EduMIPS64 Documentation

发行版本 *1.3.0*

Andrea Spadaccini (and the EduMIPS64 development team)

2023 年 10 月 16 日

Contents

1 源文件格式 **3**

1.1 内存限制 4
1.2 *.data* 部分 4
1.3 *.code* 部分 5
1.4 *#include* 命令 7

2 指令集 **9**

2.1 ALU 指令 9
2.2 加载/存储指令 14
2.3 流控制指令 15
2.4 SYSCALL 指令 16
2.5 其他指令 18

3 浮点单元 **19**

3.1 特殊价值 19
3.2 异常配置 20
3.3 .double 指令 20
3.4 FCSR 寄存器 21
3.5 指令集 21

4 用户界面 **27**

4.1 菜单栏 28
4.2 窗口 29
4.3 对话框 30
4.4 命令行选项 31

5 代码示例 **33**

5.1 SYSCALL 33

If the characters in the document are badly garbled, please set the environment variable JAVA_TOOL_OPTIONS to -Dfile.encoding=UTF8, or load the jar file with -Dfile.encoding=utf-8.

For more information on this bug, please see here:

https://github.com/EduMIPS64/edumips64/issues/785

译者注：

中文文档使用了 DeepL 机翻，不能保证完全准确。如发现错误，请以英文文档为准，并在 GitHub 上提交修改 Pull Request。

EduMIPS64 是 MIPS64 指令集架构 (ISA) 模拟器。它设计用于执行使用模拟器实现的 MIPS64 ISA 子集的小程序，允许用户查看指令在流水线中的行为、CPU 如何处理停滞、寄存器和内存的状态等。它既是一个模拟器，也是一个可视化调试器。

该项目的网站是 http://www.edumips.org，代码托管在 http://github.com/EduMIPS64/edumips64。如果发现任何错误，或有任何改进模拟器的建议，请在 github 上发布问题，或发送电子邮件至 bugs@edumips.org。

EduMIPS64 是由卡塔尼亚大学（意大利）的一群学生开发的，起初是 WinMIPS64 的克隆版，尽管现在这两个模拟器之间有很多不同之处。

本手册将向您介绍 EduMIPS64，并详细介绍如何使用它。

本手册第一章介绍了模拟器接受的源文件格式，除了命令行参数外，还描述了数据类型和指令。

第二章概述了 EduMIPS64 可接受的 MIPS64 指令集子集，以及所有需要的参数和使用说明。

第三章介绍浮点运算单元及其指令集。

第四章是 EduMIPS64 用户界面的说明，解释了每个框架和菜单的目的，以及配置对话框、Dinero 前端对话框、手册对话框和命令行选项的说明。

第五章包含一些有用的示例。

本手册介绍了 EduMIPS64 版本 1.3.0。

CHAPTER 1

源文件格式

EduMIPS64 尝试遵循其他 MIPS64 和 DLX 模拟器中使用的惯例，这样老用户就不会对其语法感到困惑。

源文件中有两个部分，即 *data* 部分和 *code* 部分，分别由 *.data* 和 *.code* 指令引入。在下面的列表中，你可以看到一个非常基本的 EduMIPS64 程序:

```
; 这是注释
        .data
label:  .word   15      ; This is an inline comment

        .code
        daddi   r1, r0, 0
        syscall 0
```

为了区分每行源代码的不同部分，可以使用空格和制表符的任意组合，因为解析器会忽略多个空格，只空格来分隔标记。

注释可以使用 “;” 字符指定，该字符后面的所有内容都将被忽略。因此，注释可以内联（指令之后）或单独一行中使用。

标签可在代码中用于引用内存单元或指令。标签不区分大小写。每行源代码只能使用一个标签。标签可以在有效数据声明或指令的上方指定一行或多行，前提是标签和声明之间除了注释和空行外没有其他内容。

1.1 内存限制

EduMIPS64 为数据（*.data* 部分，上限为 640 kB，即 80000 个 64 位值）和指令（*.code* 部分，上限为 128 kB，即 32000 条指令，每条指令占用 32 位）设置了固定的内存大小。

这些限制是模拟器硬编码的。

1.2 *.data* 部分

.data 部分包含的命令指定了程序执行开始前内存的填充方式。*.data* 命令的一般格式为:

```
[label:] .datatype value1 [, value2 [, ...]]
```

EduMIPS64 支持不同的数据类型，详见下表。

Type	Directive	Bits required
Byte	*.byte*	8
Half word	*.word16*	16
Word	*.word32*	32
Double Word	*.word* or *.word64*	64

请注意，双字可以由 *.word* 指令或 *.word64* 指令引入。

所有数据类型都解释为带符号。这意味着，*.data* 部分中的整数字面量必须介于 -2^(n-1) 和 2^(n-1) - 1 之间（包括 2^(n-1)）。

使用单个指令声明数据元素列表与使用相同类型的多个指令声明数据元素列表有很大区别。EduMIPS64 一旦发现数据类型标识符，就会从下一个 64 位双字开始写入，因此下面列表中的第一条 *.byte* 语句将把数字 1、2、3 和 4 放在 4 个字节的空间中，占用 32 位，而后面四行的代码将把每个数字放在不同的内存单元中，占用 32 个字节:

```
.data
.byte    1, 2, 3, 4
.byte    1
.byte    2
.byte    3
.byte    4
```

在下表中，内存使用字节大小的单元表示，每行宽 64 位。表中每行左侧的地址指的是最右侧的内存单元，它是每行 8 个单元中地址最低的单元。

0	0	0	0	0	4	3	2	1
8	0	0	0	0	0	0	0	1
16	0	0	0	0	0	0	0	2
24	0	0	0	0	0	0	0	3
36	0	0	0	0	0	0	0	4

有一些特殊指令需要讨论：*.space*、*.ascii* 和 *.asciiz*。

.space 指令用于在内存中留出一些空余空间。它接受一个整数作为参数，表示必须留空的字节数。当您必须为计算结果在内存中留出一些空间时，该指令非常方便。

.ascii 指令接受包含任何 ASCII 字符的字符串，以及下表中描述的一些类似 C 语言的特殊转义序列，并将这些字符串存入内存。

Escaping sequence	Meaning	ASCII code
\0	Null byte	0
\t	Horizontal tabulation	9
\n	Newline character	10
\"	Literal quote character	34
\	Literal backslash character	92

.asciiz “指令的行为与”.ascii “命令完全相同，不同之处在于它会自动以空字节结束字符串。

1.3 *.code* 部分

.code 部分包含的命令指定了程序启动时内存的填充方式。*.code* 命令的一般形式是:

```
[label:] instruction [param1 [, param2 [, param3]]]
```

代码 * 部分可以用 *.text* 别名指定。

参数的数量和类型取决于指令本身。

指令可以使用三种类型的参数:

- 寄存器寄存器参数用大写或小写的 *r* 或 *$* 表示，后面跟寄存器的编号（0 到 31 之间），如 “r4”、” R4” 或” $4”;
- 立即值立即值可以是一个数字或标签；数字可以以 10 进制或 16 进制指定：10 进制数字只需写入数字即可，而 16 进制数字则需在数字前加上前缀 “0x”。立即值的前面可以字符。
- 地址地址由一个立即值和一个寄存器名称组成，寄存器名称用括号括起来。寄存器的值将作为基数，立即值将作为偏移量。

立即值的大小受指令位编码中可用位数的限制。

当可以使用 16 位立即值时，例如在 ALU I-Type 指令中，也可以使用内存标签作为立即值。汇编程序将把标签指向的内存地址作为立即值。

你可以使用标准的 MIPS 汇编别名来寻址前 32 个寄存器，将别名附加到标准寄存器前缀之一，如 “r”、” $” 和 “R “等标准寄存器前缀。请参见下表。

Register	Alias
0	*zero*
1	*at*
2	*v0*
3	*v1*
4	*a0*
5	*a1*
6	*a2*
7	*a3*
8	*t0*
9	*t1*
10	*t2*
11	*t3*
12	*t4*
13	*t5*
14	*t6*
15	*t7*
16	*s0*
17	*s1*
18	*s2*
19	*s3*
20	*s4*
21	*s5*
22	*s6*
23	*s7*
24	*t8*
25	*t9*
26	*k0*
27	*k1*
28	*gp*
29	*sp*
30	*fp*
31	*ra*

1.4 *#include* 命令

源文件可以包含 *#include filename* 命令，其作用是将文件 *filename* 中的内容替换为命令行。

如果你想包含外部例程，这条命令是非常有用的，而且它还带有循环检测算法，如果你试图在文件 *B.s* 中执行”*#include A.s*”，又在文件 *A.s* 中执行”*#include B.s*”，它就会发出警告。

CHAPTER 2

指令集

在本节中，我们将介绍 EduMIPS64 所认知的 MIPS64 指令集子集。我们可以进行两种不同的分类：一种基于指令的功能，一种基于指令的参数类型。

第一种分类法将指令分为三类：ALU 结构指令、加载/存储指令、流控制指令。接下来的几个小节将介绍每一类指令以及属于这些类别的每一条指令。

第四小节将介绍不属于上述任何类别的指令。

2.1 ALU 指令

算术逻辑单元（简称 ALU）是 CPU 执行单元的一部分，负责进行算术和逻辑运算。因此，在 ALU 指令组中，我们可以找到进行此类运算的指令。

ALU 指令可分为两组：*R 型 * 和 I 型*。

其中四条指令使用两个特殊寄存器：LO 和 HI。它们是 CPU 的内部寄存器，可以通过 FLO‘ 和 MFHI‘ 指令访问其值。

下面是 R 型 ALU 指令的列表。

- *AND rd, rs, rt*

  在 rs 和 rt 之间执行比特 AND，并将结果放入 rd。

- *ADD rd, rs, rt*

  将 32 位寄存器 rs 和 rt 的内容相加，将其视为带符号值，并将结果放入 rd。如果出现溢出，则捕获。

- *ADDU rd, rs, rt*

  将 32 位寄存器 rs 和 rt 的内容相加，并将结果放入 rd。在任何情况下都不会发生整数溢出。

- *DADD rd, rs, rt*

  将 64 位寄存器 rs 和 rt 的内容相加，认为它们是带符号的值，并将结果放入 rd。如果出现溢出，则进行陷阱处理。

- *DADDU rd, rs, rt*

  将 64 位寄存器 rs 和 rt 的内容相加，并将结果存入 rd。在任何情况下都不会发生整数溢出。

- *DDIV rs, rt*

  执行 64 位寄存器 rs 和 rt 之间的除法运算，将 64 位商放入 LO，将 64 位余数放入 HI。

- *DDIVU rs, rt*

  执行 64 位寄存器 rs 和 rt 的除法运算，将它们视为无符号值，并将 64 位商放入 LO，将 64 位余数放入 HI。

- *DIV rs, rt*

  执行 32 位寄存器 rs 和 rt 之间的除法运算，将 32 位商放入 LO，将 32 位余数放入 HI。

- *DIVU rs, rt*

  执行 32 位寄存器 rs 和 rt 的除法运算，将它们视为无符号值，并将 32 位商放入 LO，将 32 位余数放入 HI。

- *DMUHU rd, rs, rt*

  执行 64 位寄存器 rs 和 rt 之间的乘法，将它们视为无符号值，并将结果的高位 64 位双字放入寄存器 rd。

- *DMULT rs, rt*

  执行 64 位寄存器 rs 和 rt 之间的乘法运算，将运算结果的低阶 64 位双字放入特殊寄存器 LO，将运算结果的高阶 64 位双字放入特殊寄存器 HI。

- *DMULU rd, rs, rt*

  执行 64 位寄存器 rs 和 rt 之间的乘法运算，将它们视为无符号值，并将结果的低阶 64 位双字放入特殊寄存器 LO，将结果的高阶 64 位双字放入寄存器 rd。

- *DMULTU rs, rt.*

  执行 64 位寄存器 rs 和 rt 之间的乘法运算，将它们视为无符号值，并将结果的低阶 64 位双字放入特殊寄存器 LO，将结果的高阶 64 位双字放入特殊寄存器 HI。

- *DSLL rd, rt, sa*

  将 64 位寄存器 rt 左移，移动量由立即（正）值 sa 指定，并将结果存入 64 位寄存器 rd。空位用零填充。

- *DSLLV rd, rt, rs*

对 64 位寄存器 rt 进行左移，左移量为 rs 的低阶 6 位无符号值，并将结果存入 64 位寄存器 rd。空位用零填充。

- *DSRA rd, rt, sa*

  对 64 位寄存器 rt 进行右移，移动量为即时（正）值 sa 指定的值，并将结果存入 64 位寄存器 rd。如果 rt 的最左边位为 0，则空位用 0 填充，否则用 1 填充。

- *DSRAV rd, rt, rs*

  对 64 位寄存器 rt 进行右移，将 rs 的低阶 6 位数指定为无符号值，并将结果放入 64 位寄存器 rd。如果 rt 的最左边位为 0，则空位用 0 填充，否则用 1 填充。

- *DSRL rd, rs, sa*

  对 64 位寄存器 rs 进行右移，移动量为即时（正）值 sa 指定的量，并将结果放入 64 位寄存器 rd。空位用零填充。

- *DSRLV rd, rt, rs*

  对 64 位寄存器 rt 进行右移，将 rs 的低阶 6 位数指定为无符号值，并将结果存入 64 位寄存器 rd。空位用零填充。

- *DSUB rd, rs, rt.*

  将 64 位寄存器 rt 的值减去 64 位寄存器 rs 的值，将它们视为有符号值，并将结果放入 rd。如果出现溢出，则捕获。

- *DSUBU rd, rs, rt*

  将 64 位寄存器 rt 的值减去 64 位寄存器 rs 的值，并将结果存入 rd。在任何情况下都不会发生整数溢出。

- *MFLO rd*

  将特殊寄存器 LO 的内容移入 rd。

- *MFHI rd.*

  将特殊寄存器 HI 的内容移入 rd。

- *MOVN rd, rs, rt‘*

  如果 rt 与零不同，则将 rs 的内容移入 rd。

- *MOVZ rd, rs, rt*

  如果 rt 等于零，则将 rs 的内容移入 rd。

- *MULT rs, rt*

  在 32 位寄存器 rs 和 rt 之间执行乘法运算，将运算结果的低阶 32 位字放入特殊寄存器 LO，将运算结果的高阶 32 位字放入特殊寄存器 HI。

- *MULTU rs, rt‘*

执行 32 位寄存器 rs 和 rt 之间的乘法运算，将它们视为无符号值，并将运算结果的低阶 32 位字放入特殊寄存器 LO，将运算结果的高阶 32 位字放入特殊寄存器 HI。

- *OR rd、rs、rt‘*

  在 rs 和 rt 之间执行比特 OR，并将结果放入 rd。

- *SLL rd, rt, sa.*

  对 32 位寄存器 rt 进行左移，左移量为即时（正）值 sa 指定的量，并将结果放入 32 位寄存器 rd。空位用零填充。

- *SLLV rd, rt, rs*

  对 32 位寄存器 rt 进行左移，左移量为 rs 的低阶 5 位无符号值，并将结果存入 32 位寄存器 rd。空位用零填充。

- *SRA rd, rt, sa*

  对 32 位寄存器 rt 进行右移，移动量为立即（正）值 sa 指定的值，并将结果放入 32 位寄存器 rd。如果 rt 的最左边位为 0，则空位用 0 填充，否则用 1 填充。

- *SRAV rd, rt, rs*

  对 32 位寄存器 rt 进行右移，将 rs 的低阶 5 位指定为无符号值，并将结果放入 32 位寄存器 rd。如果 rt 的最左边位为 0，则空位用 0 填充，否则用 1 填充。

- *SRL rd, rs, sa*

  对 32 位寄存器 rs 进行右移，移动量为即时（正）值 sa 指定的量，并将结果放入 32 位寄存器 rd。空位用零填充。

- *SRLV rd, rt, rs*

  对 32 位寄存器 rt 进行右移，将 rs 的低阶 5 位数指定为无符号值，并将结果存入 32 位寄存器 rd。空位用零填充。

- *SUB rd, rs, rt.*

  将 32 位寄存器 rt 的值减去 32 位寄存器 rs 的值，将它们视为有符号值，并将结果放入 rd。如果出现溢出，则捕获。

- *SUBU rd, rs, rt*

  将 32 位寄存器 rt 的值减去 32 位寄存器 rs 的值，并将结果存入 rd。在任何情况下都不会发生整数溢出。

- *SLT rd, rs, rt*

  如果 rs 的值小于 rt 的值，则将 rd 的值设置为 1，否则设置为 0。该指令执行带符号比较。

- *SLTU rd, rs, rt*

  如果 rs 的值小于 rt 的值，则将 rd 的值设置为 1，否则将其设置为 0。该指令执行无符号比较。

- *XOR rd, rs, rt*

在 rs 和 rt 之间执行位排他性 OR（XOR），并将结果存入 rd。

以下是 I 型 ALU 指令列表。

- *ADDI rt, rs, immediate* 执行 32 位寄存器 rs 与立即值的和，并将结果存入 rt。

  执行 32 位寄存器 rs 与立即值之和，并将结果存入 rt。该指令将 rs 和立即值视为带符号值。如果发生溢出，则捕获。

- *ADDIU rt, rs, immediate*

  执行 32 位寄存器 rs 与立即值的和，并将结果存入 rt。在任何情况下都不会发生整数溢出。

- *ANDI rt, rs, immediate*

  执行 rs 与立即值之间的位和运算，并将结果存入 rt。

- *DADDI rt, rs, immediate‘*

  执行 64 位寄存器 rs 与立即值的和，将结果存入 rt。如果发生溢出，则捕获。

- *DADDIU rt, rs, immediate*

  执行 64 位寄存器 rs 与立即值的和，将结果存入 rt。在任何情况下都不会发生整数溢出。

- *DADDUI rt, rs, immediate.*

  执行 64 位寄存器 rs 与立即值的和，并将结果存入 rt。在任何情况下都不会发生整数溢出。

- *LUI rt, immediate.*

  将立即值中定义的常量加载到 rt 下 32 位的上半部分（16 位），并对寄存器的上 32 位进行符号扩展。

- *ORI rt, rs, immediate*

  执行 rs 和立即值之间的位操作 OR，将结果放入 rt。

- *SLTI rt, rs, immediate* 设置 rt 的值为 0。

  如果 rs 的值小于立即值，则将 rt 的值设置为 1，否则设置为 0。

- *SLTIU rt, rs, immediate*

  如果 rs 的值小于立即值，则将 rt 的值设置为 1，否则设置为 0。

- *XORI rt, rs, immediate‘*

  在 rs 和立即值之间执行位排他性 OR（XOR），并将结果存入 rt。

2.2 加载/存储指令

这类指令包含所有在寄存器和内存之间进行传输操作的指令。所有这些指令的形式都是::

    [label:] 指令 rt, 偏移量（基数）

其中，rt 是源寄存器或目标寄存器，取决于我们使用的是存储指令还是加载指令；offset 是标签或立即值，base 是寄存器。在寄存器 *base* 的值上加上立即值 *offset*，就得到了地址。

指定的地址必须根据处理的数据类型对齐。以“U “结尾的加载指令将寄存器 rt 的内容视为无符号值。

加载指令列表：

- *LB rt, offset(base)*

  将寄存器 rt 中偏移量和基数指定地址处的存储单元内容作为有符号字节加载。

- *LBU rt, offset(base)*

  将寄存器 rt 中偏移量和基数指定地址处的存储单元内容作为无符号字节加载。

- *LD rt, offset(base)*

  将寄存器 rt 中偏移量和基数指定地址处的存储单元内容作为双字加载。

- *LH rt, offset(base)*

  加载内存单元的内容。将寄存器 rt 中偏移量和基数指定地址处的存储单元内容作为带符号的半字加载。

- *LHU rt, offset(base)*

  加载内存单元的内容。将寄存器 rt 中偏移量和基数指定地址处的存储单元内容作为无符号半字加载。

- *LW rt, offset(base)*

  将寄存器 rt 中偏移量和基数指定地址处的存储单元内容作为有符号字加载。

- *LWU rt, offset(base)*

  将寄存器 rt 中偏移量和基数指定地址处的存储单元内容作为带符号字加载。

存储指令列表：

- *SB rt, offset(base)*

  将寄存器 rt 的内容存储到偏移量和基数指定的存储单元中，将其视为字节。

- *SD rt, offset(base)*

  将寄存器 rt 的内容存储到由偏移量和基数指定的存储单元中，并将其视为双字。

- *SH rt, offset(base)*

  存储寄存器 rt 的内容。将寄存器 rt 的内容存储到由偏移量和基数指定的存储单元中，并将其视为半字。

- *SW rt, offset(base)*

  存储寄存器 rt 的内容。将寄存器 rt 的内容存储到偏移量和基数指定的存储单元中，将其视为一个字。

2.3 流控制指令

流控制指令用于改变 CPU 抓取指令的顺序。我们可以对这些指令进行区分：R 型、I 型和 J 型。

这些指令实际上是在 ID 阶段执行跳转，因此往往会执行无用的取指。在这种情况下，会从流水线中移除两条指令，分支执行停滞计数器会增加两个单位。

R 型流控制指令列表：

- *JALR rs*

  将 rs 的内容放入程序计数器，并将 JALR 指令后的指令地址（即返回值）放入 R31。

- *JR rs*

  将 rs 的内容放入程序计数器。

I 型流控制指令列表：

- *B offset*

  无条件跳转到偏移量

- *BEQ rs, rt, offset*

  如果 rs 等于 rt，则跳转到偏移量。

- *BEQZ rs, offset*

  如果 rs 等于零，则跳转到偏移量。

- *BGEZ rs, offset.*

  如果 rs 大于或等于零，执行 PC 相对跳转到偏移量。

- *BNE rs, rt, offset.*

  如果 rs 不等于 rt，则跳转到偏移量。

- *BNEZ rs, offset.*

  如果 rs 不等于零，则跳转到偏移量。

J 型流控制指令列表：

- *J target*

  将立即值 target 放入程序计数器。

- *JAL target*

  将即期目标值放入程序计数器，并将 JAL 指令后的指令地址（即返回值）放入 R31。

2.4 SYSCALL 指令

SYSCALL 指令为程序员提供了一个类似操作系统的接口，可进行六种不同的系统调用。

系统调用希望将其参数地址寄存在寄存器 R14（$t6）中，并将其返回值寄存在寄存器 R1（$at）中。

系统调用尽可能遵循 POSIX 协议。

2.4.1 *SYSCALL 0 - exit()*

SYSCALL 0 不需要任何参数，也不返回任何值。它只是停止模拟器。

请注意，如果模拟器在源代码中找不到 SYSCALL 0 或其任何等效代码（HALT - TRAP 0），它将自动添加到源代码的末尾。

2.4.2 *SYSCALL 1 - open()*

SYSCALL 1 需要两个参数：一个以零结尾的字符串，表示必须打开的文件的路径名；一个包含整数的双字，表示必须用来指定如何打开文件的标志。

这个整数必须是你想使用的标志的总和，从以下列表中选择：

- *O_RDONLY (0x01)* 以只读模式打开文件；
- *O_WRONLY (0x02)* 以只写模式打开文件；
- *O_RDWR (0x03)* 以读/写模式打开文件；
- *O_CREAT (0x04)* 如果文件不存在，则创建该文件；
- *O_APPEND (0x08)* 在写模式下，在文件末尾追加已写入的文本；
- *O_TRUNC (0x08)* 在写模式下，打开文件后立即删除文件内容。

必须指定前三种模式之一。第四和第五种模式是排他性的，如果指定 O_TRUNC，则不能指定 O_APPEND（反之亦然）。

只需将这些标志的整数值相加，就可以指定模式组合。例如，如果想以只写模式打开文件，并将写入的文本追加到文件末尾，则应指定模式 2 + 8 = 10。

系统调用的返回值是与文件相关联的新文件描述符，可以进一步用于其他系统调用。如果出现错误，返回值将为-1。

2.4.3 *SYSCALL 2 - close()*

SYSCALL 2 只需要一个参数，即被关闭文件的文件描述符。

如果操作成功结束，SYSCALL 2 将返回 0，否则将返回-1。可能的失败原因是试图关闭一个不存在的文件描述符，或试图关闭分别与标准输入、标准输出和标准错误相关联的文件描述符 0、1 或 2。

2.4.4 *SYSCALL 3 - read()*

SYSCALL 3 需要三个参数：要读取的文件描述符、读取数据的地址、读取的字节数。

如果第一个参数为 0，模拟器将通过输入对话框提示用户输入。如果输入的长度大于需要读取的字节数，模拟器将再次显示信息对话框。

如果读取操作失败，模拟器将返回已有效读取的字节数或-1。可能的失败原因包括试图从一个不存在的文件描述符中读取、试图从文件描述符 1（标准输出）或 2（标准错误）中读取或试图从一个只写文件描述符中读取。

2.4.5 *SYSCALL 4 - write()*

SYSCALL 4 需要三个参数：要写入的文件描述符、必须读取数据的地址以及要写入的字节数。

如果第一个参数为 2 或 3，模拟器将弹出输入/输出窗口，并在那里写入读取的数据。

如果写入操作失败，模拟器将返回已写入的字节数或-1。失败的可能原因是尝试向不存在的文件描述符写入数据、尝试向文件描述符 0（标准输入）写入数据或尝试向只读文件描述符写入数据。

2.4.6 *SYSCALL 5 - printf()*

SYSCALL 5 需要多个参数，第一个参数是所谓“格式字符串”的地址。格式字符串中可以包含一些占位符，如下表所示：

- *%s* 表示字符串参数；
- *%i* 表示整数参数；
- *%d* 行为类似于 *%i*；
- *%%* 字面意义为 *%*

对于每一个 *%s*、*%d* 或 *%i* 占位符，SYSCALL 5 都期望一个参数，从上一个占位符的地址开始。

如果 SYSCALL 找到一个整数参数的占位符，它就希望相应的参数是一个整数值；如果 SYSCALL 找到一个字符串参数的占位符，它就希望参数是字符串的地址。

结果打印在输入/输出窗口中，写入的字节数放入 R1。

如果出现错误，则向 R1 写入-1。

2.5 其他指令

在本节中，有一些指令不属于前几类。

2.5.1 *BREAK‘*

如果模拟器正在运行，BREAK 指令抛出的异常具有停止执行的效果。它可用于调试目的。

2.5.2 *NOP*

NOP 指令不执行任何操作，用于在源代码中创建间隙。

2.5.3 *TRAP*

TRAP 指令是 SYSCALL 指令的弃用别名。

2.5.4 *HALT*

HALT 指令是 SYSCALL 0 指令的弃用别名，用于停止模拟器。

CHAPTER 3

浮点单元

本章[1] 介绍 EduMIPS64 仿真的浮点运算单元（FPU）。

在第一段中，我们将介绍双倍格式、IEEE 754 标准中定义的特殊浮点数值以及浮点计算可能引发的异常情况。

第二段介绍 EduMIPS64 如何允许用户启用或禁用 IEEE 浮点陷阱。

第三段介绍如何在源程序中指定双精度数和特殊值。

在第四段中，我们将介绍 FPU 用来表示其状态的 FCSR 寄存器。它包含四舍五入、比较运算的布尔结果以及处理 IEEE 浮点异常的策略等信息。

在第五段，也是最后一段，我们将介绍 EduMIPS64 中实现的所有 MIPS64 浮点指令。

在开始讨论 FPU 之前，我们将浮点双精度数域定义为 [-1.79E308,-4.94E-324] ∪ {0} ∪ [4.94E-324,1.79E308]。

3.1 特殊价值

浮点运算允许程序员选择在进行无效运算时是否停止计算。在这种情况下，零与零之间的除法或负数的平方根等运算必须产生一个结果，而这个结果如果不是数字（NaN），就会被视为不同的结果。

[1] 本章是马西莫-特鲁比亚（Massimo Trubia）学士学位论文 “Progetto e implementazione di un modello di Floating Point Unit per un simulatore di CPU MIPS64 “的一部分。

3.1.1 NaN 或无效操作

IEEE 浮点运算标准（IEEE 754）规定，无效的运算操作可以在程序运行时发出错误信号（使用 IEEE 异常 ** 无效操作 ** 的陷阱），或者返回特殊值 QNan（退出非数值）。另一个 NaN 值是 SNan（Signalling Not a Number），一旦检测到它是操作数之一，就会无条件地引发相同的陷阱。这个值在应用程序中很少使用，历史上一直用于初始化变量。

3.1.2 零或下溢

该标准定义的另一个特殊值是零。由于 double 格式的域中不包含零，因此它被视为一个特殊值。零值有正零值和负零值两种：前者用于表示负数 ∈]-4.94E-324,0[) 时，并且需要一个结果（而不是允许 ** 下溢 ** 陷阱）；后者用于表示的数字 ∈ [0,4.94E-324[] 时，并且禁止下溢陷阱。

3.1.3 无穷大还是溢出

当程序试图表示一个绝对值极大的数值（∈]-∞,-1.79E308[∪]1.79E308,+∞[），而这个数值超出了双数值的范围时，CPU 会返回 -∞ 或 +∞。另一种方法是触发异常 ** 溢出 ** 条件陷阱。

在除以零的情况下，也可以返回无穷小数值；在这种情况下，无穷小数值的符号由零的符号和红利的符号的乘积给出。除以零 ** 的陷阱也会出现。

3.2 异常配置

EduMIPS64 允许用户通过 配置 → 设置窗口中的 *FPU* 异常选项卡，启用或禁用 5 个 IEEE 异常中 4 个的陷阱。如果禁用了其中任何一个，将返回相应的特殊值（如特殊价值 中所述）。

3.3 .double 指令

`.double` 指令必须在源文件的 `.data` 部分使用，它允许为一个 *double* 值分配一个内存单元。

该指令有两种使用方式:

```
变量名: .double double_number
变量名: .double 关键字
```

其中，`double_number` 可以用扩展符号（`1.0,0.003`）或科学符号（*3.7E-12,0.5E32*）表示。关键字 “可以是 “POSITIVEINFINITY“、`NEGATIVEINFINITY`、`POSITIVEZERO`、`NEGATIVEZERO`、*SNAN*“和 *'QNAN*‘，因此可以直接在内存中插入特殊值。

3.4 FCSR 寄存器

FCSR（浮点控制状态寄存器）是控制 FPU 多个功能方面的寄存器。它的长度为 32 位，在统计窗口中表示。

FCC 字段宽 8 位，从 0 到 7。条件指令（`C.EQ.D`,`C.LT.D`）使用它来保存两个寄存器之间比较的布尔结果。

Cause、Enables 和 Flag 字段用于处理特殊价值 中描述的 IEEE 异常动态。每个字段由 5 个位组成，分别是 V（无效操作）、Z（除以零）、O（溢出）、U（下溢）和 I（不精确）；后者尚未使用。

如果在程序执行过程中出现相应的 IEEE 异常，**Clause** 字段位将被置位。

启用 ** 字段位通过配置窗口设置，显示启用陷阱的 IEEE 异常。

Flag 字段显示已发生的异常，但由于陷阱未针对该异常启用，因此返回了特殊值（特殊价值 中描述的特殊值）。

RM 字段描述了当前使用的将浮点数转换为整数的舍入方法（请参阅“CVT.L.D “指令的描述）。

3.5 指令集

本节介绍 EduMIPS64 实现的 MIPS64 FPU 指令；它们按字母顺序排列。指令执行的操作使用以下符号描述：第 i 个存储单元表示为 `memory[i]`，FCSR 寄存器的 FCC 字段表示为 `FCSR_FCC[cc]`，`cc` ∈ [0,7]。

在某些指令中，为了避免歧义，寄存器被表示为 `GPR[i]` 和 `FPR[i]`，`i` ∈ [0,31]，但在大多数情况下，我们只使用 `rx` 或 `fx` 符号，其中 x``∈ {d,s,t}。三个字母用来表示每个寄存器的用途（目的寄存器、源寄存器、第三寄存器）。最后，转换操作返回的值用以下符号表示：``convert_conversiontype(register[, rounding_type])、其中 `rounding_type` 参数是可选的。

有关 FPU 指令的一些示例，请访问 `http://www.edumips.org/attachment/wiki/Upload/FPUMaxSamples.rar`。

- *ADD.D fd, fs, ft*

  描述：*fd =fs+ft*。

  异常：如果结果无法根据 IEEE 754 表示，将产生溢出和下溢陷阱。如果 fs 或 ft 包含 QNaN 或 SNan，或者执行了无效操作（+∞ - ∞），则会产生无效操作。

- *BC1F cc, offset.*

  描述：*if FCSR_FCC[cc] == 0 then branch.*

  如果 `FCSR_FCC[cc]` 为 false，则执行 PC 相关分支。

  示例：

```
C.EQ.D 7,f1,f2
BC1F 7,label
```

在本例中，C.EQ.D 检查 “f1“和 “f2“是否相等，并将比较结果写入 FCSR 寄存器 FCC 字段的第 7 位。之后，如果比较结果为 0（假），*BC1F‘* 将跳转到 ‘label‘。

- *BC1T cc, offset*

  描述：*if FCSR_FCC[cc] == 1 then branch‘.*

  如果 FCSR_FCC[cc] 为真，则执行 PC 相关分支。

  示例:

```
C.EQ.D 7,f1,f2
BC1T 7,label
```

  在本例中，C.EQ.D 检查 “f1“和 “f2“是否相等，并将比较结果写入 FCSR 寄存器 FCC 字段的第 7 位。之后，如果比较结果为 1（假），则 BC1F 跳转到 label 。

- *C.EQ.D cc, fs, ft.*

  描述：*FCSR_FCC[cc] = (fs==ft)‘*

  检查 *fs* 是否等于 *ft*，并将比较结果保存在 *FCSR_FCC[cc]* 中。请参阅 BC1T, BC1F 的示例。

  异常：如果 fs 或 ft 包含 QNaN（如果启用则触发陷阱）o SNaN（总是触发陷阱），则可能抛出无效操作。

- *C.LT.D cc,fs,ft*。

  描述：*FCSR_FCC[cc] = (fs<ft)‘*

  检查 *fs* 是否小于 *ft*，并将比较结果保存在 *FCSR_FCC[cc]*。

  示例::

      C.LT.D 2,f1,f2 BC1T 2,target

  在本例中，*C.LT.D‘* 检查 *f1* 是否小于 *f2*，并将比较结果保存在 FCSR 寄存器 FCC 字段的第二位。之后，如果 BC1T 位设置为 1，则跳转到 target 位。

  异常：如果 fs 或 ft 包含 QNaN（陷阱启用时触发）o SNaN（陷阱总是触发），则会抛出无效操作。

- *CVT.D.L fd,fs*。

  描述：*fd = convert_longToDouble(fs)‘*

  将 long 转换为 double。

  示例::

      DMTC1 r6,f5 CVT.D.L f5,f5

  在此示例中，*DMTC1‘* 将 GPR r6 的值复制到 FPR f5；然后，*CVT.D.L‘* 将存储在 f5 中的值从 long 转换为 double。例如，如果 r6 包含值 52，在执行 DMTC1 之后，52 的二进制表示将被复制到 f5。在执行 CVT.D.L 之后，f5 包含 52.0 的 IEEE 754 表示。

  异常：如果 fs 包含 QNaN、SNaN 或无限值，则会抛出无效操作。

- *CVT.D.W fd,fs*。

  描述： `fd = convert_IntToDouble(fs)`

  将 int 转换为 double。

  示例::

  MTC1 r6,f5 CVT.D.W f5,f5

  在本例中，`MTC1` 将 GPR r6 的低 32 位复制到 FPR f5 中。然后，`CVT.D.W` 读取 f5 作为 int，并将其转换为 double。

  如果我们有 `r6=0xAAAAAAAABBBBBBBB`，在执行 `MTC1` 后，我们会得到 `f5=0xXXXXXXXXBBBBBBB`；其上 32 位（`XX..X`）现在是未定义的（未被覆盖）。CVT.D.W“将 f5 解释为 int (`f5=-1145324613`) 并转换为 double (`f5=0xC1D11111400000 =-1.145324613E9`).

  异常：如果 fs 包含 QNaN、SNaN 或无限值，则会抛出无效操作。

- *CVT.L.D fd,fs*

  描述： `fd = convert_doubleToLong(fs, CurrentRoundingMode)`

  将 double 转换为 long，在转换前进行四舍五入。

  示例::

  CVT.L.D f5,f5 DMFC1 r6,f5

  CVT.L.D 将 f5 中的 double 值转换为 long 值；然后“DMFC1 “将 f5 复制到 r6；此操作的结果取决于当前的舍入模式，可在“配置“* → “设置“* 窗口的“*FPU* 舍入“选项卡中进行设置。

表 1: 四舍五入示例

Tipo	RM field	f5 register	r6 register
To nearest	0	6.4	6
To nearest	0	6.8	7
To nearest	0	6.5	6 (to even)
To nearest	0	7.5	8 (to even)
Towards 0	1	7.1	7
Towards 0	1	-2.3	-2
Towards ∞	2	4.2	5
Towards ∞	2	-3.9	-3
Towards -∞	3	4.2	4
Towards -∞	3	-3.9	-4

- *CVT.W.D fd, fs*

  描述： `fd = convert_DoubleToInt(fs, CurrentRoundingMode)`

  使用当前舍入模式将 double 转换为 int。

异常：如果 fs 包含一个无限值、任何 NaN 或结果超出有符号 int 域 [-2 ^{63}, 2 ^{63} -1] 则会抛出无效操作。

- *DIV.D fd, fs, ft*

  描述： *fd = fsdiv ft*

  异常：如果结果不能用 IEEE 754 标准表示，则会出现溢出或下溢。如果 fs 或 ft 包含 QNaN 或 SNan，或者执行了无效操作（0div0,∞ div ∞），则会出现无效操作。如果试图用非 QNaN 或 SNaN 的红利除以零，则会出现除以零的提示。

- *DMFC1 rt,fs.*

  描述： *rt = fs*

  将 FPR fs 按位复制到 GPR rt 中。

- *DMTC1 rt, fs*

  描述： `fs = rt`

  将 GPR rt 按位复制到 FPR fs 中。

- *L.D ft, offset(base)*

  描述： *ft = memory[GPR[base] + offset]‘*

  从内存中加载一个双字，并将其存储在 ft 中。

注：*L.D* 不存在于 MIPS64 ISA 中，它是 *LDC1* 的别名，存在于 EduMIPS64 中，以便与 WinMIPS64 兼容。

- LDC1 ft, offset(base)‘

  描述： `memory[GPR[base] + offset]`

  从内存中加载一个双字，并将其存储在 ft 中。

- *LWC1 ft, offset(base)*

  描述： *ft = memory[GPR[base] + offset]‘*

  从内存中加载一个字并将其存储在 ft 中。

- *MFC1 rt,fs*

  描述： `rt = readInt(fs)`

  读取 fs FPR 的 int 值，并将其写入 rt GPR 的 long 值。示例::

  ```
  MFC1 r6,f5 SD r6,mem(R0)
  ```

  让 `f5=0xAAAAAAAABBBBBB`; `MFC1` 读取 f5 作为 int（低 32 位），将 `BBBBBBBB` 解释为 `-1145324613`，并将值写入 f6（64 位）。执行 “MFC1“后，`r6=0xFFFFFFFFBBBBBBBB=-1145324613`。因此，由于 r6 中的符号被扩展，*SD‘*指令将向内存写入一个具有此值的双字。

- *MOVF.D fd, fs, cc*

  描述： `if FCSR_FCC[cc] == 0 then fd=fs`

如果 FCSR_FCC[cc] 为假，则将 fs 复制到 fd。

- *MOVT.D fd, fs, cc*

  说明：`if FCSR_FCC[cc] == 1 then fd=fs`

  如果 FCSR_FCC[cc] 为真，则将 fs 复制到 fd。

- *MOV.D fd,fs*

  描述：*fd = fs*

  将 fs 复制到 fd。

MOVN.D fd, fs, rt

描述：`if rt != 0 then fd=fs`

如果 rt 不为零，则将 fs 复制到 fd。

- *MOVZ.D fd, fs, rt*

  说明：`if rt == 0 then fd=fs`

  如果 rt 等于零，则将 fs 复制到 fd。

- MTC1 rt, fs

  描述：$fs = rt_{0..31}$

  将 rt 的低 32 位复制到 fs。

  示例::

  ```
  MTC1 r6,f5
  ```

  让 `r5=0xAAAAAAABBBBBBB``B;MTC1` 读取 r5 的低 32 位，并将其复制到 f5 的低 32 位。f5 的高 32 位不会被覆盖。

- *MUL.D fd, fs, ft*

  描述：`fd =fs×ft`。

  异常：如果结果不能用 IEEE 754 标准表示，则会出现溢出或下溢。如果 fs 或 ft 包含 QNaN 或 SNan，或执行了无效操作（乘以 ∞ 或 BY QNaN），则会出现无效操作。

- *S.D ft，offset(base)*。

  描述：*memory[base+offset] = ft‘*

  将 ft 复制到内存中。

注意：MIPS64 ISA 中没有 ‘S.D‘，它是 ‘SDC1‘的别名，EduMIPS64 中有 ‘SDC1‘，以便与 WinMIPS64 兼容。

- *SDC1 ft, offset(base)*

  描述：`memory[base+offset] = ft`

  将 ft 复制到内存。

- *SUB.D fd, fs, ft*

  描述： `fd = fs-ft`

  异常：如果结果无法根据 IEEE 753 表示，则会产生溢出和下溢陷阱。如果 fs 或 ft 包含 QNaN 或 SNan，或者执行了无效操作（+∞ - ∞），则会产生无效操作。

- *SWC1 ft, offset(base)*

  描述： `memory[base+offset] = ft`

  将 ft 的低 32 位复制到内存中。

CHAPTER 4

用户界面

EduMIPS64 的图形用户界面借鉴了 WinMIPS64 的用户界面。事实上，除了一些菜单外，主窗口是完全相同的。

EduMIPS64 主窗口由一个菜单栏和六个框架组成，显示仿真的不同方面。此外，还有一个状态栏，它有双重作用，一是在点击内存单元和寄存器时显示它们的内容，二是在模拟已启动但未选择 verbose（详细）模式时通知用户模拟器正在运行。

状态栏还显示 CPU 状态。它可以显示以下四种状态之一：

- *READY*（准备就绪） CPU 未执行任何指令（未加载程序）。
- *RUNNING*（运行中） CPU 正在执行一系列指令。
- *STOPPING*（停止中） CPU 已找到终止指令，正在执行流水线中已有的指令，然后终止执行。
- *HALTED*（关闭） CPU 已停止：程序刚刚运行完毕。

请注意，CPU 状态与模拟器状态不同。模拟器可能会执行若干次 CPU 循环，然后停止执行，允许用户检查内存和寄存器：在这种状态下，在 CPU 循环之间，CPU 保持 *RUNNING* 或 *STOPPING* 状态。一旦 CPU 进入 *HALTED* 状态，如果不重新加载程序（同一程序或不同程序），用户就无法运行任何 CPU 循环。

更多详情请参见以下章节。

4.1 菜单栏

菜单栏包含六个菜单：

4.1.1 文件

文件菜单包含有关打开文件、重置或关闭模拟器、写入跟踪文件的菜单项。

- 打开…打开一个对话框，允许用户选择要打开的源文件。打开一个对话框，允许用户选择要打开的源文件。
- 打开最近的文件显示模拟器最近打开的文件列表，用户可以从中选择要打开的文件。
- 重置重置模拟器，保留已加载的文件，但重置执行。重置模拟器，保留已加载的文件，但重置执行。
- 写入 *Dinero* 跟踪文件以 xdin 格式将内存访问数据写入文件。
- 退出关闭模拟器。

Write Dinero Tracefile…*（写入 Dinero Tracefile…*）菜单项只有在执行了整个源文件并已结束时才可用。

4.1.2 执行

执行（Execute）菜单包含有关仿真执行流程的菜单项。

- 单周期执行一个仿真步骤
- 运行开始执行，当模拟器执行到 SYSCALL 0（或类似指令），或 BREAK 指令，或用户点击停止菜单项（或按 F9 键）时停止。
- 多周期执行一些仿真步骤。执行的步数可通过设置对话框进行配置。
- 停止当模拟器处于运行或多循环模式时停止执行，如前所述。

该菜单仅在加载源文件且模拟尚未结束时可用。*停止 * 菜单项仅在运行或多循环模式下可用。

请注意，在更新用户界面时模拟器的运行速度会减慢。如果想快速执行较长（数千周期）的程序，请禁用在多步执行中图形与 CPU 同步选项。

4.1.3 配置

配置菜单为自定义 EduMIPS64 的外观和行为提供了便利。

- 设置…打开设置”对话框，本章后续章节将对其进行介绍；
- 更改语言允许用户更改用户界面使用的语言。这一更改会影响图形用户界面的各个方面，从框架标题到在线手册和警告/错误信息。

当模拟器处于运行”或多循环”状态时，”设置…”菜单项不可用。”运行”或多循环”模式时，”设置…”菜单项不可用，因为可能会出现竞争冒险 (Race Condition)。

4.1.4 工具

该菜单只包含一个项目，用于调用 Dinero Frontend 对话框。

- *Dinero Frontend*…打开 Dinero Frontend 对话框。

在未执行程序且执行结束之前，此菜单不可用。

4.1.5 窗口

该菜单包含与窗口操作有关的项目。

Tile 对可见窗口进行平铺，使一排中的窗口不超过三个。它会尽量扩大每个窗口所占的空间。

其他菜单项只是切换每个窗口的状态，使其可见或最小化。

4.1.6 帮助

该菜单包含与帮助相关的菜单项。

- 手册…显示帮助对话框。
- 关于我们…显示一个可爱的对话框，其中包含项目贡献者的姓名及其角色。

4.2 窗口

图形用户界面由七个窗口组成，其中六个默认可见，一个（输入/输出窗口）隐藏。

4.2.1 循环

周期窗口显示执行流程在一段时间内的演变情况，显示每个时间段内哪些指令处于流水线中，以及这些指令处于流水线的哪个阶段。

4.2.2 寄存器

寄存器窗口显示每个寄存器的内容。左键单击寄存器可在状态栏中看到其十进制（带符号）值，双击寄存器可弹出对话框，允许用户更改寄存器的值。

4.2.3 统计数据

统计窗口显示程序执行的一些统计数据。

请注意，在最后一个执行周期内，循环计数器不会递增，因为最后一个执行周期不是一个完整的 CPU 周期，而是一个伪周期，其唯一的任务是从流水线中移除最后一条指令，并递增已执行指令的计数器。

4.2.4 流水线

流水线窗口显示流水线的实际状态，显示哪条指令处于哪个流水线阶段。不同的颜色突出显示不同的流水线阶段。

4.2.5 内存

内存窗口显示内存单元的内容，以及来自源代码的标签和注释。与寄存器一样，内存单元的内容也可以双击修改，点击内存单元会在状态栏中显示其十进制值。第一列显示内存单元的十六进制地址，第二列显示单元值。其他列显示源代码中的其他信息。

4.2.6 代码

代码窗口显示内存中加载的指令。第一列显示指令的地址，第二列显示指令的十六进制表示。其他列显示源代码中的其他信息。

4.2.7 输入/输出

输入/输出窗口为用户提供了一个界面，以查看程序通过 SYSCALL 4 和 5 创建的输出。实际上，它并不用于输入，因为 SYSCALL 3 试图从标准输入读取时会弹出一个对话框，但未来的版本将包括一个输入文本框。

4.3 对话框

EduMIPS64 使用对话框以多种方式与用户交互。以下是最重要对话框的摘要：

4.3.1 设置

在设置对话框中可以对模拟器的各个方面进行配置。单击 OK（确定）”按钮将保存选项，而单击 Cancel（取消）（或直接关闭窗口）将忽略更改。如果要保存更改，请不要忘记点击确定”。

主设置选项卡允许配置转发和多循环模式下的步数。

行为选项卡允许启用或禁用解析阶段的警告。”多步执行中图形与 CPU 同步选项选中后，将使窗口的图形状态与模拟器的内部状态同步。这意味着模拟速度会变慢，但在模拟过程中会有明确的图形反馈。如果选中该

选项，”循环间隔选项将影响模拟器在开始一个新循环之前需要等待多少毫秒。这些选项只有在使用运行或执行菜单中的多循环选项运行模拟时，这些选项才会生效。

最后两个选项设置了同步异常发生时模拟器的行为。如果选中屏蔽同步异常选项，模拟器将忽略任何除以零或整数溢出异常。如果选中同步异常时终止选项，模拟器将在同步异常发生时停止模拟。请注意，如果同步异常被屏蔽，即使选中了终止选项，也不会发生任何情况。如果未屏蔽异常，也未选中终止选项，则会弹出对话框，但对话框关闭后模拟将继续进行。如果未屏蔽异常且选中终止选项，则会弹出对话框，关闭对话框后模拟将立即停止。

最后一个选项卡可以更改用户界面的外观。其中包括更改不同流水线阶段相关颜色的选项、选择内存单元显示为长数值还是双数值的选项以及设置用户界面字体大小的选项。

需要注意的是，用户界面与字体大小之间的比例关系远非完美，但足以让模拟器在高分辨率显示器（如 4K）上使用。

4.3.2 Dinero 前端

通过 Dinero Frontend 对话框，可以向 DineroIV 进程提供程序执行时内部生成的跟踪文件。第一个文本框中是 DineroIV 可执行文件的路径，第二个文本框中必须是 DineroIV 的参数。

关于 DineroIV 缓存模拟器的更多信息，请参阅 ~cite{dinero-web}。

下部包含 DineroIV 进程的输出，你可以从中获取所需的数据。

4.3.3 帮助

通过帮助（Help）对话框可以查看在线手册，该手册是本文档的 HTML 副本。

4.4 命令行选项

有四个命令行选项。下面的列表对它们进行了说明，长名称用圆括号括起来。长名称和短名称的使用方法相同。

- *-v (–version)* 打印模拟器版本并退出。
- *-h (–help)* 打印命令行选项的帮助信息，然后退出。
- *-f (–file) filename* 在模拟器中打开 ‘filename‘。
- *-r (–reset)* 将存储的配置重置为默认值
- *-d (–debug)* 进入调试模式
- *-hl (–headless)* 在无头模式下运行 EduMIPS64（无 gui）

-debug 的作用是激活调试模式。在该模式下，会出现一个新的窗口，即调试窗口，显示 EduMIPS64 的内部活动日志。它对最终用户没有用处，仅供 EduMIPS64 开发人员使用。

CHAPTER 5

代码示例

在本章中，您将看到一些对理解 EduMIPS64 如何工作非常有用的示例列表，以了解 EduMIPS64 如何工作。

5.1 SYSCALL

重要的是要了解 SYSCALL 1-4 的示例引用了 'print.s' 文件，这是 SYSCALL 5 的示例。如果要运行示例，应将该示例的内容复制到一个名为 'print.s' 的文件，并将其包含在代码中。

有些示例会使用已存在的文件描述符，即使它并不真正存在。如果要运行这些示例，请使用 SYSCALL 1 示例打开一个文件。

5.1.1 SYSCALL 0

调用 SYSCALL 0 时，程序停止执行。示例:

```
.code
    daddi   r1, r0, 0    ; saves 0 in R1
    syscall 0            ; exits
```

5.1.2 SYSCALL 1

打开文件的示例程序:

```
                .data
error_op:       .asciiz     "Error opening the file"
ok_message:     .asciiz     "All right"
params_sys1:    .asciiz     "filename.txt"
                .word64     0xF

                .text
open:           daddi       r14, r0, params_sys1
                syscall     1
                daddi       $s0, r0, -1
                dadd        $s2, r0, r1
                daddi       $a0,r0,ok_message
                bne         r1,$s0,end
                daddi       $a0,r0,error_op

end:            jal         print_string
                syscall 0

                #include    print.s
```

在前两行中，我们将包含错误信息和成功信息的字符串写入内存，并将其传递给 print_string 函数。给它们加上两个标签。print_string 函数包含在 print.s 文件中。

接下来，我们向内存写入 SYSCALL 1 所需的数据（第 4、5 行）、要打开的文件的路径（第 6、7 行要打开的文件的路径（如果我们使用读取或读/写模式），并在下一个内存单元中写入一个定义打开模式的整数。

在本例中，文件使用以下模式打开：| *o_rdwr* | *o_creat* | *_append*。数字数字 15（0xF，以 16 为基数）来自这三种模式的值之和（3 + 4 + 8）。

我们给这些数据加上一个标签，以便以后使用。

在.text 部分，我们将 params_sys1 的地址（对编译器来说是一个数字）保存在寄存器中。在.text 部分，我们将 params_sys1 的地址（对编译器来说是一个数字）保存在寄存器 r14 中；接下来我们可以调用 SYSCALL 1 并将 r1 的内容保存在 $sys1 中。将 r1 的内容保存在 $s2 中，这样我们就可以在程序的其余部分中使用它了（例如，使用其他 SYSCALL）。

然后调用 print_string 函数，将 error_op 作为参数传递，如果 r1 等于-1（第 13-14 行），否则将 ok_message 作为参数（第 12 和 14 行）。则将 ok_message 作为参数（第 12 和 16 行）。

5.1.3 SYSCALL 2

关闭文件的示例程序:

```
                .data
params_sys2:    .space 8
error_cl:       .asciiz     "Error closing the file"
ok_message:     .asciiz     "All right"

                .text
close:          daddi       r14, r0, params_sys2
                sw          $s2, params_sys2(r0)
                syscall     2
                daddi       $s0, r0, -1
                daddi       $a0, r0, ok_message
                bne         r1, $s0, end
                daddi       $a0, r0, error_cl

end:            jal         print_string
                syscall     0

                #include    print.s
```

首先，我们为 SYSCALL 2 的唯一参数，即必须关闭的文件的文件描述符（第 2 行）保存一些内存，并给它一个标签，以便以后访问。

接下来，我们将包含错误信息和成功信息的字符串放入内存，这些字符串将传递给 print_string 函数（第 3、4 行）。

在.text 部分，我们将 params_sys2 的地址保存在 r14 中；然后我们就可以调用 SYSCALL 2。

现在我们使用 error_cl 作为参数调用 print_string 函数，如果 r1 则调用 print_string 函数（第 13 行），如果一切顺利，则使用 ok_message 作为参数调用 print_string 函数（第 11 行）。则使用 ok_message 作为参数调用该函数（第 11 行）。

注：此列表需要寄存器 $s2 包含要调用的文件的文件描述符。文件描述符。

5.1.4 SYSCALL 3

从文件读取 16 个字节并保存到内存的示例程序:

```
                .data
params_sys3:    .space      8
ind_value:      .space      8
                .word64     16
error_3:        .asciiz     "Error while reading from file"
```

(续下页)

(接上页)

```
ok_message:     .asciiz     "All right"

value:          .space      30

                .text
read:           daddi       r14, r0, params_sys3
                sw          $s2, params_sys3(r0)
                daddi       $s1, r0, value
                sw          $s1, ind_value(r0)
                syscall     3
                daddi       $s0, r0, -1
                daddi       $a0, r0,ok_message
                bne         r1, $s0,end
                daddi       $a0, r0,error_3

end:            jal         print_string
                syscall     0

                #include    print.s
```

.data 部分的前 4 行包含 SYSCALL 3 的参数、我们必须从中读取的文件描述符、SYSCALL 必须保存读取数据的内存地址以及要读取的字节数。我们给那些稍后必须访问的参数加上标签。接下来，像往常一样，我们将包含错误信息和成功信息的字符串放入其中。

在.text 部分，我们将 params_sys3 的地址保存到寄存器 r14 中，并在 SYSCALL 参数的内存单元中保存文件描述符（我们假设保存在 $s2 中）和用于保存读取字节的地址。

接下来，我们可以调用 SYSCALL 3，然后调用 print_string 函数根据操作成功与否，将 error_3 或 ok_message 作为参数传递。

5.1.5 SYSCALL 4

向文件写入字符串的示例程序:

```
                .data
params_sys4:    .space      8
ind_value:      .space      8
                .word64     16
error_4:        .asciiz     "Error writing to file"
ok_message:     .asciiz     "All right"
value:          .space      30

                .text
```

(续下页)

(接上页)

```
write:          daddi       r14, r0,params_sys4
                sw          $s2, params_sys4(r0)
                daddi       $s1, r0,value
                sw          $s1, ind_value(r0)
                syscall     4
                daddi       $s0, r0,-1
                daddi       $a0, r0,ok_message
                bne         r1, $s0,end
                daddi       $a0, r0,error_4

end:            jal         print_string
                syscall     0

                #include    print.s
```

.data 部分的前 4 行包含 SYSCALL 4 的参数、我们必须读取的文件描述符、SYSCALL 必须读取的内存地址、要写入的字节数。我们给那些稍后必须访问的参数加上标签。接下来，像往常一样，我们将包含错误信息和成功信息的字符串放入其中。

在.text 部分，我们将 params_sys4 的地址保存到寄存器 r14 中，在 SYSCALL 参数的内存单元中保存文件描述符（我们假设保存在 $s2 中）和我们必须读取写入字节的地址。

接下来我们可以调用 SYSCALL 3，然后根据操作的成功与否调用 print_string 函数，参数为 error_3 或 ok_message。

5.1.6 SYSCALL 5

包含将 $a0 中的字符串打印到标准输出的函数的示例程序:

```
                .data
params_sys5:    .space  8

                .text
print_string:
                sw      $a0, params_sys5(r0)
                daddi   r14, r0, params_sys5
                syscall 5
                jr      r31
```

第二行用于为 SYSCALL 必须打印的字符串保存空间，由.text 部分的第一条指令填充，该指令假定 $a0 中有要打印的字符串地址。

下一条指令将字符串的地址放入 r14，然后我们就可以调用 SYSCALL 5 打印字符串了。最后一条指令将程序计数器设置为 r31 中的内容，这是 MIPS 通常的调用习惯。

5.1.7 一个更复杂的 SYSCALL 5 使用示例

SYSCALL 5 使用了一种并不简单的参数传递机制，下面的示例将对此进行说明:

```
                .data
format_str:     .asciiz   "%dth of %s:\n%s version %i.%i is being tested!"
s1:             .asciiz   "June"
s2:             .asciiz   "EduMIPS64"
fs_addr:        .space    4
                .word     5
s1_addr:        .space    4
s2_addr:        .space    4
                .word     0
                .word     5
test:
                .code
                daddi     r5, r0, format_str
                sw        r5, fs_addr(r0)
                daddi     r2, r0, s1
                daddi     r3, r0, s2
                sd        r2, s1_addr(r0)
                sd        r3, s2_addr(r0)
                daddi     r14, r0, fs_addr
                syscall   5
                syscall   0
```

格式字符串的地址被放入 R5，其内容随后被保存到内存中的 fs_addr 地址。字符串参数的地址被保存到 s1_addr 和 s2_addr。这两个字符串参数与格式字符串中的两个%s 占位符相匹配。

从内存来看，与占位符相匹配的参数显然是紧跟在格式字符串地址之后存储的：数字与整数参数相匹配，而地址与字符串参数相匹配。在 s1_addr 和 s2_addr 位置，存放着我们要打印的两个字符串的地址，而不是%s 占位符。

示例的执行将显示 SYSCALL 5 如何处理复杂的格式字符串，如存储在 format_str 中的字符串。