ECS 50: Programming Assignment #2

Instructor: Aaron Kaloti

Fall Quarter 2021

Contents

1 Changelog **1**

2 General Submission Details **1**

3 Preliminary Remarks **1**

4 Relevant Details Regarding RISC-V and RV32EM **2**

5 RV32EM Simulator **2**

6 Programming Problems **2**

6.1 File Organization and Compilation . . . 2
6.2 Restrictions . . . 6
6.3 General Advice on Writing Assembly Code . . . 6
6.4 Use of Registers . . . 6
6.5 Part #1: Count Target . . . 6
6.6 Part #2: Sum Within Range . . . 7
6.7 Part #3: Maxes . . . 8

7 Grading Breakdown **9**

8 Submission/Autograder Details **9**

1 Changelog

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

- v.1: Initial version.

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 Tuesday, 11/02. Gradescope will say 12:30 AM on Wednesday, 11/03, 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 Preliminary Remarks

Before you start this assignment, you should make sure to understand all of the lectures up to but not including the 10/22 lecture (in which we moved on to functions). Understand every single line of assembly code that was in our solutions for the examples we did in slide deck #3. If you try to do this assignment without such an understanding, then you'll find yourself overwhelmed by all of the little details, e.g. regarding file organization, and you'll also find yourself needlessly introducing all kinds of bugs in your assembly code programs, which would especially be bad because – as you might unfortunately find out – bugs are so much harder to fix in assembly code.

4 Relevant Details Regarding RISC-V and RV32EM

While going through slide deck #2, whenever we referenced specifics of an ISA (instruction set architecture), assembly language, or CPU internals, we typically talked about the x86-64 ISA. x86-64 is what is known as an *incremental ISA*. This means it has to maintain backwards compatibility with old versions of the ISA. As you'll see when we talk about the x86-64 ISA and assembly language towards the end of the quarter, this results in an assembly language that isn't particularly easy to deal with for the CPU or for the assembly language programmer[1].

In contrast, RISC-V is what is known as a *modular ISA*. Every 32-bit RISC-V processor has a base (called RV32I, where the "I" presumably stands for "integer"), which is to say that every RISC-V processor supports all of the basic instructions for unsigned/signed integers, branches, etc. Because RISC-V is modular, we can choose to add additional capabilities/hardware to the processor in order to support more instructions or different use cases. When we say "RV32EM" (the "I" is included but not written), we are including the "E" extension and the "M" extension. The "M" extension means that the process supports instructions for multiplication and division, although we did not talk about such instructions during lecture. The "E" extension is meant to help low-end cores / embedded systems and means that there are only 16 registers. (RISC-V typically has 32 registers.) In short, a RV32EM processor is a 32-bit RISC-V processor for an embedded system that supports basic operations (regarding integers, branching, etc.) and multiplication/division.

5 RV32EM Simulator

For this assignment, as was done in the slide deck #3 examples during lecture, you will use Professor Chris Nitta's RV32EM simulator, the source code for which is on GitHub here.

Please go through the PDF `using_simulator.pdf` on Canvas here for directions/advise on using the simulator. As stated in that PDF, I do NOT recommend that you install the simulator yourself, unless you have quite a bit of time on your hands.

6 Programming Problems

6.1 File Organization and Compilation

You will ONLY submit the following three files. In other words, these are the only files that you can modify. (Skeleton versions of them – to help get you started – are provided on Canvas.)

- `count_target.s`
- `sum_within_range.s`
- `maxes.s`

There are many files involved in this assignment. **Below are the ones that are involved regardless of which of the three parts of the assignment we are looking at. You should not modify any of these files.** When your submission is graded, our own/unmodified versions of these files will be used. You don't need to go through these files' contents to try to understand them; we might talk about them during lecture if time permits, but it isn't necessary for doing this assignment.

- `crt0.s`: starting point of the program. Notice that it contains `_start`, which – as I've said at least once or twice during lecture – is the true starting point of a program, unlike `main`.
- `startup.c`: contains `init()`, which is called in the `_start` function in `crt0.s`.
- `Makefile`: for compiling executables from your code for each of the three parts.
- `riscv32-console.ld`: linker script that is necessary (for reasons we may talk about later) since we are compiling code for a *bare metal* simulator that lacks an operating system.

As was the case for the examples in slide deck #3, the inputs to your programs will be global variables in the `.data` section. In order to make it easier to change the inputs (for testing multiple inputs on your submission quickly), the `.data` section will be in a different file that is compiled with the file containing your `.text` section for that part. For instance, for part #1, the file that you are to submit is `count_target.s`. A skeleton version of this file is provided on Canvas and looks as follows:

[1]As an analogy, think of all of the awkward aspects of C++ (specifically recent standards such as C++11) that come from C++ being – for the most part – an "extension" of C. C++ supports strong enumerators, but it still has to support the weak enumerators from C. C++ has new libraries for supporting random number generation, but it still has to support the library functions for random number generation that C has. C++ has the `class` keyword, but it still has to support the `struct` keyword from C.

```
.text

.global main
main:

    # YOU SHOULD NOT ADD ANY CODE PAST THIS POINT.
end:
    j end
    .end
```

One of the example input files, `input_count_target1.s` – also provided on Canvas – looks as follows:

```
.data

.global arr, arrlen, target, count
arr:
    .word 5, 13, -8, 19, 20, 13, 5, 13
arrlen:
    .word 8
target:
    .word 13
count:
    .word -1
```

Notice the `.data` section is in the latter file, whereas the `.text` section is in the former. **You are not allowed to add a `.data` section to any of the three files that you are supposed to submit for this assignment.** The `.global` directive is used on the symbols/labels `arr`, `arrlen`, `target`, and `count` in order to make them accessible outside of the `input_count_target1.s` file. For example, in `count_target.s`, you could do `lui t1, %hi(arr)`. *Do not* copy/paste the contents of an input file into your program (whether `count_target.s` or the ones for the other two parts), at least not when you do your final submission. Make sure to still use the `%hi`/`%lo` setup that we used during lecture, when applicable.

During lecture, I typically showed how the `.data` section values appeared in memory. The variables/arrays from this section will not appear in memory until after the `init()` function (called in `crt0.s` and defined in `startup.c`) is done, so you may find it helpful to set a breakpoint at the first instruction of your `main()` function (i.e. the first instruction after the `main` label) and go straight to that breakpoint. Note too that if your code does not access any variables at all (i.e. because you're just starting the part), the `.data` section will not be loaded into memory at all, regardless of the variables/arrays in it[2].

Continuing our discussion of part #1 specifically (a lot of this applies to part #2 and part #3 too, but with `sum_within_range.s` or `maxes.s` instead of `count_target.s`), compilation of the full RV32EM executable requires the following code files:

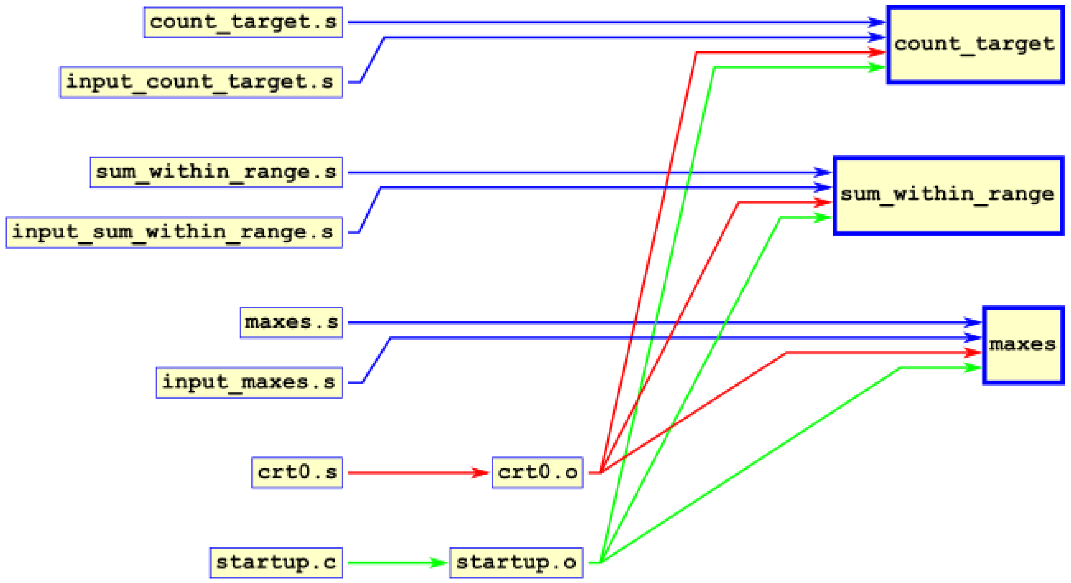
- `crt0.s`
- `startup.c`
- `count_target.s`
- `input_count_target.s`

On Canvas, I provide you a makefile that properly compiles the above four files together with the *necessary* flags in order to create an RV32EM executable that can run on the simulator. (All of the above four files, the makefile, and `riscv32-console.ld` should be in the same directory.) The default/`all` rule will attempt to build all three executables (i.e. `count_target`, `sum_within_range`, and `maxes`). You could just build a specific target by doing `make` followed by whichever of the three executables you want to build, e.g. `make sum_within_range`. (I won't review makefiles here, since you should have learned about them in ECS 32C, ECS 36B, or an equivalent course. In case it helps, I've uploaded my own slide deck on makefiles to Canvas here.)

Note that the input file needs to be called `input_count_target.s`, not `input_count_target1.s` or `input_count_target2.s`, unless you change the makefile. You can just rename whatever input file you want to use to have the right name when you use it.

Below is a diagram that shows how – thanks to the provided makefile – the different files come together to form the executables `count_target`, `sum_within_range`, and `maxes`.

[2] In case you are curious, this is due to the use of the `--gc-sections` flag in the provided makefile. This flag stands for "garbage collect sections", and it tells the linker (which runs after the assembler) to remove unused sections. This would include the `.data` section if your code does not access any of the variables.



Below is an example of what using the makefile with the appropriate files should look like on a Linux command line that is logged on to the CSIF. First, notice that all of the needed files are in the current directory. (Of course, if you only care about compiling the code for one of the three parts, that's fine too; you can omit the files for the other parts. Also, it of course doesn't matter if there are other files in the directory.)

```
$ ls -1
count_target.s
crt0.s
input_count_target.s
input_maxes.s
input_sum_within_range.s
Makefile
maxes.s
riscv32-console.ld
startup.c
sum_within_range.s
```

Below, we run the makefile.

```
$ make
riscv32-unknown-elf-gcc -O0 -g -ggdb -ffreestanding  -nostartfiles -nostdlib -nodefaultlibs -DDEBUG -c
    crt0.s -o crt0.o
make: riscv32-unknown-elf-gcc: Command not found
Makefile:22: recipe for target 'crt0.o' failed
make: *** [crt0.o] Error 127
```

Because `riscv32-unknown-elf-gcc` – the compiler for converting C and RISC-V assembly code into machine code for a RISC-V processor – is not recognized, compilation fails. We need to make sure that the path to this compiler – which I have also installed in my CSIF files – is known. We can do this by modifying the `PATH` environment variable like so[3]:

```
export PATH=$PATH:/home/aaron123/opt/riscv32/bin
```

I believe that you would have to type the above line on the command line each type you log in to the CSIF. If you want to avoid having to do this, then I would recommend adding the above line to a `.bash_aliases` file (in your home directory on the CSIF) and putting the following line in the `.bashrc` file (that should *already* be in your home directory on the CSIF): `source ~/.bash_aliases`. Each time that you open a new session on the CSIF, the `.bashrc` file will be checked, which in turn means that the `.bash_aliases` file will be checked.

Now, we can try to compile again, and assuming no assembler errors, we should see something like the below.

```
$ make
riscv32-unknown-elf-gcc -O0 -g -ggdb -ffreestanding  -nostartfiles -nostdlib -nodefaultlibs -DDEBUG -c
    crt0.s -o crt0.o
riscv32-unknown-elf-gcc -O0 -g -ggdb -ffreestanding  -nostartfiles -nostdlib -nodefaultlibs -DDEBUG -c
    startup.c -o startup.o
riscv32-unknown-elf-gcc crt0.o startup.o count_target.s input_count_target.s -o count_target -O0 -g -ggdb
    -ffreestanding  -nostartfiles -nostdlib -nodefaultlibs -DDEBUG -Wl,--gc-sections -Wl,-T,riscv32-
    console.ld
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension ‘i’
```

[3]Be careful about copying this text from this PDF. When you paste it, there might be many whitespaces inserted in certain parts.

```
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
riscv32-unknown-elf-gcc crt0.o startup.o sum_within_range.s input_sum_within_range.s -o sum_within_range -
    O0 -g -ggdb -ffreestanding  -nostartfiles -nostdlib -nodefaultlibs -DDEBUG -Wl,--gc-sections -Wl,-T,
    riscv32-console.ld
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
riscv32-unknown-elf-gcc crt0.o startup.o maxes.s input_maxes.s -o maxes -O0 -g -ggdb -ffreestanding  -
    nostartfiles -nostdlib -nodefaultlibs -DDEBUG -Wl,--gc-sections -Wl,-T,riscv32-console.ld
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
/home/aaron123/opt/riscv32/lib/gcc/riscv32-unknown-elf/10.2.0/../../../../riscv32-unknown-elf/bin/ld:
    cannot find default versions of the ISA extension 'i'
```

I have no idea how to get rid of the repeated messages complaining about default versions of ISA extensions, but it is safe to ignore those messages. Compilation was successful, and we can now see the compiled executables (and object files) in our current directory, as shown below.

```
$ ls -1
count_target
count_target.s
crt0.o
crt0.s
input_count_target.s
input_maxes.s
input_sum_within_range.s
Makefile
maxes
maxes.s
riscv32-console.ld
startup.c
startup.o
sum_within_range
sum_within_range.s
```

If you get an assembler error message, it will look similar to a compiler error message. Below is an example of one caused by my making an honest mistake in my `count_target.s` file.

```
$ make
riscv32-unknown-elf-gcc -O0 -g -ggdb -ffreestanding  -nostartfiles -nostdlib -nodefaultlibs -DDEBUG -c
    crt0.s -o crt0.o
riscv32-unknown-elf-gcc -O0 -g -ggdb -ffreestanding  -nostartfiles -nostdlib -nodefaultlibs -DDEBUG -c
    startup.c -o startup.o
riscv32-unknown-elf-gcc crt0.o startup.o count_target.s input_count_target.s -o count_target -O0 -g -ggdb
    -ffreestanding  -nostartfiles -nostdlib -nodefaultlibs -DDEBUG -Wl,--gc-sections -Wl,-T,riscv32-
    console.ld
count_target.s: Assembler messages:
count_target.s:3: Error: unrecognized opcode 'aaron is too cool for school'
```

```
Makefile:19: recipe for target 'count_target' failed
make: *** [count_target] Error 1
```

6.2 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[4], 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.***

6.3 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.

6.4 Use of Registers

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

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

As I've said from time to time, 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, one of those two registers would be `x0` (or `zero`), which is hardwired (by circuitry) to always be zero, no matter what you do to it. The other is `x3` (or `gp`). This register keeps track of where the global variables (the data in the `.data` section) is in memory. It is already set to the correct value in `crt0.s`, and you should avoid changing the value of this register; changing its value could result in reading/changing the wrong data when you load/store from/to memory. For the next assignment (which will deal with functions/subroutines), there will be more registers that I will advise you to not modify.

Whether you use the registers' special names or generic names is up to you, but you should probably be consistent within a given assembly code file, so as to avoid a mistake in which, for instance, you modify `x10` and then modify `a0` without realizing you modified the same register twice.

6.5 Part #1: Count Target

File: `count_target.s`

For this part, any use of the term "integer" refers to a 32-bit, signed integer.

Write a program that counts the number of times that a target integer (given in a variable `target`) appears in an array of integers called `arr` whose length is given in a variable called `arrlen`. The final count should be placed in a variable `count` that will already be defined in the input file.

Your program should be written in a file called `count_target.s`. You should start with the skeleton version that is provided on Canvas and that looks as follows:

```
.text

.global main
main:

```

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

```
    # YOU SHOULD NOT ADD ANY CODE PAST THIS POINT.
end:
    j end
    .end
```

As was done in the last two or three examples in slide deck #3, your program will end in an infinite loop. This is to keep the simulator from going off into "nowhere" (i.e. random instructions that could cause chaos) after reaching the end of your program. (Remember: the fetch-decode-execute instruction cycle *does not stop* so long as the computer/simulator is on! Something will be fetched, decoded, and executed; it's just a question of what that "something" is.)

The first example on Canvas – `input_count_target1.s` – is shown below. (As a reminder, you would want to rename this to `input_count_target.s` in order to use it with the provided makefile as is.) The number of times that 13 appears in `arr` is 3, so your program should set the `count` variable to 3. Your program should not assume (or care about) the initial value of `count`.

```
.data

.global arr, arrlen, target, count
arr:
    .word 5, 13, -8, 19, 20, 13, 5, 13
arrlen:
    .word 8
target:
    .word 13
count:
    .word -1
```

The second example on Canvas – `input_count_target2.s` – is shown below. With this input, the `count` variable should be set to 2 by your program.

```
.data

.global arr, arrlen, target, count
arr:
    .word 19, 20, 14, 19, 15
arrlen:
    .word 5
target:
    .word 19
count:
    .word -1
```

During lecture, we hard-coded the array length. Here, you will have to load the array length from the `arrlen` variable instead. (We had to do it this way since your submission will be tested with different inputs in which arrays of different lengths are involved.)

6.6 Part #2: Sum Within Range

File: `sum_within_range.s`

For this part, any use of the term "integer" refers to a 32-bit, signed integer.

Write a program that iterates through an array of integers called `arr` and calculates the sum of all integers in the array that are between (inclusive) a lower boundary (given by the variable `low`) and an upper boundary (given by `high`). (You may assume that `low` is always less than or equal to `high`.) The final sum should be placed in a variable `sum` that will already be defined in the input file.

Your program should be written in a file called `sum_within_range.s`. You should start with the skeleton version that is provided on Canvas and that looks as follows:

```
.text

.global main
main:

    # YOU SHOULD NOT ADD ANY CODE PAST THIS POINT.
end:
    j end
    .end
```

The first example on Canvas – `input_sum_within_range1.s` – is shown below. (As a reminder, you should want to rename this to `input_sum_within_range.s` in order to use it with the provided makefile as is.) The values in the array `arr` that are between 5 and 15 (inclusive) are 7, 12, and 5, and the sum of these values is 24, so your program should place 24 into the `sum` variable.

```
.data

.global arr, arrlen, low, high, sum
arr:
    .word 7, -8, 12, 5, 100, 4
arrlen:
    .word 6
low:
    .word 5
high:
    .word 15
sum:
    .word -2
```

The second example on Canvas – `input_sum_within_range2.s` – is shown below. The only value in the array `arr` that is between -100 and 0 (inclusive) is -8, so that is what your program should place into the `sum` variable.

```
.data

.global arr, arrlen, low, high, sum
arr:
    .word 7, -8, 12, 5, 100, 4
arrlen:
    .word 6
low:
    .word -100
high:
    .word 0
sum:
    .word -3
```

6.7 Part #3: Maxes

File: `maxes.s`

For this part, any use of the term "integer" refers to a signed integer, but *NOT necessarily* a 32-bit integer. (The size will be specified.)

Write a program that iterates through an array of **16-bit** integers called `arr` – whose length is given by the **32-bit** integer variable `arrlen` – and calculates the max of all integers at odd indices and the max of all integers at even indices. (You may assume that the length of the array is always at least 2.) These maxes should be placed in the `oddMax` and `evenMax` variables, respectively.

Your program should be written in a file called `maxes.s`. You should start with the skeleton version that is provided on Canvas and that looks as follows:

```
.text

.global main
main:

    # YOU SHOULD NOT ADD ANY CODE PAST THIS POINT.
end:
    j end
    .end
```

The first example on Canvas – `input_maxes1.s` – is shown below. Since `arr` is an array of 16-bit integers and `oddMax` and `evenMax` are 16-bit variables, the `.half` directive (standing for "halfword"), instead of the `.word` directive, is used. When loading/storing values that are *not* 32 bits, `lw` and `sw` ("load word" and "store word") will be inappropriate[5]; for 16 bit values, you should use `lh` and `sh` ("load halfword" and "store halfword"). You should think carefully about what else might differ (in the typical code you have for dealing with an array) due to the use of 16-bit elements (instead of 32-bit elements) in the array.

Regarding the specifics of the example below, the elements at odd indices (i.e. 1, 3, 5, etc.) are 14, 18, 25, and 13. The max of these is 25, so that is what the `oddMax` variable should be set to by your program. The elements at even indices are 5, -8, 20, and 5. The max of these is 20, so that is what the `evenMax` variable should be set to by your program.

```
.data

.global arr, arrlen, oddMax, evenMax
arr:
```

[5]You should ask yourself – or, even better, experiment to determine – what happens if you do use `lw` or `sw` with a 16-bit variable.

```
    .half 5, 14, -8, 18, 20, 25, 5, 13
arrlen:
    .word 8
oddMax:
    .half 0xFFFF
evenMax:
    .half 0xFFFF
```

The second example on Canvas – `input_maxes2.s` – is shown below. With this input, the `oddMax` variable should be set to 40 (the max of 10 and 40) by your program, and the `evenMax` variable should be set to 30 (the max of 30, 20, and 10).

```
.data

.global arr, arrlen, oddMax, evenMax
arr:
    .half 30, 10, 20, 40, 10
arrlen:
    .word 5
oddMax:
    .half 500
evenMax:
    .half 500
```

7 Grading Breakdown

As stated in the updated syllabus, this assignment is worth 4% of your final grade. Below is the approximate worth of each individual part of this assignment.

- Part #1: 1.25%
- Part #2: 1.25%
- Part #3: 1.5%

8 Submission/Autograder Details

You will only submit the following files. (Do not submit any other files!)

- `count_target.s`
- `sum_within_range.s`
- `maxes.s`

There might not be an autograder in the way that there was for P1. I'll release more details about this later.

UCDAVIS
COMPUTER SCIENCE