MIPS 模拟器

指令级 MIPS 模拟器。代码已上传至 GitHub

实现

该模拟器实现的所有指令如下：

J	JAL	BEQ	BNE	BLEZ	BGTZ
ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI
XORI	LUI	LB	LH	LW	LBU
LHU	SB	SH	SW	BLTZ	BGEZ
BLTZAL	BGEZAL	SLL	SRL	SRA	SLLV
SRLV	SRAV	JR	JALR	ADD	DDU
SUB	SUBU	AND	OR	XOR	NOR
SLT	SLTU	MULT	MFHI	MFLO	MTHI
MTLO	MULTU	DIV	DIVU		SYSCALL

首先，使用几个不同的函数实现了一个简单的指令解码的功能。

```
uint32_t extract_op(uint32_t inst) { return inst >> 26; }
uint32_t extract_rs(uint32_t inst) { return (inst >> 21) & 0x1f; }
uint32_t extract_rt(uint32_t inst) { return (inst >> 16) & 0x1f; }
uint32_t extract_rd(uint32_t inst) { return (inst >> 11) & 0x1f; }
uint32_t extract_target(uint32_t inst) { return inst & 0x3ffffff; }
uint32_t extract_imm(uint32_t inst) { return inst & 0xffff; }
uint32_t extract_shamt(uint32_t inst) { return (inst >> 6) & 0x1f; }
uint32_t extract_funct(uint32_t inst) { return inst & 0x3f; }
```

在指令执行前，先提取所有指令中的字段。

```
uint32_t op = extract_op(inst);
uint32_t rs = extract_rs(inst);
uint32_t rt = extract_rt(inst);
uint32_t imm = extract_imm(inst);
uint32_t rd = extract_rd(inst);
uint32_t shamt = extract_shamt(inst);
uint32_t funct = extract_funct(inst);
```

此外，还实现了一些用于符号扩展和零扩展的辅助函数。因为加载指令的种类很多，所以分别实现了对 word，half word 和 byte 的扩展。此处的符号扩展使用指针和类型转换的方式。

```
/// Sign extend the 16 bits immediate
uint32_t sign_ext(uint32_t imm) {
    int32_t signed_imm = *((int16_t*)&imm);
    uint32_t extended_imm = *((uint32_t*)&signed_imm);
    return extended_imm;
}

/// Sign extend a byte to 32 bits
uint32_t sign_ext_byte(uint8_t imm) {
    int32_t signed_imm = *((int8_t*)&imm);
    uint32_t extended_imm = *((uint32_t*)&signed_imm);
    return extended_imm;
}

uint32_t sign_ext_half(uint16_t imm) {
    int32_t signed_imm = *((int16_t*)&imm);
    uint32_t extended_imm = *((uint32_t*)&signed_imm);
    return extended_imm;
}

uint32_t zero_ext(uint32_t imm) { return imm; }

uint32_t zero_ext_byte(uint8_t imm) { return imm; }

uint32_t zero_ext_half(uint16_t imm) { return imm; }
```

在 `process_instruction` 中，首先从内存中根据程序计数器的值来获取一条指令。

```
uint32_t inst = mem_read_32(CURRENT_STATE.PC);
```

之后使用一整个 `switch` 语句根据 `op` 来处理所有不同的指令类型。如果需要， `funct` 也会 `switch` 处理。

在分支指令中，由于模拟器并不实现延迟槽的功能，每一次分支时程序计数器也都需要额外加四。

```
// BLEZ
uint32_t offset = sign_ext(imm) << 2;

if (rt == 0) {
    if ((CURRENT_STATE.REGS[rs] & 0x80000000) != 0 ||
        CURRENT_STATE.REGS[rs] == 0) {
        NEXT_STATE.PC = CURRENT_STATE.PC + offset + 4;
    } else {
        NEXT_STATE.PC = CURRENT_STATE.PC + 4;
    }
} else {
    // Illegal instruction
    printf("Illegal rt in BLEZ.\n");
}
break;
```

若分支没有执行则程序计数器仍然照常加四。

其余算数运算、位运算和跳转指令按照 ISA 手册进行编写，具体实现可参见代码。

访存指令中使用 `mem_read_32` 读内存，`mem_write_32` 写内存，对于写入小于 32 位数的指令（如 `sb`, `sh`）采用先读取内存内容并进行拼接的方式。

```
// SB

uint32_t addr = sign_ext(imm) + CURRENT_STATE.REGS[rs];

uint32_t val = (mem_read_32(addr) & 0xffffff00) |
               (CURRENT_STATE.REGS[rt] & 0xff);

mem_write_32(addr, val);
NEXT_STATE.PC = CURRENT_STATE.PC + 4;
break;
```

此外，在每个指令执行之后将 PC 加 4.

```
NEXT_STATE.PC = CURRENT_STATE.PC + 4;
```

测试

由于 Spim 存在一些 Bug 并且最新版本对伪指令的支持不好，所以使用 mars 来进行汇编和测试。使用 spim 进行汇编的脚本在 `tools` 文件夹下，但是注意伪指令（比如大立即数）在 spim 中不支持。同时，mars 这个工具也在 `tools` 文件夹下。由于没有找到通过命令行直接操作 mars 的方法，所以采用了手动汇编的方式生成了 `.x` 文件。测试时直接对比 sim 寄存器的输出和 mars 运行结果。

额外的测试用例在 `tests` 文件夹下。经过测试发现模拟器的输出结果和 mars 的输出结果相同。