f2022-public-labs

Lab 9: Control flow and C-to-assembly

Submission schedule: Your code for this lab must be submitted to Gradescope **before 5:30 PM on Friday, Nov 10 2023**. This is regardless of when your section is, but keep in mind that office hours will fill u p quickly if everyone puts it off until the last minute. **Start early**.

Late penalty: 20% off, flat, after the Friday deadline, until the Gradescope late deadline. After that, your score is zero.

In this lab we will first continue our discussion on assembly directives, with a focus on memory space directives, as they will be utilized in the latter portion of the lab.

After directives, we will touch on the next major category of the assembly instruction in our course, which is the control flow instruction, which enables CPUs to perform if-else statements and loops.

Finally, we will have some short C programs on strings and arrays that you will need to convert into equivalent assembly programs.

A reference manual for the RISC-V ISA can be found here. Refer to the 'Unprivileged' version, i.e., Volume 1.

1. Intro to directives and labels

You have already met with assembly directives when working on lab 1. In the `lab1.s` file, any line starts with `.` is an assembly directive. These directives instruct the assembler on how to organize the binary and hint to the linker what can be made visible to other object files. Basically, directives are like C macros, including some meta-information beyond the actual program. Below is a short list of the common directives in RISC-V:

1. `.data` : specify the following part of the assembly file as the global data section
2. `.text` : specify the following part of the assembly file as instructions
3. `.global SYMBOL` : make the symbol `SYMBOL` visible to other object files during linking
   i. The `SYMBOL` could be either a function, like what we have done so far
   ii. Or a global variable, like `lowercase_string` in the next section
4. `.balign PADDING_SIZE` : ensures that the next set of statements are aligned to a particular `PADDING_SIZE` .
   i. This is often useful to ensure that instructions start on an addresses that is 4-byte aligned. In our code below, the definition of our `lowercase_string` may end on an addresses that is not 4-byte aligned. Instructions (and generally all data types) need to

be aligned to ensure efficient access.

5. `.asciz` / `.string "SOME_STRING"` : specify a null-terminating C-string
   i. This means that the assembler will append a `\0` char to the end of the string.
   ii. `.ascii` does not append the `\0` .
6. `.word expr` : store a 32-bit value `expr` to the memory address at this line
7. `.space count [, value]` : reserve `count` bytes starting at the memory address, each with default value `value`
8. For more information on the directives, check out this reference manual

Besides directives, another common assembly construct is a `label` , which is statement ends with `:` . Labels are mnemonics of some specific memory addresses pointing to the next available data or instructions.

For instance, the label `lowercase_string:` will be a synonym to the address of the string `"ece 362 is awesome!"` .

Another example is the label `asm_strlen` . We first use `.global asm_strlen` directive to make it visible to the autograder program so that the autograder can call it. Then, we put our function statement for the directive underneath. Now, `asm_strlen` will be the address of the next instruction.

Labels could also help you organize the program by creating mnemonics for either variables or instruction snippets.

2. Assembly instruction: control flow

All the instructions we discuss in lab 1 are data operation instructions that do not change the program execution order, i.e. instructions execute one after the other in the same order they appear in the file. However, in real programs, there are all kinds of statements that can change that order, i.e. conditional statements or loops. Supporting these constructs requires hardware support, which are the control flow instructions.

Prior to diving directly into the practice problems, however, we will first recap the instructions and special hardware flags that branch instructions rely on.

2.1 Branch intro

In lab 1, all the programs we have coded follow a direct sequential flow like this:

```
li a0, #0
    │
    ▼
li a1, #1
    │
    ▼
add a2, a0, a1
    │
    ▼
   ...
```

However, what if we want to change to a different program flow based on some conditions?

```
 li a0, #0
     │
     ▼
 li a1, #1
     │
     ▼        false
 if x3 > 0 ──────────────┐
     │                   │
     │                   │
true │                   │
     │                   │
```

```
        ▼                   ▼
  add x2, x0, x1    sub x2, x0, x1
        │                   │
        ├───────────────────┘
        │
        ▼
       ...
```

Or if we want to create a loop in our program?

```
li a0, #0
    │
    ▼
li a1, #1
    │
    │
    ├────────────┐
    │            │
    ▼            │
add x0, x0, x1   │
    │            │
    └────────────┘
```

These are the cases where branch instructions kick in.

2.2 Unconditional branch

If we categorize branch instructions by how they are executed, there are two types: unconditional branch and conditional branch.

The unconditional branch is always executed, meaning that the next instruction getting executed after it is the branch target (the place where we branch to).

In the following example, we create an infinite loop in RISCV assembly utilizing the `jal` jump-and-link branch instruction in the RISCV32 ISA to branch to label `_inf_loop`. Whenever `jal x0, _inf_loop` is executed, the program counter of the CPU will set to the address of `add x10, x10, x11` and execute there instead of the `li, x10, 1` after the label `never_executed`, and it saves the address of the instruction after the `jal` (located at program counter + 4) to the register `x0`. Since `x0` is hardwired to 0 in RISC-V, however, the address is actually discarded (since we don't care about it in our infinite loop).

```
// An infinite loop to increment register x0
and x10, x10, x0 //resets the x10 register
li x11, 1
inf_loop:
   add x10, x10, x11
   jal x0, inf_loop
never_executed:
   li x10, 1

// Equivalent to
int32_t x = 0;
while(true)
   x += 1
x = 1
```

The unconditional branches are useful to construct loops as well as calling procedures via the `jal` jump-and-link instruction, which will save the address of the instruction after the `jal` to the specified register and jump to the Procedure label provided. The `jal` instruction can be used to create an unconditional loop when the return address is saved to x0, which is a read-only register, hence the return address is discarded. We will cover this in more details during lab 3 when we discuss Procedure calls.

2.3 Conditional branch

The conditional branch, as the name implies, is executed based upon whether a given condition is true or false. These conditions can test for equality and inequality between two numbers. However, there are many other relationships between two numbers. The full set of comparisions is less than (<), greater than (>), less than or equal (<=), greater than or equal (>=), equal (=) or not equal (!=).

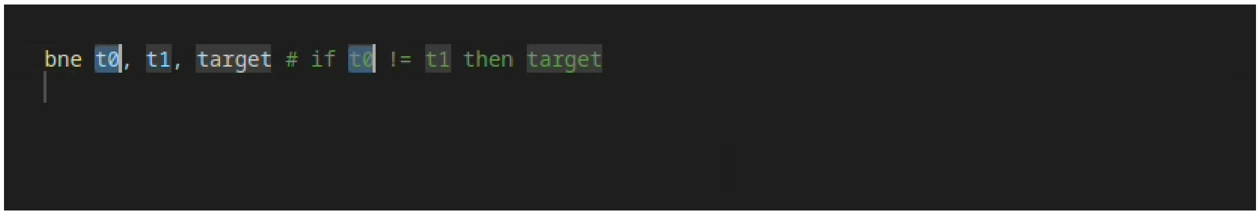
RISC-V provides instructions for these comparisions. These instructions are capable of handling the dichotomy between unsigned and signed number comparision. The instructions are `beq` , `bne` , `bge` , `blt` , `bltu` and `bgeu` . The `u` is for unsigned comparisons of values.

For example, the instruction `bge` takes two registers rs1, rs2 and a target label. The syntax for the instruction is:

```
bge rs1, rs2, target #if rs1 >= rs2 then jump to target
```

Refer to your instruction card on Piazza and the textbook to learn more about the functions performed by the other instructions and their syntax.

> VScode Tip: When you type an instruction name in VScode, a drop-down menu appears with suggestions. If you click on one of the suggestions, the syntax of the instruction will be given to you along with a comment providing information about the function of the instruction. This is also a helpful reference for you to keep in mind. If it doesn't appear, you may have to press Ctrl/Cmd + Space and start typing.



```
bne t0, t1, target # if t0 != t1 then target
```

2.4 if-else

With the compare-and-branch instructions, we could construct `if-else` clauses like this:

```
and x10, x10, x0 // resetting x10
li x11, 1

// If-else
_if:
   // Condition check
   // if (x10 == x11)
   beq x10, x11, _then
   jal x0, _else

_then:
   // x10 == x11
   some assembly code...
   // End the if
   jal x0, _end_if   // Skip the else branch

_else:
   // x10 != x11
   some other assembly code...
_end_if:

// Similar to the C program below
int x10 = 0;
int x11 = 1;

if (x10 == x11) {
   some assembly code...
} else {
   some other assembly code...
}

// Sometimes to reduce the labels of the if-else,
// we could either invert the condition
// or switch the else and then branch position

// Invert condition
_if:
   // Condition check
   // if (x10 == x11)
   bne x10, x11, _else // jump to else branch if x0 != x1

   // Noted we remove the _then branch label
   // as the b.ne will not execute if x0 == x1
   // which is the then branch we wanted
   some assembly code...
   // End the if
   jal x0, _end_if   // Skip the else branch

_else:
   // x10 != x11
   some other assembly code...

_end_if
```

2.5 Loops

In addition to `if-else`, we could also construct loops with the compare and branch instructions.

Here's a simple while loop, with compare and conditional branch instructions to implement condition checking, and an unconditional branch to jump back to the start of the loop:

```
// A while loop
_while:
   // if x10 == x11, ends the loop
   // else keep the loop
   beq x10, x11, _while_end

   some while_loop code...

   // Back to the condition check part
   jal x0, _while
_while_end:
   code after the loop...

// Similar to
while (x10 != x11) {
   some while_loop code...
}
code after the loop...

// Note the condition of the C and the assembly
// is inverted to reduce some insts and labels
// like the if-else one
// You could use either one you like (inverted or not)
// as the choice will not have impact on the
// correctness of the program
```

Here's a simple for-loop:

```
// Loop var initialization
and x10, x10, 0 //resetting x10
// Loop body
_for:
   // Loop condition check
   bge x10, x11, _for_end

   some for_loop code...

   // Increment x0 and back to loop start
   addi x10, x10, 1
   jal x0, _for

_for_end:
   code after for_loop...

// Similar to
for (int x10 = 0; x10 < x11; x10++) {
   some for_loop code...
}
code after for_loop...
```

> Note: for the practice of conditional instructions, we will have them in section 3 where we have real functions that need if-else or loops.

3. Human compiler: from C to assembly

In this section, you will be given with some C function implementations and translate them into RISC-V assembly, much like what the compiler does.

> Note: Partial credits are available for all of the problems in this section. Each problem will have 31 test cases and the partial credit is determined by `pass_count / 31 * PROBLEM_POINTS` . Some of the test cases can be passed even with an empty solution, which is normal, but this might change as you code up your own solution.

3.0 Set-up for the Lab

Add the Lab_9_Student folder to your workspace. This will ensure that the VSCode can detect the config files (present in the .vscode folder) for the debugger.

3.1 Calculating length of a string

In this problem, you will implement the `uint32_t asm_strlen(char *str)` function, which will take in a pointer to a string and return its length up to the null terminating character `\0` :

```
uint32_t asm_strlen(char *str) {
    // Set initial string length to 0
    uint32_t length = 0;

    // Check if the pointer is invalid
    if (str == NULL)
        return 0;

    // Check the string character
    // If it is '\0',
    //    we have reach the end of string
    while (*str != '\0') {
        // If not, move the pointer to
        //    next character position
        //    and increment length counter
        str++;
        length++;
    }

    // Return the length
    return length;
}
```

The `asm_strlen` works by counting the string character in a loop until a null terminating character `\0` appears, as in C, the string is terminated by appending `\0` after the last character. For instance, string `"Hello"` in memory is actually:

0x00	0x01	0x02	0x03	0x04	0x05
`H`	`e`	`l`	`l`	`o`	`\0`

Also noted the function will handle the case of null pointer, which just return `0` instead.

You will implement the function in `lab2.S` under the label `asm_strlen`. Noted like lab 8, the argument will be passed in register `x10` and you will put the result in register `x10`. Your assembly function will be verified against the above C program to ensure correct implementation.

> Note: You will not need to worry about storing the temporary variables in the above C program to memory. You could safely use registers to hold such temporary variables as we will have more than enough of them to do so.

> Note: In order to implement the multiple return statements in the C program, use the `ret` instruction in the RISCV32 ISA which is inorder to pop the return address from the stack and jump to it.

> Hint: `lw` will load a 32-bit value from the memory. However, we are facing `char` this time, which is just 8-bit!

3.2 Fibonacci with loop

A very common first program to test understanding of branching is implementing the Fibonacci sequence. Implement the `void asm_fib(int *arr, uint32_t length)` function, which will accept a pointer to an array of integer and the length of the array, then generate fibonacci series starting at `0` and store the result back to the array `arr` at corresponding indices. The C program implementation is as follow:

```
void asm_fib(int *arr, uint32_t length) {
    // Check if the array pointer is invalid
    // or if the length of array is 0
    if (arr == NULL || length == 0)
        return;

    // Initial terms for the fib series
    int prev = 0;  // Fn-2 term
    int curr = 1;  // Fn-1 term

    // Loop to put fib series term in the arr
    for (uint32_t i = 0; i < length; i++) {
        // Handle the initial 2 terms F0 and F1
        if (i == 0) {
            arr[i] = prev;
        } else if (i == 1) {
            arr[i] = curr;
        } else {
            // Fn = Fn-1 + Fn-2, n > 1

            // Save the Fn-1 value
            int tmp = curr;

            // Fn = Fn-1 + Fn-2
            curr += prev;
            // Update Fn-2 = Fn-1
            prev = tmp;

            // Save result to array
            arr[i] = curr;
        }
    }

}
```

The C program first check if the array is invalid (null pointer or size of `0`). If not, it will enter a for loop that save the fibonacci series terms in the incoming array `arr`. Inside the for loop, the program will have to handle the first two terms separately using if clauses. Starting from `i = 2`, the program will begin to calculate the fibonacci series via adding the F_{n-1} and F_{n-2} terms, denoted by `curr` and `prev` variables. It will also update the F_{n-2} to F_{n-1} at the end of the loop.

You will put your code under the `asm_fib` label in `lab2.S`. Again, the function arguments will be passed in from registers `x10` and `x11` respectively. Noted the array is 32-bit integer array. Therefore whenever, you want to have the next element in the array, you have to add 4 to the original pointer address (C handles this automatically based on pointer type, you will need to do this manually in assembly).

> Hint: If you want to access `arr[i]` in the loop, an easy way to do so is to have the multiplication of `i` and `4` stored in a register and just use that register as the base register to access `arr[i]`.
>
> ```
> // Assume x10 has the arr starting address
> // Assume x11 holds the `i` value
>
> // This will have x6 = 4 * i to match the offset of integer array
> li x12, 4 // this is done because we are working with integer array
> mul x6, x11, x12
>
> // Add the byte offset to the starting array address
> add x13, x10, x6
>
> // Now x3 will have the address of arr[i]
> // If we want to save `1` to it:
> lw x14, 1
> sw x14, 0(x13)  // storing contents of x14 into memory address x13
>
> // Now arr[i] will have value of 1 stored
> ```

> Note: Not that if you tried to use another register to store the offset value and tried to do something like `sw x14, x12(x13)` at the end, instead of the `add x13, x10, x6` line in the code, you will get an error, as RISC-V, unlike ARM, does not have a base register + index register addressing mode. More on this later.

3.3 Advanced uppercase formatter

In this problem, we will code a more advanced version of the uppercase formatter we have in lab 8 so that it could loop through an entire string and only convert the lowercase letter to uppercase, leaving the rest of the string intact. The function you will implement is `void asm_toUppercase(char *str)`, with the following C implementation:

```
void asm_toUppercase(char *str) {
    // If string is invalid
    if (str == NULL)
        return;

    // Else loop the string to find the lowercase letters
    // and convert them to uppercase
    while (*str != '\0') {
        // Temp variable to hold the character value
        char c = *str;
```

```
        // if lowercase letter
        if (c >= 'a' && c <= 'z') {
            // Convert the character
            // at the memory pointed by str
            // to its uppercase form
            *str = c - 32;
        }

        // Increment the str pointer to next character
        str++;
    }
}
```

The function will accept a pointer to a string inside memory and will loop through each character to figure out whether uppercase formatting is needed (whether the letter is lowercase). If so, it will store the uppercase form of the character back to the memory address. Note the function should handle the case with NULL pointer and return immediately.

You will put your code in `lab2.S` under `asm_toUppercase` . The assembly function asm_toUppercase gets the pointer to the char array in x10.

> Hint: For this exercise you will find the instructions `blt` and `bge` useful. Make sure you understand what these instructions do, it comes in handy when you try make the if condition(the one that checks whether c >= 'a' and c <= 'z'>) work.

Make sure to submit your work to Gradescope to get credit for this lab.