MIPS-Like 指令集模拟器

徐逸辰 2018213555

MIPS-Like是基于MIPS32设计的一个玩具指令集。基于MIPS-Like，我在本项目中实现了一个能够运行MIPS-Like汇编代码的模拟器。除此之外，为了编写测试程序的方便，我设计了一个简化的，类似于C的玩具语言——C-Like。并且实现了一个简单的从C-Like编译到MIPS-Like的编译器。在 `examples` 文件夹中，包含三个测试程序，他们分别是：

- `fibo.cl, fibo.mem`：输出斐波那契数列
- `sort.cl, sort.mem`：数组排序
- `primes.cl, primes.mem`：实现了简单的素数筛，输出30以内的素数

其中，`.cl` 文件为C-Like语言的源文件，`.mem` 文件为编译器编译产生的MIPS-Like汇编语言，可以直接装载入虚拟机内存进行模拟运行。

模拟器与编译器实现的代码量加起来近1500行。

样例运行效果

斐波那契数列

```
> ./result/bin/model -s ./examples/fibo.mem
1
1
2
3
5
8
13
21
34
55
```

排序

```
 > ./result/bin/model -s ./examples/sort.mem
132   <-- 首先输出原始数组
42
31
91
1
2     <-- 原始数组结束
1     <-- 输出排序后的数组
2
31
42
91
132   <-- 排序后的数组结束
```

注意在输出中，为了可以看的更清楚，添加了一些注释。代码会先输出排序之前的原始数组，随后输出排序后的有序数组。

素数筛

```
> ./result/bin/model -s ./examples/primes.mem
2
3
5
7
11
13
17
19
23
29
```

MIPS-Like模拟器

MIPS-Like指令集汇编语言

汇编语言的格式如下所示：

```
inst [ op1 [ ',' op2 [ ',' op3 ] ] ]
```

指令的具体实现与含义见 `inst.pdf` ，也即之前提交的报告。

唯一的不同是，为了能够方便地在内存中放置数值，汇编语言中添加了一个关键字：

```
dw val
```

也即，若想在内存中的特定位置放置一个数值，则可以在汇编代码中写：

```
dw 12321
```

模拟器的实现细节

模拟器完整地模拟了所有用户可见寄存器的行为，以及一些必要的功能性寄存器，也即$R_0, R_1, \cdots, R_{31}, HI, LO, SP, PC, SR$。

由于不需要实现虚存，也没有操作系统，在开机时，PC被设置为0，其他寄存器的值也被初始化为0。将制定的内存文件（也即MIPS-Like汇编文件）载入在内存中从0开始的位置。随后开始执行。

`break` 指令会修改SR寄存器中的特定值，在公操作时会监测这一值，若被置位则会停止模拟。

对于 `syscall` ，由于没有操作系统，因而简单地规定R_1为系统调用编号，而$R_2, R_3, \cdots$可以被认为是参数。为了简便，只实现了一个 `syscall` ，调用编号为0，参数只有一个，应当存放在R_2中，系统调用的作用为将R_2中的值当作有符号整数打印出来。

具体的实现可以参考 `src` 文件夹中的源码，由Haskell实现。

C-Like语言编译器

C-Like 语法设计

C-Like为一个大大简化的，类似于C的玩具语言。看一个简单的C-Like程序：

```
var i, j;
mem x = 0, y = 1;

i = x + y
j = x - y
print i
print j
```

这段代码将会打印出

```
1
-1
```

一段C-Like程序由三个部分组成：

1. 寄存器变量定义区，也即代码中的 `var` 部分。这一部分定义最多15个寄存器变量。无法提供初始值。
2. 内存数据定义区，也即代码中的 `mem` 部分。这一部分可以定义无限多个寄存器变量。且必须提供初始值。除了实例中定义一个32位整数之外，也可以定义一个数组：

   ```
   mem arr[] = [1, 2, 3, 4, 5];
   ```

   这里定义的变量会按顺序放入内存中，紧跟在代码之后。
3. 代码定义区。由多条C-Like语句组成。语句之间无需分号。

而C-Like语句也与C一类语言非常类似：

```
// 赋值语句
```

```
x = 1

// 引用赋值语句
&arr + 4 .= 10

// 条件分支
if x < 0 {
  x = 0
} else {
  x = 1
}
// 注意 else 分支不是必须的

// 循环语句
while x < 10 {
  x = x + 1
}

// 输出语句
print x
```

唯一值得一提的是引用赋值语句。这里为了实现的简便，将对变量赋值与引用赋值（也即对某一内存地址赋值）区分开来。

除此以外，C-Like并没有实现对数组元素的下标访问，若要访问某一个数组，需要首先取出数组首元素的地址，随后对地址进行运算：

```
x = *(&arr + i * 4) // C 中的 x = arr[i]
&arr + i * 4 .= y    // C 中的 arr[i] = y
```

下面是具体的文法定义：

```
prog ::=
[ var ident (',' ident)* ';' ]
[ mem mem_entry (',' mem_entry)* ';' ]
stmts

ident ::= ('_' | alpha) (alpha | digit)*
number ::= [ '-' ] digit+
list ::= '[' number (',' number)* ']'
mem_entry ::= ident '=' number | ident '[]' '=' list
expr ::= expr '+' expr
       | expr '-' expr
       | expr '*' expr
       | expr '/' expr
       | expr '&&' expr
       | expr '||' expr
       | '!' expr
       | '-' expr
       | '*' expr
       | '&' ident
       | expr '>' expr
```

```
       | expr '<' expr
       | expr '<=' expr
       | expr '>=' expr
       | expr '==' expr
       | expr '!=' expr
       | ident
       | number
       | '(' expr ')'
stmt ::= assign_stmt
       | ref_assign_stmt
       | while_stmt
       | if_stmt
       | print_stmt
stmts ::= stmt*
assign_stmt ::= ident '=' expr
ref_assign_stmt ::= expr '.=' expr
while_stmt ::= 'while' expr '{' stmts '}'
if_stmt ::= 'if' expr '{' stmts '}' [ 'else' '{' stmts '}' ]
print_stmt ::= 'print' expr
```

编译器的具体实现可以参考 `compiler-src` 中的源码。由Haskell实现。

测试样例的实现

斐波那契数列

```
var a, b, i, t;
mem n = 10;

a = 0
b = 1
i = 0
while i < n {
  print b
  t = a + b
  a = b
  b = t
  i = i + 1
}
```

排序

```
var k, i, x, y, t;
mem a[] = [132, 42, 31, 91, 1, 2],
    len = 6;

// print out the array before sorting
i = 0
while i < len {
  print *(&a + i * 4)
  i = i + 1
```

```
}

// sort the array
k = 0
while k < len {
  i = 0
  while i < len - 1 {
    x = *(&a + i * 4)
    y = *(&a + (i + 1) * 4)
    if y < x {
      t = x
      x = y
      y = t
    }
    &a + i * 4 .= x
    &a + (i + 1) * 4 .= y
    i = i + 1
  }
  k = k + 1
}

// print out the sorted array
i = 0
while i < len {
  print *(&a + i * 4)
  i = i + 1
}
```

这里使用了最简单的冒泡排序。

素数筛

```
var i, t, k;
mem vis[] = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
             1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
             1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    len = 30;

i = 2
while i < len {
  t = *(&vis + i * 4)
  if t {
    k = 2
    while i * k < len {
      &vis + i * k * 4 .= 0
      k = k + 1
    }
  }
  i = i + 1
}

i = 2
```

```
while i < len {
  t = *(&vis + i * 4)
  if t {
    print i
  }
  i = i + 1
}
```