Computer Architecture I Mid-Term I

Chinese Name: _____

Pinyin Name: _____

Student ID: _____

E-Mail ... @shanghaitech.edu.cn: _____

Question	Points	Score
1	1	
2	18	
3	11	
4	11	
5	12	
6	6	
7	17	
8	10	
9	14	
Total:	100	

- This test contains 22 numbered pages, including the cover page, printed on both sides of the sheet.
- We will use gradescope for grading, so only answers filled in at the obvious places will be used.
- Use the provided blank paper for calculations and then copy your answer here.
- Please turn **off** all cell phones, smartwatches, and other mobile devices. Remove all hats and headphones. Put everything in your backpack. Place your backpacks, laptops and jackets out of reach.
- The total estimated time is 105 minutes.

- You have 120 minutes to complete this exam. The exam is closed book; no computers, phones, or calculators are allowed. You may use one A4 page (front and back) of handwritten notes in addition to the provided green sheet.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided.
- Do **NOT** start reading the questions/ open the exam until we tell you so!

[1] 1. First Task (worth one point): Fill in you name **[1 point]**
Fill in your name and email on the front page and your ShanghaiTech email on top of every page (without @shanghaitech.edu.cn) (so write your email in total 22 times).

2. **MISC [18 points]**

[2] (a) (_____) The assembler takes two passes over a piece of code to resolve all addresses. **True or False?**

Solution: F

[2] (b) (_____) An RV32I CPU can load 8-bit values from memory, but it can only store the 32-bit word to memory as the smallest unit. **True or False?**

Solution: F

[2] (c) (_____) An RV32I CPU requires all memory stores to be 32-bit aligned, making 8-bit aligned stores impossible. **True or False?**

Solution: F

[2] (d) In C programming, which of the following statements about memory management is correct? (_____)

A. Variables allocated on the stack stay throughout the program's execution.
B. Memory allocated on the heap should be manually freed to prevent memory leaks.
C. Heap memory is used for function call frames.
D. The code segment contains uninitialized global variables.

Solution: B

[2] (e) Which of the following statements correctly describes an instruction in RV32I? (_____)

A. **`sw rs2, offset(rs1)`** stores a value from the register **`rs2`** to memory and requires **`offset`** to be a 4-byte aligned value.
B. **`beq rs1, rs2, offset`** means that if **`x[rs1]`** equals **`x[rs2]`**, the **`PC`** is set to **`sext(offset)`**.
C. **`slli rd, rs1, shamt`** performs a zero-filled left shift on the value in **`rs1`** and stores the result in **`rd`**.
D. **`lw rd, offset(rs1)`** loads a 16-bit value from the address **`offset + x[rs1]`** into **`rd`**; in RV64I, the value is sign-extended to 32 bits.

Solution: C

[2] (f) In the RV32I instruction set, about instruction **`jalr ra, a0, 0x10`**, which of the following statement(s) is (are) false? (_____)

A. The next instruction to be executed is located in memory at address `(0x10 + x[a0])& 0xFFFFFFFE`.

B. After executed, the value of register `ra` is the PC of this `jalr` instruction.

C. In RV32I, `jalr` can be used to implement the operation of returning from a function.

D. In RV32I, `jalr` can be used to implement the operation of calling a function.

Solution: B

2 (g) What will the following `C` code print? (_____)

```c
#include <stdio.h>
int func(int* p) {p = p + 1;}
void main() {
   int ar[2] = {4, 6};
   func(&ar[0]);
   printf("%d", *ar);
   return 0;
}
```

A. 4

B. 5

C. 6

D. 7

Solution: A

2 (h) (_____) Regarding static linking, which of the following statement(s) is/are correct?

A. Static linking requires recompilation to incorporate library updates.

B. Static linking typically results in smaller executable files compared to dynamic linking.

C. Static linking allows sharing of library code between running programs.

D. Static linking includes the entire library in the executable file even if only a portion is used.

E. Static linking resolves all references at runtime.

Solution: A and D. (0 for all the other cases)

2 (i) (_____) In the RISC-V instruction set, the following are conditional branch (B-type) and unconditional jump (J-type) instructions. Choose **all** instructions that will **definitely** change the content of the `ra` (return address) register.

A. `beq t0, t1, label`

B. `beqz t0, label`

C. `j label`

D. **`jal label`**

E. **`jr ra`**

Solution: D

- `beq t0, t1, label` and `beqz t0, label` are conditional branch instructions. They only affect the PC (program counter) and do **not** modify the `ra` register.
- `j label` (which is equivalent to `jal x0, label`) is an unconditional jump, but it does **not** store a return address, so it does **not** change `ra`.
- `jal label` (Jump and Link) **stores** the address of the next instruction in `ra` before jumping, so it **always** modifies `ra`.
- `jr ra`: This is a jump to the address in ra, but it does not modify ra itself. It only changes the PC.

3. **Number Rrepresentation [11 points]**

(a) **`Bfloat16`** (BF16) is a new floating-point number representation. It is commonly used in deep learning to boost the training and inference efficiency, especially in fused multiply-and-add operations (e.g., $A \times B + C$). Compared to the standard IEEE 754 single-precision (FP32) representation, it reduces mantissa from 23 bits to 7 bits, while the other parts remain unchanged. It can be seen as a short version of FP32, skipping the least significant 16 bits of mantissa, including the special cases such as **`nan`**, $\pm$**`inf`**, **`zero`** and denormalized numbers.

Given the bits in hexadecimal, what do these bits represent in BF16? Write down your answer in decimal form.

[2] i. In BF16, **`0xC2C8`** represents _____.

[2] ii. In BF16, **`0x7FC0`** represents _____.

[3] (b) Consider a BF16 addition. Assume the result is still a BF16 number, compute $6.0+1024.0$ in BF16 and write down the result in hexadecimal. Throughout the calculation, round-to-nearest-ties-to-even is used if rounding is required.

In hexadecimal, _____.

What is the value in decimal then?

In decimal, _____.

Hint: 6.0 is **`0x40C0`** and 1024.0 is **`0x4480`** in BF16 representation.

[2] (c) Write down the smallest positive number that can be represented in BF16 representation in 2's power.

The smallest positive number in BF16 is $2^{(\quad)}$.

[2] (d) **True or False** If a number is representable in n-bit 2's complement, its absolute value is also representable in n-bit 2's complement format. Note that this question is not related to BF16. (_____)

If you choose **"False"**, write down the number(s) whose absolute value cannot be represented in n-bit 2's complement format using a expression with n.

_____.

_____.

Solution:

a. 1. -100.

2. **`NaN`**.

b. 1. **`0x4481`**.

2. 1032.

c. -133.

d. F; -2^{n-1}.

4. **C Basics [11 points]**

2 (a) Finish the **add_two** function, where we want to add 2 to an **int** variable passed by the user, and return its previous value. For example, if **x** is 0, after calling **add_two**, the return value is 0 and **x** is 2. Only one statement is allowed for each line, and comma (,) is not allowed.

```
int add_two(_____ x) {

    _____

    _____
}
```

Solution: One possible answer

```
int add_two(int *x) {
    *x += 2;
    return *x - 2;
}
```

3 (b) `#define MUL(a, b) a * b`

The above macro calculates the product of **a** and **b**. What is the value of `16 / MUL(1 + 1, 3 - 1)`? Is the result the same as $16 \div ((1+1) \times (3-1))$? If not, please fix the macro to get the same result.

_____

_____

Solution: 18, No. `#define MUL(a, b) ((a) * (b))`

1 (c) Write a macro that returns the minimum value between **a** and **b**. You should use ternary conditional operator, i.e., `cond ? x : y` returns **x** when **cond** is true, **y** otherwise.

`#define MIN(a, b)` _____

Solution: `#define MIN(a, b) ((a) < (b) ? (a) : (b))` or
`#define MIN(a, b) ((a) > (b) ? (b) : (a))`

3 (d) Based on your **add_two** function in (a) and **MIN** macro in (c), if `a = 15`, `b = 10`, what is the value of **a** after the execution of `MIN(add_two(&a), b)`? What if `a = 10`, `b = 15`? Why?

_____

Solution: 17 and 14, respectively. Because when **a** < **b**, **add_two** is executed twice.

2 (e) Recall in Lab 1, you are required to write a C program to print the sizes of different types. Please write a C macro **PS** that can print the size of the required type, i.e., **PS(long long)** will print the following (**Hint**: the format specifier for **size_t** is **%zu**):

```
Size of long long:  8
```

```
#define PS(type) printf( _____ )
```

Solution: Two possible answers:

```
#define PS(type) printf("Size of "#type":  %zu",
sizeof(type))
#define PS(type) printf("Size of %s:  %zu", #type,
sizeof(type))
```

5. CALL [12 points]

Consider a 32-bit C program solving a LeetCode problem. The code with keyword **`leetcode`** is declared by **`leetcode.h`** and is linked as a dynamic library, while code with keyword **`hash`** declared in **`hash.h`** is statically linked. The compilation of the program does not record debugging information. Please answer the questions based on the provided code and file structure.
Note: Here are some details about the code: the hash map is an integer-to-integer mapping with a default value of 0 if a key does not exist. However, it's irrelevant to the questions.

```c
#include "hash/hash.h"
#include "leetcode/leetcode.h"

#define SUB_ARRAY_SUM 560

static const char *user = "SAKIKO";

void solution(int i) {
    struct { int *nums, size, k, result; } data;
    data.result = 0;

    leetcode_init(SUB_ARRAY_SUM, i, &data);

    int prefix = 0;
    struct hash prefix_hash = hash_init();
    *hash_ref(&prefix_hash, 0) = 1;
    for (int i = 0; i < data.size; i++) {
        prefix += data.nums[i];
        data.result += *hash_ref(&prefix_hash, prefix - data.k);
        ++*hash_ref(&prefix_hash, prefix);
    }

    leetcode_submit(SUB_ARRAY_SUM, i, &data, user);
}

int main(int argc, char *argv[]) {
    int test_cases = leetcode_cases(SUB_ARRAY_SUM);
    for (int i = 0; i < test_cases; i++)
        solution(i);
    return 0;
}
```

5 (a) Lightning Round – Put True or False of the statements in the table provided below:

i. The symbol **`SUB_ARRAY_SUM`** exists after compilation.

ii. Separating source files into **`hash.c`** and **`main.c`** is solely for coding convenience

with no additional benefits.

iii. **`hash_init`** exists in the symbol table of **`hash.o`**, while **`user`** exists in **`main.o`**'s.

iv. Data pointed by **`argv[0]`** is stored in the **`.data`** segment.

v. For the Windows SDK (12 GiB) used by nearly every program on Windows, it is preferable to statically link it.

i	ii	iii	iv	v

Solution: F, F, T, F, F

4 (b) At which stage of the process are all the machine code bits determined for the following statements:

i. `struct hash prefix_hash = hash_init();`

ii. `prefix += data.nums[i];`

iii. `leetcode_init(SUB_ARRAY_SUM, i, &data);`

iv. `++*hash_ref(&prefix_hash, prefix);`

A. After compile B. After assembly C. After linking D. After loading

i	ii	iii	iv

Solution: C, B, C, C

(c) There is a saying: Any problem in computer science can be solved with another layer of indirection. Let's dive into the procedure of compiling and executing a dynamically linked function and answer the following questions.

```
1  000103c0 <_PROCEDURE_LINKAGE_TABLE_>:
2  103c0:    00002397          auipc t2,0x2
3  103c4:    41c30333          sub   t1,t1,t3
4  103c8:    c383ae03          lw    t3,-968(t2) # 11ff8
5  103cc:    fd430313          addi  t1,t1,-44
6  103d0:    c3838293          addi  t0,t2,-968
7  103d4:    00235313          srli  t1,t1,0x2
8  103d8:    0042a283          lw    t0,4(t0)
9  103dc:    000e0067          jr    t3
10                     ...
11 00010400 <leetcode_cases@plt>:
12 10400:    00002e17          auipc t3,0x2
```

```
10404:    c08e2e03          lw    t3,-1016(t3) # 12008
10408:    000e0367          jalr  t1,t3
1040c:    00000013          nop
                       ...
000105f2 <main>:
                       ...
105fe:    3509              jal   10400
                       ...
00011ff8 <.got.plt>:
11ff8:    ffffffff
11ffc:    00000000
12000:    000103c0
12004:    000103c0
12008:    000103c0
```

1 i. (_____) What is the value in the **PC** register when the given code ends?

A. **11ff8.** B. **12008.**
C. **ffffffff.** D. **10400.**

Hint: the program begins execution from **main()**. Please read the assembly code and try to analyze the calling flow of the **dynamically linked functions**.

2 ii. (_____) When actually running the C program, the value you selected in the last question is modified to the real address of the function to call to realize dynamic linking when it ends; which of the following stages does this modification the most likely happen at?

A. Compiler. B. Assembler.
C. Dynamic linker. D. Loader.

Solution: i. C; ii. C or D.

6. **Logic [6 points]**

2 (a) (_____) How many input variable combinations does a four-input **NOR** gate have that result in an output of 1?

A. 15.
B. 8.
C. 7.
D. 1.

Solution: D.

2 (b) **(Multiple Choice)** (_____) Which of the following statement(s) are(is) true about boolean algebra? (where $\oplus$ denotes **XOR**)

A. $X(Y \oplus Z) = (XY) \oplus (XZ)$.
B. $(X \oplus \overline{Y}) \oplus Z = X \oplus (\overline{Y} \oplus Z)$.
C. $X + YZ = (X + Y)(X + Z)(\overline{X} + Y + Z)$.
D. $(X + Y)(\overline{X} + Z)(Y + Z) = (X + Y)(\overline{X} + Z)$.

Solution: ABD. 2 for exactly the same with the answer. 0 for all the other cases.

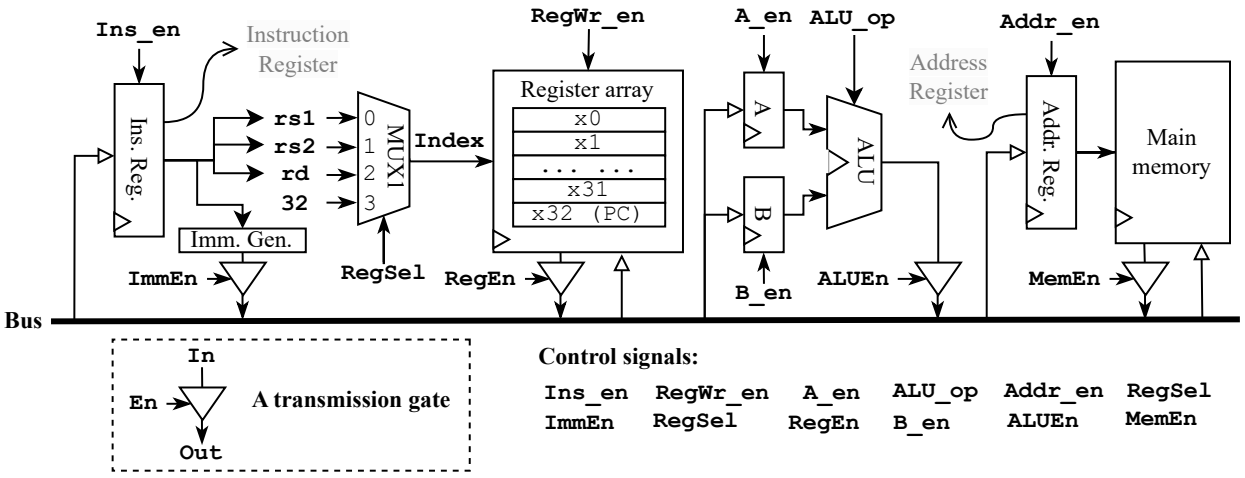
2 (c) (_____) Build a logic circuit with only 2-input AND and 2-input **OR** gates to implement the function shown in the truth table below. What are the minimal numbers of needed **AND** gates and **OR** gates?

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

A. 3 AND gates and 3 OR gates.
B. 3 AND gates and 2 OR gates.
C. 2 AND gates and 3 OR gates.
D. 2 AND gates and 2 OR gates.
E. 2 AND gates and 1 OR gate.
F. 1 AND gate and 2 OR gates.

Solution: D.

7. **Datapath [17 points]**
Except the datapath we learned from class, there are also other ways to build a computer. We will explore a bus-based design as shown in the figure below. It supports RV32I R- and I-type arithmetic and logic operations, and part of the load and store instructions only. As covered in the lectures, we assume that the main memory and the register array change their states at the rising edge of clock, while memory/register array read is purely combinational logic.



You can consider "bus" as a bunch of metal wires. It is used to pass digital signals from one device to another. In this design, the bus contains 32 wires, which means that it can only pass 1 32-bit digital signal at a time. There are 4 devices that can set value to the bus, namely the immediate generator (Imm. Gen.), the register array, the ALU and the Main memory. There are also several 32-bit registers that can read from the bus, such as the instruction register (Ins. Reg.), the register array, the A and B registers (A and B in the figure) and the address register (Addr. Reg.), controlled by enable signals **`Ins_en`**, **`RegWr_en`**, **`A_en`**, **`B_en`**, **`Addr_en`**, respectively. At one time, at most one device can set value or write to the bus (this is achieved by enable/disable the transmission gates), while all the devices can read from the bus.

To reduce the hardware cost, the PC register is integrated into the register array along with the 32 general purpose registers in RV32I ISA. It is marked as **`x32 (PC)`** in the figure, and can be indexed by 32. The general purpose registers can still be indexed by its corresponding numbers. We also assume that when the 4 devices are all disconnected from the bus, arbitrary signals can be assigned to the bus.

2 (a) To avoid signal conflict, only 1 device is allowed to write to the bus at one time. This is achieved by using the transmission gates. The MOSFET diagram of a transmission gate is shown below. When the enable signal (**`En`**) is _____, the corresponding device is connected to the bus and thus can write to the bus; while the enable signal (**`En`**) is _____, the corresponding device is disconnected to the bus and thus cannot modify the signal on the bus.

Solution: __1__; __0__

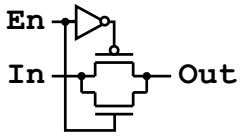
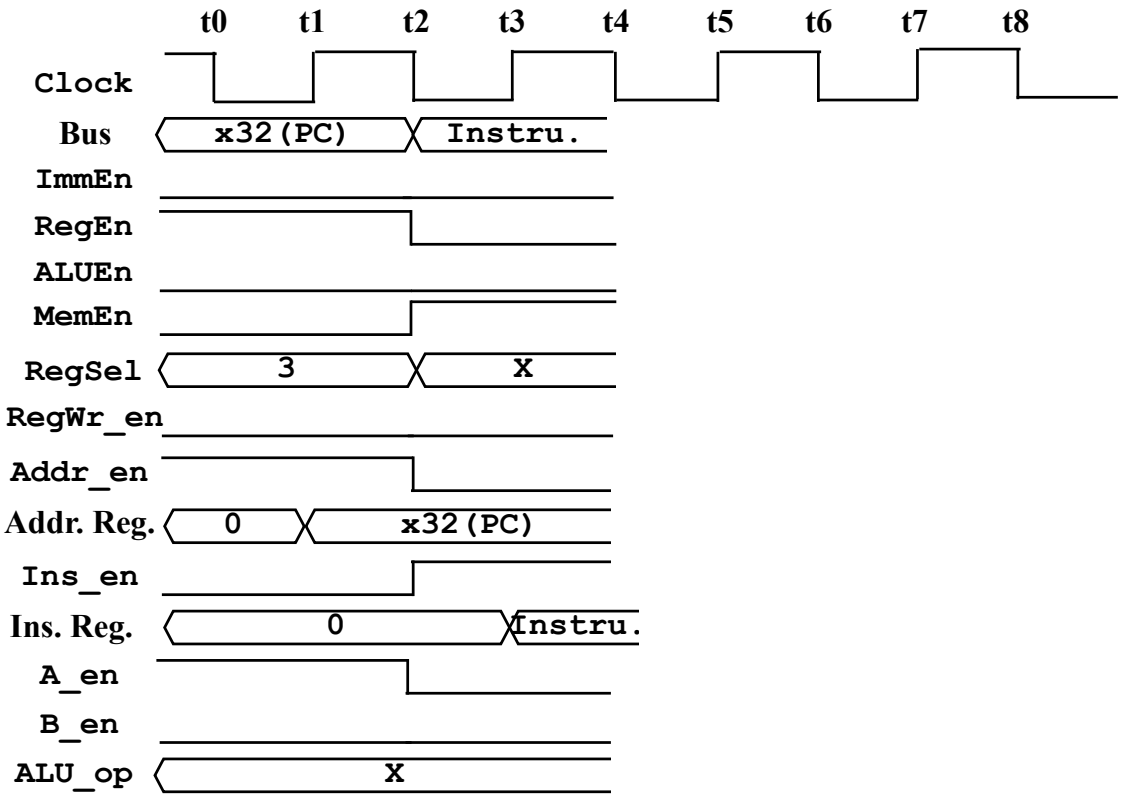


Figure 1: A transmission gate.

10 (b) Since the signals are transmitted through the bus instead of directly from device to device as in our lectures, the signals have to be transmitted serially and it takes multiple clock cycles to complete 1 instruction. For example, "instruction fetch" itself would take 2 clock cycles as shown in the timing diagram below. For ease of read, we list the waveform of relevant signals only in the diagram. We use **`X`** to indicate that we do not care what the signal is, and it does not affect the function of the circuit. **`Instru.`** stands for the instruction read from the Main memory at address **`PC`**.



From time period **t0** to **t1**, **`RegEN`** is enabled and **`RegSel`** is set to 3 so that **`x32 (PC)`** is read from the register array and the value is sent to the bus. When a clock rising edge arrives at time **t1**, since the address register is enabled, it loads data from the bus and updates its value to **`x32 (PC)`**. Correspondingly, the instruction is read from the main memory to the bus at time **t2**, with **`RegEN`** disabled and **`MemEN`** enabled. At the next clock rising edge (**`t3`**), the instruction register then fetches the instruction from the bus with **`Ins_en`** enabled. A controller (not shown in the figure) then decodes the instruction and generates necessary control signals to execute the instruction.

Assume an R-type instruction is fetched, we can load the operands from the register array to the A and B registers for computation by the ALU. To load **rs1** to A register, please fill in the control signals properly. **ImmEn** is _____, **RegSel** is _____, **RegWr_en** is _____, **RegEn** is _____, **A_en** is _____, **B_en** is _____, **ALUEn** is _____ and **MemEn** is _____.

This R-type instruction takes at least _____ clock cycles to complete. (Hint: count the total number of rising clock edges in total to complete the instruction fetch, operand fetch (instruction decode), execution, write-back phases. You should also setup the registers so that the computer is able to execute the next instruction automatically.)

Solution: 0; 0; 0; 1; 1; 0; 0; 0.
7 clock cycles
Clock cycle 1: `A=PC; Addr.Reg.=PC;`
Clock cycle 2: `Ins.Reg.=instruction;`
Clock cycle 3: `Bus=4 and B=4;`
Clock cycle 4: `PC (x32)=PC+4;`
Clock cycle 5: `A=x[rs1];`
Clock cycle 6: `B=x[rs2];`
Clock cycle 7: `x[rd]=R[A,B].`

2 (c) The bit width of **Index** should be _____ to access all the registers in the register array. Therefore, **rs1**, **rs2** and **rd** should _____ (A. make no change; B. pad 0 at the least significant bit; C. pad 0 at the most significant bit; D. pad 1 at the most significant bit). The bit width of **RegSel** should be _____.

Solution: 6; C; 2.

3 (d) By inserting pipeline registers between the given components only, is the above datapath able to pipeline? If yes, please provide a simple example; if no, explain your reason.

Solution: No. The instructions already are processed stage by stage in this implementation. What is more important, to implement pipeline, the components for each stage should work in parallel. However, with the current microarchitecture design, it is not possible to provide enough data to these components simulteneously by the bus. Therefore, it incurs structural hazard. In an actual computing system, communication between components is always time-consuming and in many cases becomes the bottleneck when improving the performance. This question serves as a concrete example.
No for 1 point. As long as you mention "structural hazard", you got the other 2 points.

8. **Calling Convention [10 points]**
Xiao Ming is solving a math problem:

Problem Statement: You are given a rectangular board of size $1 \times n$. You need to tile this board using two types of tiles:

- A 1×1 tile (a small square).
- A 1×2 tile (a domino).

How many distinct ways can you tile the board when $n = 9$?

Xiao Ming discovered that this problem follows the recurrence relation:

$$f(n) = f(n-1) + f(n-2)$$

He wrote the following program to solve it:

```
.data
# Store n = 9.
n: .word 9

.text
# load n, pass n to f.
la a0, n
lw a0, 0(a0)
jal f

# print the result
mv a1, a0
li a0, 1
ecall

# exit the program
li a0, 10
ecall

f:
   # A: Prologue
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________

   # Termination condition: if (n <= 2) return n;
   li t0, 2
   ble a0, t0, base_case

   # Compute f(n-1)
```

```
    addi a0, a0, -1 # Pass n - 1
    call f          # Call f(n-1)
    mv s0, a0       # Save return value to s0

    # B: Pass paramters for f(n-2)
    ____________________
    ____________________
    ____________________

    call f          # call f(n-2)
    add a0, a0, s0 # f(n) = f(n-1) + f(n-2)

    # C: Epilogue (restoring saved registers)
    ____________________
    ____________________
    ____________________
    ____________________
    ret
# Base case: when n <= 2, return n
base_case:
    # D: Set the correct return value
    ____________________
    ____________________
    ____________________

    # E: Epilogue
    ____________________
    ____________________
    ____________________
    ____________________
    ret
```

10 (a) Please complete the RV32I assembly implementation of the function **f**, following the comments and the calling conventions. Use the minimal stack space required for saving registers and ignore stack alignment requirements. Save only the minimal registers necessary. We may provide more blank lines than you need.

Solution:

```
// A: prologue
addi sp, sp, -12
sw ra, 8(sp)
sw a0, 4(sp)
sw s0, 0(sp)

// B: Compute f(n-2)
lw a0, 4(sp)
```

```
addi a0, a0, -2

// C: Epilogue (restore saved registers)
lw s0, 0(sp)
lw ra, 8(sp)
addi sp, sp, 12

// D: Set the return value
(lw a0, 4(sp)) // Not necessary to have this

// E: Epilogue
(lw ra, 8(sp)) // Not necessary to have this
addi sp, sp, 12
```

9. **RISC-V [14 points]**

12 (a) Convolution is a fundamental mathematical operation widely used in signal processing, image processing, and machine learning. Convolution performs a weighted sum of one array using the weights defined by another array (the kernel or filter). The 1D convolution can be mathematically expressed by the following formula:

$$y[i] = \sum_{n=0}^{K-1} x[i+n] \cdot h[n] \tag{1}$$

Here, K represent the size of the convolution kernel. The indices i denote the position of the output element in the output array.

Fill in the following RISC-V codes to implement the 1D convolution operation. Assume the assembly is for a 32-bit machine. To simplify, we consider that multiplication (**`x[rd]=x[rs1]*x[rs2]`**) can be implemented with **`mul rd, rs1, rs2`** instruction and you do not have to consider the overflow.

```
# Input array address: x10
# Kernel (filter) address: x11
# Output array address: x12
# Input array size: x13
# Kernel size: x14
main:
   addi x5, x0, 0  # Initialize outer loop counter (i)
   addi x15, x13, 1
   sub x15, x15, x14

outer_loop:
   beq x5, x15, end
   addi x6, x0, 0  # Initialize inner loop counter (n)
   addi x7, x0, 0  # Initialize sum

inner_loop:
   beq x6, x14, store_result
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   addi x6, x6, 1  # Increment inner loop counter
```

```
   jal x0, inner_loop

store_result:
   _________________
   _________________
   _________________
   _________________
   addi x5, x5, 1  # Increment outer loop counter
   jal x0, outer_loop # Jump back to outer_loop

end:
   # End program
```

Solution:

```
   inner_loop:
    beq x6, x14, store_result
    addi x8, x0, 4
    mul x9, x8, x6
    add x9, x9, x11
    lw x9, 0(x9)
    add x16, x5, x6
    mul x16, x8, x16
    add x16, x16, x10
    lw x16, 0(x16)
    mul x9, x9, x16
    add x7, x7, x9
    addi x6, x6, 1  # Increment inner loop counter
    jal x0, inner_loop

store_result:
   mul x9, x8, x5
   add x9, x9, x12
   sw x7, 0(x9)
   addi x5, x5, 1  # Increment outer loop counter
   jal x0, outer_loop # Jump back to outer_loop
```

2 (b) Translate the machine code in **hexadecimal** below to RV32I instructions.

```
0x40E787B3 _____________________

0x02E30863 _____________________
```

Solution: `sub x15, x15, x14;`
`beq x6, x14, 0x30 or 48.`