Lab03: RISC-V assembly introduction

Friday, July 22, 2022 12:12 AM

Exercise2: Introduction to **real** RISC-V assembly with *Fibonacci*

1. Directives

At the top of the ***fib.s***: **.data**, **.word**, and **.text** are **directives:**

- **.data**: Denotes where the **global variables are declared**
- **.word**: **Allocates** and **initialize space** for a 4-byte variable in **data segment**
    - In this example, global variable **n** is initialized as: ***n: .word 9***
        - Where ***n:*** is the **label**, and ***.word 9*** is the ***.word directive***
- **.text**: indicate the **start of the code**

2. More instructions in real program

2.1. Load address from directives

la n: **loads** the address of the **label where n is located** (n is specified using label and directives)

2.2. **ecall** instructions

ecall instruction is used to **perform system calls** or **request other privileged operation**s: such as accessing the file system or writing output to console. Here, we mostly use ***ecall*** for **print** or **exit**

- The action of ***ecall*** instruction **depends upon** the **value in *a0***
    - When set ***a0* to 10** and execute ***ecall:* terminate** the program
    - When set ***a0* to 1**, and set **a1 to the integer** to print: Print an integer stored in ***a1***

```
1  .data
2  n: .word 9
3
4  .text
5  main:
6      add t0, x0, x0 # curr_fib = 0
7      addi t1, x0, 1 # next_fib = 1
8      la t3, n # load the address of the label n
9      lw t3, 0(t3) # get the value that is stored at the adddress denoted by the label n
10 fib:
11     beq t3, x0, finish # exit loop once we have completed n iterations
12     add t2, t1, t0 # new_fib = curr_fib + next_fib;
13     mv t0, t1 # curr_fib = next_fib;
14     mv t1, t2 # next_fib = new_fib;
15     addi t3, t3, -1 # decrement counter
16     j fib # loop
17 finish:
18     addi a0, x0, 1 # argument to ecall to execute print integer
19     addi a1, t0, 0 # argument to ecall, the value to be printed
20     ecall # print integer ecall
21     addi a0, x0, 10 # argument to ecall to terminate
22     ecall # terminate ecall
```

Exercise3: Conversion from C to RISC-V

The complex part of this program is main function

```c
1  int source[] = {3, 1, 4, 1, 5, 9, 0};
2  int dest[10];
3
4  int fun(int x) {
5      return -x * (x + 1);
6  }
7
8  int main() {
9      int k;
10     int sum = 0;
11     for (k = 0; source[k] != 0; k++) {
12         dest[k] = fun(source[k]);
13         sum += dest[k];
14     }
15     return sum;
16 }
```

Line 1 - Line 22 **specifies a few directive** for later usage
.main: addr of main function
.source: memory location of array of source variables
.dest: memory location of array of destination variables

```
1  .globl main
2
3  .data
4  source:
5      .word   3
6      .word   1
7      .word   4
8      .word   1
9      .word   5
10     .word   9
11     .word   0
12 dest:
13     .word   0
14     .word   0
15     .word   0
16     .word   0
17     .word   0
18     .word   0
19     .word   0
20     .word   0
21     .word   0
22     .word   0
```

Line 24 - Line 29 defines function ***fun***

```
24 .text
25 fun: # Function itself is simple: take argument a0(which denotes x), and do tasks
26     addi t0, a0, 1 # t0 = a0 + 1
27     sub t1, x0, a0 # t1 = -x
28     mul a0, t0, t1 # res = t0 * t1 = -x * (x + 1)
29     jr ra # equivalent to jalr x0, ra, 0 and finally equivalent to ret
```

Line31-Line43 initializes the main function:

```
31 main:
32     # BEGIN PROLOGUE
33     addi sp, sp, -20 # Just in case, store a copy of registers that might be used
34     sw s0, 0(sp)
35     sw s1, 4(sp)
36     sw s2, 8(sp)
37     sw s3, 12(sp)
38     sw ra, 16(sp)
39     # END PROLOGUE
40     addi t0, x0, 0 # initiate iteration variable k at t0
41     addi s0, x0, 0 # initiate returned value sum at s0
42     la s1, source # load the ptr pointing to source array
43     la s2, dest # load the ptr pointing to dest array
```

Line63-Line73 warps up the program:

```
63 exit:
64     add a0, x0, s0 # store sum s0 to a0, which is the return value of main function
65     # BEGIN EPILOGUE
66     lw s0, 0(sp)
67     lw s1, 4(sp)
68     lw s2, 8(sp)
69     lw s3, 12(sp)
70     lw ra, 16(sp)
71     addi sp, sp, 20 # Recover register values being used
72     # END EPILOGUE
73     jr ra
```

Line44-Line62 defines the loop body

```
44 loop:
45     slli s3, t0, 2 # calculate offset using shifting left by 2bits (*4), put offset value in s3
46     add t1, s1, s3 # pointer arithmetic, t1 = s1 + s3 <=> ptr2source2handle = ptr2source + offset
47     lw t2, 0(t1) # load the source *ptr2source2handle (which is source[k]) to t2
48     beq t2, x0, exit # terminate the loop if source[k] == 0
49     add a0, x0, t2 # assign sum = source[k]
50     addi sp, sp, -4 # I believe the Line52 and line 55 are unnecessary, sp decrement is reduced to 4
51     sw t0, 0(sp) # t0 being used in fun, store its original value
52     # sw t2, 4(sp)
53     jal fun # a0 will be overwriten anyway later by s0, we don't need to save a copy of a0 in this case
54     lw t0, 0(sp) # retrieve t0 after usage
55     # lw t2, 4(sp)
56     addi sp, sp, 4 # mv stack pointer back
57     add t2, x0, a0 # a0 stores fun(source[k]), recall source[k] is in t2, set source[k] to func(source[k])
58     add t3, s2, s3 # t3 = base_addr_dest + offset, i.e. dest[k]
59     sw t2, 0(t3) # store func[source[k]] into dest[k]
60     add s0, s0, t2 # accumulate s0, which is sum
61     addi t0, t0, 1 # update iterative variable k
62     jal x0, loop # j loop
```

Exercise4: Factorial

Line1-Line21 specify directives, and give the main function

```
1  .globl factorial
2
```

Exercise4: Factorial

```c
#include <stdio.h>

int factorial(int n) {
    int res = 1;
    for (int i = 1; i <= n; i++) {
        res = res * i;
    }
    return res;
}

int main() {
    int n = 8;
    int f_res = 0;
    f_res = factorial(n);
    printf("factorial of %d = %d\n", n, f_res);

    return 0;
}
```

Line1-Line21 specify directives, and give the main function

```
1  .globl factorial
2
3  .data
4  n: .word 3
5
6  .text
7  main:
8      la t0, n
9      lw a0, 0(t0)
10     jal ra, factorial
11
12     addi a1, a0, 0 # a0 is the actual result
13     addi a0, x0, 1
14     ecall # Print Result
15
16     addi a1, x0, '\n'
17     addi a0, x0, 11
18     ecall # Print newline
19
20     addi a0, x0, 10
21     ecall # Exit
```

Implementation**(a)** of ***factorial(n)*** function - rigorous mapping

```
factorial:
    # YOUR CODE HERE
    addi t0, a0, 0 # upper bound of for loop in t0
    addi t1, x0, 1 # iterative variable i in t1
    addi a0, x0, 1 # set a0 to 1
loop:
    blt t0, t1, exit # i <= upperbound is equivalent to upperbound > i
    mul a0, a0, t1 # res = res * i
    addi t1, t1, 1 # update iterative variable i
    jal x0, loop
exit:
    ret
```

Implementation**(b)** of ***factorial(n)*** function - optimized a little bit

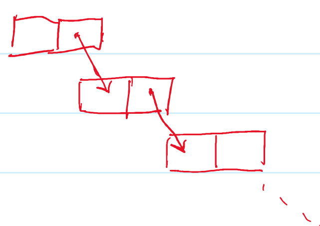
```
factorial:
    # YOUR CODE HERE
    addi t0, a0, 0 # upper bound of for loop in t0
    addi t1, x0, 1 # iterative variable i in t1
loop:
    bge t1, t0, exit # when i=upperbound, exit the loop
    mul a0, a0, t1 # res = res * i
    addi t1, t1, 1 # update iterative variable i
    jal x0, loop
exit:
    ret
```

Exercise 5: Convert **customized** data types to RISC-V functions

- In this task, a **customized** data type ***node*** is defined:

```c
struct node {
    int value;
    struct node *next;
};
```

  - Where **value** is a 32-bit integer
  - **next** is the *ptr* to next node



- The recursive **map** function to implement is like:

```c
void map(struct node *head, int (*f)(int))
{
    if (!head) { return; }
    head->value = f(head->value);
    map(head->next,f);
}
```

  - The first parameter is the **address of the head node** (ptr2head)
  - The second parameter is the **address of a function**, this function is called by ***jalr*** (offset can be too large)
    - `int (*f)(int))`
      - In the above declaration: **f** is a **pointer to a function**, which takes an **int** as an argument

5.0 Helper functions

```
# === Definition of the "square" function ===
square:
    mul a0, a0, a0
    jr ra

# === Definition of the "decrement" function ===
decrement:
    addi a0, a0, -1
    jr ra
```

```
# === Helper functions ===
# You don't need to understand these, but reading them may be useful

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      #     Allocate 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)
```

5.1 Main Function

```
.text
main:
    jal ra, create_default_list
    add s0, a0, x0 # a0 (and now s0) is the head of node list

    # Print the list
    add a0, s0, x0
    jal ra, print_list
    # Print a newline
    jal ra, print_newline

    # === Calling `map(head, &square)` ===
    # Load function arguments
    add a0, s0, x0 # Loads the address of the first node into a0

    # Load the address of the "square" function into a1 (hint: check out "la" on the green sheet)
    ### YOUR CODE HERE ###
    la a1, square

```

*Before calling map function **sqr***

```
    # Issue the call to map
    jal ra, map

```

```
    # Print the squared list
    add a0, s0, x0
    jal ra, print_list
    jal ra, print_newline

    # === Calling `map(head, &decrement)` ===
    # Because our `map` function modifies the list in-place, the decrement takes place after
    # the square does
```

*After calling map function **sqr***

```
    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, print_me_and_recurse
    jr ra               # Nothing to print
print_me_and_recurse:
    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 char syscall
    ecall
    lw a0, 4(t0)        # a0 gets address of next node
    jal x0, print_list  # Recurse. The value of ra hasn't been changed.

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
```

```
    jal ra, print_list
    jal ra, print_newline

    # === Calling `map(head, &decrement)` ===
    # Because our `map` function modifies the list in-place, the decrement takes place after
    # the square does

    # Load function arguments
    add a0, s0, x0 # Loads the address of the first node into a0

    # Load the address of the "decrement" function into a1 (should be very similar to before)
    ### YOUR CODE HERE ###
    la a1, decrement
```

*Before calling map function **decrement***

```
    # Issue the call to map
    jal ra, map
```

*After calling map function **decrement***

```
    # Print decremented list
    add a0, s0, x0
    jal ra, print_list
    jal ra, print_newline

    addi a0, x0, 10
    ecall # Terminate the program
```

5.3. Map function

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

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

    add s0, a0, x0 # Save address of this node in s0
    add s1, a1, x0 # 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).
    # Hint: Where do we keep track of the function to call? Recall the parameters of "map".
    ### YOUR CODE HERE ###
    jalr ra, s1, 0
    # 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 ###
    add a1, s1, x0
    # Recurse
    ### YOUR CODE HERE ###
    jal ra, map
done:
    # Epilogue: Restore register values and free space from the stack
    ### YOUR CODE HERE ###
    lw ra, 0(sp)
    lw s0, 4(sp)
    addi sp, sp, 8

    jr ra # Return to caller
```

1. Make copies of values that might be used after calling map function

Register resources being used before calling function ***map***

- Argument register: ***a0 for the first node, a1 for the subfunction***
- Save register: ***s0***
- Return address register: ***ra***

As a0, s0, ra could be used recursively by the recursion, let's consider reserve space for all of them

- Note that ***s0*** stores the **very first node** of the Linklist, and ***a0*** is initialized with ***s0***
- Then register ***a0*** could be overwritten, no need to reserve space

2. Specify the usage of argument registers

Argument registers declarations:

- Register ***a0*** for the ***ptr to LList Node***
- Register ***a1*** for the ***ptr to sub function***

Based upon the definition of ***node struct***

- Node->value is at 0(s0)
- Node->next is at 4(s0)

3. Calling the sub function in map function

By calling convention, ***sqr***/***decrement*** function asks argument from register **a0** which change the value of ***a0*** in place, and return **to *ra*'s** next instruction

After **MANY** recursions, **PC** could be **TOO FAR** away from the label of square

- Jump to the **function pointed by** value in register ***a1***, function could be **sqr** or **decrement** depending on what's stored in ***a1***
- **Return address** should be remembered, otherwise, you will go to ***PC=0x0 after returning from sqr/decre functions***

Then store the returned value ***a0*** back to memory **at address[s0]**

4. Making **recursive calls** of ***map*** function

Node->next is at 4(s0), load the address of **next node** into ***a0***
Set the address of function at ***a1, start the recursion***

Jump to next function call ***with jal ra, LABEL***

5. Restore copies of overwritten registers from memory

At least if we do not store **ra, we cannot return to the caller**