1. Assume we have an array in memory that contains **`int* arr = {1,2,3,4,5,6,0}`**. Let the values of arr be a multiple of 4 and stored in register s0. What do the snippets 片段 of RISC-V code do? Assume that all the instructions are run one after the other in the same context. (15 points)

   a) **`lw t0, 12(s0) --> 读s0[12] 到t0,也就是t 0 =4`**

   b)
   ```
   slli t1, t0, 2   t1 = t0 <<2  t1 = 16;
   add t2, s0, t1    t2 = s0+ t1,
   lw t3, 0(t2) --> t3 = t2[0]  t3=  5;
   addi t3, t3, 1    t3++;
   sw t3, 0(t2)      t2[0] = t3; 也就是让数组中5变为6
   ```

   c)
   ```
   lw t0, 0(s0)    t0 = 1
   xori t0, t0, 0xFFF -->  t0 = 0000 1111 1111 1110
   addi t0, t0, 1  #t0 =  0000 1111 1111 1111
   ```

   d)

[inst]	[destination register] [argument register 1] [argument register 2]
add	Adds the two argument registers and stores in destination register
xor	Exclusive or's the two argument registers and stores in destination register
mul	Multiplies the two argument registers and stores in destination register
sll	Logical left shifts AR1 by AR2 and stores in DR
srl	Logical right shifts AR1 by AR2 and stores in DR
sra	Arithmetic right shifts AR1 by AR2 and stores in DR
slt/u	If AR1 < AR2, stores 1 in DR, otherwise stores 0, u does unsigned comparison
[inst]	[register] [offset]([register with base address])
sw	Stores the contents of the register to the address+offset in memory
lw	Takes the contents of address+offset in memory and stores in the register
[inst]	[argument register 1] [argument register 2] [label]
beq	If AR1 == AR2, moves to label
bne	If AR1 != AR2, moves to label
[inst]	[destination register] [label]
jal	Stores the current instruction's address into DR and moves to label

   You may also see that there is an "i" at the end of certain instructions, such as addi, slli, etc. This means that AR2 becomes an "immediate" or an integer instead of using a register.

   While only using the instructions (and their "i" forms 就是加 i) given above, how can we branch on the following conditions:

`s0 < s1,`

```
slt t0,s0,s1

beq t0,x0
```

`s0 ≥ s1`

```
slt t0,s1,s0

beq t0,x0
```

s0 > 1

slti t0,s0,1 # s0>1 存 0

beq t0,x0 #如果是 0 就 br

2. Write a function quadruple 四倍 in RISC-V that, when given an integer x, returns 4x. (6 points)

Quadruple

slli t1, t0, 2

3. Write a function power in RISC-V that takes in two numbers x and n, and returns x^n. You may assume that n≥0 and that multiplication will always result in a 32-bit number. (8 points)

N ==1 , 返回 x, n ==2 return x^2

T =1;// x 存在 t 中

While (n >= 1) {

t = t * x;

n--;

}

Return t;

```
pow: addi sp,sp,-8 # adjust stack for 2 items
      sw s1, 4(sp)  # save s1 for use afterwards
      sw s0, 0(sp)  # save s0 for use afterwards

      addi s0,x0,1  # t = 1;
      add s1,a1,0 # s1= n
      sub s1,s1,1
Loop:
      mul s0,s0,a0; # t= t *x
      sub s1,s1,1
      bge s1,x0,Loop; # if n >= 0 ,because n >= 0 so we could do  t= t
*x at lease times.

      add a0,s0,x0  # return value s0

      lw s0, 0(sp)  # restore register s0 for caller
      lw s1, 4(sp)  # restore register s1 for caller
      addi sp,sp,8  # adjust stack to delete 2 items
```

```
    jr ra          # jump back to calling routine
```

4. Translate between the C and RISC-V verbatim (15 points)

C	RISC-V

`// computes s1 = 2^30`
 	
`s1 = 1;`
 	
`for(s0=0;s0<30;s++) {`
 	
`  s1 *= 2;`
 	
`}`
	`Add x10,x0,x0  # s1 = 0`
	
	`addi x10, x0, 1 # s1 = 1;`
	
	`Add x9,x0,x0  # s0 = 0`
	
	`Loop:`
	
	`  slli x10,x10,2 #  s1<<1`
	
	`  addi x9,x9,1 # i++`
	
	`  addi x13,x0,30 # x13=30`
	
	`  blt x9,x13,Loop`

`S0[0] = 0;`
 	
`S1 = 2;`
 	
`s0[1] = s1;`
 	
`s1= 8;`
 	
`t0 = 8;`
 	
`t0 = s1;`
	`sw x0, 0(s0) # s 0[0] = 0`
	
	`addi s1, x0, 2 # s1 = 2`
	
	`sw s1, 4(s0)#  s0[1] = 2`
	
	`slli t0, s1, 2 # s1= 8`
	
	`add t0, t0, s0 # t0 = 8`
	
	`sw s1, 0(t0) #  s1 = 8`

`S0  =5;`
`S1 = 10;`
`T0 = 10;`
`If( t0 != s1){`
`  s1 = s0 -1;`
`}`
`else{`
`  s0 = 0^0;// s0 = 0`
`}`
`exit;`
	`  addi s0, x0, 5`
	
	`  addi s1, x0, 10`
	
	`  add t0, s0, s0`
	
	`  bne t0, s1, else`
	
	`  xor s0, x0, x0`
	
	`  jal x0, exit`
	
	`else:`
	
	`  addi s1, s0, -1`
	
	`exit:`

`s0 = 4;`
`s1 = 5;`
`s2 = 6;`
	`addi s0, x0, 4`
	
	`addi s1, x0, 5`

s3 = s0+s1; // s3 = 9
s3 = s3+s2;
s3 = s3+10;
	addi s2, x0, 6
	
	add s3, s0, s1
	
	add s3, s3, s2
	
	addi s3, s3, 10

s1 = s1;
while(s0 != 0){
  s1 = s1+ s0;
  s0 = s0 -1;
}
	addi s1, s1, 0
	
	loop:
	
	beq s0, x0, exit
	
	add s1, s1, s0
	
	add s0, s0, -1
	
	jal x0, loop
	
	exit:

5. Computing a Fibonacci Number. The Fibonacci number F_n is recursively defined as

```
F(n) = F(n - 1) + F(n - 2);
```

where **F(1) = 1 and F(2) = 1. So, F(3) = F(2) + F(1) = 1 + 1 = 2**, and so on. Write the RISC-V assembly for the fib(n) function, which computes the Fibonacci number F(n): (12 points)

```
int fib(int n){
    int a = 0;
    int b = 1;
    int c = a + b;
    while (n > 1) {
        c = a + b;
        a = b;
        b = c;
        n--;
    }// n == 1, c  =1 ; n ==2 ; 循环一次
return c;
}
```

```
fib: addi sp,sp,-8 # adjust stack for 2 items
      sw s1, 4(sp)  # save s1 for use afterwards
      sw s0, 0(sp)  # save s0 for use afterwards

      addi s0,x0,x0 # int a = 0;
```

```
      addi s1,x0,1  #int b = 1;
      addi t1,x0,1  # int c = a + b;
      sub a0,a0,1  # n--
Loop:
      ble s1,x0,end     # n == 1, 不循环
      add t1,s1,s0      # c= a+ b
      add s0,s1,x0      # a= b
      add s1,t1,x0      # b  =c
      sub a0,a0,1       #n--
      bgt a0,x0,Loop; # if n >= 0 ,because n >= 0 so we could do  t= t
*x at lease times.

end:
      add a0,t1,x0 # return
      lw s0, 0(sp)  # restore register s0 for caller
      lw s1, 4(sp)  # restore register s1 for caller
      addi sp,sp,8  # adjust stack to delete 2 items
      jr ra         # jump back to calling routine
```

6. We have several addressing modes to access memory (immediate not listed):
    a. Base displacement addressing: Adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb)
    b. PC-relative addressing: Uses the PC and adds the immediate value of the instruction (multiplied by 2) to create an address (used by branch and jump instructions)
    c. Register Addressing: Uses the value in a register as a memory address (jr)

(1) What is range of 32-bit instructions that can be reached from the current PC using a branch instruction? (4 points)

Each bits target 2bytes.

32bit = 4bytes

The immediate field of the branch instruction is 12bits. Thus, branch instruction can reach [-2^11, 2^11-1]. 32-bit = 4bytes. We can reach [-2^10, 2^10-1] instructions.

(2) What is the range of 32-bit instructions that can be reached from the current PC using a jump instruction? (4 points)

2^20 bits = > [-2^19 , 2^19-1]

Each bits target 2bytes.

32bit = 4bytes

The current PC use a jump instruction can reach [-2^18, 2^18-1] instructions

(3) Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your RISC-V green card!). (4 points)

0x002cff00: loop: sub t1, t2, t0 |___01000_____|__00_____|__0x05(x5)____
__|___0x07___|__0x06_ t1= x6, _____|__0x33__|

0x002cff04: jal ra, foo|_____0x14(不是28因为省略了一个

2)_____|____rd= ra= 0x01 _____|__0x6F__|
0x002cff08: bge t1, zero, loop |_ 1 [12]
111111[10:5]____|___0x00_____|__0x06_____|__101_____|_1100_[4:1]_
1[11]_____|__0x63__|
... imm = -8 1111 1111 1000,

0x002cff2c: foo: jr ra ra=__ pc+4 =__0x002cff08_____

7. Which step in CALL resolves relative addressing? Absolute addressing? (5 points)

Assembler resolves relative addressing,

Linker resolves absolute addressing.

(1) The result converted by Assembler is an .o file. The .o file includes the object file header, which is used to describe the size and location of other files of the object file. The text segment contains the machine code, and the static data segment contains the programs used in the program. relocation information is written in the address of instructions and data when the program is loaded into memory, symbol table contains undefined labels of some external parameters, and debugging information is used to describe the compilation between C and machine code.

(2) An executable file is realized through Linker. The main job of such an executable file is to allocate real memory addresses for code and data. Link multiple object files together.

8. What does RISC stand for? How is this related to pseudo instructions? (5 points)

Reduced Instruction Set Computer.

Reduced Instruction needs more lines of assemble codes, pseudo instructions could help programmer to accelerate their coding process without lengthy code writing.