ECS 50: Programming Assignment #3[*]

Instructor: Aaron Kaloti

Fall Quarter 2021

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2:
  - Updated second example for `rotate()` such that it now resets `a0` and `a1` before the second call to `rotate()`.
  - For part #3, clarified that the reason that the `ra` and `gp` registers need not be preserved is that your code should not modify them in the first place.
- v.3: Updated the Submission Details section to talk about submitting on Gradescope.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Wednesday, 11/10. Gradescope will say 12:30 AM on Thursday, 11/11, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

3 Programming Problems

You will be using the same RV32EM simulator as before. Most of the details regarding file organization, compiling code, and running the simulator are the same as in P2, so refer to `prog_hw2.pdf` if you need to see those details again.

[*]This content is protected and may not be shared, uploaded, or distributed.

3.1 File Organization and Compilation

You will ONLY submit the file `functions.s`. A skeleton version of this file is provided on Canvas in order to help you get started. In this assignment, you will implement three functions (`rotate()`, `interleave()`, and `transpose()`), all in `functions.s`.

3.2 Restrictions

In this assignment, all code that you write will be RISC-V code for a RV32EM processor. ***You must write this assembly code yourself; you cannot use a tool, such as a compiler[1], to generate assembly code. For instance, you cannot write C++ code and then use a compiler to convert it to RISC-V assembly code. We will not accept that, and we will easily be able to tell if you submitted tool-generated assembly code.***

None of your functions can use global variables. This means that you cannot add a `.data` section to `functions.s`.

3.3 General Advice on Writing Assembly Code

You may find it helpful to quickly write a solution to each part using a high-level programming language such as C, C++, Java, or Python and then – *without using any tools/compilers to convert such code to RISC-V assembly code* – come up with the corresponding assembly code.

Don't be afraid to make extensive use of comments in your assembly code. You will not be graded on comments.

3.4 Use of Registers

Recall that a RV32EM processor has 16 32-bit so-called general-purpose registers. Here are the relevant details of the registers.

Generic Name	Special Name	Use (for this assignment)
x0	zero	Hardwired to be zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	
x5-x7	t0-t2	
x8	s0/fp	
x9	s1	
x10-x11	a0-a1	Function arguments / return values
x12-x15	a2-a5	Function arguments

As I've said repeatedly, some of these registers effectively are not general-purpose, in the sense that you cannot use them for anything you want without causing some problems and sneaky bugs. For this assignment, the registers that you should not modify are `zero` (since you can't), `ra`, `sp`, and (for reasons explained in `prog_hw2.pdf`) `gp`.

3.5 Part #1: Rotate Array

For this part, you will implement a function called `rotate()` in `functions.s`. (As mentioned above, there is a skeleton version of `functions.s` on Canvas that I recommend that you start with.) This function takes the following two arguments:

1. `a0`: The starting address of an array of 32-bit signed integers.
2. `a1`: The length of this array.

Your function should rotate the array once to the right. That is, each element in the array should be slid one slot to the right, except for the last element, which should become the new first element.

This function does not return any value.

Compilation: The makefile provided to you on Canvas is already set up to compile `functions.s`, `call_rotate.s`, `crt0.s`, and `startup.c` together into a RV32EM executable called `call_rotate`. You still must have the linker script `riscv32-console.ld` in the same directory too.

Below is one example of caller code for this function. In this case, after `rotate()` has finished, the contents of `arr` should be: 50, 10, 20, 30, 40.

[1]Here, by "compiler", we mean the program that does the step of converting source code to assembly code.

```
.data

arr:
    .word 10, 20, 30, 40, 50

.text
.global main
main:
    lui a0, %hi(arr)
    addi a0, a0, %lo(arr)
    li a1, 5
    jal ra, rotate
end:
    j end
```

Below is another example of caller code for this function. In this case, after both calls of `rotate()` have finished, the contents of `arr` should be: 0xCCCCCCCC, 0xDDDDDDDD, 0xAAAAAAAA, 0xBBBBBBBB.

```
.data

arr:
    .word 0xAAAAAAAA
    .word 0xBBBBBBBB
    .word 0xCCCCCCCC
    .word 0xDDDDDDDD

.text
.global main
main:
    lui a0, %hi(arr)
    addi a0, a0, %lo(arr)
    li a1, 4
    jal ra, rotate
    lui a0, %hi(arr)
    addi a0, a0, %lo(arr)
    li a1, 4
    jal ra, rotate
end:
    j end
```

3.6 Part #2: Interleave

Random tip about usage of labels: Before I talk about part #2, now would be a good time for me to say that you should avoid repeating labels. If you do repeat labels, it can lead to issues, and it is easy to accidentally repeat labels when you are implementing three independent functions in the same file.

For this part, you will implement a function called `interleave()` in `functions.s`. This function takes the following four arguments:

1. `a0`: The starting address of an array of 32-bit signed integers.
2. `a1`: The starting address of *another* array of 32-bit signed integers. *This array will have the same length as the first array.*
3. `a2`: The starting address of an output array of 32-bit signed integers. *This array will have double the length of the first array.*
4. `a3`: The address of a 32-bit variable containing the length of the first array (or the second array)[2].

This function should interleave the values of the first two arrays and place the result in the output array. That is, the output array should end up containing the first element of the first array, the first element of the second array, the second element of the first array, the second element of the second array, the third element of the first array, etc.

This function does not return any value. Your function is not allowed to modify the two input arrays.

Compilation: The makefile provided to you on Canvas is already set up to compile `functions.s`, `call_interleave.s`, `crt0.s`, and `startup.c` together into a RV32EM executable called `call_interleave`. You still must have the linker script `riscv32-console .ld` in the same directory too.

Below is one example of caller code for this function. In this case, after `interleave()` has finished, the contents of `outputArr` should be: 1, 5, 2, 6, 3, 7, 4, 8.

[2]Would it have made more sense to pass the length directly as was done in part #1? Yes, it would have, but I just thought I'd make things a bit more interesting.

```
.data

inputArr1:
    .word 1, 2, 3, 4
inputArr2:
    .word 5, 6, 7, 8
outputArr:
    # Repeat `.word -1` eight times.
    .rept 8
    .word -1
    .endr
arrlen:
    .word 4

.text
.global main
main:
    lui a0, %hi(inputArr1)
    addi a0, a0, %lo(inputArr1)
    lui a1, %hi(inputArr2)
    addi a1, a1, %lo(inputArr2)
    lui a2, %hi(outputArr)
    addi a2, a2, %lo(outputArr)
    lui a3, %hi(arrlen)
    addi a3, a3, %lo(arrlen)
    jal ra, interleave
end:
    j end
```

3.7 Part #3: Transpose

For this part, you will implement a function called `transpose()` in `functions.s`.

This function will perform matrix transposition. If you are not familiar with matrix transposition, you can find everything you need to know on the Wikipedia page. The short version is that you can find the transpose of a matrix by reflecting its elements along its main diagonal. For example, if you have this matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

The transpose would be:

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

This function takes the following four arguments:

1. `a0`: The starting address of the *input matrix* (which is assumed to be in row-major order).
2. `a1`: Number of rows.
3. `a2`: Number of columns.
4. `a3`: The starting address of the *output matrix* (which is assumed to be in row-major order as well).

The function should find the transpose of the input matrix and place the result in the output matrix.

The matrices (which are effectively statically allocated two-dimensional arrays) contain 32-bit signed integers.

This function does not return any value. Your function is not allowed to modify the original matrix.

Compilation: The makefile provided to you on Canvas is already set up to compile `functions.s`, `call_transpose.s`, `crt0.s`, and `startup.c` together into a RV32EM executable called `call_transpose`. You still must have the linker script `riscv32-console.ld` in the same directory too.

Below is an example of caller code for this function. (It uses the example from above.) In this case, after `transpose()` has finished, the contents of `output` should be 1, 3, 5, 2, 4, 6.

```
.data

input:
    .word 1
    .word 2
    .word 3
    .word 4
    .word 5
    .word 6

output:
    .rept 6
    .word 0
    .endr

.text
.global main
main:
    lui a0, %hi(input)
    addi a0, a0, %lo(input)
    li a1, 3
    li a2, 2
    lui a3, %hi(output)
    addi a3, a3, %lo(output)
    jal transpose
end:
    j end
```

You may find the `mul` instruction useful for this part. I talked about it in a previous Canvas announcement here.

Register preservation: For this part, your function must preserve any registers it uses, *except for*:

- `a0` through `a3`, since these are the arguments.
- `ra`, `sp`, and `gp`.
  - The `ra` and `gp` registers are listed here because your code should not modify them!

Here is an example about the goal of register preservation: if `t0` has the value 18 and `t1` has the value 30 before your function is called, then – after your function is done – those two registers must still have the values 18 and 30, respectively.

If you wish, you can preserve registers that your function does not use too. For example, you could still write code in your function to preserve the `s0` register even if your function does not use it. This may be useful in avoiding forgetting to preserve any register that you might find yourself using eventually.

4 Grading Breakdown

TBA

5 Submission Details

As with the previous programming assignment, there is no autograder for this assignment.

Submitting on Gradescope: There is a place on Gradescope that you can submit to. Only submit `functions.s`. You may be penalized for submitting other files.

Note that only your active submission (i.e. your latest submission, unless you manually activate a different submission) will count. I'll only see the file that you submitted in your active submission.