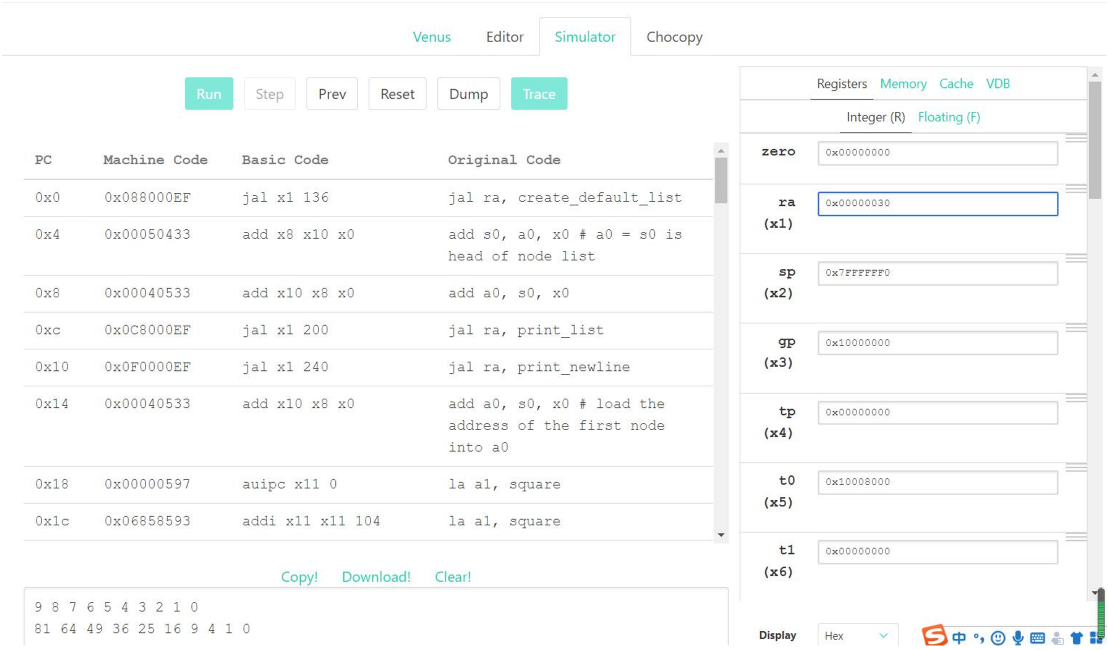
# Q3:

```
.globl map

.text
main:
	jal ra, create_default_list
	add s0, a0, x0	# a0 = s0 is head of node list

	#print the list
	add a0, s0, x0
	jal ra, print_list

	# print a newline
	jal ra, print_newline

	# load your args
	add a0, s0, x0	# load the address of the first node into a0

	# load the address of the function in question into a1 (check out la on the green sheet)
	### YOUR CODE HERE ###
	la a1, square
	# issue the call to map
	jal ra, map

	# print the list
	add a0, s0, x0
	jal ra, print_list

	# print another newline
	jal ra, print_newline

	addi a0, x0, 10
	ecall #Terminate the program

map:
	# Prologue: Make space on the stack and back-up registers
	### YOUR CODE HERE ###
	addi sp sp -12
	sw s0, 8(sp)
	sw s1, 4(sp)
	sw ra, 0(sp)

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

```
    mv s0, a0 # Save address of this node in s0
    mv s1, a1     # Save address of 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?
    ### YOUR CODE HERE ###
    lw a0, 0(s0)
    # Call the function in question on that value. DO NOT use a label (be prepared to answer why).
    # What function? Recall the parameters of "map"
    ### YOUR CODE HERE ###
    jalr s1
    # store the returned value back into the node
    # Where can you assume the returned value is?
    ### YOUR CODE 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.
    ### YOUR CODE 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?
    ### YOUR CODE HERE ###
    mv a1, s1
    # recurse
    ### YOUR CODE HERE ###
    jal ra, map
done:
    # Epilogue: Restore register values and free space from the stack
    ### YOUR CODE HERE ###
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw ra, 0(sp)
    addi sp sp 12
    jr ra # Return to caller

square:
```

```
        mul a0 ,a0, a0
        jr ra

create_default_list:
        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              # number of nodes handled
loop:     #do...
        li    a0, 8
        jal ra, malloc           # get memory for the next node
        sw    s1, 0(a0)      # node->value = i
        sw    s0, 4(a0)      # node->next = last
        add s0, a0, x0    # last = node
        addi        s1, s1, 1      # i++
        addi t0, x0, 10
        bne s1, t0, loop        # ... while i!= 10
        lw    ra, 0(sp)
        lw    s0, 4(sp)
        lw    s1, 8(sp)
        addi sp, sp, 12
        jr ra

print_list:
        bne a0, x0, printMeAndRecurse
        jr ra              # nothing to print
printMeAndRecurse:
        add t0, a0, x0    # t0 gets current node address
        lw    a1, 0(t0)      # a1 gets value in current node
        addi a0, x0, 1           # prepare for print integer ecall
        ecall
        addi        a1, x0, ' '          # a0 gets address of string containing space
        addi        a0, x0, 11           # prepare for print string syscall
        ecall
        lw    a0, 4(t0)      # a0 gets address of next node
        jal x0, print_list     # recurse. We don't have to use jal because we already have where we
want to return to in ra

print_newline:
        addi        a1, x0, '\n' # Load in ascii code for newline
        addi        a0, x0, 11
        ecall
```

```
    jr    ra

malloc:
    addi      a1, a0, 0
    addi      a0, x0 9
    ecall
    jr    ra
```




# Q4:

```
    .globl map

    .data
    arrays: .word 5, 6, 7, 8, 9
                .word 1, 2, 3, 4, 7
                .word 5, 2, 7, 4, 3
                .word 1, 6, 3, 8, 4
                .word 5, 2, 7, 8, 1

    start_msg:    .asciiz "Lists before: \n"
    end_msg:       .asciiz "Lists after: \n"

    .text
    main:
        jal create_default_list
        mv s0, a0       # v0 = s0 is head of node list

        #print "lists before: "
```

```
        la a1, start_msg
        li a0, 4
        ecall

        #print the list
        add a0, s0, x0
        jal print_list

        # print a newline
        jal print_newline

        # issue the map call
        add a0, s0, x0          # load the address of the first node into a0
        la    a1, mystery       # load the address of the function into a1

        jal ra, map

        # print "lists after: "
        la a1, end_msg
        li a0, 4
        ecall

        # print the list
        add a0, s0, x0
        jal print_list

        li a0, 10
        ecall

map:
        addi sp, sp, -12
        sw ra, 0(sp)
        sw s1, 4(sp)
        sw s0, 8(sp)

        beq a0, x0, done        # if we were given a null pointer, we're done.

        add s0, a0, x0          # save address of this node in s0
        add s1, a1, x0          # save address of function in s1
        add t0, x0, x0          # t0 is a counter

        # remember that each node is 12 bytes long:
        # - 4 for the array pointer
        # - 4 for the size of the array
```

```
        # - 4 more for the pointer to the next node

        # also keep in mind that we should not make ANY assumption on which registers
        # are modified by the callees, even when we know the content inside the functions
        # we call. this is to enforce the abstraction barrier of calling convention.
    mapLoop:
        lw t1, 0(s0)        # load the address of the array of current node into t1
        lw t2, 4(s0)        # load the size of the node's array into t2

        slli t3, t0, 2
        add t1, t1, t3      # offset the array address by the count
        lw a0, 0(t1)        # load the value at that address into a0

        mv t4, t1
        jalr s1             # call the function on that value.
        mv t1, t4

        sw a0, 0(t1)        # store the returned value back into the array
        addi t0, t0, 1      # increment the count
        bne t0, t2, mapLoop # repeat if we haven't reached the array size yet

        lw a0, 8(s0)        # load the address of the next node into a0
        mv a1, s1           # put the address of the function back into a1 to prepare for the
recursion

        jal ra, map         # recurse
    done:
        lw s0, 8(sp)
        lw s1, 4(sp)
        lw ra, 0(sp)
        addi sp, sp, 12
        jr ra

    mystery:
        mul t1, a0, a0
        add a0, t1, a0
        jr ra

    create_default_list:
        addi sp, sp, -4
        sw ra, 0(sp)
        li s0, 0    # pointer to the last node we handled
        li s1, 0    # number of nodes handled
        li s2, 5    # size
```

```
	la s3, arrays
loop: #do...
	li a0, 12
	jal malloc		# get memory for the next node
	mv s4, a0
	li a0, 20
	jal    malloc		# get memory for this array

	sw a0, 0(s4)		# node->arr = malloc
	lw a0, 0(s4)
	mv a1, s3
	jal fillArray	# copy ints over to node->arr

	sw s2, 4(s4)		# node->size = size (4)
	sw    s0, 8(s4)	# node-> next = previously created node

	add s0, x0, s4	# last = node
	addi s1, s1, 1	# i++
	addi s3, s3, 20 # s3 points at next set of ints
	li t6 5
	bne s1, t6, loop # ... while i!= 5
	mv a0, s4
	lw ra, 0(sp)
	addi sp, sp, 4
	jr ra

fillArray: lw t0, 0(a1) #t0 gets array element
	sw t0, 0(a0) #node->arr gets array element
	lw t0, 4(a1)
	sw t0, 4(a0)
	lw t0, 8(a1)
	sw t0, 8(a0)
	lw t0, 12(a1)
	sw t0, 12(a0)
	lw t0, 16(a1)
	sw t0, 16(a0)
	jr ra

print_list:
	bne a0, x0, printMeAndRecurse
	jr ra	# nothing to print
printMeAndRecurse:
	mv t0, a0 # t0 gets address of current node
	lw t3, 0(a0) # t3 gets array of current node
```

```
        li t1, 0    # t1 is index into array
    printLoop:
        slli t2, t1, 2
        add t4, t3, t2
        lw a1, 0(t4) # a0 gets value in current node's array at index t1
        li a0, 1    # preparte for print integer ecall
        ecall
        li a1, ' ' # a0 gets address of string containing space
        li a0, 11    # prepare for print string ecall
        ecall
        addi t1, t1, 1
      li t6 5
        bne t1, t6, printLoop # ... while i!= 5
        li a1, '\n'
        li a0, 11
        ecall
        lw a0, 8(t0) # a0 gets address of next node
        j print_list # recurse. We don't have to use jal because we already have where we want
to return to in ra

    print_newline:
        li a1, '\n'
        li a0, 11
        ecall
        jr ra

    malloc:
        mv a1, a0 # Move a0 into a1 so that we can do the syscall correctly
        li a0, 9
        ecall
        jr ra
```

Q5:

```
.globl f

.data
neg3:    .asciiz "f(-3) should be 6, and it is: "
neg2:    .asciiz "f(-2) should be 61, and it is: "
neg1:    .asciiz "f(-1) should be 17, and it is: "
zero:    .asciiz "f(0) should be -38, and it is: "
pos1:    .asciiz "f(1) should be 19, and it is: "
pos2:    .asciiz "f(2) should be 42, and it is: "
pos3:    .asciiz "f(3) should be 5, and it is: "

output: .word    6, 61, 17, -38, 19, 42, 5
.text
main:
    la a0, neg3
    jal print_str
    li a0, -3
    la a1, output
    jal f                  # evaluate f(-3); should be 6
    jal print_int
    jal print_newline

    la a0, neg2
    jal print_str
    li a0, -2
    la a1, output
    jal f                  # evaluate f(-2); should be 61
    jal print_int
    jal print_newline

    la a0, neg1
    jal print_str
    li a0, -1
    la a1, output
    jal f                  # evaluate f(-1); should be 17
    jal print_int
    jal print_newline

    la a0, zero
    jal print_str
    li a0, 0
```

```
        la a1, output
        jal f                    # evaluate f(0); should be -38
        jal print_int
        jal print_newline

        la a0, pos1
        jal print_str
        li a0, 1
        la a1, output
        jal f                    # evaluate f(1); should be 19
        jal print_int
        jal print_newline

        la a0, pos2
        jal print_str
        li a0, 2
        la a1, output
        jal f                    # evaluate f(2); should be 42
        jal print_int
        jal print_newline

        la a0, pos3
        jal print_str
        li a0, 3
        la a1, output
        jal f                    # evaluate f(3); should be 5
        jal print_int
        jal print_newline

        li a0, 10
        ecall

# f takes in two arguments:
# a0 is the value we want to evaluate f at
# a1 is the address of the "output" array (defined above).
# Think: why might having a1 be useful?
f:
        # YOUR CODE GOES HERE!
        addi t0, a0, 3       # index
        slli t0, t0, 2
        add t0 a1, t0
        lw a0, 0(t0)
        jr ra                    # Always remember to jr ra after your function!
```

```
print_int:
	mv a1, a0
	li a0, 1
	ecall
	jr	ra

print_str:
	mv a1, a0
	li a0, 4
	ecall
	jr	ra

print_newline:
	li a1, '\n'
	li a0, 11
	ecall
	jr	ra
```