Assembly : Writing our first assembly program

In this lab, we will write our first assembly program. While the program will be very simple, it is still a milestone to write an assembly program from scratch. There are actually very few people who know how to do this and even fewer who really understand what they are doing. You will be able to count yourself among them.

The goal of the lab is to both get us familiar with strange syntax of assembly programming as well as bridging it to the concepts we have been covering.

Setup

Follow the post on piazza to create your Github Classroom lab repository.

Log into your UNIX environment and clone the repository and change directories into the repository working copy.

Create a file called `mov.s` in that directory.

> **Example files**
>
> We have included a version of the first two programs that you develop in this lab and a makefile in the lab repository. These are the files that begin with the name `example_`. It is best, however, if you try and write your own version first without looking at these.

Reference Sheet

We have written a reference sheet that summarizes many important facts that will make our job of writing and debugging assembly code easier. A copy is included with this handout.

The pdf is included in the text book and can be found here:

https://CS-210-Fall-2023.github.io/UndertheCovers/textbook/images/INTELAssemblyAndGDBReferenceSheet.pdf

We recommend you keep this beside you when working on your labs and assignments.

Writing a simple “program” that loads a register

All of the smart bits of our computer live inside our CPU. To get any work done, we need to load values into the CPU registers. The operating system lets us load memory from a properly formatted file. These files are called binary executable files, but we can simply refer to them as binaries or executables. Our goal is to create an executable that encodes a value and an instruction for loading that value into CPU register.

INTEL Instruction: MOVE

INTEL CPUs have an instruction called `mov` that lets us transfer data in various ways. We are going to try and get a sense for how to use the `mov` instruction by writing some examples.

Before we get going, let's take a quick look at a snippet from the INTEL manual that documents the `mov` instruction. Given how powerful the instruction is, the documentation is very long. However, it is still worth looking at an example of the documentation provided by the CPU manufacturer, as they provide details for every operation supported by their products.

https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf#G7.865954

Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, a doubleword, or a quadword.

You can find the INTEL manuals here:
https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

To make our lives a little easier, the reference sheet has a summary of all the INTEL instructions that we will use in this class. On the reference sheet, look for "Common Intel mnemonics (instructions)". In the case of the move operations, the mnemonic is `mov`, and that is what will use in our assembly source code file.

The Registers

Chapter 3, "Basic Execution Environment" of Volume 1 of the Intel manuals provides a long and detailed discussion about the parts of the INTEL CPUs and how they interact to execute software. Given how old and varied the INTEL product line is, and the need for the manual to be very detailed, things can get complicated. Fortunately, however, writing basic user software is really not that complicated given that, like most CPUs, the INTEL products conform to the generic von Neumann Architecture we have been studying.

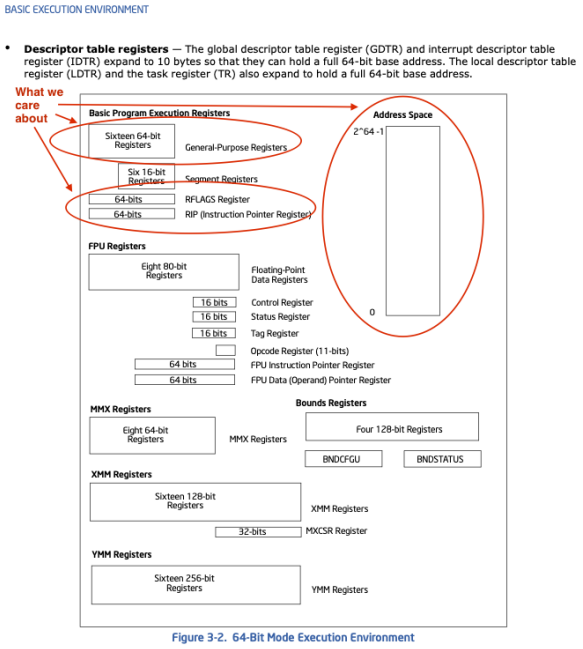


Figure 3-2. 64-Bit Mode Execution Environment

3.3 MEMORY ORGANIZATION

The memory that the processor addresses on its bus is called **physical memory**. Physical memory is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address, called a **physical address**. The **physical address space** ranges from zero to a maximum of $2^{36} - 1$ (64 GBytes) if the processor does not support Intel 64 architecture. Intel 64 architecture introduces a set of changes in physical and linear address space; these are described in Section 3.3.3, Section 3.3.4, and Section 3.3.7.

3-6 Vol. 1

We will be using an INTEL CPU in its "64-Bit Mode Execution Environment". Figure 3-2, 3-6 Volume 1 of the INTEL manual (to the left) illustrates this mode of execution. In the end, all we need to care about to program an INTEL processor is the GPRs, the RIP register, the EFLAGS register, and the range of memory addresses supported. As illustrated, the range of addresses is $0 - 2^{64}$ (in other words memory addresses are 64 bit numbers).

So, what we really need to focus on are the 16 General Purpose Registers (GPRs) of an INTEL processor, as they are the locations in the CPU that we can use to hold the values we want to work on. On the reference sheet, you will see the 16 INTEL GPRs illustrated ("INTEL Registers: Names, sizes and bit positions"). The names are the labels within each box. These will be the register names that we use in our assembly code to say where we want a value to be move to or from. It will be our job to remember what data we have moved into what register.

> **Note**
>
> At first, you might find it really useful to keep a piece of paper beside you and draw boxes for the registers so that you can keep track of what you put in each.

The Source Code mov.s

Ok, finally lets write some “code”! Our job is to create an ascii file that contains INTEL assembly code that our assembler can translate into

1. INTEL opcodes (the binary values that correspond to the instructions we want to execute)
2. Definitions of any extra memory space we want for data

Let’s start with something really easy. Let’s write code that gets the value `1` into the register called `rax`.

Now there are two obvious ways for us to do this:

1. Using an ‘immediate’ to encode the value
2. Placing the value 1 into the data section and then moving it from there

---

1. Using an immediate

This is about as easy as it gets. You specify the `mov` instruction and tell the assembler that the destination of your move is the register you want, followed by the value you want to move. By default, if you specify a number, the assembler will assume signed decimal notation. However, if you prefix the value with `0x`, you can use hex notation for the value.

So the line of code that we can use to load the `rax` gpr with the value of 1 would be:

```
mov rax, 1
```

Setting the syntax

Now to turn this into a full blown program, we need to add some directives before our line of code. First, as we did in our prior lab, we need to tell the assembler that we are using the same syntax as the INTEL manuals for our code.

```
.intel_syntax noprefix
```

Putting our code into the text section

We need to tell the assembler that we want to place the resulting binary opcodes in the area of memory where instructions should go – the TEXT section. Remember to keep stay organized and have the operating system help us catch bugs by separating our instructions from the data of our program.

To do this, we tell the assembler that what follows should go into the text section with the following line of code.

```
 .section .text
```

You can shorten this to just `.text`, but being more explicit can help us remember what the line is doing. It says what follows should be placed in the text section of memory.

> **Note**
>
> Be careful to notice that there is a dot (`.`) at the beginning of the word text.

At this point, any values we specify, with additional lines of code, will continue to to placed one after the other into the text section of memory. This will continue to be true unless we switch sections. We will look at an example of how to do this later.

Ensuring that execution starts at the right instruction

By default, the linker on our operating system (Linux) will look to see if there is a label called `_start` in the symbol table. If there is, then the linker will mark, in the binary, that the address associated that this label is the binary's "entry" point. When the OS loads the binary, it will ensure that the PC register will be initialized with entry point address.

> **_start: The default entry point for a program on Linux**
>
> You can think of `_start` like the `main` method of a java program in that it is where execution starts, with the exception that we can be more exact in knowing how things are working. We place a label called `_start` in our text section to mark the starting code of our program. The linker will ensure that the address of this code is recorded in the binary as the "entry" point. This entry point is the address that the PC register, called `RIP` on INTEL, will get set to prior the to the process starting to execute on the CPU. This means that the instruction we encode at the _start label will be the first instruction to execute in our program.

So, to ensure that our binary will start by executing our move instruction, we need to mark its location with the label `_start`

```
   .global _start
_start:
    mov rax, 1
```

> **Note**
>
> Placement of `_start`: We can place the `_start` label anywhere in the text section. In our examples, we happen to place it as the very first instruction that we put into the text section, but this need not be the case.

So, the complete source code for our program should look something like this:

```
    .intel_syntax noprefix
    .section .text
    .global _start
_start:
    mov rax, 1
```

Assemble and link this program

Write a makefile to automate the translation of our source code into a binary called `mov` Our makefile should first create the object file `mov.o` from `mov.s` using the assembler as follows

```
as -g mov.s -o mov.o
```

and to create the `mov` executable, you can run the linker on our object file `mov.o` as follows

```
ld -g mov.o -o mov
```

If you can't remember how to write the makefile here is the what it should look like

```
mov: mov.o
    ld -g mov.o -o mov
mov.o: mov.s
    as -g mov.s -o mov.o
```

Don't forget that the recipe commands of our "rules" need to be indented with a single tab.

> **Make and Rules**
>
> If you are having troubles with makefiles be sure to review the following:
>
> - https://www.gnu.org/software/make/manual/make.html#Introduction
> - https://www.gnu.org/software/make/manual/make.html#Rule-Introduction
> - https://www.gnu.org/software/make/manual/make.html#Rule-Example
> - https://www.gnu.org/software/make/manual/make.html#Rule-Syntax

Use gdb to explore the binary

1. Start gdb eg. `gdb -x setup.gdb mov`
2. Set a break a break point `b _start`
3. Start a process from our binary `run`
4. What what happens to `rax` when you single step our instruction?

> **gdb.setup**
>
> Remember the gdb.setup file is just a set of gdb commands that we have gdb run before we start using it. These commands just help configure gdb in a way that we find makes things easier. We have provided the file for you in our lab and assignment repositories. The following is its contents, and you are free to simply create the file yourself with the commands in it:
>
> ```
> set disassembly-flavor intel
> tui new-layout mylayout regs 3 {src 1 asm 1} 2 cmd 1 status 0
> layout mylayout
> winh cmd 4
> focus cmd
> ```

Exiting and Exit Status

Now, modify your code to call the OS kernel exit system call after your instruction by adding the following to the end of your code

```
    /* Call OS EXIT system call */
    mov rax, 60  # move exit system call number 60 into rax
    mov rdi, 0   # move exit status code into rdi
    syscall      # call OS kernel (process will terminate)
```

> **Exit Status**
>
> The value we place in the `rdi` register prior to calling the OS exit system call is our program's exit status. Changing this value will change the value of the exit status.

This will cause your program to request that the OS terminate your process right after your single move instruction. See https://CS-210-Fall-23.github.io/UndertheCovers/lecturenotes/assembly/L08.html#exit-an-os-service-to-terminate-a-process for details.

---

2. Using a data value

Now that we learned to load a register using an immediate, let's try something a little fancier. Let's first place a value into the data section of our binary. This way, the OS will load the value into memory. Then, we can use a different version of the move instruction to load a register from the location in memory that our new value ends up at. Let's load this new value into the `rbx` register. This way, we will have one value in `rax` and another in `rbx`.

> **Note**
>
> It is a good idea to draw a diagram to help you understand what you are doing. Eg. draw boxes for rax, rbx, and some boxes for memory. Update the diagram as you go along to understand how things get laid out in memory and the registers.

To place our data value in memory, we will need to switch sections to the `.data` section and then use an other directive to place our value.

To switch sections we use

```
    .section .data
```

> **Keep data and text organized in the source files**
>
> We can add our `.data` section anywhere in our source code file. We can even switch back and forth mixing text and data through out our source files. However, it is generally a good idea to place all our data together, either before or after our `.text` section just so that it is easier to read. Remember however, if you place it at the start, you need to switch to the `.text` section to write instructions.

Given that we are loading a 64 bit register, we want the assembler to encode an 8 byte value in the data section. To do this, we use the `.quad` directive. Eg. to encode the value 2 as a 64 bit value, we would use the following code

```
    .quad 2
```

> **.quad, numbers, and other directives**
>
> 1. You can find information about the `.quad` directive here: https://sourceware.org/binutils/docs/as/Quad.html
> 2. You can find information about the syntax for numbers here: https://sourceware.org/binutils/docs/as/Numbers.html
>
> In general, we will be using a very small subset of simple gnu assembler directives and we will introduce them as needed. If, however, you would like to see the full list, they can be found here: https://sourceware.org/binutils/docs/as/Pseudo-Ops.html

At this point, our code should look something like this:

```
    .intel_syntax noprefix

    .section .data
    .quad 2

    .section .text
    .global _start
_start:
    mov rax, 1

    /* Call OS EXIT system call */
    mov rax, 60  # move exit system call number 60 into rax
    mov rdi, 0   # move exit status code into rdi
    syscall      # call OS kernel (process will terminate)
```

Great! We have now placed a value into memory. Let's add an instruction to load the value into the INTEL `rbx` register.

Working with data

Well, the instruction and destination of what we have to write is easy, which will just be the INTEL `rbx` register, but what should go to the source of this move instruction?

```
    mov rbx, WhatGoesHere
```

This is where labels come in.

> **Labels and references**
>
> In this case, the source needs to be the address of where our data value ends up in memory. However, it would be very complicated if we had to decide where everything should go in memory. We would have to know all the rules the the OS might have and we would need to coordinate with any extra code or libraries that we might include in our binary. To avoid all this hassle, we can simply add a label in our code before our data value and then use the label in our instruction. The assembler and linker will cooperate to ensure that any "uses" of the label in our instructions will get updated with the final address of where the data at the label gets placed. "Uses" of a label are lines of our code that have name of label written instead of a numeric address. Uses of a label are also called **references**.
>
> This might seem confusing at first, but as we write some examples, it will get easier.

Let's modify our code to add a label, and then let's modify our new `mov` line to reference that label. For the lack of creativity, let's use the symbolic name `x`. Feel free to use whatever you want.

```
    .intel_syntax noprefix

    .section .data
x:  .quad 2

    .section .text
    .global _start
_start:
    mov rax, 1
    mov rbx, WhatGoesHere

    /* Call OS EXIT system call */
    mov rax, 60  # move exit system call number 60 into rax
    mov rdi, 0   # move exit status code into rdi
    syscall      # call OS kernel (process will terminate)
```

Notice that all we needed to do was add the label `x` to the beginning of the `.quad 2` line. Remember, we need to suffix the name with `:` to let the assembler know it is a label.

> **Linking – Symbols and Symbol Resolution**
>
> When we run the the linker to process our code, it will "resolve" the references we have made to the labels. The names of our labels are called symbols. The linker will assign every value in the program to memory addresses. For values that have labels, it will remember the location and this will become the symbol's address. Then, the linker will go back and "fix up" all the code that has references to symbols, `x` in our case, where "fix up" means replacing the symbolic references to the label with the address of the label. Notice there are various errors that might crop up during this step if our code is not "correct". Eg. there are references to symbols that we did not define with a label, there is more than one definition for symbol, etc.

Ok, but now, how do we update date our `mov rbx,` line to use the label? For this, we need to use INTEL's memory addressing syntax. Using this syntax, we need to do two things:

1. specify the number of bytes that we want to move from memory.
2. specify the starting address.

To do the first part, we need to use one of the following:

1. `BYTE PTR`
2. `WORD PTR`
3. `DWORD PTR`
4. `QWORD PTR`

You can think of these as saying what follows is a “pointer” – a number that is a memory address of a value that is a single byte, two bytes, four bytes or eight bytes in size respectively. In our case, our value is eight bytes in size, so we are going to use `QWORD PTR`.

After this, we then need to say the value of the address. In our case, we simply need to state the name of our label and the linker will substitute the correct address when it runs. Eg.

```
   mov rbx, QWORD PTR [x]
```

Strictly speaking, if we are just using a symbolic name of a label, we don’t need the square brackets. Using them, however, can help us remember that memory is really like an array, and that the label is really just an index into the memory array.

> **Memory Address Syntax**
>
> The syntax for specifying a memory address can support much more advanced variations. Where the address is the result of a calculation that uses values in registers and constants to specify the location in memory that we want to load (or store) a value from (or to). This is where a lot of the power and difficulty comes from when writing assembly code. We will learn more about this in time.

Ok, so our code now should look like the following:

```
     .intel_syntax noprefix

    .section .data
x:  .quad 2

    .section .text
    .global _start
_start:
    mov rax, 1
    mov rbx, QWORD PTR [x]

    /* Call OS EXIT system call */
    mov rax, 60  # move exit system call number 60 into rax
    mov rdi, 0   # move exit status code into rdi
    syscall      # call OS kernel (process will terminate)
```

At this point, you should rebuild your binary and make sure you don’t have any assembler or linker errors. If all is good, you should be able to create your new binary and use gdb to play with it.

A complete example of this can be found in the `example_mov.s` included in the discussion repository. Also, you will find that the `example_Makefile` has rules to build the `example_mov`.

Doing something with our values

Lets use our `mov.s` as the start for a new program called `addtwo` which adds the values we have loaded into the registers. To do this

1. use `cp` to copy `mov.s` to `addtwo.s` (eg. `cp mov.s addtwo.s`)
2. add new targets to your makefile (one for `addtwo` and one for `addtwo.o`). So that you can run `make addtwo` to build you new program
3. Use the reference sheet to add a line of code that will add the value in `rbx` to the value in `rax`

Use gdb to see if you program behaves the way you expect.

Then, update your code by introducing a new data value called `result`, then after the two values have been placed into `rax` and `rbx`, sum up the value and store it in `rax`. Lastly, use `mov` that saves the value that is currently in `rax` to the location of result. Hint: the source of `mov` will be `rax` and the destination will be a quad word pointer to the address of `result`.

If you got this far and everything worked, great job!

Small challenge

See if you can modify your code so that result you calculated is the exit status for your program. If you get this to work, then when you run without the debugger and then `echo $?`, you will see 3. Eg.

```
$ ./addtwo
$ echo $?
3
$
```

A complete example of this code can be found in `example_addtwo.s`.

Trace and draw

Now that we have "real" program, build it with your makefile and use the debugger to explore how it works using `gdb`

Use `gdb` to explore things and draw a diagram, on paper, that illustrates the step by step behavior or your code. Draw a box for `rax`, `rbx`, `rip`, and then draw two arrays of bytes for your memory. Label one of the arrays as `.text` and the other `.data`. For the `.text` array, label one of the cells on the right with `_start`. Similarly, label one of the `.data` memory locations with `x` beside it. At this stage, try and draw every thing in terms of bytes. See the last page of this hand out for an example of how your drawing should look.

Now start up `gdb`, set your break point, and start your program as a process. Then, use `gdb` commands to figure out where the symbols are in memory and what byte values are at those locations. The following two commands will print out the address of our two symbols

```
p /x & _start
p /x & x
```

Now, update the address of the starting boxes for the two symbols on the diagram. Then, look at the contents of memory at these locations and again update the diagram appropriately. Here is how you might do this.

1. First you can use `disassemble _start` to ask gdb to interpret the bytes at _start symbol. They should look a lot like the assemble code we wrote.
2. Then, you can use a command like `x/32xb &  _start` to see what byte values are at those addresses and you can update you diagram. Normally, we don't really need to look at the byte values of our instructions, but it is useful to help us understand exactly what is going on.
3. Now, for our data value, we can again use the examine command `x /8xb & x`. Since we know that the value is 8 bytes in length, we know how many bytes to display. Notice displaying things as bytes we can see that the value is in little endian order in memory. We can also ask gdb to view the value as an 8 byte quantity in the same way it would get loaded into a register. To do this, we use the following gdb command `x/1xg & x`. `g` stands for giant, which is what gdb calls 64bit values.

Now, before we start tracing our program, we can print the starting values of our registers and update our diagram

```
p /x $rax
p /x $rbx
p /x $rip
```

At this point, we have the initial state of our process captured on our diagram. Now, every time we execute an instruction using `stepi`, we should update our diagram with the changes that happen. To know what changed, you will need to think about the instruction that was just executed. Determine what its destination is and then use print (`p`) or examine (`x`) gdb command to inspect the destination and update your diagram.

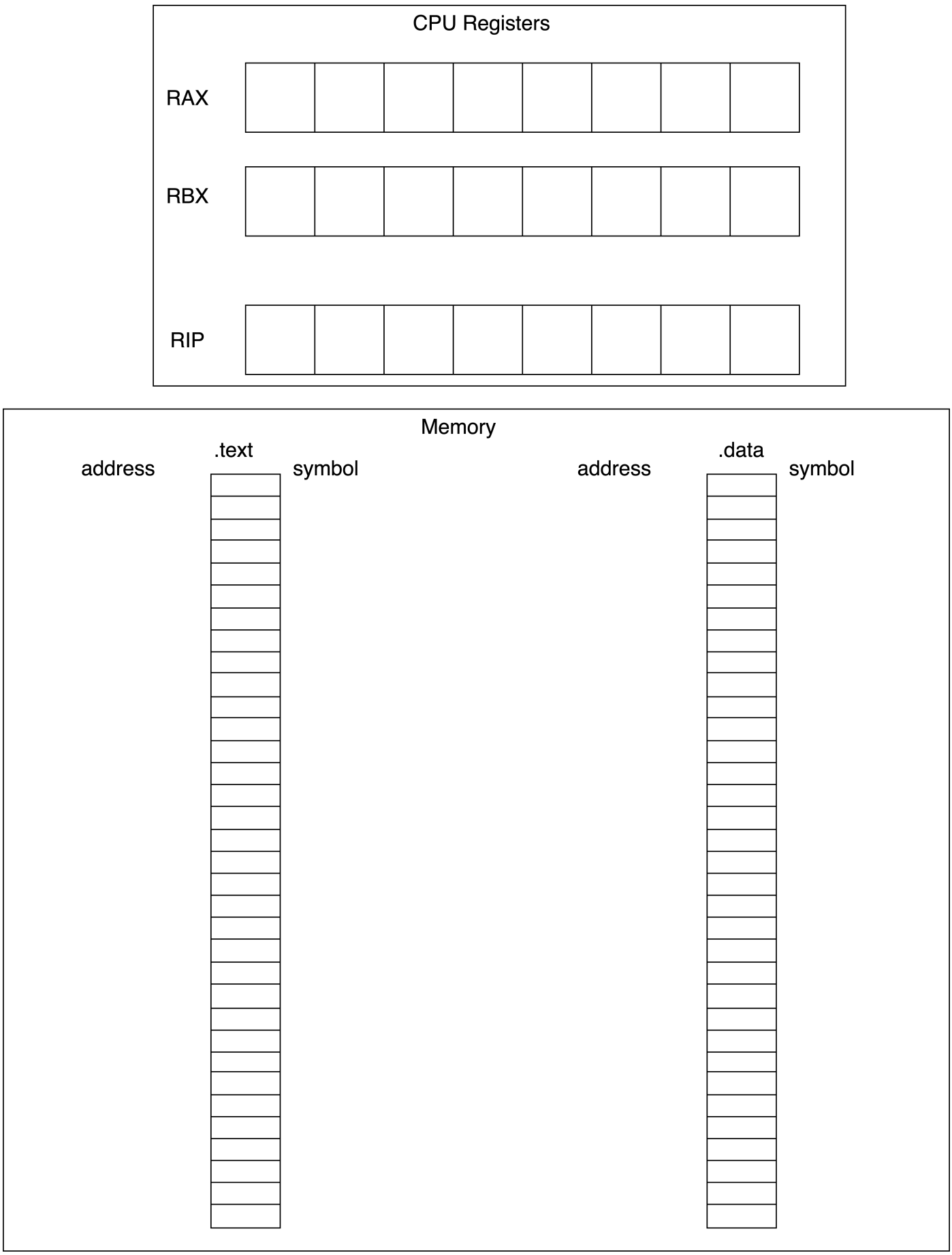
Going further

Now, using only the `mov` and `add` instructions along with any registers you like, write a new program called `addten` that adds the following numbers together.

```
599
586
122
478
58
821
926
578
55
72
```

You can use any form of `mov` you like, any combination of registers, and any type of `add`. Remember that when using `add`, you can have either its source or destination be a memory location, rather than just using registers for both. However, you should note that the you may not use a memory location for both the destination and the source within the same instruction. You will find that there are many different ways you can write this program even though we are only using two instructions. We encourage you to write several different versions.

Again, save the final result to a memory location and return the result as the exit status of your program.



By Jonathan Appavoo