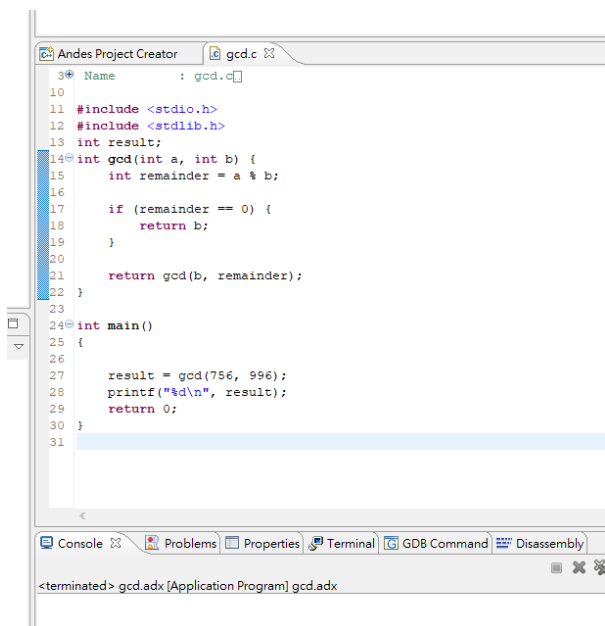


Department of Computer Science
National Tsing Hua University
EECS403000 Computer Architecture
Spring 2023 Homework 2
Deadline: 2023/03/30

1. (40 points)
 - (1) Create an Andes C project and replace it with the gcd.c file.
 - (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 Project Creator IDE. The main window displays the source code for gcd.c. The code is as follows:

```
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 int result;
14 int gcd(int a, int b) {
15     int remainder = a % b;
16
17     if (remainder == 0) {
18         return b;
19     }
20
21     return gcd(b, remainder);
22 }
23
24 int main()
25 {
26
27     result = gcd(756, 996);
28     printf("%d\n", result);
29     return 0;
30 }
31
```

At the bottom, the Disassembly window is open, showing the output of the program: <terminated> gcd.adx [Application Program] gcd.adx.

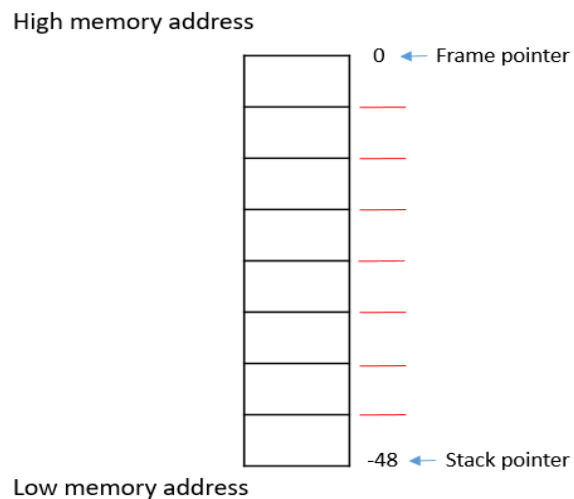
Note: When you check the assembly code of this project, you may see some instructions that start with a “c.”. These instructions are *RISC-V standard compressed instruction set extension*. The extension reduces program code size by adding short 16-bit instruction encodings for common operations. For 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 operands 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) Find the memory addresses of variable/procedure names: `result` and `gcd`, from the assembly code. Show how they are referenced in the instructions. The `gp` register contains the address of the beginning of the static data segment.
- (b) (4 points) You will see the `remw` instruction in the assembly code. Please explain what the purpose of this instruction is.
- (c) (10 points)
 - i. Record the assembly code from memory address 0x104a8 to 0x104ca. Fill in the stack

block by register name and the corresponding memory offsets of the stack in the figure below.

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



- (d) (6 points) Record the assembly code corresponding to the C statement: `return gcd(b, remainder)` ; Briefly explain what these instructions do. (You can use a screenshot or just write down the instructions.)
- (e) (10 points) Do the same as (d) but change the optimization level to `-Og`, and compare the difference between them.

2. (15 points) Answer the following questions based on the figure below.
- (a) (5 points) Give the hexadecimal representation of “beq x10, x0, L1”.
- (b) (5 points) Assume the program executes to “jal x1, 4”. What is the next instruction to execute?
- (c) (5 points) Use lui or auipc to write a sequence of instructions to jump from the memory location 0x0000000020000000 to execute “beq x10, x0, L1” in the figure. Show the memory locations of your instructions and explain your code. You cannot use more than 3 instructions and can only use RV64I instructions.

address	
0x0000000080000000	00000000000100010011000000100011
0x0000000080000004	00000000101000010011010000100011
...	11111111111101010000010100010011
...	jal x1, 4
...	00000000100000010011001010000011
	00000000101000011011110000100011
	11111111111000101000010100010011
	00000000000000000000000011101111
L1:	ld x5, 8(x2)
	00000000000101010001010100010011
	00000000010101010000010100110011
	000000000000000010011000010000011
	00000001000000010000000100010011
	000000000000000001000000001100111
	beq x10, x0, L1
	00000000000100000000001010010011
	00000000010101010000000001100011
	11111111000000010000000100010011

3. (5 points)
- Show how the word “RISC-V2v” coded in 8-bit ascii code would be arranged in the memory of a little-endian and a big-endian machine respectively. Assume that the machines are byte-addressable and the data are stored starting at address 0x00000000. You can use the hex representation instead of binary.
4. (5 points)
- For the following C statement, write the corresponding RISC-V assembly code. Assume that the base addresses of long long int arrays A and B are in registers x6 and x7 respectively. Each element of A or B is 8 bytes, and the variables i and j are assigned to registers x5 and x10 respectively.

```
j = B[A[i*2]] - 16;
```

5. (10 points)

Translate the following RISC-V assembly code to C code and indicate for each line of the assembly code the corresponding C code. Assume that the variables *m*, *n*, *i* and *j* are in registers *x3*, *x4*, *x11*, and *x12*, respectively. Also, assume that each element of array *D* is a 4-byte integer and register *x14* holds the base address of *D*.

```
    addi x11, x0, 0
    addi x30, x14, 0
LOOPI:
    bge x11, x3, ENDI
    addi x12, x0, 0
    lw x31, 0(x30)
LOOPJ:
    bge x12, x4, ENDJ
    add x27, x11, x12
    slli x28, x27, 3
    sub x28, x28, x27
    add x31, x31, x28
    addi x12, x12, 1
    jal x0, LOOPJ
ENDJ:
    sw x31, 0(x30)
    addi x11, x11, 1
    addi x30, x30, 4
    jal x0, LOOPI
ENDI:
```

6. (13 points) Translate the following C code into RISC-V assembly code. You can only use RV64I and RV64M instructions and cannot use pseudoinstructions. Note that 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.”.

```
long long int func(int n) {
    if (n == 0 or n == 1 or n == 2)
        return n;
    else
        return n*n + func(n - 1) + 8*func(n - 2) + func(n - 3);
}
```

7. (12 points) True or false. Give your reasons to get points.

- (a) Assume we use a big-endian, RV64I based machine. Register `t2` contains `0x0000000000000008` and `t3` contains `0x0000B3AA75C316AB`. After executing the following code, `t1` will store `0x00000000000000C3`.

```
add t0, zero, t3
sd  t0, 0(t2)
lb  t1, 2(t2)
```

- (b) If we want to use registers `t0` and `t1` in a function, what we need to do is to store their value in the stack before using them and restore the value before returning.

- (c) Let `x6 = 0x000000007777AAAA`, `x7 = 0x0000000088880000`, `x28 = 0x00000000AAAAEEEE`, `x29 = 0x00000000000000FFF`. After executing the code below, `x6` will contain

`0x0000AAAAAAAAA0000`.

```
or    x6, x6, x7
and   x6, x6, x28
slli  x6, x6, 32
srai  x6, x6, 16
xor   x5, x6, x7
lui   x5, 0xFFFFF
sd    x5, 0(x6)
```

- (d) If we execute the instructions below, the memory (not registers) will be accessed 3 times. Assume the first instruction is stored in `0x0000000000000000`.

```
ld x12, 32(x11)
slli x12, x12, 3
add x12, x12, x10
ld x13, 0(x12)
sub x7, x13, x8
sd x7, 64(x11)
```