实验目的

- 理解RISC-V常用32位整数指令功能
- 熟悉RISC-V汇编仿真软件RARS，掌握程序调试的基本方法
- 设计用于生成斐波那契数列的RISC-V汇编程序设计，以及存储器初始化文件(COE)的生成方法
- 理解CPU调试模块PDU的使用方法

移位寄存器

代码

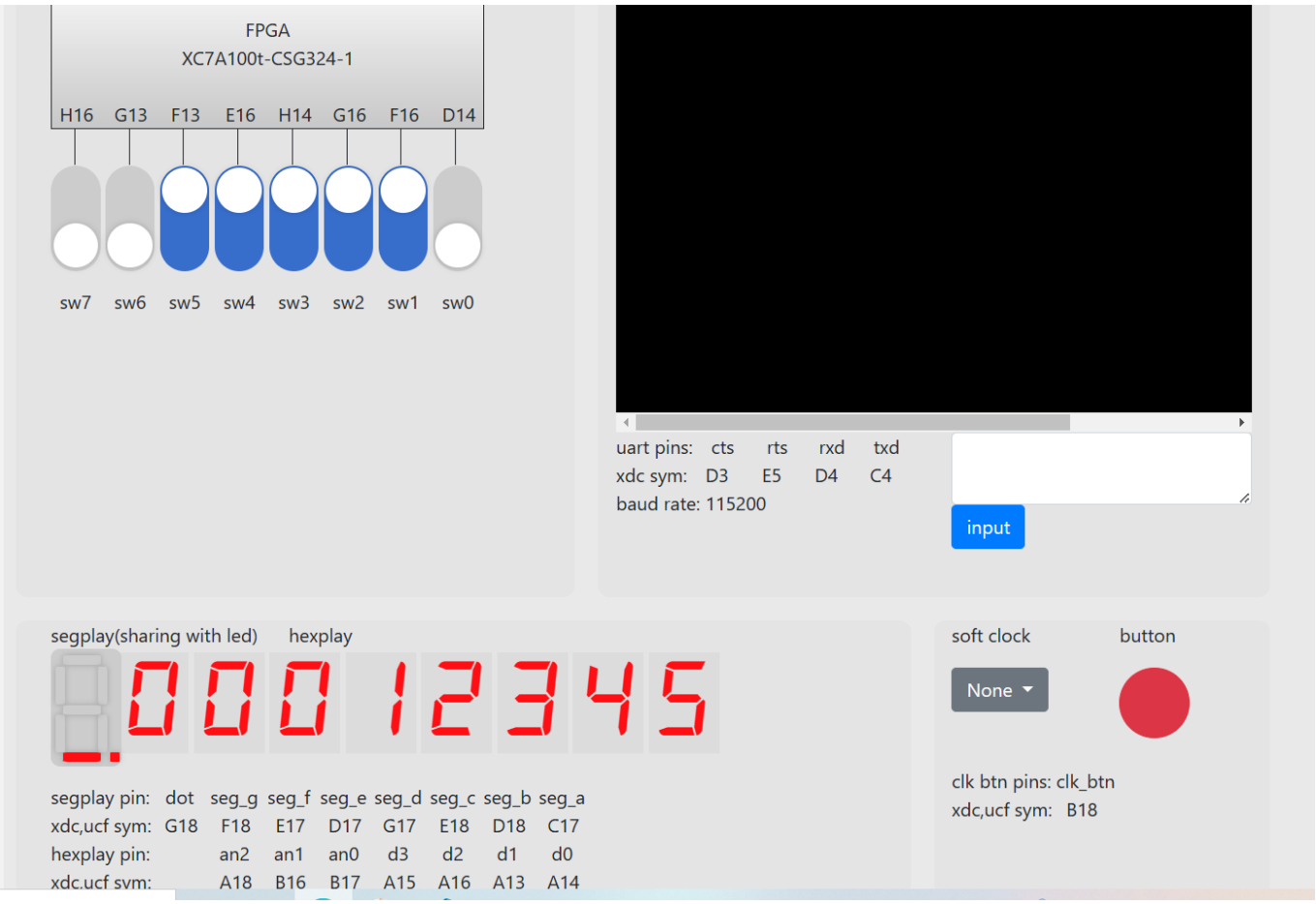
Verilog

```
module Shift_reg(
    input rst,
    input clk,          // Work at 100MHz clock

    input [31:0] din,   // Data input
    input [3:0] hex,    // Hexadecimal code for the switches
    input add,          // Add signal
    input del,          // Delete signal
    input set,          // Set signal

    output reg [31:0] dout  // Data output
);

    // TODO
    reg [31:0] dout_mid;
    always @(posedge clk) begin
        if(rst) dout_mid <= 0;
        else if(set) dout_mid <=  din;
        else if(add) begin
            dout_mid <= {dout_mid[27:0], hex};
        end
        else if(del) begin
            dout_mid <= {4'b0, dout_mid[31:4]};
        end
```

```
25          else
26              dout_mid <= dout_mid;
27      end
28      always @(*) begin
29          dout <= dout_mid;
30      end
31  endmodule
```

使用位拼接实现移位。

上板结果



与演示视频一致。

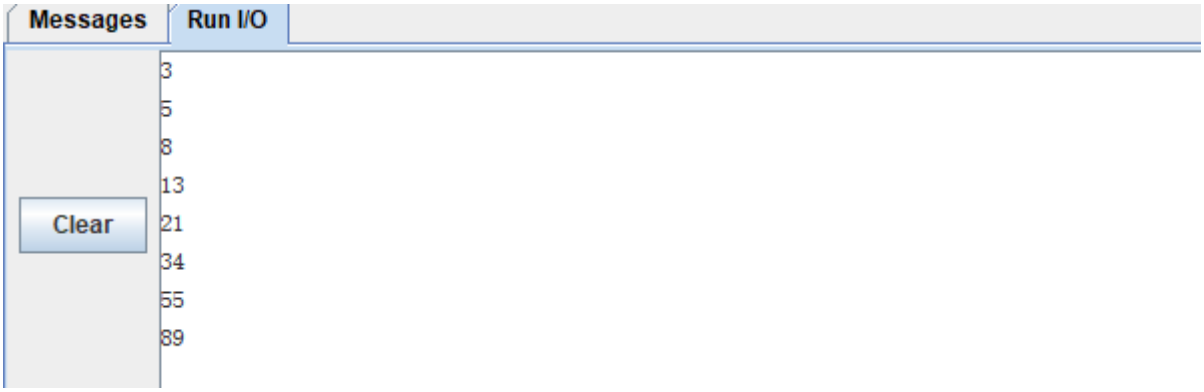
计算斐波那契-卢卡斯数列的代码

```asm
1   .data
2   elem1: .word 0x0001
3   elem2: .word 0x0001
```

```
4   .text
5   #load
6     lw a1, elem1
7     lw a2, elem2
8     addi a4, x0, 10 #a4 holds the number n
9   #print elem1, elem2
10    addi a3, a1, 0
11    jal print
12    addi a3, a2, 0
13    jal print
14    addi a4, a4, -1
15  loop:
16    add a3, a1, a2
17    jal print
18  #update elem1, elem2
19    addi a1, a2, 0
20    addi a2, a3, 0
21  #update n
22    addi a4, a4, -1
23  #loop
24    blt zero, a4, loop
25    j exit
26  print:#print a2
27    addi a0, a3, 0
28    addi a7, x0, 1
29    ecall
30    #newLine
31    addi a0, x0, 10
32    addi a7, x0, 11
33    ecall
34    ret
35  exit:
36    addi     a7, x0, 10
37    ecall
```

`.data`段存储第一个和第二个元素，`.text`
段先读取第一个和第二个元素，调用`ecall`输出，随后进入循环，计算下一项的值。输出，更新前两项的值。

最终结果如图：



外设输入和输出

完整代码贴在最后，此处分段说明。

数据段

```Nasm
.data
kb_ready_addr:  .word 0x00007f00
kb_register_addr:       .word 0x00007f04
display_register_addr:  .word 0x00007f0c
elem1: .word 0x0001
elem2: .word 0x0001
string: .string "\n"
```

输入轮询

```Nasm
setup:
    # s5 <- kb ready bit addr
        # s6 <- kb register addr
    # s0 <- ';'
        la      t0, kb_ready_addr
        lw      s5, 0(t0)
        la      t0, kb_register_addr
        lw      s6, 0(t0)
    li  s0, 59
    li  s1, 10

wait_for_ready:
        # while(ready_bit == 0);
```

```
14          lw      t0, 0(s5)
15          beq     t0, zero, wait_for_ready
16
17  ready:
18          lw      t0, 0(s6)
19      bge t0, s0, OK
20      addi    t0, t0, -48
21      mul a4, a4, s1
22      add a4, a4, t0
23      j wait_for_ready
24  OK:
25      jal FLS
```

在wait_for_ready段，循环检测kb_ready_addr是否为1，是则跳到ready段处理输入。

在ready段，将输入减去'0'，拼接到a4中，回到wait_for_ready等待下一个输入。

当输入';'时，跳到FLS处理，结束轮询。

斐波那契数列生成

Asm

```
1       lw a1, elem1
2       lw a2, elem2
3
4       addi a4, a4, -2
5       blez    a4, exit
6       addi a0, a1, 0
7       jal s4, output
8       addi a0, a2, 0
9       jal s4, output
10  loop0:
11      add a0, a1, a2
12      jal s4, output
13
14      addi a1, a2, 0
15      addi a2, a0, 0
16
17      addi a4, a4, -1
18
19
```

```
20      bgtz a4, loop0
        j exit
```

与之前类似，只是输出从ecall变为调用output段。此外，还需要检测输入的n是否大于2.

输出

输出的基本思路是先用sw指令将数据存入.data段里的string，随后用lbu指令每次读取一个二位16进制数，处理后输出。

21-24行将二位16进制数分为两个一位16进制数，分别存入t4和t5。

40-45行分别输出两位。

46-52行判断是否输出结束。

27-38行用于消除无效位数的0.当s11为0时，不断循环，直到碰到第一个非0的数时，将s11中的判断变量置1。

print段判断输出0-9还是a-f，并将其处理为相应的ASCII码。

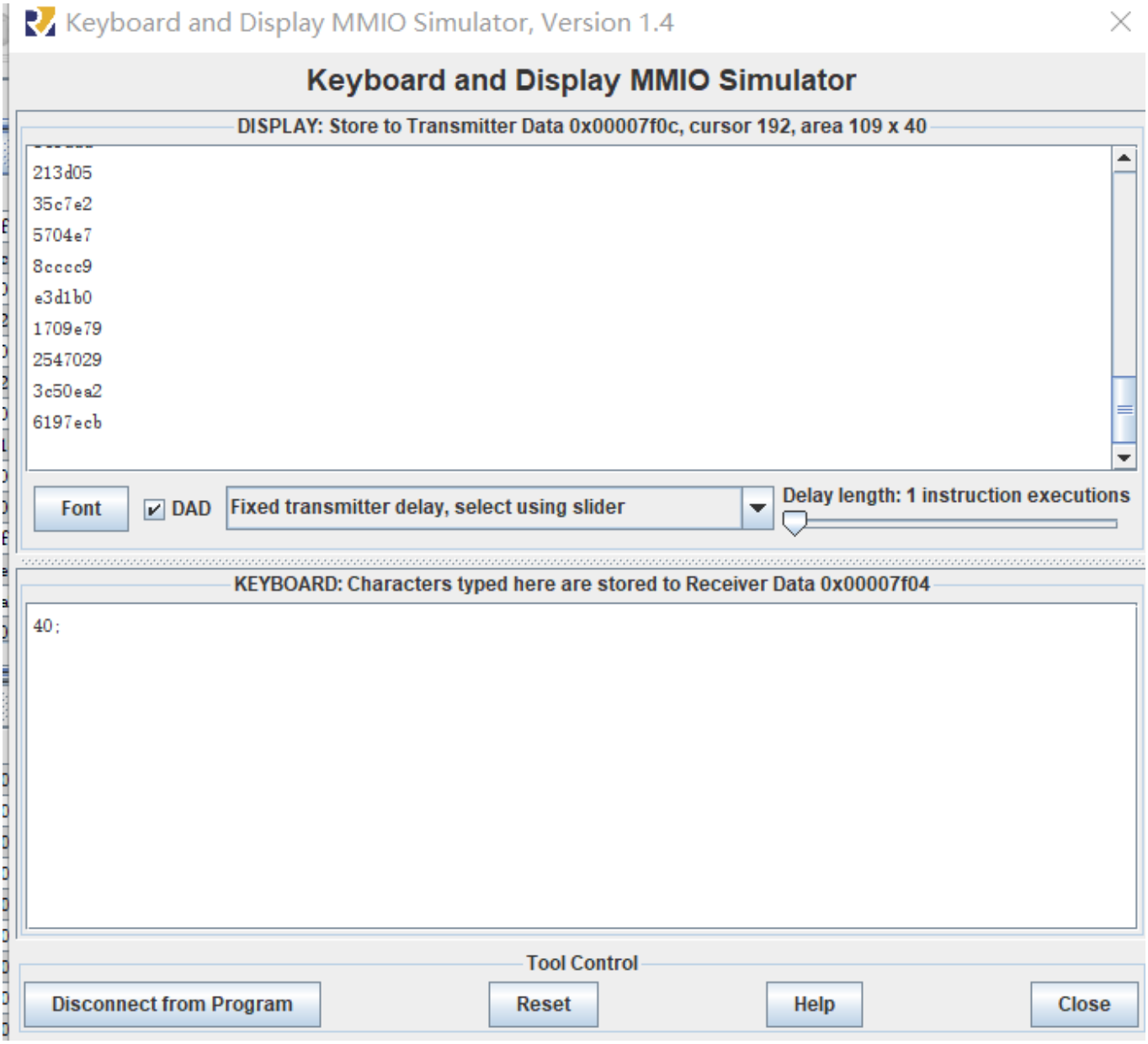
Nasm

```
1   output:
2       # a0 <- src
3       # t0 <- string address
4       # t1 <- number of bit
5       # t2 <- pointer
6       # s1 <- display_register_addr\
7       # s2 <- 16
8       # s3 <- 10
9       # s4 <- return Address
10      # s11 <- judge
11      li  s11, 0
12      li  s3, 10
13      li  s2, 16
14      la  t0, display_register_addr
15      lw  s1, 0(t0)
16      la  t0, string
17      sw  a0, 0(t0)
18      li  t1, 3
19      add t0, t0, t1
20  loop1:
21      lbu t3, 0(t0)
```

```
22      divu    t4, t3, s2
23      mul t5, t4, s2
24      sub t5, t3, t5
25      #
26      bgtz    s11, printHigh
27      Judge:
28      beqz t4, judgeLow
29      addi s11, x0, 1
30      j printHigh
31      judgeLow:
32      beqz t5, notPrint
33      addi s11, x0, 1
34      j printLow
35      notPrint:
36      addi    t0, t0, -1
37      addi    t1, t1, -1
38      beqz s11, loop1
39
40      printHigh:
41      addi    t3, t4, 0
42      jal print
43      printLow:
44      addi    t3, t5, 0
45      jal print
46      addi    t0, t0, -1
47      addi    t1, t1, -1
48      bgez    t1, loop1
49      #finish output
50      addi    t3, x0, 10
51      sw  t3, 0(s1)
52      jr  s4
53  print:
54      bge t3, s3, ge9 # if t3 >9 then target
55      addi    t3, t3, 48
56      j return  # jump to return
57  ge9:
58      addi    t3, t3, 87
59  return:
60      sw  t3, 0(s1)
61      ret
```

结果

40位结果如下：



总结

- 本次实验难度适中。
- 使用外设输出时，记得将delay length设为1，以及设置memory configuration。

外设输入的完整代码

Nasm

```
.data
kb_ready_addr:  .word 0x00007f00
kb_register_addr:       .word 0x00007f04
display_register_addr:  .word 0x00007f0c
elem1: .word 0x0001
elem2: .word 0x0001
string: .string "\n"
.text
setup:
    # s5 <- kb ready bit addr
        # s6 <- kb register addr
    # s0 <- ';'
        la      t0, kb_ready_addr
        lw      s5, 0(t0)
        la      t0, kb_register_addr
        lw      s6, 0(t0)
    li  s0, 59
    li  s1, 10

wait_for_ready:
        # while(ready_bit == 0);
        lw      t0, 0(s5)
        beq     t0, zero, wait_for_ready

ready:
        lw      t0, 0(s6)
    bge t0, s0, OK
    addi    t0, t0, -48
    mul a4, a4, s1
    add a4, a4, t0
    j wait_for_ready
OK:
    jal FLS

FLS:
    #load
    lw a1, elem1
    lw a2, elem2
    #print elem1, elem2
```

```
40      addi a4, a4, -2
41      blez    a4, exit
42      addi a0, a1, 0
43      jal s4, output
44      addi a0, a2, 0
45      jal s4, output
46  loop0:
47      add a0, a1, a2
48      jal s4, output
49      #update elem1, elem2
50      addi a1, a2, 0
51      addi a2, a0, 0
52      #update n
53      addi a4, a4, -1
54      #loop
55      bgtz a4, loop0
56      j exit
57  output:
58      # a0 <- src
59      # t0 <- string address
60      # t1 <- number of bit
61      # t2 <- pointer
62      # s1 <- display_register_addr\
63      # s2 <- 16
64      # s3 <- 10
65      # s4 <- return Address
66      # s11 <- judge
67      li  s11, 0
68      li  s3, 10
69      li  s2, 16
70      la  t0, display_register_addr
71      lw  s1, 0(t0)
72      la  t0, string
73      sw  a0, 0(t0)
74      li  t1, 3
75      add t0, t0, t1
76  loop1:
77      lbu t3, 0(t0)
78      divu    t4, t3, s2
79      mul t5, t4, s2
```

```
80      sub t5, t3, t5
81      #
82      bgtz    s11, printHigh
83      Judge:
84      beqz t4, judgeLow
85      addi s11, x0, 1
86      j printHigh
87      judgeLow:
88      beqz t5, notPrint
89      addi s11, x0, 1
90      j printLow
91      notPrint:
92      addi    t0, t0, -1
93      addi    t1, t1, -1
94      beqz s11, loop1
95
96      printHigh:
97      addi    t3, t4, 0
98      jal print
99      printLow:
100     addi    t3, t5, 0
101     jal print
102     addi    t0, t0, -1
103     addi    t1, t1, -1
104     bgez    t1, loop1
105     #finish output
106     addi    t3, x0, 10
107     sw  t3, 0(s1)
108     jr  s4
109 print:
110     bge t3, s3, ge9 # if t3 >9 then target
111     addi    t3, t3, 48
112     j return  # jump to return
113 ge9:
114     addi    t3, t3, 87
115 return:
116     sw  t3, 0(s1)
117     ret
118 exit:
119
```

```
120       addi        a7, x0, 10
          ecall
```