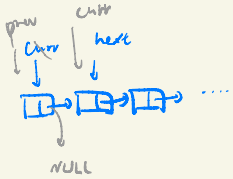
Note: x_0 的值恒为 0

a.

2021

The equivalent C code for this operation is provided below.

```
void reverse(struct node **head)
{
    struct node *prev = NULL, *curr = *head;
    while (curr) {
        struct node *next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
}
```



The RISC-V assembly implementation is shown as following:

```
# head is passed in a0
# t0 holds prev
# t1 holds next
    lw a0, 0(a0)
    beqz __A01__
    addi t0, t0, 0?
loop:
    lw t1, 0(a0)
    sw t0, 0(a0)
    addi t0, a0, 0
    addi a0, t1, 0
    __A02__
done:
```

// 把 head 的 value load 到 a_0，即 $a_0 = *head$ (address)

// A01: beqz a_0, done —— 如果为空，直接跳到 done

// $t_0 = t_0$?

// $t_1 = *head$

// $*head = t_0$

// $t_0 = a_0$

// $a_0 = t_1$

// A02: bnez t_1, loop (bnez a_0, loop ?)

beqz: branch if equal to zero

B.

```
unsigned multiply(unsigned x, unsigned y) {
    unsigned ret = 0;
    for (; y != 0; y--)
        ret += x;
    return ret;
}
```

x 存在 x_1 reg.
y 存在 x_2 reg.

We can translate into the optimized RISC-V assembly, so that fewer dynamic instructions are executed for large values of x and y.

A simple optimization is loop unrolling. With some careful bookkeeping, this can reduce the number of additions. This implementation keeps x + x as a temporary value to cut the number of additions approximately in half.

The following is more efficient than simply translating from C.

x_3 : result

```
      addi x3, x0, 0   # result = 0
      andi __B01__     # x? = y - (y%2)
      add x7, x1, x1   # x7 = x + x
      beq x2, x0, done # y is 0
      beq __B02__      # skip ahead if y is even
      add x3, x0, x1   # set result to x if y is odd
loop: add __B03__    # result = result + 2x
      addi x6, x6, -2  # x6 = x6 - 2
      bge x6, x0, loop # repeat if x6 != 0
done:
```

C.

The function `strtriple` receives as input a null-terminated string, and writes a string where every letter is repeated three times. For example, if we called stringtriple on the input string `"Hello World!"`, we would get the output string `"HHHeeellllllooo WWWooorrrllldddd!!!"`.

More specifically: `strtriple` has the following inputs: `a0` stores a null-terminated string. It is guaranteed that the string is well-formatted. Let the string have strlen of `n`. `a1` contains a pointer to a buffer of length at least 3n+1 characters, potentially containing garbage data. You may assume that the buffer does not overlap with the string stored in `a0`.

`strtriple` outputs the following: `a0` should return the pointer to the buffer we initially received in `a1`. The buffer should contain the string originally provided in `a0`, with every character tripled. You must properly null-terminate the string, and for full credit must NOT replicate the null terminator at the end. For full credit, you must follow proper calling convention.

```
strtriple:
    mv t2, a1
loop:
    lbu __C01__
    beq t0, x0, end
    slli __C02__
    add t1, t1, t0
    sh t1, 0(a1)
    sb __C03__
    addi a0, a0, 1
    __C04__
    j loop
end:
    sb x0, 0(a1)
    __C05__
    jr ra
```

sh: save half word

slli : shift left logic immediate

D,

```
int SL_find(void *find, int level,
            struct SkipListNode *sl,
            (int)(*cmp)(void *, void *) {
    if (cmp(find, sl->data) == 0) return 1;
    if (level == 0 && sl->next[level] == NULL) return 0;
    if (sl->next[level] == NULL)
        return SL_find(find, level - 1, sl, cmp);
    if (cmp(find, sl->next[level]->data)) > 0))
        return SL_find(find, level - 1, sl, cmp);
    return SL_find(find, level, sl->next[level], cmp);
}
```

In translating the code `cmp(find, sl->next[level]->data))` into RISC-V, we need to store arguments on the stack. So we will have `find` at `sp(0)`, `level` at `sp(4)`, `sl` at `sp(8)` and `cmp` at `sp(12)`. We are going to break up the translation into pieces. Please write down the RISC-V assembly accordingly.

1 Load find into `a0`
- D01 = ? lw a0 , 0(sp) // argument of find

2 The next thing we need to do is get `sl->next[level]->data` into `a1`. The RISC-V code for that would be: (Hint: Load `sl` into `t0`)
- D02 = ? lw t0 , 8(sp) // argument of sl

3 Load `level` into `t1`
- D03 = ? lw t1, 4(sp) // argument of level

4 load `sl->next` into `t0`
- D04 = ? lw t0, 4(t0) // move it to next (4 byte) (1 word), and load

5 load `sl->next[level]` into `t0`

```
slli __D05__
add t0, t0, t1
lw t0, t0(4)
```

slli t1, t1, ~~4~~ 2 // level offset 16倍?? 2吧? 一般不是 4 byte 吗?

level addr. load sl-> next [level]

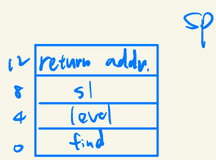
- D05 = ?

6 load data into `a0`
- D06 = ? lw a0, 0(t0)

7 Finally, how do we call `cmp` (using `t0` as a temporary):

```
lw __D07__
__D08__
```

- D07 = ? lw t0, 12(sp)
- D08 = ? jalr ra, t0



Problem E

You are supposed to implement RISC-V code per requested as less instructions as possible. Follow calling convention, use register mnemonic names (e.g., refer to `t0` rather than `x6`), and add commas and a single space between registers/arguments (e.g. `addi a0, a1, 2`). If you do not follow this, you may be misgraded.

1. Implement less-than for floating-point numbers.

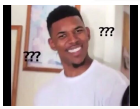
- Inputs: `a0` and `a1` are positive non-NaN IEEE-754 32-bit floating point numbers.
- Output: `a0` returns `1` if `a0 < a1` and `0` otherwise.
- Example: If the inputs `a0` and `a1` were `1.2` and `1.25`, the expected output would be `1`. If `a0` and `a1` were `1.1` and `1.1` respectively, the expected output would be `0`.

```
fp_less_than:
    __E01__ slt a0, a0, a1
    ret
```

ret: return from subroutine (pseudo instruction)

2. Find the end position of C-style string.

- Inputs: `a0` is a pointer to a nonempty string
- Output: `a0` returns a pointer immediately after the end of the string.
- Example: Let `a0` be the pointer `0x10000000` to string s, which is the string `"Hello"`. Then the expected output would be `0x10000006`. 0x10000005
- NOTE: You may NOTE use `strlen` or any other library function.




```
end_of_str:
    lb t0, 0(a0)
    __E02__ addi a0, a0, 1 // move to next byte
    __E03__ bne t0, x0, end_of_str
```

loop

- E02 = ?
- E03 = ?

3. Find the length of a null-terminated string in bytes. The function should accept a pointer to a null-terminated string and return an integer. Your solution must be recursive.

```
strlen:
        __E04__
        beq t0, zero, basecase
        addi sp, sp, -4
        __E05__
        __E06__
        jal strlen
        addi a0, a0, 1
        __E07__
        __E08__
        ret
basecase:
        addi a0, zero, 0
        ret
```

- E04 = ?
- E05 = ?
- E06 = ?
- E07 = ?
- E08 = ?

4. Arithmetically negate a Two's Complement 32-bit integer without using the `sub`, `mul` or pseudo instructions.

```
negate:
    __E09__
    addi a0, a0, 1
    ret
```

- E09 = ?

5.

For example, if the input to this function is `10`, then the output would be `1010`. The equivalent RISC-V code:

```
find_binary:
    addi sp, sp, -8 # a0 will have arg and be where we return
    sw ra, 4(sp) # saving return address on the stack
    sw s0, 0(sp) # saving "decimal % 2" on the stack
    __E10__ #  we will just return 0
    andi s0, a0, 1 # set s0 to a0 % 2
    __E11__ # set a0 to a0 / 2
    jal ra, find_binary # recursive call
    li t0, 10
    __E12__
    add a0, a0, s0 # accumulating the stored return value "decimal % 2"
postamble:
    lw ra, 4(sp) # Restore ra
    __E13__ # restore s0
    __E14__ # restore sp
end:
    jr ra
```

Hint:

- The recursive part of the function stores all of the first part of the return value “decimal % 2” on the stack
- The second part of the function and postamble are combing all the return values by `+` and `10 *`

6.

```
strncpy:
    add t0, x0, x0 # Current length
loop:
    __E15__ # Check if we have iterated over the length of the string
    add t1, a1, t0 # calculate the address of the t0-th (count-th) char in the string
    lb t2, 0(t1) # load this char
    __E16__ # calculate the same address, but off of the dst string
    sb t2, 0(t1) # store into the dst string
    addi t0, t0, 1 # increment our counter
    bne t1, x0, loop # check if the loaded char was null
end:
    jr ra
```

F

```
auipc t0, 0xABCDE # Assume this instruction is at 0x100
addi t0, t0, 0xABC
```

Write down the value of `t0` in HEX.

- F01 = ?

0xABCDE100 + 0xABC

= 0xABCDEBBC

```
li t0, 0xABCDEFAD
sw t0, 0(s0)
lb t0, 0(s0)
```

Write down the value of `t0` in hex. Assume big-endianness.

- F02 = ? 0xFFFFFFAB

auipc: add upper immediate to pc

auipc rd, uimm20

20 bit，填12bit 補0，共32bit，與 pc 相加 寫入 rd

G.

We define the following structure.

uint32_t : unsigned integer 32 bits

```
#include <stdint.h>
struct BloomFilter {
    uint32_t size; /* Size is # of bits, NOT BYTES, in the bloom filter */
    uint16_t itercount;
    uint64_t (*)(void *data, uint16_t iter) hash;
    uint8_t *data;
};
```

For the insertion function, we need to set the appropriate bit for each iteration.

```
void insert(struct BloomFilter *b, void *element) {
    uint64_t bitnum; /* which bit we need to set */
    for (int i = 0; i < b->itercount; ++i) {
        bitnum = b->hash(element, (uint16_t) i) % b->size;
        b->data[bitnum >> 3] = b->data[bitnum >> 3] | (1 << (bitnum & 0x7));
    }
}
```

We also have the following function to allocate a new bloom filter

prolog

body

```
struct BloomFilter *alloc(uint64_t (*)(void *data, uint16_t iter) hash,
                          uint32_t size,
                          uint16_t itercount) {
    struct BloomFilter *ret = malloc(64);
    ret->size = size;
    ret->data = calloc(size >> 3, 1);
    ret->hash = hash;
    ret->itercount = itercount;
}
```

Complete the RISC-V translation necessary to allocate this: We will put `ret` in `s0`.

ret : jalr x0, 0(x1)

return from subroutine

```
alloc:
# Prolog
    addi sp, sp, __G01__ -20
    sw ra, 0(sp)
    sw s0, 4(sp) data
    sw a0, 8(sp) hash
    sw a1, 12(sp) size
    sw a2, 16(sp) itercount
# body
    addi a0, x0, 64
    jal malloc
    mv s0, a0 # put ret in s0
    __G02__ # load size into t0
    __G03__ # store it
    __G04__ # div size by 8 with a shift
    li a1, 1
    jal calloc
    sw a0, 12(s0) # store data
    __G05__ # load hash to t0
    __G06__ # store it: Use the right type
    __G07__ # load itercount to t0
    sh t0, 4(s0) # store it
    mv a0, s0
# epilog
    lw ra, 0(sp)
    lw s0, 4(sp)
    addi sp, sp, 20
    jr ra
```

Struct

size	0
itercount	4
hash	8
data	12

H. Please complete the translation by filling in the lines below with RISC-V assembly. The prologue and epilogue have been written correctly but are not shown.

`strlen()`, both as a C function and RISC-V procedure, takes in one string as an argument and returns the length of the string (not including the null terminator).

```c
/* Returns 1 if s2 is a substring of s1, and 0 otherwise. */
int is_substr(char *s1, char *s2) {
    int len1 = strlen(s1), len2 = strlen(s2);
    int offset = len1 - len2;
    while (offset >= 0) {
        int i = 0;
        while (s1[i + offset] == s2[i]) {
            i += 1
            if (s2[i] == '\0') return 1;
        }
        offset -= 1;
    }
    return 0;
}
```

The translated RISC-V Assembly:

```
is_substr:
    mv s1, a0
    mv s2, a1
    jal ra, strlen
    mv s3, a0
    mv a0, s2
    jal ra, strlen
    sub s3, s3, a0
Outer_Loop:
    blt s3, x0, False
    add t0, x0, x0
Inner_Loop:
    add t1, t0, s3
    add t1, s1, t1
    lbu t1, 0(t1)
    __H01__
    __H02__
    bne t1, t2, __H03__
    addi t0, t0, 1
    add t2, t0, s2
    lbu t2, 0(t2)
    beq t2, x0, True
    jal x0, Inner_Loop
Update_Offset:
    addi s3, s3, -1
    __H04__
False:
    xor a0, a0, a0
    jal x0, End
True:
    addi a0, x0, 1
End:
```

2022

A.

```
node_t *partition(node_t *head, int x)
{
    node_t *L = NULL, **p1 = &head, **p2 = &L;

    for (node_t *node = head; node; node = node->next) {
        node_t ***indir = (node->val >= x) ?
            (A01 /* fill your code */) : (A02 /* fill your code */);
        **indir = node;
        *indir = A03 /* fill your code */;
    }

    *p1 = L, *p2 = NULL;
    return head;
}
```

bge : branch if greater than or equal

A01: &p2

A02: &p1

A03: &node->next

B.

This code has to be converted to C code below.

```
#include <stdlib.h>

typedef struct vector { int x, y; } vector_t;

vector_t *transform(vector_t *self, int (*f)(int))
{
    vector_t *nv = malloc(sizeof(vector_t));
    nv->x = f(self->x), nv->y = f(self->y);
    return nv;
}
```

Translate the `transform` function to RISC-V. The function takes inputs `self` in `a0` and `f` in `a1`, and returns output in `a0`. You may assume that `vector_t` is as defined in the C code. You may also assume that you have access to `malloc`, and that `malloc` and f each receive their argument in `a0`, and return their result in `a0`. Your solution MUST fit within the lines provided.

```
transform:
    addi sp, sp, B01
    B02
    B03
    B04
    B05
    mv s0, a0
    mv s1, a1
    li a0, 8
    jal ra, malloc
    mv s2, a0
    lw a0, 0(s0)
    B06
    sw a0, 0(s2)
    lw a0, 4(s0)
    B07
    sw a0, 4(s2)
    mv a0, s2
    B08
    B09
    B10
    B11
    addi sp, sp, B12
    ret
```

self in a_0

f in a_1

returns in a_0

Both `B06` and `B07` MUST use `jalr` instruction. You MUST use ABI name for accessing registers defined in RISC-V Calling Convention. i.e, use `ra` instead of `x1` while it appears in `jalr` instruction.

- B01 = ?
- B02 = ?
- B03 = ?
- B04 = ?
- B05 = ?
- B06 = ?
- B07 = ?
- B08 = ?
- B09 = ?
- B10 = ?
- B11 = ?
- B12 = ?

C.

The stack can hold local variables in addition to the `ra` register and the `s` registers. The following labels that were externally specified are available to you:

- `password`: a pointer to a statically-stored string “secretpass”
- `get_chars`: A function defined as follows:
  - Input: `a0` is a pointer to a buffer
  - Effect: Reads characters from stdin, and fills the buffer pointed to by `a0` with the read data, null-terminating the string. Your code may assume that the input is at most `19` characters, not including the null-terminator.
  - Output: None

The function `verify_password` is defined as follows:

- Input: No register input; however, the function receives a string input from stdin.
- Output: `a0` returns `1` if the input from stdin is exactly “secretpass”, and `0` otherwise.

Complete the function `verify_password`. Each line contains exactly one instruction or pseudoinstruction.

```
verify_password:
    addi sp, sp, -24 # Make space for a 20-byte buffer
    C01 # Your code here
    mv a0, sp
    jal ra, get_chars
    la t0, password # load absolute addr.
    mv t1, sp
Loop:
    lb t2, 0(t0)
    lb t3, 0(t1)
    C02 # Your code here
    C03 # Your code here
    addi t0, t0, C04 # Your code here
    addi t1, t1, C05 # Your code here
    j Loop
Pass:
    li a0, C06 # Your code here
    j End
Fail:
    li a0, C07 # Your code here
End:
    C08 # Your code here
    C09 # Your code here
    jr ra
```

In order to enhance their models in machine learning, some data scientists add random noise to a training dataset. Here, we will take a dataset and turn it into RISC-V data that can be used.

The function `jitter` is defined as follows:

- Inputs:
  – `a0` holds a pointer to an integer array.
  – `a1` holds a buffer large enough for n integers (which does not overlap with the array in `a0`).
  – `a2` holds `n`, the length of the arrays.
- Output:
  – `a0` holds a pointer to the buffer originally in a1. The buffer is filled with the result of calling `noisen` on eac element in the `a0` array.

The function `noisen` is defined as follows:

- Input: `a0` holds an integer.
- Output: `a0` returns the integer with noise added.

The implementation of `jitter` is provided below, following calling convention:

```
jitter:
    # BEGIN PROLOGUE
    addi sp, sp, -20
    # We need to make space for 5 registers on the stack,
    # and each register is 4 bytes long.
    # (multiple lines omitted)
    # END PROLOGUE
    mv s0, a0
    mv s1, a1 # Hold beginning of output arr
    mv s2, a1
    mv s3, a2 # Hold counter
loop:
    beq s3, x0, end
    lw a0, 0(s0)
    jal ra, noisen
    sw a0, 0(s1)
    addi s0, s0, 4
    addi s1, s1, 4
    addi s3, s3, -1
    j loop
end:
    mv a0, s2
    # BEGIN EPILOGUE
    # (multiple lines omitted)
    addi sp, sp, 20
    # END EPILOGUE
    ret
```

Part 1: List all registers that we need to save on the stack in order to follow calling convention. __ D01 __

- D01 = ? ra, s0, s1, s2, s3

Part 2: Now, we want to implement `noisen` to add some random offset to an integer `a0`. Unfortunately, we only have access to the `randomBool` function, which takes in no inputs and randomly returns either `1` or `0` in `a0`. If we implemented `noisen` to return `a0 + randomBool()`, the integer would always get shifted in the positive direction. Instead, we suggest implementing `noisen` so that noisen alternates between returning `a0 + randomBool()` and returning `a0 - randomBool()`.

Fill in the blanks to complete the above suggested implementation of `noisen`. Assume that you can read from and write to any memory addresses, in any section of memory.

D.

```
noisen:
    addi sp, sp, -8 # Prologue
    sw ra, 0(sp)
    sw s0, 4(sp)
    mv s0, a0
    jal ra, randomBool
    add s0, s0, a0
    D02 # Your code here
    xori t0, t0, 64
    D03 # Your code here
    mv a0, s0 # Epilogue
    lw ra, 0(sp)
    lw s0, 4(sp)
    addi sp, sp, 8
    ret
```

lb t0, 3(ra)

sb t0, 3(ra)

- D02 = ?
- D03 = ?

Extremely difficult

E

```
float_lt
    E01 # Your code here
    ret

E01: slt a0, a0, a1

skipline:
    E02 # Your code here
    ret

E02: addi ra, ra, 4
```