Lab 02

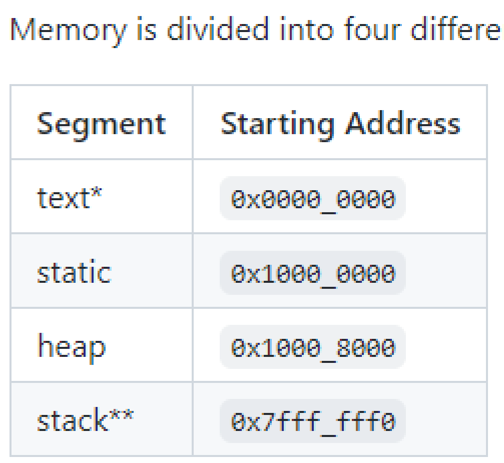
Exercise1

1. What do the `.data`, `.word`, `.text` directives mean ?

- **`.data` :** Store subsequent items in the static segment at the next available address which mean that .data store in data section of memory (starting at 0x10000000)

- **`.word` :** Store listed values as unaligned 32-bit words
Unaligned memory access is the access of data with a size of **N** number of bytes from an address that is not evenly divisible by the number of bytes **N**.

- **`.text` :** Store subsequent instructions in the text segment at the next available address which mean that .text store in text section of memory where you can write your program.

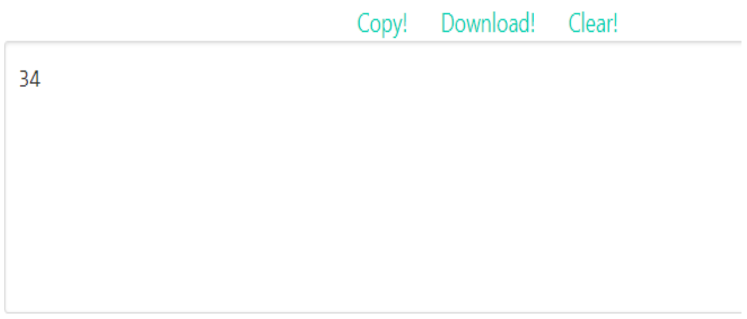


Memory is divided into four different segments: text, static, heap and stack.

Segment	Starting Address
text*	`0x0000_0000`
static	`0x1000_0000`
heap	`0x1000_8000`
stack**	`0x7fff_fff0`

2. Run the program to completion. What number did the program output? What does this number represent?

The program out is 34



```
34
```

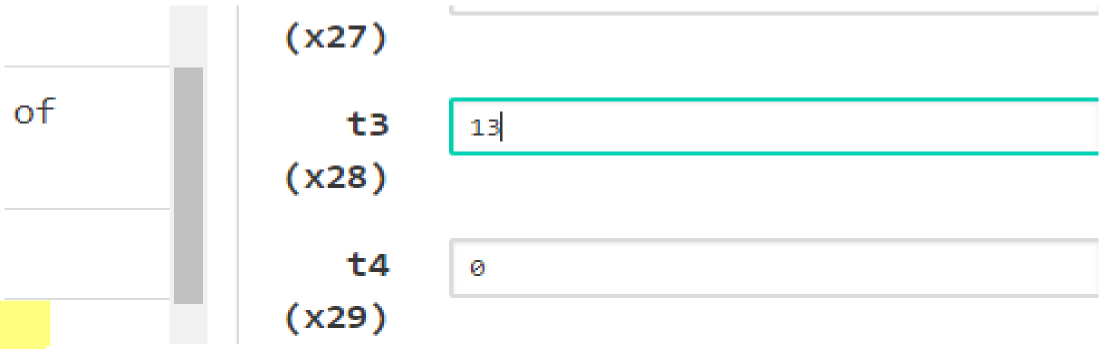
This number represent the value of t0 register which represent one of Fibonacci sequence register

3. At what address is `n` stored in memory? Hint: Look at the contents of the registers.

n stored at address= 0x10000010 in memory

4. Without using the "Edit" tab, have the program calculate the 13th fib number (0-indexed) by *manually* modifying the value of a register.

Yes , 13th fib number = 233



```
t3 (x28)   13
t4 (x29)   0
```

```
0x3c        0x00000073    ecall                         ecall # termi
```

```
233
```

Exercise 2: Translating from C to RISC-V

Task: Find/explain the following components of this assembly file.

- **The register representing the variable `k`.**

  Because I need to change the address each cycle of loop and I can't do a variable offset, so I need to add the offset to the base address each cycle by using the instructions
  add t4, t1, t3
  add t6, t2, t3
  to have the absolute address in a register

- **The registers acting as pointers to the `source` and `dest` arrays**

  Because I load the addresses of source and dest in registers, so these registers become pointers to them
  t1 = source base address
  t2 = dest base address

t4 = the new address of source element each cycle
t6 = the new address of dest element each cycle

➢ The assembly code for the loop found in the C code.

```
loop:
slli t3, t0, 2          # each char is 4B, so we mult by 4 (i.e., << 2)
add t4, t1, t3          # t4 = absolute source array index
lw t5, 0(t4)            # t5 = current element
beq t5, x0, exit        # end when element is 0
add t6, t2, t3          # t6 = absolute dest array index
sw t5, 0(t6)
addi t0, t0, 1
j loop                  # jal x0, loop
exit:
```

➢ How the pointers are manipulated in the assembly code.

We load the base addresses of arrays to registers using load absolute address (psuedo instruction)
and calculate the offset of moving in another register then add these two values and store them in another register to have the new position of pointers

the instructions that used in manipulating the pointers

```
la t1, source
la t2, dest
```

slli t3, t0, 2 # each char is 4B, so we mult by 4 (i.e., << 2) we calculate the offset

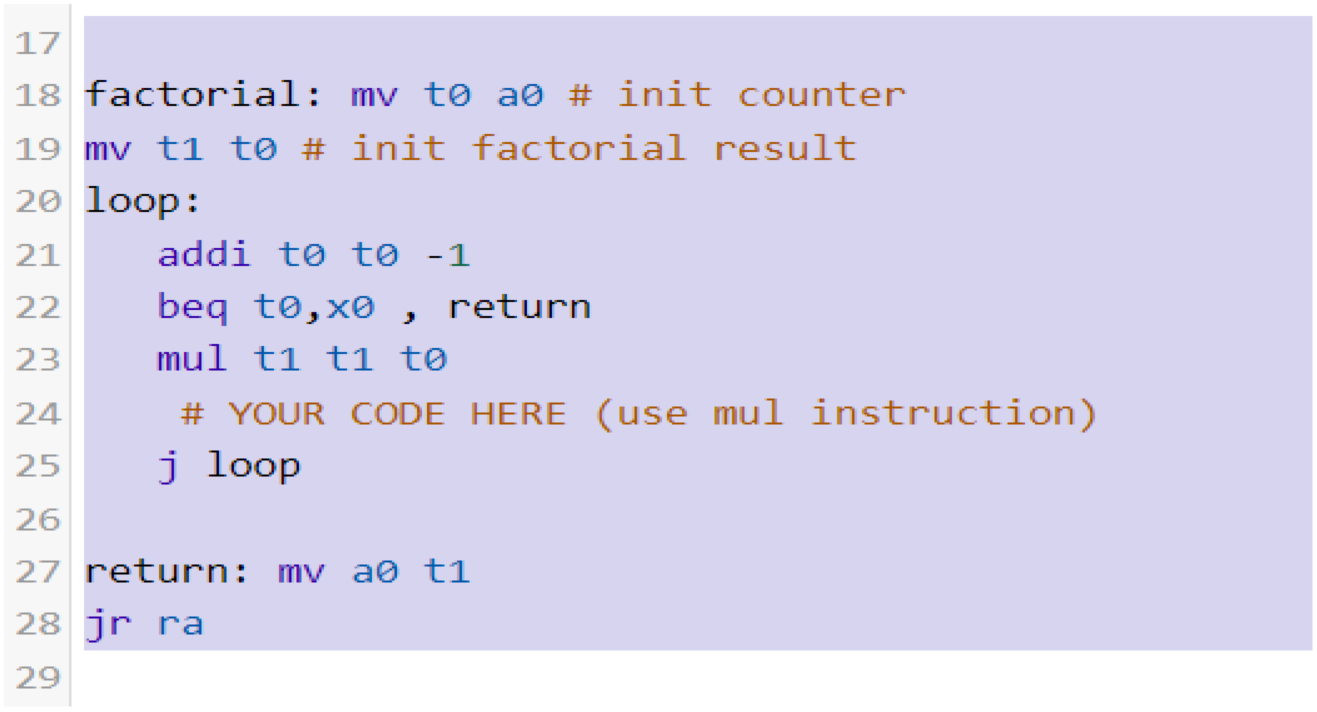
add t4, t1, t3 # t4 = absolute source array index

lw t5, 0(t4) # t5 = current element

add t6, t2, t3 # t6 = absolute dest array index

Exercise 3: Factorial

The factorial code is :



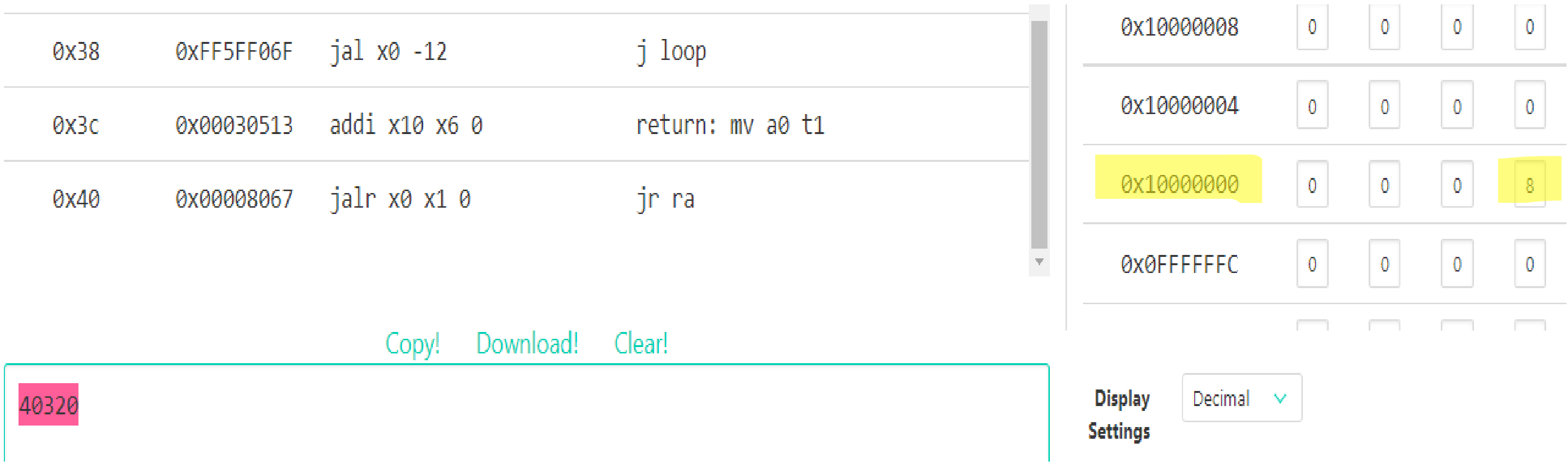
```

factorial: mv t0 a0 # init counter
mv t1 t0 # init factorial result
loop:
   addi t0 t0 -1
   beq t0,x0 , return
   mul t1 t1 t0
    # YOUR CODE HERE (use mul instruction)
   j loop

return: mv a0 t1
jr ra

```

The output when n = 8 is 40320



```
0x38    0xFF5FF06F   jal x0 -12        j loop
0x3c    0x00030513   addi x10 x6 0     return: mv a0 t1
0x40    0x00008067   jalr x0 x1 0      jr ra
```

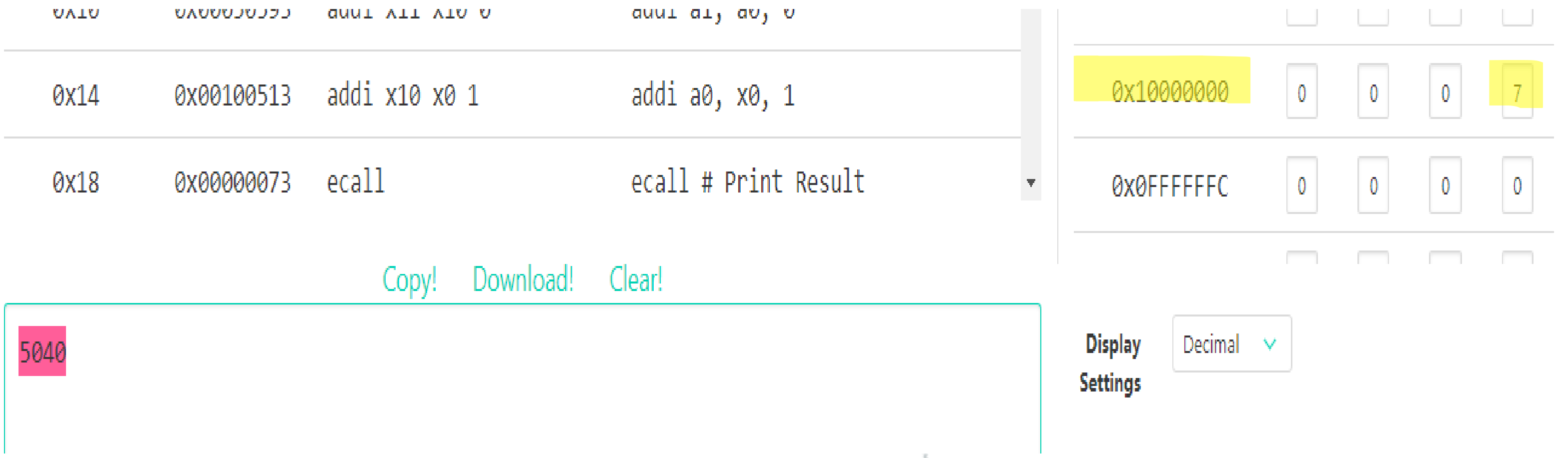
```
0x10000008   0  0  0  0
0x10000004   0  0  0  0
0x10000000   0  0  0  8
0x0FFFFFFC   0  0  0  0
```

Copy! Download! Clear!

```
40320
```

Display Settings: Decimal

the out when n=7 is 5040



```
0x14    0x00100513   addi x10 x0 1     addi a0, x0, 1
0x18    0x00000073   ecall             ecall # Print Result
```

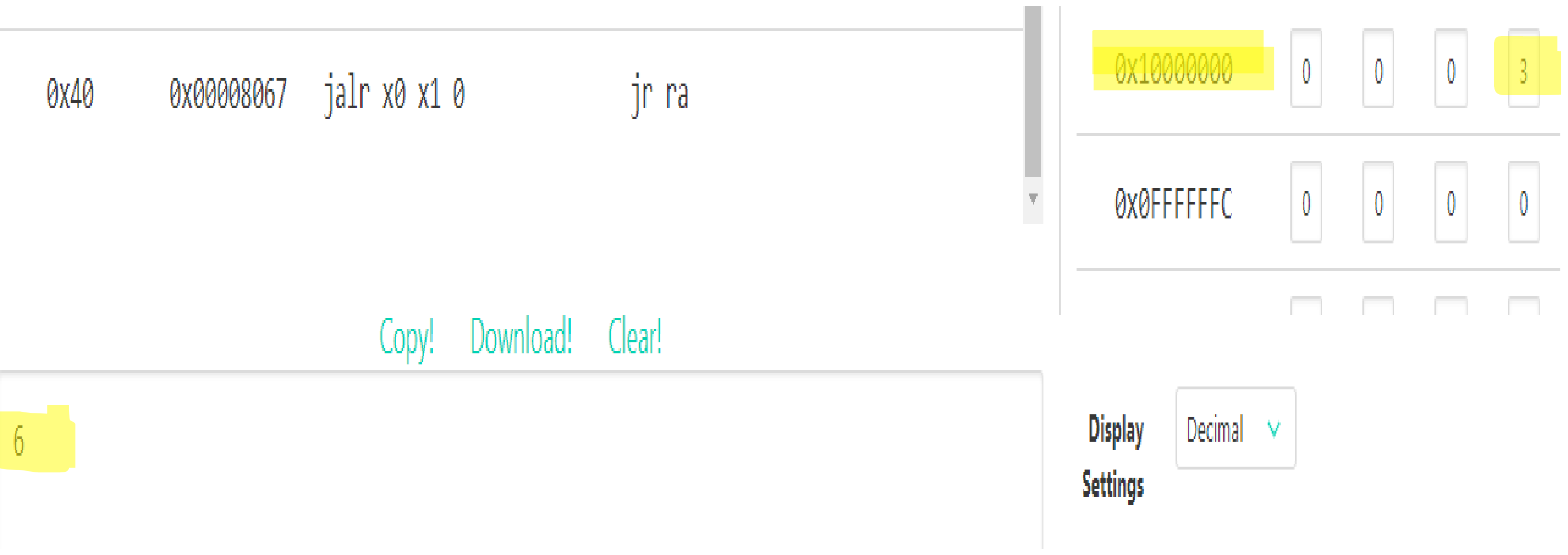
```
0x10000000   0  0  0  7
0x0FFFFFFC   0  0  0  0
```

Copy! Download! Clear!

```
5040
```

Display Settings: Decimal

the out when n=3 is 6



```
0x40    0x00008067    jalr x0 x1 0        jr ra
```

```
0x10000000    0    0    0    3
0x0FFFFFFC    0    0    0    0
```

```
6
```

Display Settings: Decimal

Exercise 4: RISC-V function calling with `map`

The modified code is :

```
.data
start_msg:  .asciiz "List before: "
end_msg:    .asciiz "List after: "

.text
main:
    jal create_default_list
    mv s0, a0  # s0 = a0 is head of node list
    #print the list
    jal ra, print_list

    # print a newline
    jal print_newline

    # load your args
    mv a0, s0  # load the pointer to the first node into a0

    # load the address of the function in question into a1 (check out la)
    # WRITE ONE LINE HERE
     la a1 square
    # issue the call to map (arguments: list pointer in a0, fn pointer in a1)
    jal map

    #print the list
    mv a0, s0
    jal ra, print_list

    addi a0, x0, 10
    ecall #Terminate the program

map:
    # map arguments: list pointer in a0, fn pointer in a1
    # backup in the stack
    # WRITE ONE LINE HERE
    addi sp, sp, -12
    sw  ra, 0(sp)
    sw  s0, 4(sp)
    sw  s1, 8(sp)

    beq a0, x0, done    # If we were given a null pointer (address 0), we're done.

    # Remember you cannot use s0 and s1 values set by the caller even if they are
what you need
    mv s0, a0  # Save the pointer to the node in s0
    mv s1, a1  # Save the pointer to the function in s1

    # Remember that each node is 8 bytes long: 4 for the value followed by 4 for
the pointer to next.
    # What does this tell you about how you access the value and how you access
the pointer to next?

    # load the value of the current node into a0
    # THINK: why a0? square function will use this parameter
    # WRITE ONE LINE HERE
    lw a0 0(s0)
    # Call the function in question on that value. DO NOT use a label (be prepared
to answer why).
    # because the the function is actually a parameter (we pass a pointer to it)
    # What function? Recall the parameters of "map"
    # WRITE ONE LINE HERE
    jalr s1

    # store the returned value back into the node
    # Where can you assume the returned value is?
    # WRITE ONE LINE HERE
    sw a0 0(s0)

    # Load the address of the next node into a0
    # The Address of the next node is an attribute of the current node.
    # Think about how structs are organized in memory.
    # WRITE ONE LINE HERE
    lw a0 4(s0)

    # Put the address of the function back into a1 to prepare for the recursion
    # THINK: why a1? What about a0?
    # a0 contains address of node
    # WRITE ONE LINE HERE
     add a1 s1 x0

    # pop saved registers
    lw  ra, 0(sp)    # h
    lw  s0, 4(sp)    # h
    lw  s1, 8(sp)    # h
    # WRITE ONE LINE HERE
    jal ra map
    # recurse
    # WRITE ONE LINE HERE

done:
    # Epilogue: Restore register values and free space from the stack
     addi sp, sp, 12
    lw  ra, 0(sp)    # h
    lw  s0, 4(sp)    # h
    lw  s1, 8(sp)    # h
    # WRITE ONE LINE HERE
    jr ra

    ret # Return to caller

square:
    mul a0 ,a0, a0
    ret

create_default_list:
    # push three words in the stack
    addi sp, sp, -12
    sw  ra, 0(sp)
    sw  s0, 4(sp)
    sw  s1, 8(sp)
    li  s0, 0      # pointer to the last node we handled
    li  s1, 0      # data index (i) to fill the node values
    # we go from last node to first node (we go backward)
    # last node pointer is S1 = 0 (termination)
loop:
    li  a0, 8       # do... (each node is 8 bytes = 4 bytes data + 4 bytes pointer)
    jal malloc     # get a0 (8 bytes) memory for the next node, returns pointer in a0
    sw  s1, 0(a0)   # node->value = i (store node value in heap)
    sw  s0, 4(a0)   # node->next = last (store pointer to next node in heap)
    mv  s0, a0        # s0 (pointer stored in prevoius node) = a0 (pointer to current
node value)
    # Note that we are going backwards: next iteration is the previous node
    addi s1, s1, 1   # i++ (increment node value for the previous node)
    li t0, 10       # a list of 10 nodes
    bne s1, t0, loop    # ... while i!= 10 (create a list of 10 nodes)
    # pop the three saved words from the stack
    lw  ra, 0(sp)
    lw  s0, 4(sp)
    lw  s1, 8(sp)
    addi sp, sp, 12
```

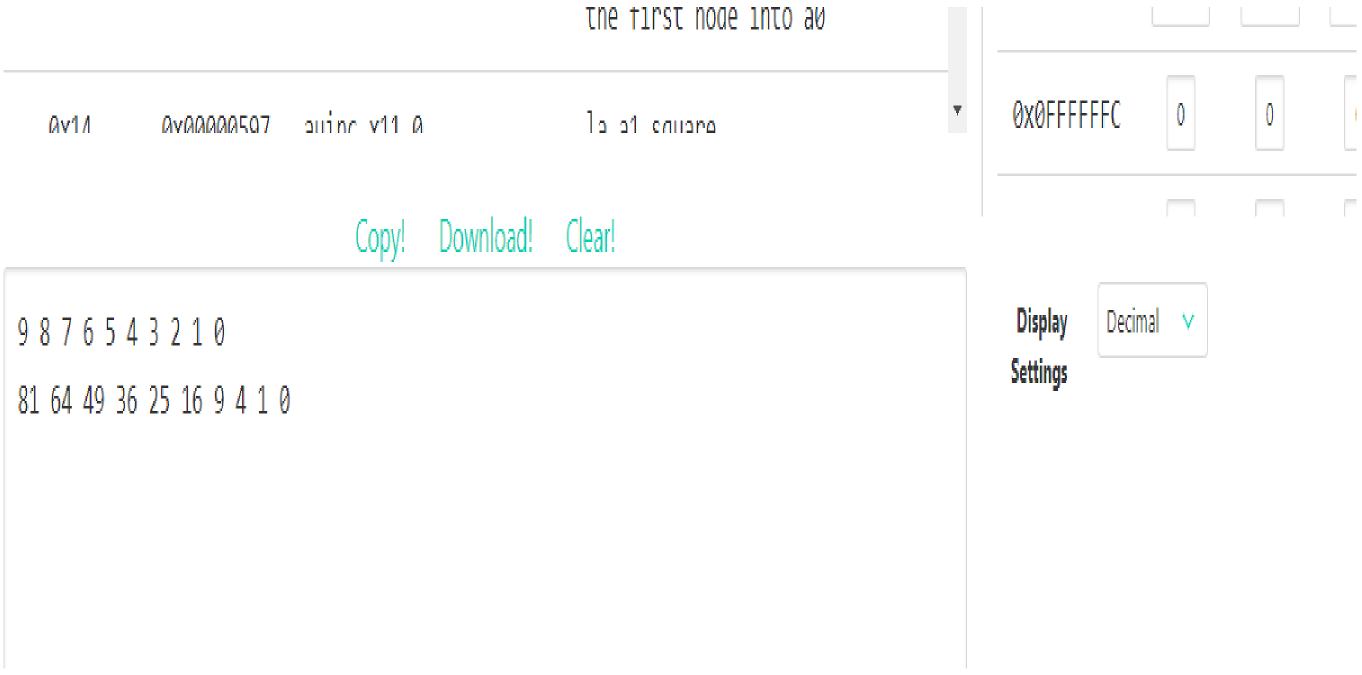
```
    # a0 (the return value) points to the first element in the list
    ret

print_list:
    # a0 points to the first element in the list
    # loop till last element (null terminated)
    bne a0, x0, printMeAndRecurse
    ret       # nothing to print

printMeAndRecurse:
    mv t0, a0  # t0 gets current node address
    lw  a1, 0(t0)   # a1 gets value in current node
    addi a0, x0, 1      # prepare for print integer ecall (prints the integer in a1 = node
value)
    ecall
    li    a1, ' '     # a0 gets address of string containing space
    li    a0, 11      # prepare for print string syscall (prints the string in a1 = space)
    ecall
    lw  a0, 4(t0)   # a0 gets pointer to next node
    j print_list  # recurse

print_newline:
    li    a1, '\n' # Load in ascii code for newline
    li    a0, 11
    ecall
    ret


malloc:
    mv      a1, a0    # a1 = a0 = 8 bytes (memory to be allocated for each node)
    li      a0, 9    # ID (a0) = 9 = malloc(): allocates a1 bytes on the heap, returns
pointer to first byte in a0
    ecall    # 1st iteration will set a0 to 0x10008000 (pointer to first address of heap)
    ret
```



```
                                        the first node into a0

   0x14      0x00000597   auipc x11 0        la a1 square
```

```
0x0FFFFFFC   0   0
```

```
9 8 7 6 5 4 3 2 1 0
81 64 49 36 25 16 9 4 1 0
```

Display Settings: Decimal