Running Assembly Programs

The purpose of this lab is to extend the use of **gdb**, the GNU debugger program, to the analysis and execution of x86-64 assembler programs. For debugging C programs, `gdb` is an optional tool, although it can improve the efficiency of the C programmer in finding execution errors. For the assembly language programmer however, a debugger is essential to tracking the behavior of a program during execution. This is because, while it is easy to add "print" statements to a C program, a much more elaborate and cumbersome set of statements that can call ouput routines is required each time you wish to print the contents of a location of either external or internal memory in an assembly program.

An Example Using 8-Byte Operands (64 bits):

1. The following program is intended to sum the values 1, 2, and 3, storing the result in `ans`. The values are stored in consecutive locations of memory, with the first value stored at location `Z`. The sum is to be stored at location `ans`. Type the source code exactly as shown into a file called `testprogq.s`:

```
        .data
Z:      .quad 1,2,3
ans:    .quad

        .text
        .global main
main:
        mov $Z, %rbx
        mov 0(%rbx), %rax
        add 8(%rbx), %rax
        add 16(%rbx), %rax
        mov %rax,24(%rbx)
        ret
```

2. Assemble the program to obtain an object program as follows but do not attempt to execute the program:

   ```
   as -g  testprogq.s -o testprogq.o
   ```

   NOTE: Remember, it is important that the `-g` flag be included to insure that breakpoints are provided for gdb.

3. After all copying errors have been corrected, load it with the following command:

   ```
   gcc -o runtestq testprogq.o
   ```

   NOTE: The "`ld`" command to load an object file should not be used. Using `gcc` to load the file insures that the necessary instructions are added to the machine code so that it will terminate properly. Otherwise a segmentation fault may occur.

   Since the program calls no input or output functions, data can only be entered by declaring it in the `.data` area of the program. Similarly results can only be observed by examining memory in the `.data` area as well. `Gdb` makes it possible to examine directly what is stored in the data and program areas of memory, as well as the contents of each register in the CPU. Therefore, gdb will be used to execute the program.

4. Instead of running the executable file, `runtestq`, with the linux `run` command, launch gdb with the command line:

   ```
   gdb runtestq
   ```

5. Following the preamble, rather than run your program, enter the following gdb instruction:

   ```
   (gdb) disassemble main
   ```

   You should observe output somewhat similar to that obtained with `objdump`, as follows:

   ```
   Dump of assembler code for function main:
      0x00000000004004ed <+0>:  mov    $0x601038,%rbx
      0x00000000004004f4 <+7>:  mov    (%rbx),%rax
      0x00000000004004f7 <+10>: add    0x8(%rbx),%rax
      0x00000000004004fb <+14>: add    0x10(%rbx),%rax
      0x00000000004004ff <+18>: mov    %rax,0x18(%rbx)
      0x0000000000400503 <+22>: retq
      0x0000000000400504 <+23>: nopw   %cs:0x0(%rax,%rax,1)
      0x000000000040050e <+33>: xchg   %ax,%ax
   End of assembler dump.
   (gdb)
   ```

   NOTES:

   - The 16-hexadecimal-digit number at the left is the actual address in memory where the first byte of the instruction, shown at the right, is stored.
   - The immediate mode value 0x601038 in the first instruction is the actual address in memory assigned to the label `Z` in the `.data` area of the program.
   - The numbers in "angle brackets" define the distance in byte locations from the label `main`. Because this label immediately precedes the first instruction, these values define how many bytes each instruction is from the first byte of the first instruction.
   - The machine language instructions are not shown. The gcc command objdump, described in an earlier lab is needed for this.
   - The original instruction, `ret` has been replaced by `retq` by the disassembler. You may ignore any instructions that follow `retq` as they are introduced by the assembler to facilitate termination of execution.

6. The information provided in the disassembled program can be used to execute the program in a stepwise manner. To do this, breakpoints must be established using the location of each instruction relative to the address, `main`, as shown in the disassembled program in the previous step. A message indicating the establishment of the breakpoint will be printed between each command line. Enter the following `break` commands, pausing each time for the acknowlegment message:

   ```
   (gdb) break *main+7
   (gdb) break *main+10
   (gdb) break *main+14
   (gdb) break *main+18
   (gdb) break *main+22
   ```

   In this example a breakpoint has been established prior to the execution of each instruction following the first instruction of the program. So any information printed when the program pauses at a breakpoint will convey information about the contents of memory or a register prior to the instruction where the breakpoint occurs. The next steps illustrate this.

   NOTE: If you wish to pause execution after each instruction you can do so without setting breakpoints on each instruction with the gdb command "`next`". This will execute the next instruction in your program and then pause.

7. Initiate execution of your program with a `run` command. The CPU will execute all instructions until it reaches the first breakpoint:

```
(gdb) run
Starting program: /home/dixon/sfuhome/CMPT295/runtestq

Breakpoint 1, main () at datatestq.s:9
9 mov 0(%rbx), %rax
(gdb)
```

   The instruction shown will be the next one to be executed (but not yet!), as explained above.

8. To check the contents of any register during any breakpoint pause, the `print/x` instruction can be used. The following commands display the contents of registers `rax` and `rbx`:

```
(gdb) print/x $rax
$1 = 0x4004ed
(gdb) print/x $rbx
$2 = 0x601038
(gdb)
```

   A register is specified by its name, without the "%" symbol. Instead the name is prefixed with a "$" symbol. Since the first program instruction loaded the address, `Z`, into register `rbx`, the content of `rbx` is the address `0x601038`, as shown in the disassembled program above. Further, since nothing has been placed in register `rax`, the value is not yet defined. However, there is always some value in a register, and in this case, it is the meaningless value `0x4004ed`. Finally, both addresses and the contents of registers in this example are 64 bit values, so all the hexadecimal numbers shown in the output represent 64-bit values.

9. To display the contents of the `.data` area where the values 1, 2, and 3 are stored, enter the following command:

```
(gdb) x/24xb &Z
```

   NOTES:

   (a) The expression "`x/24xb`" specifies that the output should be displayed in hexadecimal bytes (as indicated by the character `b` in the command) and that 24 bytes should be printed. The expression "`&Z`" specifies the address label of the first byte of the 24 to be displayed.

   The output should be as follows:

```
0x601038: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x601040: 0x02 0x00 0x00 0x00 0x06 0x00 0x00 0x00
0x601048: 0x03 0x00 0x00 0x00 0x00 0x00 0x00 0x00
(gdb)
```

   Recall that x86-64 external memory is a $2^{64} \times 8$ memory. In the first line of output, the content of address 0x601038 is the 8-bit value immediately to its right. The rest of the line indicates the values stored in the next 7 bytes that make up a 64-bit value. The next line, beginning with address 0x601040, defines the contents of the next 8 bytes of memory that specify the second value in memory, "2". Similarly the third line of output defines the third input value, "3".

   (b) The starting address of the first 8 bytes, 0x601038, matches the address value in register `rbx` as printed previously.

   (c) Each 8-byte value is stored in <u>in little endian order</u>; that is, with the least significant byte first.

(d) The values to be summed are each stored in 8 bytes of storage because of the pseudo-op, **quad**, used to declare the three values 1,2, and 3 in the `.data` area of the program.

(e) Each value is stored beginning with an address that is divisible by 8. Recall that all 8-byte values must be aligned on addresses that are divisible by 8.

10. As can be seen, because the values are displayed in little endian order, it is easy to misread them. Gdb provides formats for outputting the contents of memory in a more readable order if the size of data is known. Since the data are stored as "quad" values, rather than printing out 24 bytes, they can be printed out in hexadecimal as three 8-byte values (as indicated by the chararacter `g` in the command). Enter the following:

    ```
    (gdb) x/3xg &Z
    ```

    The resulting output is:

    ```
    0x601038: 0x0000000000000001 0x0000000000000002
    0x601048: 0x0000000000000003
    (gdb)
    ```

    Each line of output specifies two 64-bit values. The address of the leftmost value is shown at the left (i.e., `0x601038`). The rightmost address is 8 bytes after the left value; that is, beginning at address `0x601040`.

11. To continue execution of the program to the next breakpoint, type:

    ```
    (gdb) continue
    ```

    NOTE: Typing `run` will restart the program from the beginning.

12. Display the contents of register `rax` again, as you did previously. The value displayed should now be a valid value, namely "1" because the execution of the the instruction on the line displayed at the previous breakpoint has now been executed. This instruction moved the 64-bit value starting at address `0x601038` to register `rax`.

13. Repeat the previous two steps until the fifth breakpoint is reached. Each time, observe the value accumulating in register `rax`.

14. Breakpoint 5 occurs prior to the execution of the `retq` instruction of the disassembled program. The previous instruction that was executed stored the 64-bit sum at the address specified by the operand `24(%rbx)`. This is a "base+displacement" operand and therefore the effective address is 24_{10} `+ 0x601038 = 0x601050`. Thus, the value of the sum is stored in 64-bits beginning at location `0x601050`.

15. To confirm the result, display the contents of the value stored beginning at `ans` as follows:

    ```
    (gdb)  x/xg &ans
    ```

    The following output should be observed:

    ```
    0x601050 <completed.6973>: 0x0000000000000006
    (gdb)
    ```

An Example Using 4-Byte Operands (32 bits)

1. The previous example operated on 64-bit values, producing a 64-bit result. Since the magnitude of the numbers used was not large, the program can be written to use 32-bit operands rather than 64-bit operands. The following program sums the three numbers using 32-bit memory values and 32 bit registers:

```
        .data
Z:      .long 1,2,3
ans:    .long

        .text
        .global main
main:
        mov $Z, %rbx
        mov 0(%rbx), %eax
        add 4(%rbx), %eax
        add 8(%rbx), %eax
        mov %eax,12(%rbx)
        ret
```

   NOTES:

   - The register accumulating the sum is now the 32-bit register, `eax`.
   - The base+displacement register, `rax` is still a 64-bit register because it holds addresses and addresses are always 64 bits.
   - The pseudo-op `.long` is now used to allocate memory for 1, 2, and 3, and `ans`, rather than `.quad`.

   Enter this program into a file called testprogl.s, assemble it, and create an executable file, `runtestl`, following the same steps that were used for `testprogq.s` above.

2. Launch gdb with:

```
gdb runtestl
```

3. Disassemble the program. The result should be:

```
Dump of assembler code for function main:
   0x00000000004004ed <+0>:  mov    $0x601038,%rbx
   0x00000000004004f4 <+7>:  mov    (%rbx),%eax
   0x00000000004004f6 <+9>:  add    0x4(%rbx),%eax
   0x00000000004004f9 <+12>: add    0x8(%rbx),%eax
   0x00000000004004fc <+15>: mov    %eax,0xc(%rbx)
   0x00000000004004ff <+18>: retq
End of assembler dump.
(gdb)
```

4. Set one breakpoint on the `retq` statement. Then run the program.

5. When the program pauses at the breakpoint, 32 bytes of the `.data` area can be displayed with

```
(gdb) x/4xw &Z
```

   Your output should be as follows:

```
0x601038: 0x00000001 0x00000002 0x00000003 0x00000006
(gdb)
```

NOTES:

- The command `x/4xw` specifies that four 32-bit values are to be printed in hexadecimal in groups of 32 bits (`w`).
- By using the pseudo-op `.long`, each value is now stored in a 4 bytes of memory.
- Since the location `ans` immediately follows the three 4 byte locations allocated for the values 1, 2, and 3, by printing out four 32-bit words beginning with the address `Z`, the value in `ans` is also shown since the address `ans` is the same as the address Z + 0xc = 0x0601038 + 0xc; that is 0x0601044.