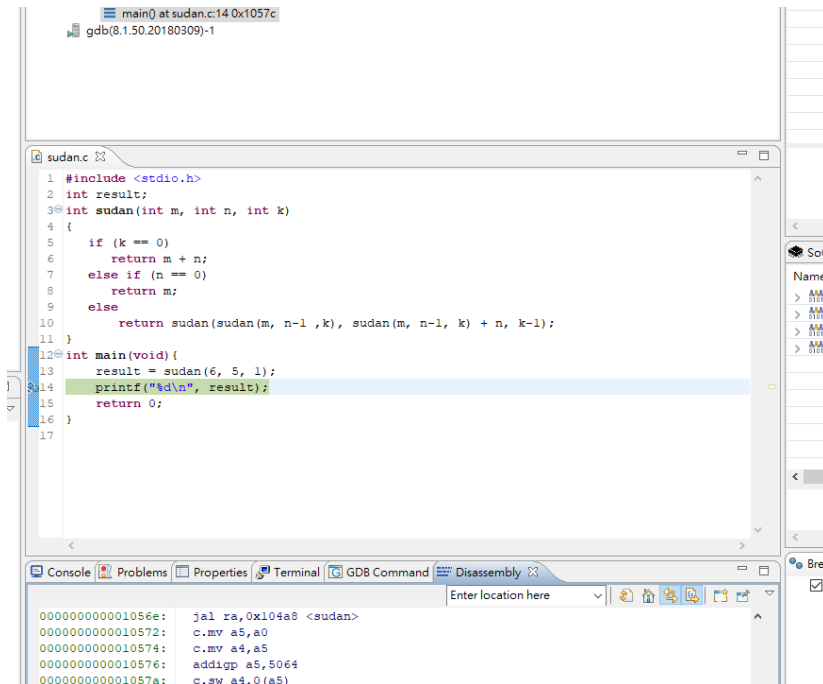


**Department of Computer Science**  
**National Tsing Hua University**  
**CS4100 Computer Architecture**  
**Spring 2022 Homework 2**

**109062320 discuss with 109062123, 109062318, 109062233, 109062101, 109062302**

Deadline: 2022/3/31 10:00

1. (35 points)
  - (1) Create an Andes C project and replace it with the sudan.c file on eLearn.
  - (2) Change optimization level to -O0.
  - (3) Press the button “Build” in the toolbar.
  - (4) Press the button “Debug” in the toolbar.
  - (5) Open the Disassembly window.



The screenshot shows the Andes Studio IDE. The main window displays the source code for `sudan.c`. The code defines a recursive function `sudan` and a `main` function that calls it with arguments `6, 5, 1`. The `main` function prints the result and returns 0. The bottom panel shows the disassembly window, which displays the assembly code for the `main` function. The instructions are in RISC-V compressed instruction set extension format, starting with 'c.'.

```
1 #include <stdio.h>
2 int result;
3 int sudan(int m, int n, int k)
4 {
5     if (k == 0)
6         return m + n;
7     else if (n == 0)
8         return m;
9     else
10        return sudan(sudan(m, n-1, k), sudan(m, n-1, k) + n, k-1);
11 }
12 int main(void) {
13     result = sudan(6, 5, 1);
14     printf("%d\n", result);
15     return 0;
16 }
17
```

```
00000000001056e: jal ra,0x104a8 <sudan>
000000000010572: c.mv a5,a0
000000000010574: c.mv a4,a5
000000000010576: addiwp a5,5064
00000000001057a: c.sw a4,0(a5)
```

**Note:** When you check the assembly code of this project, you may see some instructions that start with a “c.”. These instructions are *RISC-V standard compressed instruction set extension*. The extension reduces program code size by adding short 16-bit instruction encodings for common operations. For example, the instruction `c.add rd,rs2` adds the values in registers `rd` and `rs2` and writes the result to register `rd`. It expands into `add rd,rd,rs2` for normal 32-bit RISC-V instruction encoding. Since there are only two registers in `c.add rd,rs2` (2-address), it is possible to encode the instruction with 16 bits, thereby reducing the code size. You can read the “RISC-V\_C\_Extension\_Instruction\_Set.pdf” for more details.

Answer the following questions.

- (a) (10 points) First, show how the memory address of the gp register is initialized. Next, show how to get the memory address of the “result” variable by referencing the gp register in “main”. Finally, find the memory address of the procedure “sudan”.

AUIPC (add upper immediate to pc) is used to build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register rd.

The AUIPC instruction supports two-instruction sequences to access arbitrary offsets from the PC for both control-flow transfers and data accesses. The combination of an AUIPC and the 12-bit immediate in a JALR can transfer control to any 32-bit PC-relative address, while an AUIPC plus the 12-bit immediate offset in regular load or store instructions can access any 32-bit PC-relative data address.

The current PC can be obtained by setting the U-immediate to 0. Although a JAL +4 instruction could also be used to obtain the local PC (of the instruction following the JAL), it might cause pipeline breaks in simpler microarchitectures or pollute BTB structures in more complex microarchitectures.

### Image1-1-1:RISC Spec pg.37

```

start:
00000000000010450: auipc a0,0x2ff0
00000000000010454: addi a0,a0,-1104 # 0x3000000
00000000000010458: c.beqz a0,0x1045c <_start+12>
0000000000001045a: c.mv sp,a0
0000000000001045c: auipc gp,0x0
00000000000010460: addi gp,gp,-652 # 0x101d0
00000000000010464: auipc a0,0xfffff0

```

### Image1-1-2:the assembly code about initializing the gp register

Firstly, by Image 1-1-2, we can see that there are two instructions include the gp register, then by Image 1-1-1, we can see that the usage of AUIPC is “to build pc-relative address...”. As a result, we know that when initializing the memory address of the gp register, we firstly use “AUIPC” to let gp (as know as global pointer) to be allocated into memory, then since gp is a pointer, we secondly use “ADDI” to make space for the stack.

```

00000000000010576: addigp a5,5064
0000000000001057a: c.sw a4,0(a5)
24 printf("%d\n", result);
0000000000001057c: addigp a5,5064

```

### Image1-1-3:the assembly code about getting the memory address of the “result” variable

I take reference from

[https://www.ovpworld.org/modeldocs/OVP\\_Model\\_Specific\\_Information\\_andes\\_riscv\\_N25.pdf](https://www.ovpworld.org/modeldocs/OVP_Model_Specific_Information_andes_riscv_N25.pdf)

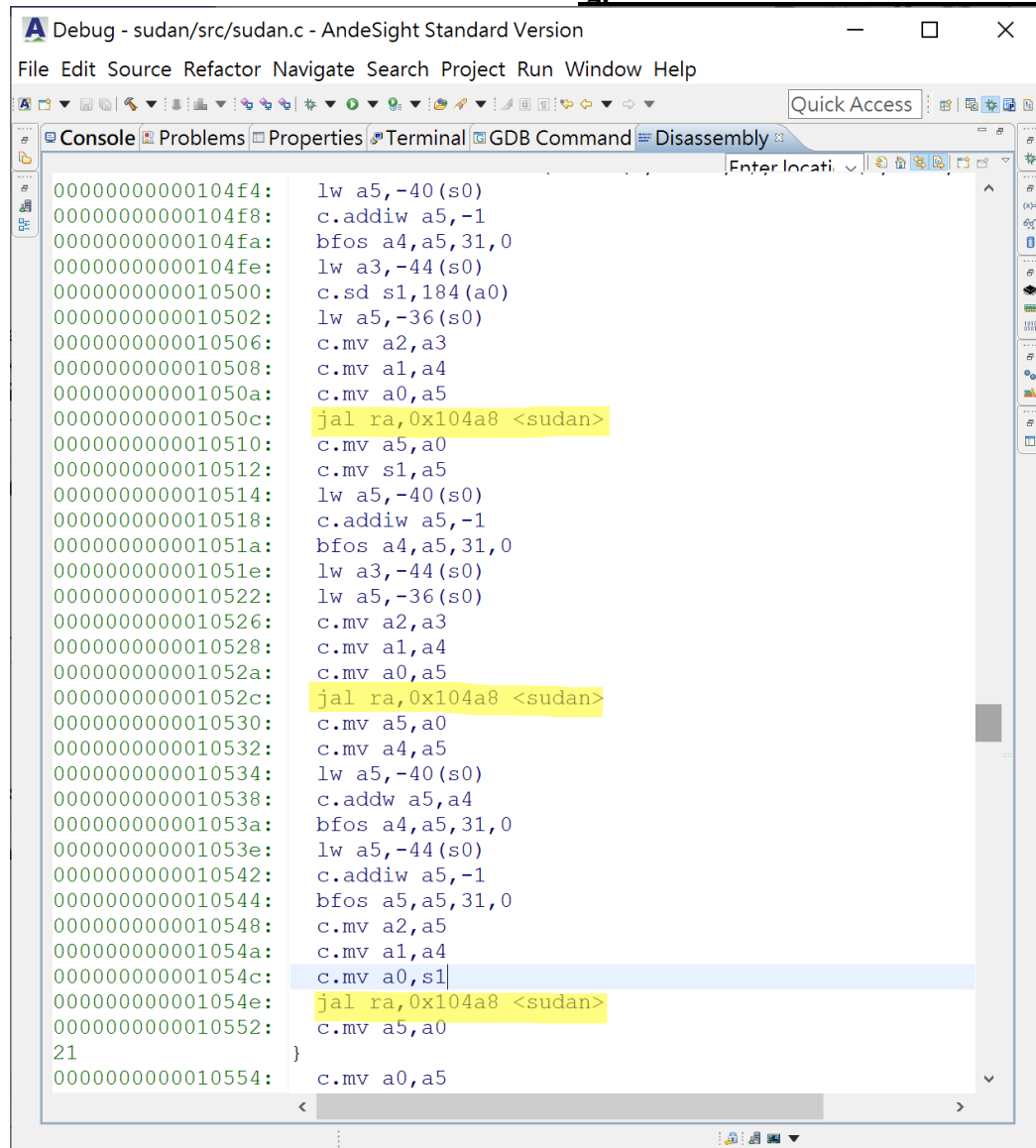
#### **2.7.1.1 ADDIGP**

31	30	21	20	19	17	16	15
imm[17]	imm[10:1]		imm[11]	imm[14:12]		imm[16:15]	
14		13	12	11	7	6	0
imm[0]		01		Rd		Custom0 0001011	

Add the content of the implied GP (x3) register with a signed constant.

### Image1-1-4:introduction of “addigp”

By Image1-1-4, we know that “addigp” adds an immediate to GP register and store the result in Rd, so we have the address of “result” variable :  $gp + 5064 = 0x101d0 + 0x13c8 = 0x11598$



```

Debug - sudan/src/sudan.c - AndeSight Standard Version
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Console Problems Properties Terminal GDB Command Disassembly
Enter location:

00000000000104f4: lw a5,-40(s0)
00000000000104f8: c.addiw a5,-1
00000000000104fa: bfos a4,a5,31,0
00000000000104fe: lw a3,-44(s0)
0000000000010500: c.sd s1,184(a0)
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>
0000000000010510: c.mv a5,a0
0000000000010512: c.mv s1,a5
0000000000010514: lw a5,-40(s0)
0000000000010518: c.addiw a5,-1
000000000001051a: bfos a4,a5,31,0
000000000001051e: lw a3,-44(s0)
0000000000010522: lw a5,-36(s0)
0000000000010526: c.mv a2,a3
0000000000010528: c.mv a1,a4
000000000001052a: c.mv a0,a5
000000000001052c: jal ra,0x104a8 <sudan>
0000000000010530: c.mv a5,a0
0000000000010532: c.mv a4,a5
0000000000010534: lw a5,-40(s0)
0000000000010538: c.addw a5,a4
000000000001053a: bfos a4,a5,31,0
000000000001053e: lw a5,-44(s0)
0000000000010542: c.addiw a5,-1
0000000000010544: bfos a5,a5,31,0
0000000000010548: c.mv a2,a5
000000000001054a: c.mv a1,a4
000000000001054c: c.mv a0,s1
000000000001054e: jal ra,0x104a8 <sudan>
0000000000010552: c.mv a5,a0
21 }
0000000000010554: c.mv a0,a5

```

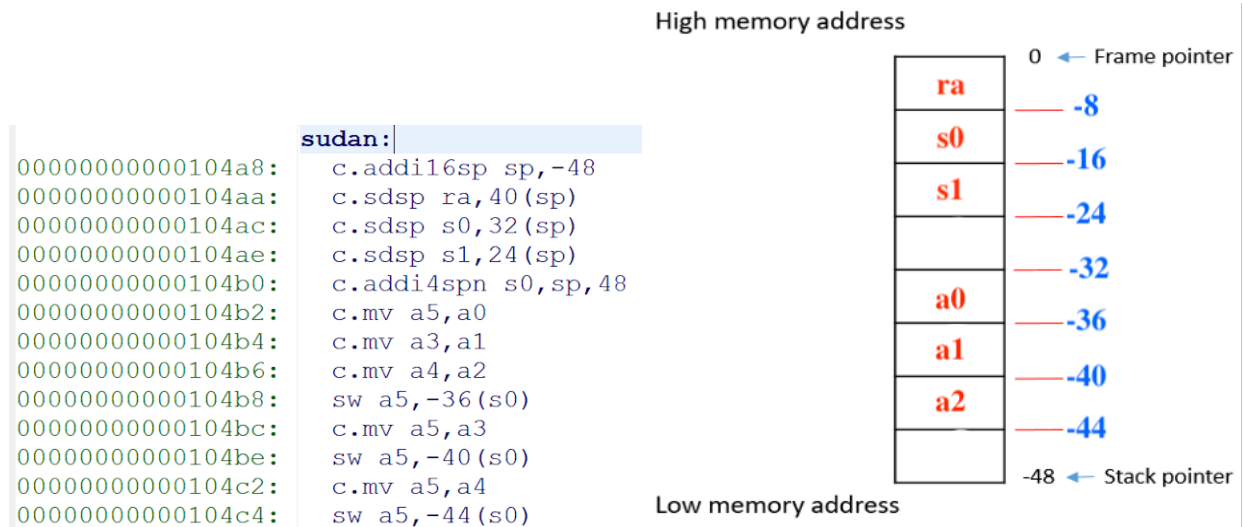
**Image1-1-5:the instructions that call the function “sudan”**

By image 1-1-5, we can recognize the memory address of the procedure “sudan” by the places which are highlighted ----- “jal ra, 0x104a8 <sudan>” and we know what we want is “0x104a8”

(b) (10 points)

**i.** Record the assembly code from memory address 0x104a8 to 0x104c4. Fill in the stack block by register name and the corresponding memory offsets of the stack in the below figure.

**Ans:**



**Image1-2-1:assembly code from 0x104a8 to 0x104c4**

**Image1-2-2:the original image with the answer**

According to the RISC\_C\_INSTRUCTION\_SET manual, we firstly recognize the meaning of each instruction that starts with “c.”

**c.addi16sp** --> used to adjust the stack pointer in procedure prologues and epilogues

**c.sdsp** --> computes an effective address by adding the zero-extended offset, scaled by 8, to the stack pointer

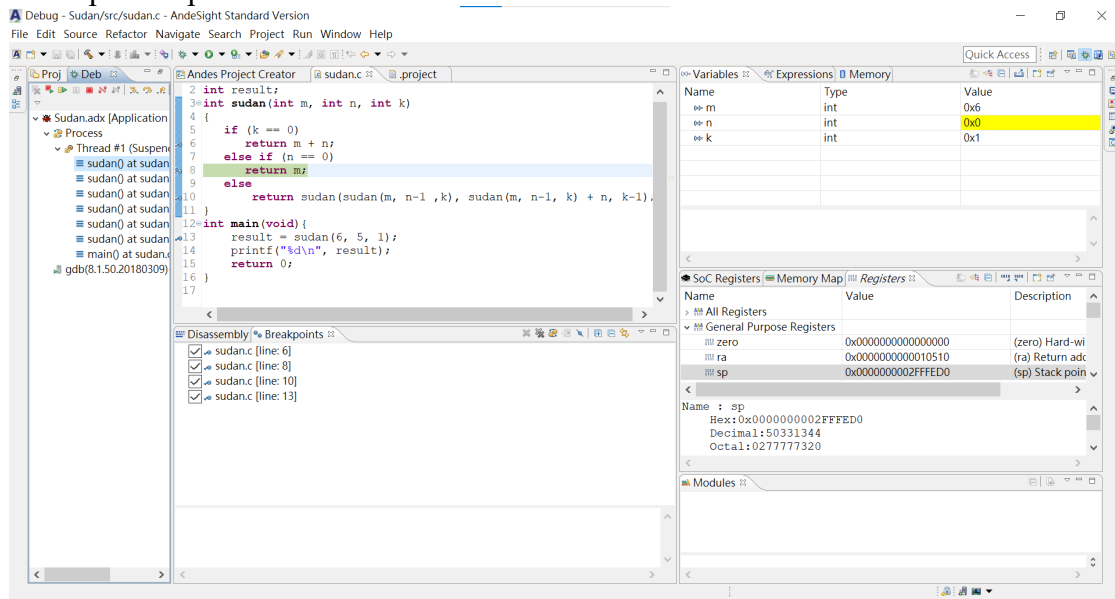
**c.addi4pn** --> used to generate pointers to stack-allocated variables

**c.mv** --> copies the value in register rs2 into register rd

**sw** --> store word from register to memory

By the above assembly codes and their meanings, we know that it firstly set the stack pointer(sp) by c.addi16sp, then set the address of ra, s0 and s1's value to the stack by the address of sp, and since the value of s0 is set to sp+48=0 in 104b0, the values of a0, a1, a2 are store to the stack by the reference of s0(e.g. s0-44 = -44 but the original value of s0 that stored in the stack doesn't change), so we have them fill in the stack block ( the red words)

ii. During the execution of the sudan function, what is the lowest memory address that the stack pointer pointed to?



**Image1-2-3:the lowest memory that sp point to**

Because when running the recursion, we have to store more variables, the stack pointer will point to the lower place than what we just saw in the assembly code.

As a result, we need to set break point to observe the lowest memory address that the stack pointer point to, and by the Image1-2-3, we can see that the address is **0x0000 0000 02FF FED0**

- (c) (5 points) Record the assembly code from memory address 0x104f4 to 0x1050c, which correspond to part of the statement: `return sudan(sudan(m,n-1,k), sudan(m,n-1,k)+n,k-1)`; Briefly explain what these instructions do. (You can use a screenshot and explain each line correspondingly.)

```
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>
```

**Image1-3-1:assembly code from 0x104f4 to 0x1050c**

We have already know that a0 stores m, a1 stores n, a2 stores k  
104f4:Load word value at s0-40(offset) to a5 register

104f8:a5 = a5 - 1 (c.addiw: performs the same computation but produces a 32-bit result, then sign-extends result to 64 bits )

**104fa:bit field operation**(take reference from

[https://www.gowinsemi.com/upload/database\\_doc/586/document\\_ja/5de4c10ca33c9.pdf](https://www.gowinsemi.com/upload/database_doc/586/document_ja/5de4c10ca33c9.pdf)

Chapter19.1) **move a5中0~31bits的值到a4(即move a4,a5)**

104fe:Load word value at s0-44 to a3 register

10502:Load word value at s0-36 to a5 register

10506:move the value in a3 to a2  
 10508:move the value in a4 to a1  
 1050a:move the value in a5 to a0  
 1050c:jump to the <sudan> label in 0x104a8 to execute it and store the address of the next line (1050e) to ra

(d) (10 points)

- i. Change the optimization level to -Og, and do the same as (c) but with the whole statement:  
`return sudan(sudan(m,n-1,k), sudan(m,n-1,k)+n, k-1);`

10	<code>return sudan(sudan(m, n-1, k), sudan(m, n-1, k) + n, k-1);</code>	
00000000000104c8:	<code>addiw s3,a1,-1</code>	the value of s3 = a1-1
00000000000104cc:	<code>c.mv a1,s3</code>	move the value in s3 to a1
00000000000104ce:	<code>jal ra,0x104a8 &lt;sudan&gt;</code>	jump to the label <sudan> located in 0x104a8 and store 104d2(the next line's address) to ra
00000000000104d2:	<code>c.mv s4,a0</code>	move the value in a0 to s4
00000000000104d4:	<code>c.mv a2,s1</code>	move the value in s1 to a2
00000000000104d6:	<code>c.mv a1,s3</code>	move the value in s3 to a1
00000000000104d8:	<code>c.mv a0,s0</code>	move the value in s0 to a0
00000000000104da:	<code>jal ra,0x104a8 &lt;sudan&gt;</code>	jump to the label <sudan> located in 0x104a8 and store 104de(the next line's address) to ra
00000000000104de:	<code>addiw a2,s1,-1</code>	the value of a2 = s1-1
00000000000104e2:	<code>addw a1,a0,s2</code>	the value of a1 = a0+s2
00000000000104e6:	<code>c.mv a0,s4</code>	move the value in s4 to a0
00000000000104e8:	<code>jal ra,0x104a8 &lt;sudan&gt;</code>	jump to the label <sudan> located in 0x104a8 and store 104ec(the next line's address) to ra
11	<code>}</code>	
00000000000104ec:	<code>c.ldsp ra,40(sp)</code>	load the value stored in sp+40 to ra
00000000000104ee:	<code>c.ldsp s0,32(sp)</code>	load the value stored in sp+32 to s0
00000000000104f0:	<code>c.ldsp s1,24(sp)</code>	load the value stored in sp+24 to s1
00000000000104f2:	<code>c.ldsp s2,16(sp)</code>	load the value stored in sp+16 to s2
00000000000104f4:	<code>c.ldsp s3,8(sp)</code>	load the value stored in sp+8 to s3
00000000000104f6:	<code>c.ldsp s4,0(sp)</code>	load the value stored in sp+0 to s4
00000000000104f8:	<code>c.addi16sp sp,48</code>	sp = sp+48, restore the space of stack we have set before
00000000000104fa:	<code>c.jr ra</code>	jump to the address that stored in ra

**Image1-4-1: the assembly code under level -Og**

- ii. Compare the difference of the assembly codes according to the return statements that are generated by the different optimization levels -Og and -O0.

Lets compare image 1-1-5 and 1-4-1 to compare the code generated by -Og and -O0. Firstly, it's obviously that the code in Og is less than O0 .

Then we also have to compare the codes in detail, and we separate it to three parts by the condition

1. `k==0` , return `m+n`

2. `n==0` ,return `m`

3. else, return `sudan(sudan(m,n-1,k),sudan(m,n-1,k)+n,k-1)`

00000000000104c4:	<code>sw a5, -44(s0)</code>
15	<code>if (k == 0)</code>
00000000000104c8:	<code>lw a5, -44(s0)</code>
00000000000104cc:	<code>bfos a5,a5,31,0</code>
00000000000104d0:	<code>c.bnez a5,0x104e4 &lt;sudan+60&gt;</code>
16	<code>return m + n;</code>
00000000000104d2:	<code>lw a4, -36(s0)</code>
00000000000104d6:	<code>lw a5, -40(s0)</code>
00000000000104da:	<code>c.addw a5,a4</code>
00000000000104dc:	<code>bfos a5,a5,31,0</code>
00000000000104e0:	<code>j 0x10554 &lt;sudan+172&gt;</code>
17	<code>else if (n == 0)</code>
00000000000104e4:	<code>lw a5, -40(s0)</code>
00000000000104e8:	<code>bfos a5,a5,31,0</code>
00000000000104ec:	<code>c.bnez a5,0x104f4 &lt;sudan+76&gt;</code>
18	<code>return m;</code>
00000000000104ee:	<code>lw a5, -36(s0)</code>
00000000000104f2:	<code>c.j 0x10554 &lt;sudan+172&gt;</code>



#### **Image1-4-2:assembly code for condition 1 and 2 in -O0**

```
00000000000104a6:   c.jr ra
15                if (k == 0)
                sudan:
00000000000104a8:   c.beqz a2,0x104b0 <sudan+8>
17                else if (n == 0)
00000000000104aa:   c.bnez a1,0x104b4 <sudan+12>
21                }
00000000000104ac:   ret
16                return m + n;
00000000000104b0:   c.addw a0,a1
00000000000104b2:   c.jr ra
14                {
00000000000104b4:   c.addi16sp sp,-48
00000000000104b6:   c.sdsp ra,40(sp)
00000000000104b8:   c.sdsp s0,32(sp)
00000000000104ba:   c.sdsp s1,24(sp)
00000000000104bc:   c.sdsp s2,16(sp)
00000000000104be:   c.sdsp s3,8(sp)
00000000000104c0:   c.sdsp s4,0(sp)
00000000000104c2:   c.mv s1,a2
00000000000104c4:   c.mv s2,a1
00000000000104c6:   c.mv s0,a0
```

#### **Image1-4-3:assembly code for for condition 1 and 2 in -Og**

##### **A. For condition 1**

Both of them have c.bnez to take the branch if k != 0, but we can see that the rs1 are different!

For O0, it firstly load a5 from memory, then set the bit field by bfos while in Og, it directly have a2 to be the rs1 but to load it first!

We can take reference from textbook and it shows that a2 stores function argument, so we don't have to load it first!

As for the return statement ; similarly, we have to load word , calculate the value and set bit field , then jump to 0x10554 in O0 while we only have to calculate the return value and jump back to the address stored in ra.

##### **B. For condition 2**

Similar to what happened in condition 1, O0 also have to do some extra actions such as load the word and set the bit field.

But this time, when returning the value, Og only have one instruction---RET,

And RET is used to pop the return address off the stack (which is pointed to by the stack pointer register) and then continues execution at that address.

(take reference from

<https://www.quora.com/What-does-RET-instruction-do-in-assembly-language>

)

##### **C. For condition 3**

The most significant differences are the same as the above two conditions;

besides, we can see that O0 need to store value to the stack(c.sd) while Og doesn't have this instruction.

In summary, the reason why Og can have less instruction than O0 is that Og is good at taking advantage of the usage of the registers and prevent from the redundant calculation!

2. (5 points) Respectively show how the value  $14A7CF9E_{\text{hex}}$  would be arranged in the memory of a little-endian machine and a big-endian machine. Assume that the machines are byte-addressable and the data are stored starting at address  $0x00000000$ .

**Each hexadecimal digit accounts for 4 bits and two hex digits take up 8 bits = 1 byte.  
As a result, we allocate two hex digits in each memory address.  
(Memory is byte addressed, Each address identifies an 8-bit byte)**

**little-endian machine:**

<b>0x00000000</b>	<b>9E</b>
<b>0x00000001</b>	<b>CF</b>
<b>0x00000002</b>	<b>A7</b>
<b>0x00000003</b>	<b>14</b>

**big-endian machine:**

<b>0x00000000</b>	<b>14</b>
<b>0x00000001</b>	<b>A7</b>
<b>0x00000002</b>	<b>CF</b>
<b>0x00000003</b>	<b>9E</b>

3. (10 points) For each of the following C statements, write the corresponding RISC-V assembly code. Assume that the base addresses of arrays A and B are in registers x10 and x11, respectively. Each element of A or B is **8 bytes**, and the variables g, h, i, j are assigned to registers x5, x6, x7 and x8, respectively.

(a) (5 points)  $B[8] = A[g + h];$   
**add x28, x5, x6 #add g+h and place it to the temporary register**  
**slli x28, x28, 3 #8byte offset**  
**add x9, x28, x10 #A[g+h]'s address store at x9**  
**ld x29, 0(x9) #load value of A[g+h] to x29**  
**sd x29, 64(x11) #store value at B[8]**

(b) (5 points)  $i = A[B[4]] - j;$   
**ld x28, 32(x11) #load value of B[4] to x28**  
**slli x28, x28, 3 #8byte offset**  
**add x9, x28, x10 #A[B[4]]'s address store at x9**  
**ld x29, 0(x9) #load value of A[B[4]] to x29**  
**sub x7, x29, x8 # i = A[B[4]] - j**



4. **(10 points)** Consider the following code sequence, assuming that LOOP is at memory location  $1024_{10}$ . What is the binary representation for the 4<sup>th</sup> instruction (beq) and the 8<sup>th</sup> instruction (jal)?

```

LOOP:  slli  x10, x22, 3      #1024
        add  x10, x10, x1     #1028
        ld   x9, 0(x10)      #1032
        beq  x9, x24, EXIT    #1036
        addi x22, x21, 2      #1040
        addi x22, x22, 1      #1044
        addi x23, x23, 1      #1048
        jal  x0, LOOP         #1052
EXIT:                                     #1056

```

Since target address = PC + immediate\*2, the immediate of beq -->  $1036(\text{pc}) + 2 * \text{immed} = 1056$   
Immed =  $(1056 - 1036) / 2 = 10$  --> 0000 0000 1010 --> 0 0 000000 1010

Binary representation of beq --> **imm[12, 10:5] rs2 rs1 funct3 imm[4:1, 11] opcode**

**→ 0 000000 11000 01001 000 1010 0 1100011**

Similarly, the immediate of jal -->  $(1024 - 1052) / 2 = -14$  --> 1111 1111 1111 1111 0010  
--> 1 1 11111111 1 11111110010

Binary representation of jal --> **imm[20, 10:1, 11, 19:12] rd opcode**

**→ 1 1111110010 1 11111111 00000 1101111**

5. (10 points) (show your derivation)

- (a) Give the **instruction type** and **assembly instruction** for the following RISC-V machine instruction:

0100 000 1 0111 1001 0 101 0100 0 011 0011

First, we distinguish the **instruction type** by the opcode and we recognize that the above instruction is belong to **R-type**.

opcode:0110011-->R-type

rd:01000 -->x8

funct3:101 -->"srl" or "sra"

rs1:10010 -->x18

rs2:10111 -->x23

funct7:0100000 -->"sub" or "sra"

Secondly, by the above inference, we find that what the assembly instruction is

----- **sra x8, x18, x23**

- (b) Give the instruction type and hexadecimal representation of the following RISC-V assembly instruction:

sd x6, 80(x26)

Firstly, "sd" belongs to the **"S-type"**, and we can have its opcode, and funct3 by the "RISC-V instruction encoding" figure in the textbook.

Then we have

Opcode -->0100011

Funct3 -->011

Rs1--> x26 --> 1101 0

Rs2--> x6 -->0 0110

Immed-->80 --> 0000010 10000

Then we have the binary representation:

0000 0100 0110 1101 0011 1000 0010 0011

Hexadecimal:

**0x46D3823**

6. (10 points) Translate the following RISC-V code into C code. Assume that the C-code integer `result` is held in register `x5`, `x6` holds the C-code integer `i`, and `x10` holds the base address of the integer array `MemArray`.

```
        addi x6, x0, 100
        addi x29, x0, 0
LOOP:   ld x7, 0(x10)
        add x5, x5, x7
        addi x10, x10, 8
        addi x6, x6, -4
        bgt x6, x29, LOOP
```

```
int i;
int j = 0;
for(i = 100; i > 0; i -= 4){
    result += MemArray[j];
    j++;
}
```

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

要在做完之後才減掉4

7. (10 points) Implement the following C code in RISC-V assembly. Note: According to RISC-V spec, "In the standard RISC-V calling convention, the stack grows downward and the stack pointer is always kept 16-byte aligned."

```
int fib(int n){
    if(n == 0)
        return 0;
    else if(n == 1)
        return 1;
    else
        return fib(n-1) + 2*fib(n-2);
}
```

**Note: x10-x11 function argument/return values**

**x12-x17 function argument**

**x5-x7 temporary**

**x9, x18-x27 saved registers**

**fib:**

**beq x10, x0, done #if n == 0, return 0**

**addi x5, x0, 1**

**beq x10, x5, done #if n == 1, return 1**

**addi sp, sp, -16 #allocate space on stack**

**sd x1, 8(sp) #save return address in x1 onto stack**

**sd x10, 0(sp) #save argument in x10 onto stack**

**addi x10, x10, -1 #x10 = n-1**

**jal x1, fib #fib(n-1)**

**addi x9, x10, 0 #move return value of fib(n-1) to x9**

**ld x10, 0(sp) #load old n from stack to x10**

**sd x9, 0(sp) #save fib(n-1) in x0 onto stack**

**addi x10, x10, -2 #x10 = n-2**

**jal x1, fib #fib(n-2)**

**slli x18, x10, 1 #move return value of 2\*fib(n-2) to x18**

**ld x10, 0(sp) #x10 = fib(n-1)**

**add x10, x10, x18 # fib(n-1) + 2\*fib(n-2)**

**ld x1, 8(sp) #load saved return address**

**addi sp, sp, 16 #pop two words from the stack**

**done:**

**jalr x0, 0(x1) #return**

8. (10 points) We would like to expand the RISC-V register file to 128 registers and expand the instruction set to contain four times as many instructions.
- (a) How would this affect the size of each of the bit fields in the R-type instructions?

R-type instruction: 31 25      24 20   19 15   14 12   11 7    6 0

funct7      rs2      rs1      funct3      rd      opcode

**the opcode will increase by 2 bits -->  $7+2 = 9$ bits**

**the rs2, rs1, rd will increase by 2 bits, too -->  $5+2 = 7$ bits**

- (b) How could each of the two proposed changes alone decrease the size of a RISC-V assembly program?

On the other hand, how could the two proposed changes together increase the size of a RISC-V assembly program?

**Increasing the size of each bit field potentially makes the instruction longer, and increase its code size; whereas increasing the number of registers could lead the requirement for increasing the number of bits dedicated to encode the registers, probably increasing the instruction size with effects on the cache and elsewhere.**

**However, increasing the number of registers could lead to less register spillage(storing a variable into memory instead of registers take reference from [https://en.wikipedia.org/wiki/Register\\_allocation](https://en.wikipedia.org/wiki/Register_allocation) ), which would reduce the total number of instructions, possibly reducing the code size overall.**