Lab04: RISC-V functions, pointers

Friday, July 22, 2022 12:13 AM

Exercise1: Array Practice by implementing a discrete function

Considering a discrete-valued function f defined on integers in set
{-3, -2, -1, 0, 1, 2, 3}
Where: f(-3) = 6; f(-2) = 61 ; f(-1) = 17; f(0) = -38; f(1) = 19; f(2) = 42; f(3) = 5

Implement a function **without** any branches or Loops

The directives, data, argument-value array are declared as:

```
# The .globl directive identifies functions that we want to export to other files,
# similar to including a function in a header file in C
.globl f

.data
# asciiz is a directive used to store strings
# asciiz will automatically append a null terminator to the end of the string
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
```

Main function calling example

```
main:
    ######## evaluate f(-3), should be 6 ########
    # load the address of the string located at neg3 into a0
    # this will serve as the argument to print_str
    la a0, neg3
    # print out the string located at neg3
    jal print_str
    # load the first argument to f into a0
    li a0, -3
    # load the second argument of f into a1
    # `output` is a pointer to an array that contains the possible output values of f
    la a1, output
    # execute f(-3)
    jal f
    # f will return the output of f(-3) into register a0
    # to print out the return value, we will call print_int
    # print_int expects the value that it's printing out to be in register a0
    # the output of the function is already in a0, so we don't need to move it
    jal print_int
    # print a new line
    jal print_newline
```

Implemented discrete function

```
# 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).
f:
    # YOUR CODE GOES HERE!
    addi t0, a1, 0 # make a copy of base address
    addi t1, a0, 3 # fix the index
    addi t2, x0, 4 # Int size
    mul t3, t1, t2
    add t0, t0, t3
    lw a0, 0(t0)
    jr ra               # Always remember to jr ra after your function!
```

Exercises2: Calling Convention Practice

This exercise requires us to fix all the calling convention errors in the following code
Note:

1. When to store return address in ra?

When caller calls a function, we need to **return** to the **location the function was called**

2. Calling convention checker only looks for bugs in functions that are exported with the .global **directive** :

- **.global** directive **identifies function**s that we want to **export** to other files: similar to including a function in the header file in C
- In the Venus online tool, **.import** directive can make labels marked **.global** be available

0. A few initialization directives

```
.globl pow inc_arr

.data
fail_message: .asciiz "%s test failed\n"
pow_string: .asciiz "pow"
inc_arr_string: .asciiz "inc_arr"

success_message: .asciiz "Tests passed.\n"
array:
    .word 1 2 3 4 5
exp_inc_array_result:
    .word 2 3 4 5 6
```

1. Main function

```
.text
main:
    # pow: should return 2 ** 7 = 128
    li a0, 2
    li a1, 7
    jal pow          # Equivalent to jal ra, pow
    li t0, 128 # verifies that pow returned the right value
    beq a0, t0, next_test
    la a0, pow_string
    j failure

next_test:
    # inc_arr: increments "array" in place
    la a0, array     • Load array address into a0
    li a1, 5         • Load array length into a1
    jal inc_arr
    jal check_arr # Verifies inc_arr returned the right value
    # all tests pass, exit normally
    li a0, 4
    la a1, success_message
    ecall
    li a0, 10
    ecall
```

Ignorable: Check results

```
# Checks the result of inc_arr, which should contain 2 3 4 5 6 after
# one call.
# You can safely ignore this function; it has no errors.
check_arr:
    la t0, exp_inc_array_result
    la t1, array
    addi t2, t1, 20 # Last element is 5*4 bytes off
check_arr_loop:
    beq t1, t2, check_arr_end
    lw t3, 0(t0)
    lw t4, 0(t1)
    beq t3, t4, continue
    la a0, inc_arr_string
    j failure
continue:
    addi t0, t0, 4
    addi t1, t1, 4
    j check_arr_loop
check_arr_end:
    ret


# prints a failure message, then terminates the program
# Since we don't return back to the caller, this is like executing an exception
# inputs: a0 = the name of the test that failed
failure:
    mv a3, a0 # load the name of the test that failed
    li a0, 4 # String print ecall
    la a1, fail_message

    ecall
    li a0, 10 # Exit ecall
    ecall
```

2. Compute power of 2

```
pow:
    # BEGIN PROLOGUE
    # FIXME Need to save the calle saved register(s)
    # END PROLOGUE
    addi sp, sp, -4
    sw s0, 0(sp)
    li s0, 1
pow_loop:
    beq a1, zero, pow_end
    mul s0, s0, a0
```

Be conservative, **s0** is used, then lets make a **copy of s0** in memory
And initialize **the pow result to 1,** which is temporarily stored **in s0**

Computes **x^n**. Where x is in **a0**, and n is in **a1**
Step by step: s0 = a0 * s0

```
#include <stdint.h>

uint32_t pow(uint32_t a0, uint32_t a1) {
    uint32_t s0 = 1;
    while (a1 != 0)
    {
        s0 *= a0;
        a1 -= 1;
    }
    return s0;
}
```

```
    addi sp, sp, -4
    sw s0, 0(sp)
    li s0, 1
pow_loop:
    beq a1, zero, pow_end
    mul s0, s0, a0
    addi a1, a1, -1
    j pow_loop
pow_end:
    mv a0, s0
    # BEGIN EPILOGUE
    # FIXME Need to restore the calle saved register(s)
    # END EPILOGUE
    lw s0, 0(sp)
    addi sp, sp, 4
    ret
```

Be conservative, **s0** is used, then lets make a **copy of s0** in memory
And initialize **the pow result to 1,** which is temporarily stored **in s0**

Computes **x^n**. Where x is in **a0**, and n is in **a1**
Step by step: s0 = a0 * s0
Repeat iteration **by n times (n, n-1, n-2, …… 1)**

Restore s0 from memory,
As the return address is stored in the caller with **jal, ra, label**, callee return to caller using ***jr ra (ret / jalr x0, rs, 0)***

```
while (a1 != 0)
{
    s0 *= a0;
    a1 -= 1;
}
return s0;
}
```

3. Increments the element in-place and its helper function

```
# Increments the elements of an array in-place.
# a0 holds the address of the start of the array, and a1 holds
# the number of elements it contains.
#
# This function calls the "helper_fn" function, which takes in an
# address as argument and increments the 32-bit value stored there.
inc_arr:
    # BEGIN PROLOGUE
    # FIXME What other registers need to be saved?
    addi sp, sp, -12
    sw ra, 0(sp)
    sw s0, 4(sp)
    sw s1, 8(sp)
    # END PROLOGUE
    mv s0, a0 # Copy start of array to saved register
    mv s1, a1 # Copy length of array to saved register
    li t0, 0 # Initialize counter to 0
inc_arr_loop:
    beq t0, s1, inc_arr_end
    slli t1, t0, 2 # Convert array index to byte offset
    add a0, s0, t1 # Add offset to start of array
    # Prepare to call helper_fn
    #
    # FIXME Add code to preserve the value in t0 before we call helper_fn
    add t3, t0, x0
    # Also ask yourself this: why don't we need to preserve t1?
    jal helper_fn
    # FIXME Restore t0
    mv t0, t3
    # Finished call for helper_fn
    addi t0, t0, 1 # Increment counter
    j inc_arr_loop
inc_arr_end:
    # BEGIN EPILOGUE
    # FIXME What other registers need to be restored?
    lw ra, 0(sp)
    lw s0, 4(sp)
    lw s1, 8(sp)
    addi sp, sp, 12
    # END EPILOGUE
    ret

# This helper function adds 1 to the value at the memory address in a0.
# It doesn't return anything.
# C pseudocode for what it does: "*a0 = *a0 + 1"
#
# This function also violates calling convention, but it might not
# be reported by the Venus CC checker (try and figure out why).
# You should fix the bug anyway by filling in the prologue and epilogue
# as appropriate.
helper_fn:
    # BEGIN PROLOGUE
    # FIXME: YOUR CODE HERE
    addi sp, sp, -4
    sw s0, 0(sp)
    # END PROLOGUE
    lw t1, 0(a0)
    addi s0, t1, 1
    sw s0, 0(a0)
    # BEGIN EPILOGUE
    # FIXME: YOUR CODE HERE
    lw s0, 0(sp)
    addi sp, sp, 4
    # END EPILOGUE
    ret
```

Again, store all saved registers(**s0 - sn**), return address registers (**ra**)

Copy a0, a1 to s1, s0:
- Register a0, s0: **addr of arr[0]**
- Register a1, s1: **array size**

Use iteration **counter t0** to loop through the array

When counter t0 == length s1, stop and return, Otherwise:
- Register t1 is set to t0 * 4, which is the **offset** to retrieve memory contents
- Address s0 + t1 is the address of element arr[i]

- Though in this case, t0 is not being used by ***helper_fn***. We can still preserve its value
- We still need to return to L97 after calling helper, then **ra** should be saved
  - Note ***jal helper_fn*** *<=>* ***jal ra, Label***
  - In this case: the actual assembly of ***jal*** helper is ***jal x1 32. This*** instruction jumps to ***current_PC + decimal2hex(32).*** Current PC is also stored in ***ra*** for **return from callee**
- And **restore t0** after calling helper
- Increment counter for array index, and jump to **next iteration**

Restored stored copies of ***ra, s0, s1***. Move ***sp*** back by popping

Register s0 is used to store the base address of array in the caller, we should at least store **s0** in the helper, and push the **sp**

- As we have already stored the address of **arr[i]** in **a0**, simply load the value of **arr[i]** into **t1**
- Increment **t1** by 1 and store the **incremented value in s0**
- Store **s0**, which equals to **arr[i] + 1** in memory with an address of **s0**

Restore **s0** from its copy, and pop the **stack pointer**

Exercise3: Debugging "Complex" struct mapping function

Similar to Lab3 Exercise 5, we hope to map each value in a Linked List. Now, the **linked list** is a **linked list of int arrays**.
The LinkedList Node is defined as:

```
struct node {
    int *arr;
    int size;
    struct node *next;
};
```

As it is a linked list of arrays, the new map function will traverse the linked list and for each element in each array of each node, it applies the passed-in function to it, and stores back to the array

```
void map(struct node *curr_node, int (*f)(int)) {
    if (!curr_node) { return; }
    for (int i = 0; i < curr_node->size; i++) {
      curr_node->arr[i] = f(curr_node->arr[i]);
    }
    map(curr_node->next, f);
}
```

A few initialization directives

```
.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"
```

1. Understanding the main function

```
.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
```

- Create the list, and load the head address to ***s0***

See helpers definitions here if you wish

Ref: Helper functions for list creation, debugging, and printing

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

create_default_list:
    addi sp, sp, -24
    sw ra, 0(sp)
    sw s0, 4(sp)
    sw s1, 8(sp)
    sw s2, 12(sp)
    sw s3, 16(sp)
    sw s4, 20(sp)
    li s0, 0  # pointer to the last node we handled
```

```
    #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 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
```

- Print the list using the predefined recursive function

See helpers definitions here if you wish

- Jump to map function, record the ***ra***

Map function and action items

```
map:
    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, 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    # convert count t0(index i) to offset
    add t3, t3, t1    # get arr[i] by adding the offset and the base address
    lw a0, 0(t3)      # load the value at that address into a0

    jalr ra, s1, 0          # call the function on that value.

    sw a0, 0(t3)      # 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
    add a1, s1, x0     # put the address of the function back into a1 to prepare for the recursion

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

    jr ra
```

- From the perspective of register usage, ***ra, s0*** are definitely used outside of the scope, if not stored properly, the program will crash
- Regarding ***s1***, because we are only using ***s1*** in map function, copying ***s1*** is more like a CC concern (If we do not store ***s1***, program outputs are still correct but a lot of CC convention warnings are throwed)

If current node address is 0, terminate the function, otherwise store argument registers values in saved registers

- Register ***a0*** is the node's address, stored in ***s0***
- Register ***a1*** is the function address, stored in ***s1***
- Register ***t0*** is the counter for ***node->array*** index

- Address of ***node->arr[0]*** (base address) is at memory address ***s0***
- Address of ***node->size (used for loop termination)*** is at memory address ***4(s0)***

Note *t2, t1, t0* are being accessed in the loop body, we **cannot** overwrite them in the same iteration. Need an **additional temporary register** to compute **address of arr[i]**

- Use ***t3*** for offset computing, first assign ***t3*** = **counter** * **4** = ***t0*** * **4**, i.e. **offset** = **sizeof(int)** * **i**
- Address of **arr[i]** is *base + offset = t1 + 4 * t0*
- Load value at the address of **arr[i]** into ***a0***, feed **arr[i]** to ANY function pointed by ***s1***.
- Jump to that function and save the current **PC** in ***ra***

- After ***f*** processed ***arr[i], a0*** gets modified, store ***a0*** to its original address, which is ***t3***
- ***Move counter to next index by adding 1***
- Determine whether to terminate the loop, **if index == size**, move to next node
  - Load the address of next node from ***8(s0)*** into ***a0***
  - The address of the function is put back to ***a1*** as the **argument** for recursion function
  - As we have get ***a0, a1*** ready, **make the recursive call**
- **Otherwise**, keep processing ***curr_node->arr[i]***

Restore ***s0, s1, ra*** after the last recursive call (when **a0**, which equals to **curr_node->next_node** finally reach **0**), where ***ra*** is crucial and worth discussing in detail.

```
    sw ra, 0(sp)
    sw s0, 4(sp)
    sw s1, 8(sp)
    sw s2, 12(sp)
    sw s3, 16(sp)
    sw s4, 20(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)
    lw s0, 4(sp)
    lw s1, 8(sp)
    lw s2, 12(sp)
    lw s3, 16(sp)
    lw s4, 20(sp)
    addi sp, sp, 24
    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

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

Let's denote the original ***sp*** as **sp0**; ***s0*** as **&node[i];** and ***ra*** as **SOMEONE's ra; *s1*** always holds function **pointer fn**