Assembly : Conditionals, Loops, and Multiple Source File Programs

In this lab, we will learn how to:

1. Write conditional logic, aka *if statement*, in assembly code
2. Write a loop in assembly code
3. Break up our code into two pieces:
    - a fragment that implements the *if* code we develop
    - a driving bit of code that jumps to our fragment in a loop
4. Use gdb to execute our binary (loop over the jumps to our fragment)

It is a good idea to partner up for this lab so that you can discuss and reflect upon what you are doing.

Note: In this lab, there are various questions provided to help guide your understanding. If you feel short on time, you can work on these questions later. Rather, focus on ensuring you know how to build the binaries and know how to use gdb to explore them. Ask the staff about any questions you have.

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.

Again we have provided pre-written versions of the files with the prefix `example_`. Avoid using these files to allow yourself to develop the code, but if you find yourself short on time, feel free to use these files.

Reference Sheet

It's a good idea to have a copy of the assembly and gdb reference sheet with you when working on this lab.

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

Conditionals

Writing conditional logic is perhaps the most common thing we learn how to do in any programming language. That is to say, code that executes if a certain calculation is "true" or "false". For example, let's say we want to check to if a signed integer, `x`, is positive or negative. If it is positive, then let's increment a variable we call `postive_count` by one. Otherwise, it is negative and we will increment a variable called `negative_count` by one.

There are many ways to write code that does this in Python or Java. The following are simple python and java example programs.

Python: if.py

```
postive_count = 0
negative_count = 0

x=100

if x < 0:
    negative_count += 1
else:
    postive_count += 1
```

Java: if.java

```
class ifExample {
    public static int positive_count = 0;
    public static int negative_count = 0;

    public static void main(String args[]) {
        int x = 100;

        if (x<0) {
            positive_count++;
        } else {
            negative_count++;
        }
    }
}
```

So let's look at two examples of how one might use INTEL assembly code, on Linux, to achieve the same.

INTEL Linux Assembly: if1.s

```
  .intel_syntax noprefix

  .section .data
positive_count:
  .quad 0
negative_count:
  .quad 0
x:
  .quad 100

  .section .text
  .global _start
_start:
  mov rax, QWORD PTR [x]
  cmp rax, 0
  jl is_neg
  mov rbx, QWORD PTR [positive_count]
  inc rbx
  mov QWORD PTR [positive_count], rbx
  jmp done_cond
is_neg:
  mov rcx, QWORD PTR [negative_count]
  inc rcx
  mov QWORD PTR [negative_count], rcx
done_cond:

  mov rax, 60
  mov rdi, 0
  syscall
```

INTEL Linux Assembly: if2.s

```
  .intel_syntax noprefix

  .section .data
positive_count:
  .quad 0
negative_count:
  .quad 0
x:
  .quad 100

  .section .text
  .global _start
_start:
  cmp QWORD PTR [x], 0
  jl is_neg
  inc QWORD PTR [positive_count]
  jmp done_cond
is_neg:
  inc QWORD PTR [negative_count]
done_cond:

  mov rax, 60
  mov rdi, 0
  syscall
```

Remember these are just two examples of how you might write this code – there are many others.

Create binaries

1. Use an editor and create the two above source files.
    - if you feel like you don't need the practice of creating the files, then feel free to use the example files, eg.
        - `cp example_if1.s if1.s`
        - `cp example_if2.s if2.s`
        - Then, open these in the editor so that you can read and change them if you like.
2. Write a `Makefile` that creates a binary for each: `if1` and `if2`. Look at `example_Makefile` if you need more help. (Note: the `example_Makefile` use the file names with the `example_` prefix.)

```
if1: if1.o
    ld -g if1.o -o if1
if1.o: if1.s
    as -g if1.s -o if1.o
if2: if2.o
    ld -g if2.o -o if2
if2.o: if2.s
    as -g if2.s -o if2.o
```

1. Now you should be able to build your two binaries with: `make if1` and `make if2` respectively

Questions

Trace the code execution with `gdb` and draw diagrams to help you understand how these binaries behave. Then, answer or do the following.

- How many GPRs does each version of the code use?
- If a line of code uses a register to hold a data value or address, can you write down which "variable" the register is holding at that point? Eg.

```
    mov rax, QWORD PTR [x]  # rax is holding the value of x -- rax = value of x
    cmp rax, 0              # rax = value of x
```

- Which instructions form the "if"?
- Draw a circle around the instructions that execute when the value of x is positive.
- Draw a circle around the instructions that execute when the value of x is negative.
- Is `is_neg` a label of data or a label of instructions (text)?
- Why do we need `is_neg`?
- Which lines of code reference a label?
- Using the examine command with the value at the address of the label `x`, demonstrate that INTEL is a little endian computer (see the first part of "The gdb display command" hint below for some guidance).

Here are a couple of `gdb` hints that can make your life easier.

Exploring with different values of x

To see how the code behaves with different values of `x` we could modify our source and then rebuild the binaries. However, there is an easier way to do this using the power of `gdb`. Remember that you can set the values of memory and registers in gdb. Doing so will let you quickly explore how your program behaves with different values of `x` without having to rebuild and start a new process. For example after you step you code in gdb to the `done_cond` point you can set a new value into the memory that contains `x`, then set the pc back to the address of `_start`, and then step you code again to see how things work with this new value. Eg.

```
(gdb) set {long long} & x = -42
(gdb) set $pc = & _start
```

The gdb display command

Remember we can use the `print` (`p`) command and the `examine` (`x`) gdb commands to print value and examine memory. For example example we can use the following commands to look at the value a that the location of x with the following commands:

```
(gdb) x /1gd &x
0x402010:       100
(gdb) x /1gx &x
0x402010:       0x0000000000000064
(gdb) x /8bx &x
0x402010:       0x64    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) p {long long} & x
$1 = 100
```

Respectively:

1. Use examine to view the 8 bytes at the address of `x` as a single signed decimal value – where `g` means giant, which in gdb speak is 8 bytes, and d means in signed decimal notation (aka a 2's complement value)
2. Use examine to view the 8 bytes as the address of `x` as a single hex value – where `g` means giant, which in gdb speak is 8 bytes, and x means in hex notation
3. Use examine to view the 8 bytes at the address of `x` as 8 individual hex byte values – where `b` means byte and x means in hex notation
4. Use the print to view the 8 bytes at the address of `x` as a single 8 bytes signed decimal value – `{long long}` is telling print that the value at the address of `x` (`& x`) is a 8 byte signed integer

See the reference sheet for a summary of this gdb notation. Another, really useful command in gdb is the `display` command which uses the same syntax as both the `x` and `p` commands but instead will automatically repeatedly display the value every time gdb stops. This means if you use display you can value gdb automatically print the values of memory locations for you so you don't have to constantly using `p` or `x` to see how a particular location in memory is changing. Eg.

```
(gdb) display {{long long} & x, {long long} & positive_count, {long long} &
negative_count }
1: {{long long} & x, {long long} & positive_count, {long long} &
negative_count } = {100, 0, 0}
(gdb) si
15              cmp rax, 0
1: {{long long} & x, {long long} & positive_count, {long long} &
negative_count } = {100, 0, 0}
```

Loops

Let's expand our example to see a couple of ways to write a loop in assembly. Which is really just another use of conditional logic. Consider the following two examples.

INTEL Linux Assembly: loop1.s

```
        .intel_syntax noprefix

        .section .data
positive_count:
        .quad 0
negative_count:
        .quad 0
array:
        .quad 1
        .quad 2
        .quad -4
        .quad 5
        .quad 7
        .quad -13

        .section .text
        .global _start
_start:
        xor rdi, rdi
loop_start:
        cmp rdi, 6
        je loop_done
        cmp QWORD PTR [array+rdi*8], 0
        jl is_neg
        inc QWORD PTR [positive_count]
        jmp done_cond
is_neg:
        inc QWORD PTR [negative_count]
done_cond:
        inc rdi
        jmp loop_start
loop_done:
        mov rax, 60
        mov rdi, 0
        syscall
```

INTEL Linux Assembly: loop2.s

```
        .intel_syntax noprefix

        .section .data
positive_count:
        .quad 0
negative_count:
        .quad 0
array:
        .quad 1
        .quad 2
        .quad -4
        .quad 5
        .quad 7
        .quad -13

        .section .text
        .global _start
_start:
        xor rdi, rdi
        jmp loop_test
loop_start:
        cmp QWORD PTR [array+rdi*8], 0
        jl is_neg
        inc QWORD PTR [positive_count]
        jmp done_cond
is_neg:
        inc QWORD PTR [negative_count]
done_cond:
        inc rdi
loop_test:
        cmp rdi, 5
        jle loop_start

        mov rax, 60
        mov rdi, 0
        syscall
```

Questions

Build two binaries based on the above code and then explore how they behave and answer the questions below.

- Can you identify where the loop starts?
- Can you identify the instruction that causes the loop to repeat?
- Which instructions implement the condition that determines if the loop should exit?
- What register is being used as the loop index variable?
- Why does the first example test with 6, and the other test with 5, to decide when to exit the loop?
- Where do we encode the length of the array?
- What does this syntax mean `cmp QWORD PTR [array+rdi*8]`

Another gdb hint:

> **You can use math in your gdb commands**
>
> You might find the following gdb command really useful:
>
> ```
> (gdb) p & array
> $1 = (<data variable, no debug info> *) 0x402010
> (gdb) display /1gd 0x402010 + ($rdi * 8)
> 1: x/dg 0x402010 + ($rdi * 8)  0x402010:        1
> ```

Breaking things up

Most real code is broken up to different reusable portions and across multiple files. Let's explore how to break our above code into two parts that we combine together to create our binary.

> **Fragments vs Functions**
>
> Rather than writing full blown functions at this point, we are just going to structure our code as "fragments" so that we can better understand how functions work in our next lab. In both cases, a fragment and function, each will have a label that indicates the memory address of the starting instruction. The core difference is that our fragments will just end with an `int3` instruction to return to the debugger versus a function that will use the stack to return back to the code that calls it. This means until we move on to understanding functions, we will need to use the debugger to run our binary.

So, let's break our code into two assembly source files which together iterates over an array, counting how many are positive and how many are negative.

1. `posneg.s` : The *if* fragment
    - contains a fragment of code who's starting instruction is at a label called: `POSNEG_FRAG`
        - The fragment should take as a parameter the address of an 8 byte signed integer to test. This address should in the register `rax`.
        - It should test if the number at the address is positive or negative and update the right count.
        - It should update the address in `rax` by 8 bytes before finishing – advance rax to point to the next value.
    - defines the data memory for the `positive_count` and `negative_count`.
2. `posnegtest.s`: The driver
    - contains the starting code for our program which
        - zero's the counters.
        - jumps to the fragment in a loop .
    - defines the data memory for an array of numbers to test.

posneg.s

Here is our ‘if’ code refactored and commented. It is modified to test a value that is at the address held in rax. It is the responsibility of the calling code to setup rax correctly. As such, we need to load it with the address of the first value to test before jumping to the fragment. When this code is finished, it increments the address in rax by 8. In other words, the fragment finishes by moving rax to the address of the next 8 bytes in memory. If the calling code is working on an array, to evaluate the next element, it can simply jump to the fragment as rax will already be pointing to the next element.

```
        .intel_syntax noprefix

        # switch to data section
        .section .data
        # set aside the memory for the two counters and make sure
        # the symbols are global so that they can be used in
        # posnegtest.s
        .global positive_count
        .global negative_count
positive_count:  .quad 0      # set aside 8 bytes
negative_count:  .quad 0      # set aside 8 bytes

        .section .text

        # POSNEG_FRAG:
        # a little fragment of code that tests the 8 byte
        # signed integer number, x, pointed to by the address in rax.
        # NOTE: rax does not contain x but the address of x
        # INPUTS:
        #     rax -> &x address of where the value of x is in memory
        # OUTPUTS: if y is positive increment the positive_count
        #          else increment the negative_count
        #          finally rax should be updated to equal &x + 8
        #
        # This file must provide the global symbols
        # - positive_count : start of 8 bytes for postive count
        # - negative_count : start of 8 bytes for negative count
        # - POSNEG_FRAG    : starting instruction of the fragment

        # starting instruction label for the fragment
        .global POSNEG_FRAG
POSNEG_FRAG:
        # compare 8 byte value at the address in rax to 0
        cmp QWORD PTR [rax], 0
        # if negative (x<0) jump to negative case
        jl is_neg
        # Positive Case:
        inc QWORD PTR [positive_count] # x>=0 increment positive_count
        # end of positive case goto conditon end
        jmp done_cond
        # Negative Case:
is_neg:
        inc QWORD PTR [negative_count] # x<0 increment negative_count
        # Conditional logic ends here
done_cond:
        add rax, 8       # update rax to point the next value
        # End of our Fragment
        int3             # use int3 to return to debugger
        # will have use gdb to set pc to where we
        # want execution to go next : eg set $pc = <some label>
```

posnegtest.s

To use our fragment, we factor out our loop code into its own file. This code defines a data array of numbers and then jumps to the address of POSNEG_FRAG six times. However, remember that the fragment does not actually know how to get back to the loop instead it simply hands control back to the debugger. We will discuss what to do in gdb to continue execution properly.

```
        .intel_syntax noprefix

        # switch to data section
        .section .data
data_start:            # use data_start label too mark beginning of array
        # each element is an 8 byte integer we use .quad
        # directive for each
        .quad 1           # first  element = 1
        .quad 2           # second element = 2
        .quad -4          # third  element = -4
        .quad 5           # fourth element = 5
        .quad 7           # fifth  element = 7
        .quad -13         # sixth  element = 13

        # switch to text section
        .section .text

        # define global _start symbol so linker knows where our program
        # should begin executing
        .global _start
_start:
        xor rax, rax                        # rax = 0
        mov QWORD PTR [positive_count], rax  # set positive_count to 0
        mov QWORD PTR [negative_count], rax  # set negative_count to 0

        mov rax, OFFSET data_start          # set rax to address of
                                            # first integer

        jmp loop_test                       # jump to our loop condition
loop_start:
        jmp POSNEG_FRAG
RETURN_HERE:
        inc rdi                             # increment loop index
loop_test:
        # compare index to length of array (5)
        cmp rdi, 5
        # if index <= length jump to top of loop
        jle loop_start

        # all done call OS exit system call
        mov rax, 60      # rax = 60 os exit system call number
        mov rdi, 0       # rdi = 0  exit status 0 success
        syscall          # call OS system call
```

Building our binary from two source files

To build our binary, we will need to assemble each source file into its own object file and then link the object files together to produce our new binary. The following are the commands to do this.

Assemble posneg.s

We need to assemble our `posneg.s` source file to translate it into a binary object file – `posneg.o`

```
as -g posneg.s -o posneg.o
```

Assemble posnegtest.s

We need to assemble our `posnegtest.s` source file to translate it into a binary object file – `posnegtest.o`

```
as -g posnegtest.s -o posnegtest.o
```

Link the object files

At this point, we have four files: our two source files and the object files we created from them. Now, we need to link the object files together to produce an executable binary file that we can run – `posnegtest`. The command to do this is:

```
ld -g posnegtest.o posneg.o -o posnegtest
```

Makefile recipies to automate this

It is pain, and error prone, to type all these commands every time we change our source code to update our binary. So usually, we automate these steps by adding recipes to our `Makefile` that the `make` command can run for us. Here are the recipes we would add to a `Makefile`:

```
posnegtest: posnegtest.o posneg.o
    ld -g posnegtest.o posneg.o -o posnegtest

posnegtest.o: posnegtest.s
    as -g posnegtest.s -o posnegtest.o

posneg.o: posneg.s
    as -g posneg.s -o posneg.o
```

Now, to build our binary from our source files, we simply run `make posnegtest`. With these recipes in the Makefile, make will check to see what if any source files have changed and rerun the commands as needed to assemble and link the files to create the binary `posnegtest`

Running our binary

Given that our fragment does not know where to return to on its own, we MUST use gdb to run and test our binary.

1. start gdb with the binary and setup code
    - `gdb posnegtest -x setup.gdb`
2. run some commands so that gdb will display the running values of the positive_count negative_count
    - Command to have gdb keep showing us the value of positive and negative count memory values
        - `display {{long long}&positive_count, {long long}&negative_count}`
3. get the binary running and start tracing it
    - set a break point at _start: `b _start`
    - start the binary running as a process : `run`
    - single step each instruction to see what is happening: `stepi`
4. At some point, you will get to the `int3` at the end of your fragment. At this point, you need to manually redirect execution back to the main loop. Specifically, you need to set the pc to the address of instruction right after the `jmp POSNEG_FRAG`. Fortunately, we placed a label there called `RETURN_HERE`. So, to have execution continue properly, we can issue the command:
    - `set $pc = RETURN_HERE`
5. Now, using this knowledge, you should be able to trace all six iterations of the loop and confirm that the code does what we want.

> **GDB continue command**
>
> Once you get the hang of tracing a few iterations of the loop you probably will want to speed things. You can do this by letting execution `continue` to the next time the `int3` instruction is encountered. Unlike `stepi`, the gdb `continue` command (`c`) will cause the fetch, decode, execute loop of the processor to keep on going instead of executing just one instruction. However, execution will stop if the program hits a break point, `int3`, exits, or an error occurs. With this in mind you can test your program using by continuing until we reach the `int3` instruction. This will get us to the end of our fragment. Then, to continue execution in the loop, you can set the pc back to the return point and continue again.

By Jonathan Appavoo