仿真实验一

2112614 刘心源

一、实验内容

本次仿真实验主要任务为编写一个C程序`sim.c`，这是一个针对`MIPS`指令集的有限子集的指令级模拟器，这个指令级的模拟器将对每个指令的行为进行建模并允许用户运行`MIPS`程序并查看其输出。

模拟器将处理一个包含`MIPS`程序的输入文件，输入文件的每一行对应一个写入为十六进制字符串的`MIPS`指令。我们编写的模拟器将执行输入程序，每次只执行一条指令。在每条指令之后，模拟器将修改`MIPS`架构状态，包括存储在寄存器和内存中的值。

模拟器分为两个部分：

- shell，为用户提供命令来控制模拟器的执行；
- sim.c，也就是模拟程序，我们将要实现它～

二、实验步骤

将MIPS指令转换为十六进制指令

由于无法安装使用原有代码文件中自带的`Spim`😭，我在开源平台上找到了`mars4.5.jar`进行指令转换，使用`python`编写相应的脚本文件在命令行中调用`jar`包，将`.s`文件转变为`.x`文件存储。

```
#!/usr/bin/python3

import os
import sys

os.system("java -jar Mars4_5.jar " + sys.argv[1] +" dump .text HexText " + sys.argv[1]
[:-2]+".x")
```

在`inputs`文件夹下使用命令执行脚本文件

```
./asm2hex addiu.s
```

指令处理函数编写

编写全局变量

```
char instruction[400]; //存储输入的指令
int curr_pos = 0; //当前指令的位置
char conversion[100]; //存储转换后的指令
```

编写辅助函数

根据题意，需要将原本的十六进制指令进行译码，因此编写一些辅助函数如下：

1. **uintToStr(uin32_t u)**函数
   - `MIPS`指令通常存储为32位二进制数，该函数将一个32位的无符号整数（代表指令或者内存中的数据）转换为32位二进制字符串，以便于后续的解析和执行。
   - 实现思路：将输入的十六进制数转换为字符串，用填充每一个字符串确保长度为8（也就是保证转换为二进制之后为32位）；建立查找表将十六进制的0～F分别与二进制的字符串对应，遍历字符串将一位十六进制字符转换为4位二进制字符，即完成转换。

2. **convertBin(int length)**函数
   - 在解析指令时，需要从二进制指令字符串中提取各个字段（比如操作码，寄存器编号，操作数等），该函数中当前位置（全局变量`curr_pos`）开始的指定长度（输入参数`length`）的二进制字符串转换为整数。
   - 实现思路：遍历字符串，使用位运算累加计算整数结果。

3. **convertChar(char str,int length,int start)**
   - 该函数类似于`convertBin`函数，但是它可以在指定的开始位置对于输入的字符串进行转换。可以用于处理从内存中读取的数据。

4. **ComplementToBinary(int length)**
   - 在处理有符号数（如立即数、偏移量等）时，需要将补码形式的二进制字符串转换为整数。
   - 实现思路：将`instruction`中的二进制补码字符串（从`curr_pos`开始，长度为`length`）转换为整数，判断首位（符号位）并进行相应的处理。
   - 后续的**int ComplementToBinaryChar(char* str, int length)**与其类似，但是是从给定字符串`str`中读取并转换。
   - 其代码如下：

```
int ComplementToBinary(int length){
    int result = 0;
    if(instruction[curr_pos] == '1'){
        result = convertBin(length) | 0xffff0000;
        //将result的前16位符号扩展为1
        return result;
    }
    result += convertBin(length);
    return result;
}
```

编写主要处理函数

process_instruction()

- 这是模拟器的核心函数，负责读取、解析并执行一条`MIPS`指令。通过借助上述的辅助函数来读取并解析指令，然后根据指令的类型和操作码分别编写执行相应的操作，并更新寄存器和内存的状态。
- 实现准备：

  观察文件`shell.h`,发现`NEXT_STATE`和`CURRENT_STATE`两个结构体保存了两个状态下寄存器堆的状况。

  ```
  typedef struct CPU_State_Struct {

    uint32_t PC;    /* program counter */
    uint32_t REGS[MIPS_REGS]; /* register file. */
    uint32_t HI, LO;          /* special regs for mult/div. */
  } CPU_State;

  /* Data Structure for Latch */

  extern CPU_State CURRENT_STATE, NEXT_STATE;
  ```

  在`shell.c`中，编写了`cycle()`函数，完成将下一个状态赋值给当前状态。

  ```
  void cycle() {
    process_instruction();
    CURRENT_STATE = NEXT_STATE;
    INSTRUCTION_COUNT++;
  }
  ```

  在`process_instruction()`函数执行时只需要将`NEXT_STATE`的`PC`+4即可。
- 实现思路：
  - 读取当前`PC`指向的指令，并转换为二进制字符串存储在`instruction`中；
  - 检测该指令是否为`syscall`

```
if(convertBin(32) == 12){
  if(NEXT_STATE.REGS[2] == 10){
    RUN_BIT = 0;
    return;
  }
}
```

根据实验指导中提到的

> if the register `$v0` (register 2) has value `0x0A` (decimal 10) when `SYSCALL` is executed, then the `go` command should stop its simulation loop and return to the simulator shell's prompt. If `$v0` has any other value, the instruction should have no effect.
>
> The `process_instruction()` function that you write should cause the main simulation loop to terminate by setting the global variable `RUN_BIT` to 0

根据题目要求检测寄存器`$v0`的值，如果是`0x0A`则将`RUN_BIT`置为0并返回；

- 使用`convertBin`函数取出`type`字段分辨指令类型

1.***R型指令***

由于篇幅限制，选取R型指令中的几个进行分析。其余指令与其类似～

```
switch (funct)
{
  case 32:
    //add
    printf("instruction ADD $%d $%d $%d executed\n", rd, rs, rt);
    NEXT_STATE.REGS[rd] = CURRENT_STATE.REGS[rs] + CURRENT_STATE.REGS[rt];
    break;

  case 0:
    //sll
    printf("instruction SLL $%d $%d %d executed\n", rd, rt, shamt);
    NEXT_STATE.REGS[rd] = CURRENT_STATE.REGS[rt] << shamt;
    break;

  case 3:
    // sra
    printf("instruction SRA $%d $%d %d executed\n", rd, rt, shamt);
    uint32_t result = CURRENT_STATE.REGS[rt] >> shamt;
    // 检查原寄存器中数据最高位是否为1
    if (CURRENT_STATE.REGS[rt] & 0x80000000) {
      // 构建一个掩码，将result左边的shamt个位置1，而后面的位保持不变
      uint32_t mask = (1U << (32 - shamt)) - 1;
      result = result | (~mask);
    }
    NEXT_STATE.REGS[rd] = result;

  case 9:
```

```
    // jalr
    printf("instruction JALR $%d $%d executed\n", rd, rs);
    NEXT_STATE.REGS[rd] = CURRENT_STATE.PC;
    NEXT_STATE.PC = CURRENT_STATE.REGS[rs] - 4;

  case 16:
    //mfhi
    printf("instruction MFHI $%d executed\n", rd);
    NEXT_STATE.REGS[rd] = CURRENT_STATE.HI;

  case 24:
    //mult
    printf("instruction MULT $%d $%d executed\n", rs, rt);
    int64_t result2 = (int64_t)CURRENT_STATE.REGS[rs] *
(int64_t)CURRENT_STATE.REGS[rt];
    CURRENT_STATE.HI = result2 >> 32;
    CURRENT_STATE.LO = result2 & 0xffffffff;
    printf("HI: 0x%08X\n", CURRENT_STATE.HI);
    printf("LO: 0x%08X\n", CURRENT_STATE.LO);

  case 42:
    //slt
    printf("instruction SLT $%d $%d $%d executed\n", rd, rs, rt);
    if ((int32_t)CURRENT_STATE.REGS[rs] < (int32_t)CURRENT_STATE.REGS[rt]) {
      NEXT_STATE.REGS[rd] = 1;
    }
    else {
      NEXT_STATE.REGS[rd] = 0;
    }

    break;
  default:
    break;
}
```

`ADD`指令：该指令将当前状态中`rs`和`rt`寄存器里的数值进行相加，并将结果存储到下一状态的`rd`寄存器中；其余类似指令如`SUB`,`AND`,`OR`等也是这样；

`SLL`指令：逻辑左移指令，通过使用位运算符号`<<`，该指令将当前状态下的`rt`寄存器中的数据左移`shamt`位，并将结果存储到下一状态的`rd`寄存器中；

`SRA`指令：算术右移指令，首先对`CURRENT_STATE.REGS[rt]`执行逻辑右移，并将结果存储在`result`中。检查原寄存器中数据的最高位（符号位）是否为1，如果是，通过一个掩码将`result`的左边的`shamt`个位置1，而后面的位保持不变，最后将处理后的`result`赋值给下一个状态的`rd`寄存器。

`SLT`指令：比较`CURRENT_STATE.REGS[rs]`和`CURRENT_STATE.REGS[rt]`，如果前者数值小于后者，则目标寄存器`rd`被设置为1，否则设置为0。`SLTU`指令与其类似。

`JALR`指令：跳转链接指令，常用于函数调用。将当前状态（`CURRENT_STATE`）的程序计数器（`PC`，代表当前指令地址）的值保存到下一个状态（`NEXT_STATE`）的目的寄存器（`REGS[rd]`）中。这通常用于保存返回地址。

`MFHI`指令：将`HI`寄存器的内容复制到一个普通的寄存器中。将当前状态（`CURRENT_STATE`）的HI寄存器的值复制到下一个状态（`NEXT_STATE`）的目的寄存器（`REGS[rd]`）中。

`MULT`指令：指令用于将两个寄存器中的数值相乘，并将64位的结果存储到两个特殊的寄存器：`HI`寄存器和`LO`寄存器。计算源寄存器`rs`和`rt`中的值的乘积，并将结果存储在64位整数`result2`中。将`result2`的高32位存储到`HI`寄存器中，低32位存储到`LO`寄存器中。`&`是位与操作符，`result2 & 0xffffffff`表示取`result2`的低32位。

2.***J型指令***

```
    else if(InstructionType == 2 || InstructionType == 3)
    {
        //J型指令
        int address = convertBin(26) * 4;
        //地址，由于MIPS是基于字的，地址为字对齐，因此*4
        switch (InstructionType)
        {
        case 2:
            //j
            printf("instruction J %d executed\n", address);
            NEXT_STATE.PC =(uint32_t) address - 4;
            break;

        case 3:
            //jal
            printf("instruction JAL %d executed\n", address);
            NEXT_STATE.REGS[31] = CURRENT_STATE.PC;
            NEXT_STATE.PC = address - 4;
            break;

        default:
            break;
        }
    }
```

`J`指令：无条件跳转。执行此指令时，将`address`直接赋值给下一个状态的`PC`寄存器。由于在执行每条指令后，`PC`会自动增加4，以指向下一条要执行的指令，因此实际实现中我们将`address - 4`赋值给`NEXT_STATE.PC`，以确保跳转地址的正确性。

`JAL`指令：`JAL`指令（跳转并链接）与`J`指令类似，也用于无条件跳转，但同时还会将跳转前的下一条指令的地址（即`CURRENT_STATE.PC + 4`）保存到`$ra`寄存器中。这样做的目的是，当函数或子程序执行完毕后，可以通过寄存器`$ra`中保存的地址返回到调用位置，继续执行后续的指令。

3.***I型指令***

```
case 1:
    //bltz
    printf("%d",NEXT_STATE.REGS[rs]);
    if(rt == 0){
      //bltz
      immediate = ComplementToBinary(16);
      if(NEXT_STATE.REGS[rs] & 0x80000000){
        NEXT_STATE.PC += immediate * 4 ;
      }
      printf("instruction BLTZ $%d %d executed\n", rs, immediate);
    }
    else if(rt == 1){
      //bgez
      immediate = ComplementToBinary(16);
      if(!(NEXT_STATE.REGS[rs] & 0x80000000) ){
        NEXT_STATE.PC += immediate * 4;
      }
      printf("instruction BGEZ $%d %d executed\n", rs, immediate);
    }
    else if(rt == 16){
      //bltzal
      immediate = ComplementToBinary(16);
      printf("%s %u","rs val:",NEXT_STATE.REGS[rs]);
      if(NEXT_STATE.REGS[rs] & 0x80000000){

        NEXT_STATE.REGS[31] = CURRENT_STATE.PC;
        NEXT_STATE.PC += immediate<<2 ;
      }
      printf("instruction BLTZAL $%d %d executed\n", rs, immediate);
    }
    else if(rt == 17){
      //bgezal
      immediate = ComplementToBinary(16);
      printf("%s %u","rs val:",NEXT_STATE.REGS[rs]);
      if( !(NEXT_STATE.REGS[rs] & 0x80000000) ){
        NEXT_STATE.REGS[31] = CURRENT_STATE.PC;
        NEXT_STATE.PC += immediate<<2 ;
```

```
      }
      printf("instruction BGEZAL $%d %d executed\n", rs, immediate);
    }
case 4:
    //beq
    immediate = ComplementToBinary(16);
    if(NEXT_STATE.REGS[rs] == CURRENT_STATE.REGS[rt]){
      NEXT_STATE.PC += immediate * 4;
    }
    printf("instruction BEQ $%d $%d %d executed\n", rs, rt, immediate);

case 33:
    //lh
    immediate = ComplementToBinary(16);
    uintToStr(mem_read_32(CURRENT_STATE.REGS[rs] + immediate));
    strncpy(subbuff, conversion, 16);
    NEXT_STATE.REGS[rt] = ComplementToBinaryChar(subbuff,16); //将转换后的指令存入
instruction中
    printf("instruction LH $%d (%d) $%d executed\n", rt, immediate,rs);
    break;

case 41:
    //sh
    immediate = ComplementToBinary(16);
    uintToStr(NEXT_STATE.REGS[rt]);
    strncpy(subbuff, conversion, 16);
    mem_write_32(CURRENT_STATE.REGS[rs] +
immediate,ComplementToBinaryChar(subbuff,16));
    printf("instruction SH $%d (%d) $%d executed\n", rt, immediate,rs);
```

同样由于受篇幅限制，选取几个R型指令进行分析

`BTLZ`&`BTLZAL`指令：注意到其实`BTLZ`&`BTLZAL`&`BGEZ`&`BGEZAL`这四个指令的`option`操作码其实都是000001，所以需要通过`rt`字段来进行区分。

- `BLTZAL`的`rt`字段值为`10001`（十进制的17）
- `BGEZAL`的`rt`字段值为`10000`（十进制的16）
- `BLTZ`指令，`rt`字段的值为`00000`（十进制中的 0）。
- `BGEZ`指令，`rt`字段的值为`00001`（十进制中的 1）。

这几个指令较为类似，选取**BLTZAL**进行解释：这个指令用于测试一个寄存器中的值是否小于零。如果是，则跳转到目标地址，并且将返回地址（当前指令地址加4）存储到链接寄存器`$ra`中。也就是如果寄存器`rs`中的值小于0，则程序将跳转到`PC + offset * 4`的地址，其中`PC`是当前指令的地址。注意到由于寄存器中存储的是**无符号整型**，也就是直接使用`NEXT_STATE.REGS[rs]`是会被翻译为正数，那么不可能进行跳转。因此我们需要加上一个判断：`NEXT_STATE.REGS[rs] & 0x80000000`，这条语句可以对于`NEXT_STATE.REGS[rs]`的最高位进行判断，如果最高位是1则结果为1，也就是该数是负数，进行跳转。

`beq`指令：比较两个寄存器中的值，如果相等，则程序计数器（PC）会跳转到某个新的地址，实现分支。比较两个寄存器 `rs` 和 `rt` 中的值。如果 `rs` 和 `rt` 寄存器中的值相等，那么程序计数器（PC）的值会增加 `immediate * 4`。这里 `* 4` 是因为 MIPS 指令集中的地址是以字节为单位的，而每条指令的长度是 4 字节，所以需要乘以 4。

`lh`指令：用于从内存中加载一个半字（16位）到一个寄存器中。通过`CURRENT_STATE.REGS[rs] + immediate`计算出要读取的内存地址。然后，调用`mem_read_32`函数从该地址读取32位（4字节）的数据，并将其转换为字符串格式。使用`strncpy`函数将`conversion`字符串的前16个字符复制到`subbuff`字符串中将`subbuff`字符串转换为二进制数，并将结果存储在`NEXT_STATE.REGS[rt]`寄存器中，模拟`LH`指令将数据从内存加载到寄存器。

`sh`指令：用于将寄存器中的一个半字（16 位）数据存储到内存中。将 `NEXT_STATE.REGS[rt]` 中的值转换为字符串，将 `conversion` 字符串的前 16 个字符复制到 `subbuff` 字符串中。计算内存地址 `CURRENT_STATE.REGS[rs] + immediate`，然后调用 `ComplementToBinaryChar` 函数将 `subbuff` 转换为二进制数，并将该值写入计算出的内存地址。模拟 `SH` 指令将寄存器中的数据存储到内存。

编译测试

使用指令`src/sim inputs/brtest2.x`打开模拟器，用`betest2.x`作为测试代码得到输出如下：

```
MIPS Simulator

Read 11 words from program into memory.

MIPS-SIM> go

Simulating...

00100100000000100000000000001010
instruction ADDIU $2 $0 10 executed

00001000000100000000000000000010
instruction J 4194312 executed

00010100000000000000000000000011
instruction BNE $0 $0 3 executed

00010000000000000000000000000011
instruction BEQ $0 $0 3 executed

00111100000000010000000000000000
instruction LUI $1 0 executed

00110100001000011101000000001101
instruction ORI $1 $1 53261 executed
```

```
00000000000000010011100000100001
instruction ADDU $7 $0 $1 executed

00000000000000000000000000001100
Simulator halted
```

PS：发现似乎MARS在将MIPS转换为机器码的时候似乎会将`addiu`指令分为多条指令进行执行。比如本代码原本在`BEQ $0 $0 3`之后应当是`addiu $7, $zero, 0xd00d`，但是MARS将这个指令分为三条：`LUI $1 0`,`ORI $1 $1 53261(0x0000d00d)`,`ADDU $7 $0 $1`进行实现。但是结果也是正确的。

发现由于`addiu`是无符号整型加法，但是`0xd00d`是负数，因此可能MARS会对于这种指令进行优化，从而进行符号扩展之后在进行加法。