL,
                      /* !print */FALSE, /* format */NULL, /* accrue */TRUE);
}
```

这十一个指令后缀分别为
-max:inst、-all-iclass、-iprof、-brprof、-amprof、-segprof、
-tsymprof、-taddrprof、-dsymprof、-internal、-pcstat。

他们分别表征了指令在运行时需要调用配置文件的状况。
缺省所有设置都为 false，在屏幕上打印输出 true，并将指针域置为 NULL。
至此，有关寄存器单元的，寄存器标志位的，寄存器中字符串清单的初始化设定完成。
在程序运行的开始，将上述后缀添加到命令后，相关环境变量便可以成功手动配置。

四、 154-248 行

```
/* check simulator-specific option values */
```

此函数用于检查 simulator-specific 可选项的值
struct opt_odb_t 为选项数据库

```
void
sim_check_options(struct opt_odb_t *odb, int argc, char **argv)
{
  if (prof_all)
    {
      /* enable all options */
```

当 prof_all 值为真时，使能所有性能选项

```
      prof_ic = TRUE;
      prof_inst = TRUE;
      prof_bc = TRUE;
      prof_am = TRUE;
      prof_seg = TRUE;
      prof_tsyms = TRUE;
      prof_dsyms = TRUE;
      prof_taddr = TRUE;
    }
}

/* instruction classes */
```

指令类

```
enum inst_class_t {
  ic_load,          /* load inst */
```

加载指令

```
  ic_store,         /* store inst */
```

存储指令

```
  ic_uncond,        /* uncond branch */
```

无条件分支

```
  ic_cond,          /* cond branch */
```

条件分支

```
  ic_icomp,         /* all other integer computation */
```

所有其他整型计算

```
  ic_fcomp,         /* all floating point computation */
```

所有浮点计算

```
  ic_trap,          /* system call */
```

系统调用

```
  ic_NUM
};

/* instruction class strings */
```

指令类字符串

```
static char *inst_class_str[ic_NUM] = {
  "load",       /* load inst */
```

加载指令

```
  "store",          /* store inst */
```

存储指令

```
  "uncond branch",    /* uncond branch */
```

无条件分支

```
  "cond branch",  /* cond branch */
```

条件分支

```
  "int computation",    /* all other integer computation */
```

所有其他整型计算

```
  "fp computation",     /* all floating point computation */
```

所有浮点计算

```
  "trap"        /* system call */
```

系统调用

```
};
/* instruction class profile */
```

指令类属性
struct stat_stat_t 中包括以下成员
数据库表中指向下一个状态的指针
状态名
状态描述
状态输出格式
状态类（包括整型状态变量和无符号型状态变量的定义以及其初始化）

```
static struct stat_stat_t *ic_prof = NULL;

/* instruction description strings */
```

指令描述字符串

```
static char *inst_str[OP_MAX];

/* instruction profile */
```

指令属性

```
static struct stat_stat_t *inst_prof = NULL;

/* branch class profile */
```

分支类属性

```
enum branch_class_t {
  bc_uncond_dir,  /* direct unconditional branch */
```

直接无条件分支

```
  bc_cond_dir,          /* direct conditional branch */
```

直接条件分支

```
  bc_call_dir,       /* direct functional call */
```

直接函数调用

```
  bc_uncond_indir,      /* indirect unconditional branch */
```

间接无条件分支

```
    bc_cond_indir,   /* indirect conditional branch */
```

间接条件分支

```
    bc_call_indir,      /* indirect function call */
```

间接函数调用

```
    bc_NUM
};

/* branch class description strings */
```

分支类描述字符串

```
static char *branch_class_str[bc_NUM] = {
    "uncond direct",/* direct unconditional branch */
```

直接无条件分支

```
    "cond direct",    /* direct conditional branch */
```

直接条件分支

```
    "call direct",      /* direct functional call */
```

直接函数调用

```
    "uncond indirect",    /* indirect unconditional branch */
```

间接无条件分支

```
    "cond indirect", /* indirect conditional branch */
```

间接条件分支

```
    "call indirect"     /* indirect function call */
```

间接函数调用

```
};
```

```
/* branch profile */
```

分支属性

```
static struct stat_stat_t *bc_prof = NULL;

/* addressing mode profile */
```

地址模式属性

```
static struct stat_stat_t *am_prof = NULL;

/* address segments */
```

地址段

```
enum addr_seg_t {
  seg_data,       /* data segment */
```

数据段

```
  seg_heap,       /* heap segment */
```

堆段

```
  seg_stack,      /* stack segment */
```

栈段

```
  seg_text,       /* text segment */
```

文本段

```
  seg_NUM
};

/* address segment strings */
```

地址段字符串

```
static char *addr_seg_str[seg_NUM] = {
  "data segment", /* data segment */
```

数据段

"heap segment", /* heap segment */

堆段

"stack segment", /* stack segment */

栈段

"text segment", /* text segment */

文本段

};

五、 248-294 行

```
/* address segment profile */
```

地址段概述。

```
static struct stat_stat_t *seg_prof = NULL;

/* bind ADDR to the segment it references */
```

将 ADDR 绑定到它引用的段。

```
static enum addr_seg_t

/* segment referenced by ADDR */
```

声明 ADDR 引用的段。

```
bind_to_seg(md_addr_t addr)

/* address to bind to a segment */
```

将地址绑定到一个段。

```
{
  if (ld_data_base <= addr && addr < (ld_data_base + ld_data_size))
```

```
    return seg_data;
```

如果地址 address 匹配在 data 段上，返回 seg_data。

```
  else if ((ld_data_base + ld_data_size) <= addr && addr <
ld_brk_point)
```

```
    return seg_heap;
```

如果地址 address 匹配在 data 段与 brk_point 之间，返回 seg_heap。

```
  /* FIXME: ouch! */
```

此处出现错误，需要修正。

```
  else if ((ld_stack_base - (16*1024*1024)) <= addr && addr <
ld_stack_base)
```

```
    return seg_stack;
```

如果地址 address 匹配在栈地址段上，返回 seg_stack。

```
  else if (ld_text_base <= addr && addr < (ld_text_base +
ld_text_size))
```

```
    return seg_text;
```

如果地址 address 匹配在 text 段上，返回 seg_text。

```
  else
    panic("cannot bind address to segment");
```

不能把地址绑定到段。

```
}

/* text symbol profile */
```

文本标志概述。

```
static struct stat_stat_t *tsym_prof = NULL;
static char **tsym_names = NULL;
```

```
/* data symbol profile */
```

数据标志概述。

```
static struct stat_stat_t *dsym_prof = NULL;
static char **dsym_names = NULL;

/* text address profile */
```

文本地址概述。

```
static struct stat_stat_t *taddr_prof = NULL;

/* text-based stat profiles */
```

基于文本的统计的概述。

```
static struct stat_stat_t *pcstat_stats[MAX_PCSTAT_VARS];
static counter_t pcstat_lastvals[MAX_PCSTAT_VARS];
static struct stat_stat_t *pcstat_sdists[MAX_PCSTAT_VARS];

/* wedge all stat values into a counter_t */
```

将所有统计数据放入一个计数器 counter_t。

```
#define STATVAL(STAT)                       \
```

定义 STATVAL(STAT)。

```
  ((STAT)->sc == sc_int                     \
   ? (counter_t)*((STAT)->variant.for_int.var)           \
   : ((STAT)->sc == sc_uint                 \
      ? (counter_t)*((STAT)->variant.for_uint.var)       \
      : ((STAT)->sc == sc_counter               \
     ? *((STAT)->variant.for_counter.var)             \
     : (panic("bad stat class"), 0))))
```

六、 294-491 行

/* register simulator-specific statistics */

寄存器模拟器统计数据配置

```
void sim_reg_stats(struct stat_sdb_t *sdb)
{
  int i;

  stat_reg_counter(sdb, "sim_num_insn",       //database，name，var_dep,
                   "total number of instructions executed",
                   &sim_num_insn, sim_num_insn, NULL);
  stat_reg_counter(sdb, "sim_num_refs",
                   "total number of loads and stores executed",
                   &sim_num_refs, 0, NULL);
  stat_reg_int(sdb, "sim_elapsed_time",
                   "total simulation time in seconds",
                   &sim_elapsed_time, 0, NULL);
```

记录了一个 int 类型的统计数据变量

```
  stat_reg_formula(sdb, "sim_inst_rate",
                   "simulation speed (in insts/sec)",
                   "sim_num_insn / sim_elapsed_time", NULL);

  if (prof_ic)
    {
      /* instruction class profile */
```

如果 prof_ic=TRUE，那么开始进行 instruction class 的配置

```
      ic_prof = stat_reg_dist(sdb, "sim_inst_class_prof",
                              "instruction class profile",
                              /* initial value */0,
```

初始值=0

```
                              /* array size */ic_NUM,
```

数组大小=ic_NUM

```
                              /* bucket size */1,
```

bucket 容器大小=1

```
                              /* print format */(PF_COUNT|PF_PDF),
```

输出格式定义

```
                    /* format */NULL,
                    /* index map */inst_class_str,
```

索引图

```
                    /* print fn */NULL);
  }
```

```
if (prof_inst)
```

如果 prof_inst=TRUE，则进行指令的配置

```
  {
    int i;
    char buf[512];

    /* conjure up appropriate instruction description strings */
```

选用合法的指令来描述当前字符串

```
    for (i=0; i < /* skip NA */OP_MAX-1; i++)
  {
    sprintf(buf, "%-8s %-6s", md_op2name[i+1], md_op2format[i+1]);
    inst_str[i] = mystrdup(buf);
  }

    /* instruction profile */
```

进行 instruction 的配置

```
    inst_prof = stat_reg_dist(sdb, "sim_inst_prof",
                    "instruction profile",
                    /* initial value */0,
```

初始值=0

```
                    /* array size */ /* skip NA */OP_MAX-1,
```

数组大小=OP_MAX-1

```
                    /* bucket size */1,
```

bucket 容器大小=1

```
                    /* print format */(PF_COUNT|PF_PDF),

                    输出格式

                    /* format */NULL,
                    /* index map */inst_str,

                    索引映射

                    /* print fn */NULL);
  }

if (prof_bc)
  {
    /* instruction branch profile */

    进行 branch instruction（转移指令）的配置

    bc_prof = stat_reg_dist(sdb, "sim_branch_prof",
                    "branch instruction profile",
                    /* initial value */0,
                    /* array size */bc_NUM,
                    /* bucket size */1,
                    /* print format */(PF_COUNT|PF_PDF),
                    /* format */NULL,
                    /* index map */branch_class_str,
                    /* print fn */NULL);

    bc_prof 是通过调用 stat_reg_dist 来创建一个阵列分布（array distribution）

  }

if (prof_am)
  {
    /* instruction branch profile */

    进行 addressing mode（寻址模式）的配置

    am_prof = stat_reg_dist(sdb, "sim_addr_mode_prof",
                    "addressing mode profile",
                    /* initial value */0,
                    /* array size */md_amode_NUM,
                    /* bucket size */1,
                    /* print format */(PF_COUNT|PF_PDF),
```

```
                        /* format */NULL,
                        /* index map */md_amode_str,
                        /* print fn */NULL);
    }

  if (prof_seg)
    {
      /* instruction branch profile */

      载入或存储地址段的配置文件

      seg_prof = stat_reg_dist(sdb, "sim_addr_seg_prof",
                        "load/store address segment profile",
                        /* initial value */0,
                        /* array size */seg_NUM,
                        /* bucket size */1,
                        /* print format */(PF_COUNT|PF_PDF),
                        /* format */NULL,
                        /* index map */addr_seg_str,
                        /* print fn */NULL);
    }

  if (prof_tsyms && sym_ntextsyms != 0)
    {
      int i;

      /* load program symbols */

      载入程序符号

      sym_loadsyms(ld_prog_fname, load_locals);

      /* conjure up appropriate instruction description strings */
      tsym_names = (char **)calloc(sym_ntextsyms, sizeof(char *));

      for (i=0; i < sym_ntextsyms; i++)
    tsym_names[i] = sym_textsyms[i]->name;

      /* text symbol profile */

     文本符号配置

      tsym_prof = stat_reg_dist(sdb, "sim_text_sym_prof",
                        "text symbol profile",
```

```
                    /* initial value */0,
                    /* array size */sym_ntextsyms,
                    /* bucket size */1,
                    /* print format */(PF_COUNT|PF_PDF),
                    /* format */NULL,
                    /* index map */tsym_names,
                    /* print fn */NULL);
  }

if (prof_dsyms && sym_ndatasyms != 0)
  {
    int i;

    /* load program symbols */
```

载入程序符号

```
    sym_loadsyms(ld_prog_fname, load_locals);

    /* conjure up appropriate instruction description strings */
```

找到合法的指令描述字符串

```
    dsym_names = (char **)calloc(sym_ndatasyms, sizeof(char *));

    for (i=0; i < sym_ndatasyms; i++)
  dsym_names[i] = sym_datasyms[i]->name;

    /* data symbol profile */
```

数据符号属性定义

```
    dsym_prof = stat_reg_dist(sdb, "sim_data_sym_prof",
                    "data symbol profile",
                    /* initial value */0,
```

初始量设为 0

```
                    /* array size */sym_ndatasyms,
```

数组大小变量：sym_ndatasyms

```
                    /* bucket size */1,
```

命名空间大小为 1

/* print format */(PF_COUNT|PF_PDF),

输出格式：PF_COUNT|PF_PDF

/* format */NULL,

格式

/* index map */dsym_names,

索引图变量：dsym_names

/* print fn */NULL);
}

```
if (prof_taddr)
  {
    /* text address profile (sparse profile), NOTE: a dense print format
         is used, its more difficult to read, but the profiles are *much*
     smaller, I've assumed that the profiles are read by programs, at
     least for your sake I hope this is the case!! */
```

文本定位配置，注意：这里使用了密集输出格式，这种格式阅读更困难，但是配置文件更小，我假定（建议）你通过程序来阅读配置文件，至少站在我的立场来为你着想是这样（建议的）。

```
    taddr_prof = stat_reg_sdist(sdb, "sim_text_addr_prof",
                    "text address profile",
                    /* initial value */0,
                    /* print format */(PF_COUNT|PF_PDF),
                    /* format */"0x%p %u %.2f",
                    /* print fn */NULL);
  }

for (i=0; i<pcstat_nelt; i++)
  {
    char buf[512], buf1[512];
    struct stat_stat_t *stat;

    /* track the named statistical variable by text address */
```

通过文本定位来跟踪已命名的统计变量

```
        /* find it... */
        stat = stat_find_stat(sdb, pcstat_vars[i]);
        if (!stat)
      fatal("cannot locate any statistic named `%s'", pcstat_vars[i]);
```

这里是通过文本定位来找到一个名字为 pcstat_vars[i]的统计变量，如果没有找到该字符串，则抛出异常信息

```
        /* stat must be an integral type */
        if (stat->sc != sc_int && stat->sc != sc_uint && stat->sc != sc_counter)
      fatal("`-pcstat' statistical variable `%s' is not an integral type",
            stat->name);
```

数据必须是一个具有完整定义的类型，这里对统计数据各个类变量进行了类型检查

```
        /* register this stat */
```

记录当前的统计数据

```
        pcstat_stats[i] = stat;
        pcstat_lastvals[i] = STATVAL(stat);

        /* declare the sparce text distribution */
```

稀疏文本分配的声明

```
        sprintf(buf, "%s_by_pc", stat->name);
        sprintf(buf1, "%s (by text address)", stat->desc);
        pcstat_sdists[i] = stat_reg_sdist(sdb, buf, buf1,
                          /* initial value */0,
                          /* print format */(PF_COUNT|PF_PDF),
                          /* format */"0x%p %u %.2f",
                          /* print fn */NULL);
      }
    ld_reg_stats(sdb);
    mem_reg_stats(mem, sdb);
}
```

七、 491-638 行

```
/* initialize the simulator */
```

初始化模拟器；
包括寄存器文件和存储器空间；

```
void
sim_init(void)
{
```

初始化 sim_num_refs 值为 0。

```
  sim_num_refs = 0;

  /* allocate and initialize register file */
```

regs_init(&regs)分配并初始化寄存器文件。

```
  regs_init(&regs);

  /* allocate and initialize memory space */
```

分配并初始化存储器空间；
mem_create()函数分配存储器空间给 mem；
mem_init()函数初始化存储器 mem。

```
  mem = mem_create("mem");
  mem_init(mem);
}

/* local machine state accessor */
```

本地机器状态存取器。

```
static char *                   /* err str, NULL for no err */
```

错误返回 str；
没有错误则返回 NULL。

```
profile_mstate_obj(FILE *stream,    /* output stream */
```

输出流 *stream；

```
                char *cmd,          /* optional command string */
```

操作命令字符串*cmd；

```
                struct regs_t *regs,         /* registers to access */
```

用于存取数据的寄存器 regs_t *regs；

```
                struct mem_t *mem)           /* memory to access */
```

用于存取数据的存储器 mem_t *mem。

```
{
  /* just dump intermediate stats */
```

sim_print_stats()丢弃中间缓存数据流 stream。

```
  sim_print_stats(stream);

  /* no error */
```

没有错误，返回 NULL。

```
  return NULL;
}

/* load program into simulated state */
```

将程序加载入模拟器状态。

```
void
sim_load_prog(char *fname,       /* program to load */
```

待加载的程序流*fname；

```
              int argc, char **argv, /* program arguments */
```

程序参量 argc 和 **argv；

```
              char **envp)           /* program environment */
```

程序环境envp。**

```
{
  /* load program text and data, set up environment, memory, and regs */
```

加载程序文本和数据；
设置环境、存储器和寄存器。

```
  ld_load_prog(fname, argc, argv, envp, &regs, mem, TRUE);

  /* initialize the DLite debugger */
```

dlite_init()初始化 Dlit 调试器。

```
  dlite_init(md_reg_obj, dlite_mem_obj, profile_mstate_obj);
}


/* print simulator-specific configuration information */
```

sim_aux_config()函数打印 simulator-specific 配置信息；
输出流*stream；
该函数内容目前为空。

```
void
sim_aux_config(FILE *stream)          /* output stream */
```

输出流*stream。

```
{
  /* nothing currently */
```

目前函数内容具体内容为空。

```
}

/* dump simulator-specific auxiliary simulator statistics */
```

sim_aux_stats()函数丢弃 simulator-specific 的辅助模拟器的缓存数据；
输出流*stream；
同 sim_aux_config()函数相同，该函数内容目前为空。

```
void
sim_aux_stats(FILE *stream)       /* output stream */
```

输出流，
同 void sim_aux_config(FILE *stream)一样，
函数内容为空。

```
{
}
```

```
/* un-initialize simulator-specific state */
```

sim_uninit()非初始化的 simulator - specific 状态。

```
void
sim_uninit(void)
{
  /* nada */
```

函数内容为空。

```
}
```

```
/*
 * configure the execution engine
 */
```

配置执行工具，
包括寄存器存取器、程序寄存器、目的寄存器、浮点寄存器等。

```
/*
 * precise architected register accessors
 */
```

配置已架构的寄存器存取器；

```
/* next program counter */
```

配置接下来的程序计数器为 EXPR；

```
#define SET_NPC(EXPR)           (regs.regs_NPC = (EXPR))

/* current program counter */
```

配置当前的程序计数器 CPC；

```
#define CPC             (regs.regs_PC)

/* general purpose registers */
```

配置普通目的寄存器 GPR；

```
#define GPR(N)                  (regs.regs_R[N])
#define SET_GPR(N,EXPR)         (regs.regs_R[N] = (EXPR))

#if defined(TARGET_PISA)

/* floating point registers, L->word, F->single-prec, D->double-prec */
```

配置浮点寄存器 FPR；

```
#define FPR_L(N)        (regs.regs_F.l[(N)])
#define SET_FPR_L(N,EXPR) (regs.regs_F.l[(N)] = (EXPR))
#define FPR_F(N)        (regs.regs_F.f[(N)])
#define SET_FPR_F(N,EXPR) (regs.regs_F.f[(N)] = (EXPR))
#define FPR_D(N)        (regs.regs_F.d[(N) >> 1])
#define SET_FPR_D(N,EXPR)(regs.regs_F.d[(N) >> 1] = (EXPR))

/* miscellaneous register accessors */
```

配置寄存器访问器：HI、LO、FCC；

```
#define SET_HI(EXPR)        (regs.regs_C.hi = (EXPR))
#define HI                  (regs.regs_C.hi)
#define SET_LO(EXPR)        (regs.regs_C.lo = (EXPR))
#define LO                  (regs.regs_C.lo)
#define FCC                 (regs.regs_C.fcc)
#define SET_FCC(EXPR)       (regs.regs_C.fcc = (EXPR))

#elif defined(TARGET_ALPHA)

/* floating point registers, L->word, F->single-prec, D->double-prec */
```

配置浮点寄存器 FPR；

```
#define FPR_Q(N)        (regs.regs_F.q[N])
#define SET_FPR_Q(N,EXPR)       (regs.regs_F.q[N] = (EXPR))
#define FPR(N)                  (regs.regs_F.d[N])
#define SET_FPR(N,EXPR)         (regs.regs_F.d[N] = (EXPR))

/* miscellaneous register accessors */
```

配置寄存器访问器：FPCR、UNIQ；

```
#define FPCR                (regs.regs_C.fpcr)
#define SET_FPCR(EXPR)          (regs.regs_C.fpcr = (EXPR))
```

```
#define UNIQ            (regs.regs_C.uniq)
#define SET_UNIQ(EXPR)          (regs.regs_C.uniq = (EXPR))

#else
#error No ISA target defined...
#endif

/* precise architected memory state accessor macros */
```

构架内存状态存取器的宏命令；

```
#define READ_BYTE(SRC, FAULT)                                   \
  ((FAULT) = md_fault_none, addr = (SRC), MEM_READ_BYTE(mem, addr))
#define READ_HALF(SRC, FAULT)                                   \
  ((FAULT) = md_fault_none, addr = (SRC), MEM_READ_HALF(mem, addr))
#define READ_WORD(SRC, FAULT)                                   \
  ((FAULT) = md_fault_none, addr = (SRC), MEM_READ_WORD(mem, addr))
#ifdef HOST_HAS_QWORD
#define READ_QWORD(SRC, FAULT)                                  \
  ((FAULT) = md_fault_none, addr = (SRC), MEM_READ_QWORD(mem, addr))
#endif /* HOST_HAS_QWORD */
```

定义 HOST_HAS_QWORD 的宏命令；

```
#define WRITE_BYTE(SRC, DST, FAULT)                             \
  ((FAULT) = md_fault_none, addr = (DST), MEM_WRITE_BYTE(mem, addr, (SRC)))
#define WRITE_HALF(SRC, DST, FAULT)                             \
  ((FAULT) = md_fault_none, addr = (DST), MEM_WRITE_HALF(mem, addr, (SRC)))
#define WRITE_WORD(SRC, DST, FAULT)                             \
  ((FAULT) = md_fault_none, addr = (DST), MEM_WRITE_WORD(mem, addr, (SRC)))
#ifdef HOST_HAS_QWORD
#define WRITE_QWORD(SRC, DST, FAULT)                            \
  ((FAULT) = md_fault_none, addr = (DST), MEM_WRITE_QWORD(mem, addr, (SRC)))
#endif /* HOST_HAS_QWORD */
```

定义 HOST_HAS_QWORD 的宏命令，同上。

```
/* system call handler macro */
```

系统调用处理程序的宏命令。

```
#define SYSCALL(INST)sys_syscall(&regs, mem_access, mem, INST, TRUE)
```

/* addressing mode FSM (dest of last LUI, used for decoding addr modes) */

设置寻址模式 FSM (dest of last LUI, used for decoding addr modes)；
将参数 FSM 置为 0。

static unsigned int fsm = 0;

八、 sim.h 解析

/* sim.h - simulator main line interfaces */

模拟器主线接口头文件

/* SimpleScalar(TM) Tool Suite
 * Copyright (C) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
 * All Rights Reserved.
 *
 * THIS IS A LEGAL DOCUMENT, BY USING SIMPLESCALAR,
 * YOU ARE AGREEING TO THESE TERMS AND CONDITIONS.
 *
 * No portion of this work may be used by any commercial entity, or for any
 * commercial purpose, without the prior, written permission of SimpleScalar,
 * LLC (info@simplescalar.com). Nonprofit and noncommercial use is permitted
 * as described below.
 *
 * 1. SimpleScalar is provided AS IS, with no warranty of any kind, express
 * or implied. The user of the program accepts full responsibility for the
 * application of the program and the use of any results.
 *
 * 2. Nonprofit and noncommercial use is encouraged. SimpleScalar may be
 * downloaded, compiled, executed, copied, and modified solely for nonprofit,
 * educational, noncommercial research, and noncommercial scholarship
 * purposes provided that this notice in its entirety accompanies all copies.
 * Copies of the modified software can be delivered to persons who use it
 * solely for nonprofit, educational, noncommercial research, and
 * noncommercial scholarship purposes provided that this notice in its
 * entirety accompanies all copies.
 *
 * 3. ALL COMMERCIAL USE, AND ALL USE BY FOR PROFIT ENTITIES, IS EXPRESSLY
 * PROHIBITED WITHOUT A LICENSE FROM SIMPLESCALAR, LLC (info@simplescalar.com).
 *
 * 4. No nonprofit user may place any restrictions on the use of this software,

* including as modified by the user, by any other authorized user.
*
* 5. Noncommercial and nonprofit users may distribute copies of SimpleScalar
* in compiled or executable form as set forth in Section 2, provided that
* either: (A) it is accompanied by the corresponding machine-readable source
* code, or (B) it is accompanied by a written offer, with no time limit, to
* give anyone a machine-readable copy of the corresponding source code in
* return for reimbursement of the cost of distribution. This written offer
* must permit verbatim duplication by anyone, or (C) it is distributed by
* someone who received only the executable form, and is accompanied by a
* copy of the written offer of source code.
*
* 6. SimpleScalar was developed by Todd M. Austin, Ph.D. The tool suite is
* currently maintained by SimpleScalar LLC (info@simplescalar.com). US Mail:
* 2395 Timbercrest Court, Ann Arbor, MI 48105.
*
* Copyright (C) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
*/

```
#ifndef SIM_H
#define SIM_H
```

把 C 文件中要引用的头文件的内容都放在 sim.h 中

```
#include <stdio.h>
```

引用标准输入输出头文件

```
#include <setjmp.h>
```

申明了 setjmp()和 longjmp()函数（分别承担非局部标号和 goto 作用）及同时所需的 jmp_buf 数据类型

```
#include <time.h>
```

引用日期和时间头文件

```
#include "options.h"
```

对自定义配置文件的引用

```
#include "stats.h"
```

包含封装好的程序包接口

#include "regs.h"

包含已架构的寄存器状态接口

#include "memory.h"

包含平面内存模式空间接口

/* set to non-zero when simulator should dump statistics */
extern int sim_dump_stats;

定义整形变量 sim_dump_stats， 在模拟器将清除缓存数据时把它的值设为非零

/* exit when this becomes non-zero */
extern int sim_exit_now;

定义整形变量 sim_exit_now， 在其值为非零时退出

/* longjmp here when simulation is completed */
extern jmp_buf sim_exit_buf;

定义 jmp_buf 变量 sim_exit_buf， 在模拟器完成时在这里将数组里面存储的内容恢复到寄存器里面

/* byte/word swapping required to execute target executable on this host */
extern int sim_swap_bytes;
extern int sim_swap_words;

定义整形变量 sim_swap_bytes 和 sim_swap_words， 在此主机上执行目标可执行文件所需的字节/字交换

/* execution instruction counter */
extern counter_t sim_num_insn;

定义执行指令计数器 sim_num_insn

/* execution start/end times */
extern time_t sim_start_time;
extern time_t sim_end_time;

```
extern int sim_elapsed_time;
```

定义执行的开始和结束时间

```
/* options database */
extern struct opt_odb_t *sim_odb;
```

创立选项数据库

```
/* stats database */
extern struct stat_sdb_t *sim_sdb;
```

创立统计数据库

```
/* EIO interfaces */
extern char *sim_eio_fname;
extern char *sim_chkpt_fname;
extern FILE *sim_eio_fd;
```

定义字符串指针 *sim_eio_fname 和 *sim_chkpt_fname ，
文件指针*sim_eio_fd ，实现 EIO 接口

```
/* redirected program/simulator output file names */
extern FILE *sim_progfd;
```

定义文件指针*sim_progfd， 重定向程序和模拟器输出文件名

```
/*
 * main simulator interfaces, called in the following order
 */
```

主模拟器接口，按以下顺序调用

```
/* register simulator-specific options */
void sim_reg_options(struct opt_odb_t *odb);
```

此函数用于配置 simulator-specific 的环境选项
struct opt_odb_t 为选项数据库

```
/* main() parses options next... */
```

随后主函数 main（）解析环境设定

```
/* check simulator-specific option values */
void sim_check_options(struct opt_odb_t *odb, int argc, char **argv);
```

此函数用于检查 simulator-specific 选项数据库中可选项的值

```
/* register simulator-specific statistics */
void sim_reg_stats(struct stat_sdb_t *sdb);
```

此函数用于配置 simulator-specific 的统计数据

```
/* initialize the simulator */
void sim_init(void);
```

此函数用于初始化模拟器

```
/* load program into simulated state */
void sim_load_prog(char *fname, int argc, char **argv, char **envp);
```

此函数用于将程序加载入模拟器状态。

```
/* main() prints the option database values next... */
```

随后主函数打印选项数据库的值

```
/* print simulator-specific configuration information */
void sim_aux_config(FILE *stream);
```

此函数用于打印 simulator-specific 配置信息；

```
/* start simulation, program loaded, processor precise state initialized */
void sim_main(void);
```

此函数用于开始模拟，载入程序，包括初始化处理程序的状态。

```
/* main() prints the stats database values next... */
```

随后主函数打印统计数据库的值

```
/* dump simulator-specific auxiliary simulator statistics */
void sim_aux_stats(FILE *stream);
```

此函数用于丢弃 simulator-specific 的辅助模拟器的缓存数据

```
/* un-initialize simulator-specific state */
void sim_uninit(void);
```

此函数用于去除 simulator – specific 的初始化状态

```
/* print all simulator stats */
void sim_print_stats(FILE *fd);
```

此函数用于打印模拟器的统计数据

```
/* output stream */
```

输出流*stream

```
#endif /* SIM_H */
```

开头#ifndef SIM_H #define SIM_H 的收尾