实验报告

于新雨 计25 2022010841

实验代码

1. 求前10个斐波那契数 并且保存到起始地址为 0x80400000 的 10 个字中

```
        li      a5,-2143289344
        li      a4,1
        sw      a4,0(a5)
        sw      a4,4(a5)
        addi    a5,a5,4
        li      a2,-2143289344
        addi    a2,a2,36
.L1:
        lw      a4,0(a5)
        lw      a3,-4(a5)
        addi    a5,a5,4
        add     a4,a4,a3
        sw      a4,0(a5)
        bne     a5,a2,.L1
        jr      ra
```

过程截图

```
>> a
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] li      a5,-2143289344
[0x80100004] li      a4,1
[0x80100008] sw      a4,0(a5)
[0x8010000c] sw      a4,4(a5)
[0x80100010] addi    a5,a5,4
[0x80100014] li      a2,-2143289344
[0x80100018] addi    a2,a2,36
[0x8010001c] .L2
/tmp/tmpqg1gcwgq: Assembler messages:
/tmp/tmpqg1gcwgq:1: Error: unknown pseudo-op: `.l2'
b''
[0x8010001c] .L1:
[0x8010001c] lw      a4,0(a5)
[0x80100020] lw      a3,-4(a5)
[0x80100024] addi    a5,a5,4
[0x80100028] add     a4,a4,a3
[0x8010002c] sw      a4,0(a5)
[0x80100030] bne     a5,a2,.L1
[0x80100034] jr      ra
[0x80100038]
>> g
addr: 0x80100000

elapsed time: 0.000s
```

```
>> d
addr: 0x80400000
num: 40
0x80400000: 0x00000001
0x80400004: 0x00000001
0x80400008: 0x00000002
0x8040000c: 0x00000003
0x80400010: 0x00000005
0x80400014: 0x00000008
0x80400018: 0x0000000d
0x8040001c: 0x00000015
0x80400020: 0x00000022
0x80400024: 0x00000037
```

```
>> u
addr: 0x80100000
num: 32
0x80100000:     804007b7        lui     a5,0x80400
0x80100004:     00100713        li      a4,1
0x80100008:     00e7a023        sw      a4,0(a5)
0x8010000c:     00e7a223        sw      a4,4(a5)
0x80100010:     00478793        addi    a5,a5,4
0x80100014:     80400637        lui     a2,0x80400
0x80100018:     02460613        addi    a2,a2,36
0x8010001c:     0007a703        lw      a4,0(a5)
>>
```

2. 打印所有可打印字符

```
main:
        addi    sp,sp,-16
        sw      ra,12(sp)
        sw      s0,8(sp)
        sw      s1,4(sp)
        sw      s2,0(sp)
        li      s0,33
        li      s1,127
.L2:
        mv      a0,s0

WRITE_SERIAL:
```

```
        // 串口地址是 0x10000000
        li t0, 0x10000000

        // 轮询串口状态（0x10000005）
.TESTW:
        lb t1, 5(t0)
        // 判断是否可写
        andi t1, t1, 0x20
        beq t1, zero, .TESTW

.WSERIAL:
        // 向终端（0x10000000）输出 a0 寄存器中的最低字节
        sb a0, 0(t0)
        addi    s0,s0,1
        bne     s0,s1,.L2       # here the loop goes...
        li      a0,0
        lw      ra,12(sp)
        lw      s0,8(sp)
        lw      s1,4(sp)
        lw      s2,0(sp)
        addi    sp,sp,16
        jr      ra
```

过程截图

```
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] addi    sp,sp,-16
[0x80100004] sw      ra,12(sp)
[0x80100008] sw      s0,8(sp)
[0x8010000c] sw      s1,4(sp)
[0x80100010] sw      s2,0(sp)
[0x80100014] li      s0,33
[0x80100018] li      s1,127
[0x8010001c] .L2:
[0x8010001c] mv      a0,s0
[0x80100020] WRITE_SERIAL:
[0x80100020] li t0, 0x10000000
[0x80100024] .TESTW:
[0x80100024] lb t1, 5(t0)
[0x80100028] andi t1, t1, 0x20
[0x8010002c] beq t1, zero, .TESTW
[0x80100030] .WSERIAL:
[0x80100030] sb a0, 0(t0)
[0x80100034] addi    s0,s0,1
[0x80100038] bne     s0,s1,.L2
[0x8010003c] li      a0,0
[0x80100040] lw      ra,12(sp)
[0x80100044] lw      s0,8(sp)
[0x80100048] lw      s1,4(sp)
[0x8010004c] lw      s2,0(sp)
[0x80100050] addi    sp,sp,16
[0x80100054] jr      ra
```

```
>> u
addr: 0x80100000
num: 168
0x80100000:     ff010113        addi    sp,sp,-16
0x80100004:     00112623        sw      ra,12(sp)
0x80100008:     00812423        sw      s0,8(sp)
0x8010000c:     00912223        sw      s1,4(sp)
0x80100010:     01212023        sw      s2,0(sp)
0x80100014:     02100413        li      s0,33
0x80100018:     07f00493        li      s1,127
0x8010001c:     00040513        mv      a0,s0
0x80100020:     100002b7        lui     t0,0x10000
0x80100024:     00528303        lb      t1,5(t0)
0x80100028:     02037313        andi    t1,t1,32
0x8010002c:     fe030ce3        beqz    t1,0x80100024
0x80100030:     00a28023        sb      a0,0(t0)
0x80100034:     00140413        addi    s0,s0,1
0x80100038:     fe9412e3        bne     s0,s1,0x8010001c
0x8010003c:     00000513        li      a0,0
0x80100040:     00c12083        lw      ra,12(sp)
0x80100044:     00812403        lw      s0,8(sp)
0x80100048:     00412483        lw      s1,4(sp)
0x8010004c:     00012903        lw      s2,0(sp)
0x80100050:     01010113        addi    sp,sp,16
0x80100054:     00008067        ret
0x80100058:     00000000        ...
```

```
>> g
addr: 0x80100000
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
elapsed time: 0.000s
```

3. 求fibonacci数列的第项60

```
main:
        li      a2,58
        li      a5,1
        li      a4,0
        li      a6,1
        li      a0,0
.L2:
        mv      t1,a5
        mv      a7,a4
        add     a3,a5,a6
        sltu    a5,a3,a5
        add     a4,a4,a0
        add     a1,a5,a4
        mv      a5,a3
        mv      a4,a1
```

```
        addi    a2,a2,-1
        mv      a6,t1
        mv      a0,a7
        bne     a2,zero,.L2
        li      a5,-2143289344  # -0x7fc00000
        sw      a3,0(a5)
        sw      a1,4(a5)
        li      a0,0
        ret
```

因为感觉之前第一个的汇编略麻烦，所以这个写法的思路是用的以下算法

```
    long long a=1,b=1;
    for(int i=2;i<60;i++){
        b=a+b;
        a=b-a;
    }
    return b;
```

截图如下

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] li      a2,58
[0x80100004] li      a5,1
[0x80100008] li      a4,0
[0x8010000c] li      a6,1
[0x80100010] li      a0,0
[0x80100014] .L2:
[0x80100014] mv      t1,a5
[0x80100018] mv      a7,a4
[0x8010001c] add     a3,a5,a6
[0x80100020] sltu    a5,a3,a5
[0x80100024] add     a4,a4,a0
[0x80100028] add     a1,a5,a4
[0x8010002c] mv      a5,a3
[0x80100030] mv      a4,a1
[0x80100034] addi    a2,a2,-1
[0x80100038] mv      a6,t1
[0x8010003c] mv      a0,a7
[0x80100040] bne     a2,zero,.L2
[0x80100044] li      a5,-2143289344
[0x80100048] sw      a3,0(a5)
[0x8010004c] sw      a1,4(a5)
[0x80100050] li      a0,0
[0x80100054] jr ra
[0x80100058]
```

```
>> u
addr: 0x80100000
num: 88
0x80100000:     03a00613        li      a2,58
0x80100004:     00100793        li      a5,1
0x80100008:     00000713        li      a4,0
0x8010000c:     00100813        li      a6,1
0x80100010:     00000513        li      a0,0
0x80100014:     00078313        mv      t1,a5
0x80100018:     00070893        mv      a7,a4
0x8010001c:     010786b3        add     a3,a5,a6
0x80100020:     00f6b7b3        sltu    a5,a3,a5
0x80100024:     00a70733        add     a4,a4,a0
0x80100028:     00e785b3        add     a1,a5,a4
0x8010002c:     00068793        mv      a5,a3
0x80100030:     00058713        mv      a4,a1
0x80100034:     fff60613        addi    a2,a2,-1
0x80100038:     00030813        mv      a6,t1
0x8010003c:     00088513        mv      a0,a7
0x80100040:     fc061ae3        bnez    a2,0x80100014
0x80100044:     804007b7        lui     a5,0x80400
0x80100048:     00d7a023        sw      a3,0(a5)
0x8010004c:     00b7a223        sw      a1,4(a5)
0x80100050:     00000513        li      a0,0
0x80100054:     00008067        ret
```

```
>> d
addr: 0x80400000
num: 32
0x80400000: 0x6c8312d0
0x80400004: 0x00000168
0x80400008: 0x00000000
0x8040000c: 0x00000000
0x80400010: 0x00000000
0x80400014: 0x00000000
0x80400018: 0x00000000
0x8040001c: 0x00000000
```

思考题

1. 比较 RISC-V 指令寻址方法与 x86 指令寻址方法的异同

    - 相同点：
        - 都支持立即数寻址，即是在指令中直接包含立即数 如 `addi a0,a0,1`
        - 都支持寄存器寻址，即是在指令中直接包含寄存器 如 `add a0,a0,a1`
        - 支持寄存器相对寻址，如 `lw a0, 12(a1)`
        - 支持 pc 相对寻址
    - 不同点：
        - x86/x64 指令支持更多种寻址方式，如直接寻址 `mov rax,[401234h]`，间接寻址（虽然感觉和寄存器相对寻址比较像）如 `mov rax,[rbx]`，基址+变址+offset 的寻址 如 `mov rax,[rbx+rcx*8+4]`

2. 可以有多种不同的分类方式，比如按照操作数个数分类，按照指令类型分类这样，以下将给出一种按照指令类型的分类方式，原因在指令类型的括号中给出

```
arithmetic instructions（代数运算相关）:
ADD, ADDI, AND, ANDI, OR, ORI, SLLI, SRLI, XOR
memory instructions（内存操作相关）:
LW, SW, LB, SB
branch instructions（跳转指令）:
BEQ, BNE, JAL, JALR
control instructions（寻址相关）:
LUI, AUIPC
```

按照操作数分类

```
RTri instructions (接收3个寄存器参数):
ADD, AND, OR,  XOR
ITri instructions (接收2个寄存器参数和一个 Value):
ADDI, ANDI, ORI, SLLI, SRLI
IBin instructions (接收一个寄存器参数和一个 Value):
LW, SW, LB, SB, LUI, AUIPC
Branch instructions:
BEQ, BNE, JAL, JALR
```

这里的 Value 定义为

```
pub enum Value {
    RiscvNumber(RiscvNumber),
    LongLong(i64),
    VirtMem(VirtAddr),
    Label(Label),
    OffsetReg(RiscvNumber, RiscvReg),
}
#[derive(Clone, Hash, PartialEq, Eq)]
pub enum RiscvNumber {
    Lo(Label),
```

```
    Hi(Label),
    Int(i32),
}
```

3. 结合 term 源代码和 kernel 源代码说明 term 是如何实现用户程序计时的

- term 里面是用 time_it 的 default_timer 计时，通过看输出的字符是什么，如果是我们预留的特殊字符的话就判断用户程序完成运行，算出时间间隔
- kernel 里面有下列几步操作：
- 在结束的时候进入 `USERRET_TIMEOUT`（超时的情况）或 `USERRET2`（正常终止）分别是向串口写入 "\x07" "\x81" 两个特殊字符 而这会被 term 识别为用户程序结束
- 而上面所述超时的情况是触发异常进入 trap.S 中的 `EXCEPTION_HANDLER` 根据 mcause 进一步判断是超时，在 `.HANDLE_TIMER` 中进入 `USERRET_TIMEOUT` 状态

4. 说明 kernel 是如何使用串口的

- 相关代码位于 utils.S 中，实现了 `WRITE_SERIAL` `WRITE_SERIAL_WORD` `WRITE_SERIAL_XLEN` `WRITE_SERIAL_STRING` `READ_SERIAL` `READ_SERIAL_WORD` `READ_SERIAL_XLEN` 等函数

接口	功能	实现思路
WRITE_SERIAL	向串口输出一个字节（a0 低八位）	1. 串口位于 0x1000000 的位置，我们先求取它关于写的那一位，判断它是否为0 2. 如果是0则代表串口不可写 继续忙等待 3. 否则串口可写，将数据写入串口的发送保持寄存器
WRITE_SERIAL_WORD	写入32位数	将 32 位数据分为4段8位数据，依次调用 WRITE_SERIAL 将每个字节写入串口
WRITE_SERIAL_XLEN	写入 XLEN 位数 (XLEN 的长度类似于对应架构的 size_t 的字节数)	如果是 RV64 ，则分高4字节，低4字节调用两次 WRITE_SERIAL_WORD， 否则调用 WRITE_SERIAL_WORD
WRITE_SERIAL_STRING	写入字符串	通过循环调用 WRITE_SERIAL 逐个写入字符串的每个字符，直到遇到 '\0' 代表字符串终止
READ_SERIAL	读取一个字节	1. 串口位于 0x1000000 的位置，我们先求取它关于读的那一位，判断它是否为0 2. 如果是0则代表串口不可读 继续忙等待 3. 否则串口可读，将数据读出串口的接收保持寄存器
READ_SERIAL_WORD	读取32位数	通过循环调用 READ_SERIAL 读取4个字节，再从高到低通过3次左移+存低八位的组合合成一个32位数
READ_SERIAL_XLEN	读取 XLEN 位数	如果是 RV64 ，则分高4字节，低4字节调用两次 READ_SERIAL_WORD， 否则调用 READ_SERIAL_WORD

5. term 如何检查 kernel 已经正确连入，并分别指出检查代码在 term 与 kernel 源码中的位置。

- term: 首先从输入读取一个字节，如果是 "\x06" 则代表 kernel 连入成功，否则如果是 trap 的信号 "\x80" 则代表产生错误，其他信号代表连入不成功
- kernel: 写开始计时信号"\x06" 告诉终端已开始计时，代表连入成功
- 代码在 kernel 的 `shell.S` 中 172-175 行，在 `term.py` 中 356-360 行