1. (35 points)

a. gp: 0x101d0

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

Init gp: 先使用 auipc 將 pc + 0 加至 gp，再 – 652 指到初始位置

result: 0x11598

```
22                      result = sudan(6, 5, 1);
0000000000010568:   c.li a2,1
000000000001056a:   c.li a1,5
000000000001056c:   c.li a0,6
000000000001056e:   jal ra,0x104a8 <sudan>
0000000000010572:   c.mv a5,a0
0000000000010574:   c.mv a4,a5
0000000000010576:   addigp a5,5064
000000000001057a:   c.sw a4,0(a5)
```

這段指令中，因為 result 為 global variable，因此利用 addigp a5,5064，將 result 的位置由 gp + 5064 的地址放入 a5 中，而 gp 也在程式 initial 時被定為 0x101d0，因此 result 的 memory address 便在 0x101d0(hex) + 5064(10) = 0x11598(hex)

sudan: 0x104a8

```
22                      result = sudan(6, 5, 1);
0000000000010568:   c.li a2,1
000000000001056a:   c.li a1,5
000000000001056c:   c.li a0,6
000000000001056e:   jal ra,0x104a8 <sudan>
0000000000010572:   c.mv a5,a0
0000000000010574:   c.mv a4,a5
0000000000010576:   addigp a5,5064
000000000001057a:   c.sw a4,0(a5)
```

在這段 instruction 可以看到進行 procedure call 時，明確指出要 jump 的位置，即 sudan function 的 memory address。

b. (10 points)

i. (8 points)

```
104a8:   c.addi16sp sp,-48
104aa:   c.sdsp ra,40(sp)
104ac:   c.sdsp s0,32(sp)
104ae:   c.sdsp s1,24(sp)
104b0:   c.addi4spn s0,sp,48
104b2:   c.mv a5,a0
104b4:   c.mv a3,a1
104b6:   c.mv a4,a2
104b8:   sw a5,-36(s0)
104bc:   c.mv a5,a3
104be:   sw a5,-40(s0)
104c2:   c.mv a5,a4
104c4:   sw a5,-44(s0)
```

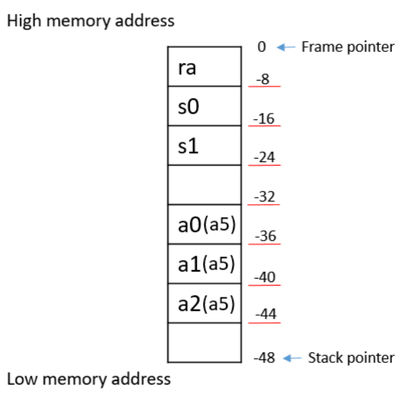


Figure1. Stack

ii. (2 points)
每次 call sudan function 時，都會先將 sp-48，以保存 register 的值。觀察可知當 n=0 時，function 會 return，因此執行時最深 call 至 5 次，另外加上初始 call sudan 時保存的 1 次，因此最低位址為原本的 sp 位址 - 48*(5+1) = 0x000000002FFFED0

c. (5 points)

```
19                         return sudan(sudan(m,n-1,k),sudan(m,n-1,k)+n,k-1);
00000000000104f4:   lw a5,-40(s0)
00000000000104f8:   c.addiw a5,-1
00000000000104fa:   bfos a4,a5,31,0
00000000000104fe:   lw a3,-44(s0)
0000000000010502:   lw a5,-36(s0)
0000000000010506:   c.mv a2,a3
0000000000010508:   c.mv a1,a4
000000000001050a:   c.mv a0,a5
000000000001050c:   jal ra,0x104a8 <sudan>
```

| | |
|---|---|
| lw a5,-40(s0) | load n 的值到 a5 |
| c.addiw a5,-1 | a5 的值-1(對應到 "n-1")後存回 a5 |
| bfos a4,a5,31,0 move a4, a5) | move a5 中 0~31 bits 的值到 a4 (即 |
| lw a3,-44(s0) | load k 的值到 a3 |
| lw a5,-36(s0) | load m 的值到 a5 |
| c.mv a2,a3 | move a3 的值到 a2(k) |
| c.mv a1,a4 | move a4 的值到 a1(n) |
| c.mv a0,a5 | move a5 的值到 a0(m) |
| jal ra,0x104a8 <sudan> | 紀錄下條指令的 address 並跳 sudan |

d. (10 points)

```
19                      return sudan(sudan(m,n-1,k),sudan(m,n-1,k)+n,k-1);
00000000000104c8:   addiw s3,a1,-1
00000000000104cc:   c.mv a1,s3
00000000000104ce:   jal ra,0x104a8 <sudan>
00000000000104d2:   c.mv s4,a0
00000000000104d4:   c.mv a2,s1
00000000000104d6:   c.mv a1,s3
00000000000104d8:   c.mv a0,s0
00000000000104da:   jal ra,0x104a8 <sudan>
00000000000104de:   addiw a2,s1,-1
00000000000104e2:   addw a1,a0,s2
00000000000104e6:   c.mv a0,s4
00000000000104e8:   jal ra,0x104a8 <sudan>
```

| | |
|---|---|
| addiw s3,a1,-1 | a1 的值-1(對應到 "n-1")存到 s3 |
| c.mv a1,s3 | move s3 的值到 a1(n) |
| jal ra,0x104a8 <sudan> | 紀錄下條指令的 address 並跳到 sudan |
| c.mv s4,a0 | move a0 的值到 s4(存 return value) |
| c.mv a2,s1 | move s1 的值到 a2(k) |
| c.mv a1,s3 | move s3 的值到 a1(n-1) |
| c.mv a0,s0 | move s0 的值到 a0(m) |
| jal ra,0x104a8 <sudan> | 紀錄下條指令的 address 並跳到 sudan |

```
addiw a2,s1,-1              a2 = s1 - 1  (k-1)
addw a1,a0,s2               a1 = a0 + s2 (return value+n)
c.mv a0,s4                  s4 移到 a0 (return value as argument)
jal ra,0x10140 <sudan>      紀錄下條指令的 address 並跳 sudan
```

2. -Og 跟-O0 比起來最大的差別就是使用多個 saved registers 來代替 load 和 move 指令，因為實際上有許多 register 的值可以重複利用，藉此來達到 speed up。

2. (5 points)

| Little-Endian | | Big-Endian | |
|---|---|---|---|
| Address | Data | Address | Data |
| 0x00000003 | 14 | 0x00000003 | 9E |
| 0x00000002 | A7 | 0x00000002 | CF |
| 0x00000001 | CF | 0x00000001 | A7 |
| 0x00000000 | 9E | 0x00000000 | 14 |

3. (10 points)

(a)
```
add x5, x5, x6          # g = g + h
slli x5, x5, 3          # multiply by 8
add x5, x10, x5         # find array index of A
ld x12, 0(x5)           # load A[g + h]
sd x12, 64(x11)         # store in B[8]
```

(b)
```
ld x12, 32(x11)         # load B[4] to x12
slli x12, x12, 3        # x12 = offset of A
add x12, x12, x10       #add A base
ld x13, 0(x12)          # x13 = A[B[4]]
sub x7, x13, x8         # i = A[B[4]] - j
```

4. (10 points)
beq : 0000000 11000 01001 000 10100 1100011

| Imm[12\|10:5] | rs2 | rs1 | funct3 | Imm[4:1\|11] | opcode |
|---|---|---|---|---|---|
| 0 000000 | 11000 | 01001 | 000 | 1010 0 | 1100011 |

jal: 11111110010111111111 00000 1101111

| Imm[20\|10:1\|11\|19:12] | rd | opcode |
|---|---|---|
| 1 1111110010 1 11111111 | 00000 | 1101111 |

5. (10 points)
   (a) R-type
       sra x8(s0), x18(s2), x23(s7)
   (b) S-type
       sd x6, 80(x26)

| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode |
|---|---|---|---|---|---|
| 0000010 | 00110 | 11010 | 011 | 10000 | 0100011 |

Instruction: 0x046D3823(hex)

6. (10 points)

```
int *ptr = Memarray;
int i=100;
do{
    result += *ptr;
    ptr++;
    i -= 4;
}while(i > 0);
```

7. (10 points)

```
# IMPORTANT! Stack pointer must remain a multiple of 16!!!!
fib:    beq     x10, x0, done       // If n==0, return 0
        addi    x5, x0, 1
        beq     x10, x5, done       // If n==1, return 1
```

```
        addi   x2, x2, -16          // Allocate 2 words of stack
        sd     x1, 0(x2)            // Save the return address
        sd     x10, 8(x2)           // Save the current n
        addi   x10, x10, -1         // x10 = n-1
        jal     x1, fib             // fib(n-1)
        ld     x5, 8(x2)            // Load old n from the stack
        sd     x10, 8(x2)           // Push fib(n-1) onto the stack
        addi   x10, x5, -2          // x10 = n-2
        jal    x1, fib              // Call fib(n-2)
        ld     x5, 8(x2)            // x5 = fib(n-1)
        slli   x10, x10 , 1         // 2*fib(n-2)
        add    x10, x10, x5         // x10 = fib(n-1)+2*fib(n-2)
         // Clean up:
        ld     x1, 0(x2)            // Load saved return address
        addi   x2, x2, 16           // Pop two words from the stack
done:   jalr   x0, 0(x1)
```

8. (10 points)

(a) The opcode would expand from 7 bits to 9. The rs1 , rs2 , and rd fields would increase from 5 bits to 7 bits.

opcode: 7bits->9bits

rs1, rs2, rd: 5bits->7bits

(b) Increasing the size of each bit field potentially makes each instruction longer, potentially increasing the code size overall. However, increasing the number of registers could lead to less register spillage, which would reduce the total number of instructions, possibly reducing the code size overall.

increase size 會使指令變長， 可能 increase code size， 然而比較多的暫存器可以比較不會被覆寫， 可能可以減少指令總數， 也可能 reduce code size.