1. (40 points)

   a. (10 points)

```
000000000001045c:   auipc gp,0x0
0000000000010460:   addi gp,gp,-652 # 0x101d0
```

Init gp: first use **auipc** to add **pc + 0** to **gp**, then – 652 to point to initial position.

result: 0x11538

```
27                    result = gcd(756, 996);
0000000000010508:   li a1,996
000000000001050c:   li a0,756
0000000000010510:   jal ra,0x104a8 <gcd>
0000000000010514:   c.mv a5,a0
0000000000010516:   c.mv a4,a5
0000000000010518:   addigp a5,4968
000000000001051c:   c.sw a4,0(a5)
```

In this part, the result is global variable, so we use **addigp a5,4968** to make the position of the result from **gp +4968** 's address to **a5**. **gp** also be set to **0x101d0** when the program **initial**, so the memory address of the result will be **0x101d0(hex) +4968(10) = 0x11538(hex)**.

gcd: 0x104a8

```
27                    result = gcd(756, 996);
0000000000010508:   li a1,996
000000000001050c:   li a0,756
0000000000010510:   jal ra,0x104a8 <gcd>
0000000000010514:   c.mv a5,a0
0000000000010516:   c.mv a4,a5
0000000000010518:   addigp a5,4968
000000000001051c:   c.sw a4,0(a5)
```

Here we can find that when during the procedure call, jump position has been point out clearly, which is gcd function's memory address.

   b. (4 points)

REMW is a RV64 instruction that provides the corresponding signed remainder operation. REMW always sign-extend the 32-bit result to 64 bits, including on a divide by zero.

   c. (10 points)

      i. (5 points)

```
gcd:
104a8:   c.addi16sp sp,-48
104aa:   c.sdsp ra,40(sp)
104ac:   c.sdsp s0,32(sp)
104ae:   c.addi4spn s0,sp,48
104b0:   c.mv a5,a0
104b2:   c.mv a4,a1
104b4:   sw a5,-36(s0)
104b8:   c.mv a5,a4
104ba:   sw a5,-40(s0)
           int remainder = a % b;
104be:   lw a4,-36(s0)
104c2:   lw a5,-40(s0)
104c6:   remw a5,a4,a5
104ca:   sw a5,-20(s0)
```

High memory address		0	Frame Pointer
	ra	-8	
	s0	-16	
	a0%a1	-20	
		-24	
		-28	
		-32	
	a0	-36	
	a1	-40	
		-44	
		-48	Stack Pointer
Low memory address			

ii. (5 points)

Each time the gcd function is called, **sp-48** will be used to save the value of the register. It can be seen that when remainder=0, the function will return, so the deepest call is 6 times during execution. In conclusion, the lowest address is the initial **sp** address - **16** (main function) - **48* 6** (gcd recur.) **= 0x3000000 - 16(dec) - 48 * 6 (dec) = 0x3000000 - 0x0130 = 0x02FFFED0**

From AndeSight disassembly,
0000000000010454: addi a0,a0,-1104 # 0x3000000
0000000000010500: c.addi sp,-16
00000000000104a8: c.addi16sp sp,-48

d. (6 points)

```
21                        return gcd(b, remainder);
00000000000104e0:   lw a4,-20(s0)
00000000000104e4:   lw a5,-40(s0)
00000000000104e8:   c.mv a1,a4
00000000000104ea:   c.mv a0,a5
00000000000104ec:   jal ra,0x104a8 <gcd>
00000000000104f0:   add a5,zero,a0
```

lw a4,-20(s0)	load remainder to a4
lw a5,-40(s0)	load b to a5
c.mv a1,a4	move a4 to a1
c.mv a0,a5	move a5 to a0
jal ra,0x104a8 <gcd>	record the address and jump to gcd
add a5,zero,a0	a5=0+a0

e. (10 points)

```
21                        return gcd(b, remainder);
00000000000104b8:   c.mv a0,a5
00000000000104ba:   jal ra,0x104a8 <gcd>
```

c.mv a0,a5	move a5 to a0
jal ra,0x104a8 <gcd>	record the address and jump to gcd

2. The main difference between -Og and -O0 is that -Og optimizes user debugging experience compared to -O0. Like -O0, -Og completely disables a number of optimization passes so that individual options controlling them have no effect. Otherwise -Og enables all -O1 optimization flags except for those that may interfere with debugging.

From the two assembly codes, we can see that -Og uses extra saved registers to reduce the number of load and move instructions, increase both debuggability and performance.

2. (15 points)

(a) (5 points)

```
beq x10, x0, -24
```

binary representation: 1111111 00000 01010 000 01001 1100011

Imm[12|10:5]	rs2	rs1	funct3	Imm[4:1|11]	opcode
1 111111	00000	01010	000	01001	1100011

hexadecimal representation: 0xFE0504E3

(b) (5 points)

0000000 01010 00011 011 11000 0100011

```
sd      x10, 24(x3)
```

(c) (5 points)

i. lui version:

```
lui x5, 524287         //524287decimal = 0x7FFFF
addi x5,x5, 4095       //4095decimal = 0xFFF,  x5 = 0x7FFFFFFF
jalr x0, 57(x5)        //x5 + 57decimal = 0x80000038
```

(If the imm of addi is decreased a bit and the offset of jalr is increased a bit, it is still considered correct as long as the final jump destination is 0x80000038. However, it should be noted that the imm of both instructions is only 12 bits long and cannot exceed 4095.)

ii. auipc version:

```
auipc x5, 393216       //0x60000 = 393216decimal
addi x5,x5, 56         //0x38 = 56decimal
jalr x0, 0(x5)

or

auipc x5, 393216
jalr x0, 56(x5)
```

3. (5 points)

Little-Endian		Big-Endian	
Address	Data	Address	Data
0x00000007	52	0x00000007	76
0x00000006	49	0x00000006	32
0x00000005	53	0x00000005	56
0x00000004	43	0x00000004	2D

0x00000003	2D	0x00000003	43
0x00000002	56	0x00000002	53
0x00000001	32	0x00000001	49
0x00000000	76	0x00000000	52

4. (5 points)

```
j = B[A[i*2]] - 16
  slli x28, x5, 4            # i*2*offset
  add x28, x6, x28           # add A base
  ld x29, 0(x28)             # x29 = A[i*2]
  slli x29, x29, 3           # x29 = A[i*2] * offset
  add x29, x7, x29           # add B base
  ld x10, 0(x29)             # x10 = B[A[i*2]]
  addi x10, x10, -16         # x10 = B[A[i*2]] - 16
```

5. (10 points)

(m, n, i, and j are in registers x3, x4, x11, and x12,array D is a 4-byte integer and register x14 holds the base address of D)

c code:

V1:

```
for (i=0; i<m; i++)
     for (j=0; j<n; j++)
          D[i] = D[i] + 7*(i+j);
```

V2:

```
int *ptr = &D;
for (i=0; i<m; i++){
     int d = ptr[0];
     for (j=0; j<n; j++){
          d = d + 7*(i+j);
     }
     ptr[0] = d;
     ptr += 4;
}
```

```
  addi x11, x0, 0          # Init i = 0
  addi x30, x14, 0         # x30 = &D
LOOPI:
```

```
    bge x11, x3, ENDI               # if i >= m, branch
    addi x12, x0, 0             # Init j = 0
    lw x31, 0(x30)              # x31 = D[i]
LOOPJ:
    bge x12, x4, ENDJ           # if j >= n, branch
    add x27, x11, x12               # x27 = i+j
    slli x28, x27, 3            # x28 = x27 * 8
    sub x28, x28, x27           # x28 = x27 * 8 - x27
    add x31, x31, x28               # x31 = x31 + 7*(i+j)
    addi x12, x12, 1            # j++
    jal x0, LOOPJ
ENDJ:
    sw x31, 0(x30)              # D[i] = x31
    addi x11, x11, 1            # i++;
    addi x30, x30, 4            # x30 = &D[i]

    jal x0, LOOPI
ENDI:
```

6. (13 points)

```
func:
    beq  x10, x0, done          // If n==0, return 0
    addi x5, x0, 1              // x5 = 1
    beq  x10, x5, done          // If n==1, return 1
    addi x5, x5, 1              // x5 = 2
    beq  x10, x5, done          // If n==2, return 2
    addi x2, x2, -32            // Allocate the space of stack
    sd   x1, 0(x2)              // store the return address
    sd   x10, 8(x2)             // store the current n
    addi x10, x10, -1           // n-1
    jal  x1, func               // func(n-1)
    sd   x10, 16(x2)            // save func(n-1) to stack
    ld   x10, 8(x2)             // load old n
    addi x10, x10, -2           // n-2
    jal  x1, func               // func(n-2)
    sd   x10, 24(x2)            // save func(n-2) to stack
    ld   x10, 8(x2)             // load old n
    addi x10, x10, -3           // n-3
    jal  x1, func               // func(n-3)
    ld   x5, 8(x2)              // load old n
    ld   x6, 16(x2)             // load func(n-1)
    ld   x7, 24(x2)             // load func(n-2)
    mul  x5, x5, x5             // n*n
    slli x7, x7, 3              // 8* func(n-2)
    add  x5, x5, x6             // n*n + func(n-1)
```

```
    add   x5, x5, x7               // n*n + func(n-1) + 8*func(n-2)
    add   x10, x10, x5             // n*n + func(n-1) + 8*func(n-2) + func(n-3)
    ld    x1, 0(x2)                // load return address
    addi  x2,x2,32                 // pop the stack
done:
    jalr x0,0(x1)
```

7. (12 points)

True or false. Please explain if your answer is false.

(a)

false, t1 = 0xFFFFFFFFFFFFFFB3

```
add t0, zero, t3
sd t0, 0(t2)   // from low address(t2) to high address : 00 00 B3 AA 75 C3 16 AB
lb t1, 2(t2)   //B3，and load byte will do sign extension
```

(b)

false，According to the RISC-V calling convention, the callee does not need to save t0 and t1.

(c)

false,

```
or x6, x6, x7        //x6 = 0x00000000FFFFAAAA
and x6, x6, x28      //x6 = 0x00000000AAAAAAAA
slli x6, x6, 32      //x6 = 0xAAAAAAAA00000000
srai x6, x6, 16      //x6 = 0xFFFFAAAAAAAA0000
xor  x5, x6, x7      //x5 = 0xFFFFAAAA22220000
lui  x5,0xFFFFF      //x5 = 0xFFFFFFFFFFFFF000
sd   x5, 0(x6)       //x6 won't be changed.
```

(d)

false, The instructions are also stored in memory. There should be 6+3 times.