Chapter 5 Registers

Computer memory is essentially an array of bytes which software uses for instructions and data. While the memory is relatively fast, there is a need for a small amount of faster data to permit the CPU to execute instructions faster. A typical computer executes at 3 GHz and many instructions can execute in 1 cycle. However for an instruction to execute the instruction and any data required must be fetched from memory. One fairly common form of memory has a latency of 6 nanoseconds, meaning the time lag between requesting the memory and getting the data. This 6 nanoseconds would equal 18 CPU cycles. If the instructions and data were all fetched from and stored in memory there would probably be about 18 nanoseconds required for common instructions. 18 nanoseconds is time enough for 54 instructions at 1 instruction per cycle. There is clearly a huge need to avoid using the relatively slow main memory.

One type of faster memory is cache memory, which is perhaps 10 times as fast as main memory. The use of cache memory can help address the problem, but it is not enough to reach the target of 1 instruction per CPU cycle. A second type of faster memory is the CPU's registers. Cache might be several megabytes, but the CPU has only a few registers. However the registers are accessible in roughly one half of a CPU cycle or less. The use of registers is essential to achieving high performance. The combination of cache and registers provides roughly half a modern CPU's performance. The rest is achieved with pipelining and multiple execution units. Pipelining means dividing instructions into multiple steps and executing several instructions simultaneously though each at different steps. Pipelining and multiple execution units are quite important but these features are not part of general assembly language programming, while registers are a central feature.

The x86-64 CPUs have 16 general purpose 64 bit registers and 16 modern floating point registers. These floating point registers are either 128 or 256 bits depending on the CPU model and can operate on multiple integer or floating point values. There is also a floating point register stack which we will not use in this book. The CPU has a 64 bit instruction pointer register (`rip`) which contains the address of the next instruction to execute. There is also a 64 bit flags register (`rflags`). There are additional registers which we probably won't use. Having 16 registers means that a register's "address" is only 4 bits. This makes instructions using registers much smaller than instructions using only memory addresses.

The 16 general purpose registers are 64 bit values stored within the CPU. Software can access the registers as 64 bit values, 32 bit values, 16 bit values and 8 bit values. Since the CPU evolved from the 8086 CPU, the registers have evolved from 16 bit registers to 32 bit registers and finally to 64 bit registers.

On the 8086 registers were more special purpose than general purpose:

`ax` - accumulator for numeric operations

`bx` - base register (array access)

`cx` - count register (string operations)

`dx` - data register item si - source index

`di` - destination index

`bp` - base pointer (for function frames)

`sp` - stack pointer

In addition the 2 halves of the first 4 registers can be accessed using `al` for the low byte of `ax`, `ah` for the high byte of `ax`, and `bl`, `bh`, `cl`, `ch`, `dl` and `dh` for the halves of `bx`, `cx` and `dx`.

When the 80386 CPU was designed the registers were expanded to 32 bits and renamed as `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`, and `esp`. Software could also use the original names to access the lower 16 bits of each of the registers. The 8 bit registers were also retained without allowing direct access to the upper halves of the registers.

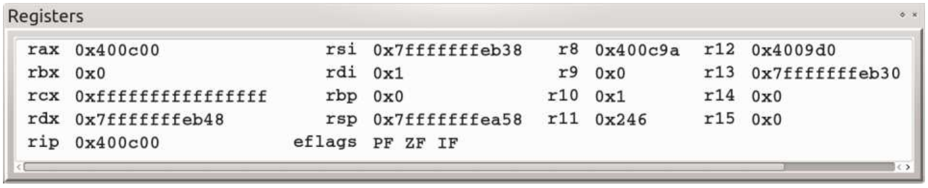
For the x86-64 architecture the registers were expanded to 64 bits and 8 additional general purpose registers were added. The names used to access the 64 bit registers are rax, rbx, rcx, rdx, rsi, rdi, rbp, and rsp for the compatible collection and `r8-r15` for the 8 new registers. As you might expect you can still use `ax` to access the lowest word of the `rax` register along with `eax` to access the lower half of the register. Likewise the other 32 bit and 16 bit register names still work in 64 bit more. You can also access registers `r8-r15` as byte, word, or double word registers by appending `b`, `w` or `d` to the register name.

The `rflags` register is a 64 bit register, but currently only the lower 32 bits are used, so it is generally sufficient to refer to `eflags`. In addition the flags register is usually not referred to directly. Instead conditional instructions are used which internally access 1 or more bits of the flags register to determine what action to take.

Moving data seems to be a fundamental task in assembly language. In the case of moving values to/from the integer registers, the basic command is `mov`. It can move constants, addresses and memory contents into registers, move data from 1 register to another and move the contents of a register into memory.

5.1 Observing registers in ebe

One of the windows managed by ebe is the register window. After each step of program execution ebe obtains the current values of the general purpose registers and displays them in the register window. Similarly ebe displays the floating point registers in the floating point register window. Below is a sample of the register window.



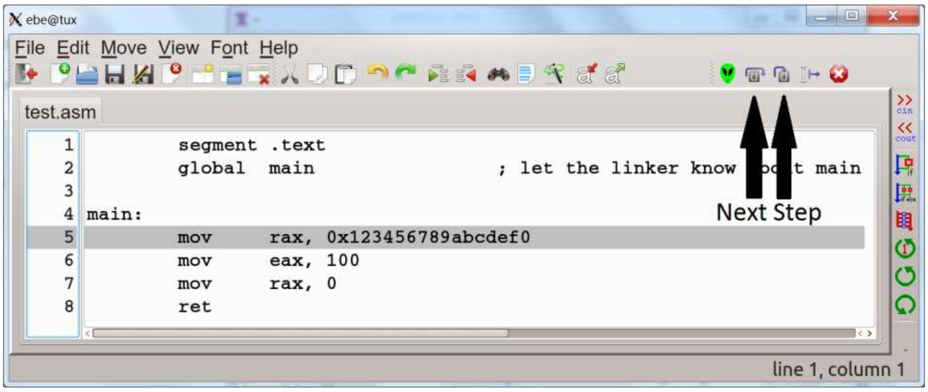
You can select a different format for the registers by right clicking on the name of a register. This will popup a list of choices. You can choose either decimal or hexadecimal format for that register or for all the general purpose registers. You can see below the general purpose registers, the instruction pointer register (`rip`) and the flags register (`eflags`). For simplicity the set bits of `eflags` are displayed by their acronyms. Here the parity flag (`PF`), the zero flag (`ZF`) and the interrupt enable flag (`IF`) are all set.

5.2 Moving a constant into a register

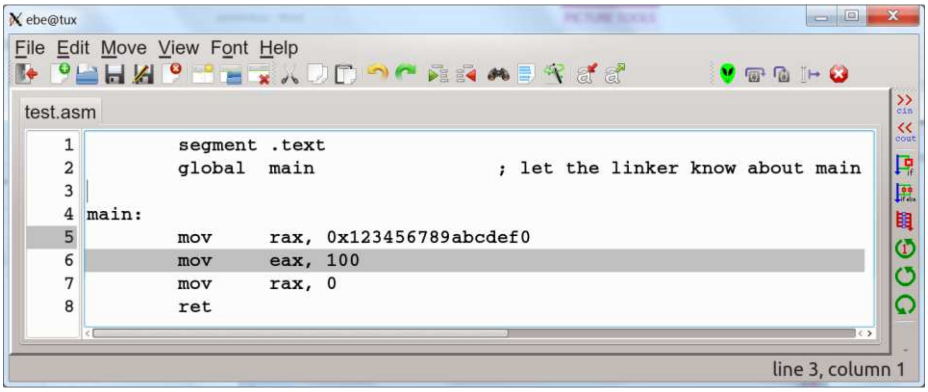
The first type of move is to move a constant into a register. A constant is usually referred to as an immediate value It consists of some bytes stored as part of the instruction. Immediate operands can be 1, 2 or 4 bytes for most instructions. The `mov` instruction also allows 8 byte immediate values.

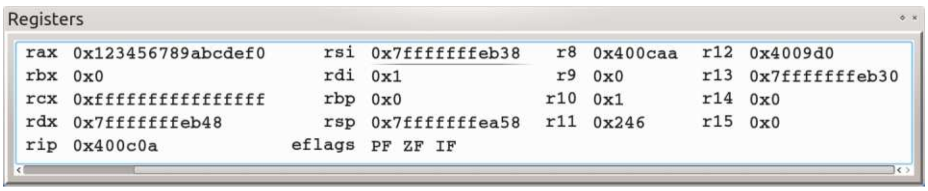
```
        mov     rax, 100
        mov     eax, 100
```

Surprisingly, these two instructions have the same effect - moving the value 100 into `rax`. Arithmetic operations and moves with 4 byte register references are zero-extended to 8 bytes. The program shown below in ebe illustrates the `mov` instruction moving constants into register `rax`.

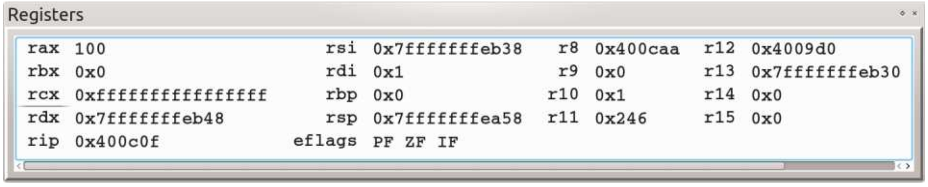


There has been a breakpoint set on line 5 and the program has been run by clicking the “Run” button. At this point the first `mov` has not been executed. You can advance the program by clicking on either “Next” or “Step” (highlighted with arrows in the picture). The difference is that “Step” will step into a function if a function call is made, while “Next” will execute the highlighted statement and advance to the next statement in the same function. The effect is the same in this code and here is the source window and the register window after executing the first `mov`:





You can observe that the value `0x123456789abcdef0` has been placed into `rax` and that clearly the next `mov` has not been executed. There is little value in repeatedly displaying the source window but here is the register window after executing the `mov` at line 6:



For convenience the display format for `rax` has been switched to decimal and you can observe that "`mov eax, 100`" results in moving 100 into the lower half of `rax` and 0 into the upper half.

The same operations can be done directly in gdb. Below is a gdb session illustrating moving constants. All the user inputs are in bold face to distinguish them from the text printed by gdb. Note also that gdb issues "`(gdb)`" for its prompt.

```
(gdb) list 5,7
 5          mov     rax, 0x123456789abcdef0
 6          mov     eax, 100
 7          mov     rax, 0
(gdb) break 5
Breakpoint 1 at 0x400508: file test.asm,line 5
(gdb) run
Starting program: /home/seyfarth/asm/test
Breakpoint 1, main () at test.asm:5
 5              mov     rax, 0x123456789abcdef0
(gdb) nexti
 6              mov     eax, 100
(gdb) print/x $rax
$1 = 0x123456789abcdef0
(gdb) nexti
 7              mov     rax, 0
(gdb) print/x $rax
$2 = 0x64
```

You can see that the gdb prompt is `(gdb)`. The first command entered is "`list 5,7`". This command lists line 5 through 7 of the source file. You can abbreviate "`list`" as "`l`".

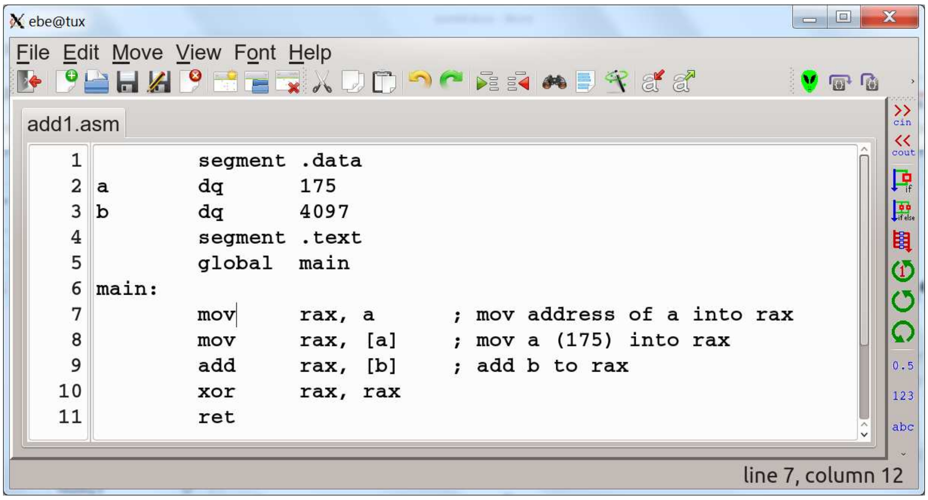
The next command is "`break 5`", which sets a break point at line 5. "`break`" can be abbreviated as "`b`". A break point is a statement which will not be executed when the program in executed. Instead the control will be passed back to the debugger. After issuing the "`run`" command the debugger starts running the program, processing instructions until it reaches line 5. It breaks there without executing that instruction.

The next command is "`nexti`" which means execute the next instruction and return to the debugger. "`nexti`" can be abbreviated as "`ni`". After executing that `mov`, the value of register `rax` is printed in hexadecimal. "`print`" can be abbreviated as "`p`". The purpose of loading the large value is to place non-zero bits in the top half of `rax`.

You can follow the sequence of statements and observe that moving 100 into `eax` will clear out the top half of `rax`. It turns out that a 32 bit constant is stored in the instruction stream for the `mov` which moves 100. Also the instruction to move into `eax` is 1 byte long and the move into `rax` is 3 bytes long. The shorter instruction is preferable. You might be tempted to move 100 into `al`, but this instruction does not clear out the rest of the register.

5.3 Moving values from memory to registers

In order to move a value from memory into a register, you must use the address of the value. Consider the program shown below



The label `a` is will be replaced by the address of `a` if included in an instruction under Linux. OS X uses relative addressing and `a` will be replaced by its address relative to register `rip`. The reason is that OS X addresses are too big to fit in 32 bits. In fact yasm will not allow moving an address under OS X. The alternative is to use the `lea` (load effective address) instruction which will be discussed later. Consider the following statement in the `.text` section.

```
        mov     rax, a
```

The instruction has a 32 bit constant field which is replaced with the address of `a` when the program is executed on Linux. When tested, the `rax` register receives the value `0x602088` as shown below:

Registers

rax	0x602088	rsi	0x7fffffffeb38	r8	0x400caa
rbx	0x0	rdi	0x1	r9	0x0
rcx	0xffffffffffffffff	rbp	0x0	r10	0x1
rdx	0x7fffffffeb48	rsp	0x7fffffffea58	r11	0x246
rip	0x400c07	eflags	PF ZF IF		

The proper syntax to get the value of `a`, 175, is from line 8 of the program and also below:

```
        mov     rax, [a]
```

The meaning of an expression in square brackets is to use that expression as a memory address and to load or store from that address. In this case it loads the value from the address represented by `a`. This is basically a different instruction from the other `mov`. The other is “load constant” and the latest one is “load from memory”.

After executing line 8 we see that `rax` has the value 175. In the register display below I have used a decimal format to make the effect more obvious.

Registers

rax	175	rsi	0x7fffffffeb38	r8	0x400caa
rbx	0x0	rdi	0x1	r9	0x0
rcx	0xffffffffffffffff	rbp	0x0	r10	0x1
rdx	0x7fffffffeb48	rsp	0x7fffffffea58	r11	0x246
rip	0x400c0f	eflags	PF ZF IF		

In line 9 of the program I have introduced the `add` instruction to make things a bit more interesting. The effect of line 9 is to add the contents of `b`, 4097, to `rax`. The result of the `add` instruction is shown below:

Registers

rax	4272	rsi	0x7fffffffeb38	r8	0x400caa
rbx	0x0	rdi	0x1	r9	0x0
rcx	0xffffffffffffffff	rbp	0x0	r10	0x1
rdx	0x7fffffffeb48	rsp	0x7fffffffea58	r11	0x246
rip	0x400c17	eflags	AF IF		

You will notice that my main routine calls no other function. Therefore there is no need to establish a stack frame and no need to force the stack pointer to be a multiple of 16.

Below is the result of running this program in gdb:

```
(gdb) b 7
Breakpoint 1 at 0x4004c0: file add1.asm, line 7.
(gdb) r
```

```
 Starting program: /home/seyfarth/asm/add1
Breakpoint 1, main () at add1.asm:7
 7               mov     rax, a      ; mov address of a to rax
 (gdb) n
 8               mov     rax, [a]    ; mov a (175) into rax
 (gdb) p/x $rax
 $1 = 0x601018
 (gdb) n
 9               add     rax, [b]    ; add b to rax
 (gdb) p $rax
 $2 = 175
 (gdb) n
 10              xor     rax, rax
 (gdb) p $rax
 $3 = 4272
 (gdb) p a+b
 $4 = 4272
```

It can be slightly confusing running gdb since it always prints the line which will be the one next to execute. For example right after it printed line 8, I printed rax which contained the value placed in it from the last executed instruction (from line 7). Ebe is more obvious if you remember that the highlighted line is the next one to execute.

We see that the correct sum is placed in rax by the add instruction. We also see that gdb knows about the labels in the code. It can print a and b, and can even compute their sum. Unfortunately the code produced by yasm does not inform gdb of the data types, so gdb assumes that the variables are double word integers. Still, this ability to print arithmetic expressions can be quite convenient.

There are other ways to move data from memory into a register, but this is sufficient for simpler programs. The other methods involve storing addresses in registers and using registers to hold indexes or offsets in arrays.

You can also move integer values less than 8 bytes in size into a register. If you specify an 8 bit register such as al or a 16 bit register such as ax, the remaining bits of the register are unaffected. However it you specify a 32 bit register such as eax, the remaining bits are set to 0. This may or may not be what you wish.

Alternatively you can use move and sign extend (movsx) or move and zero extend (movzx) to control the process. In these cases you would use the 64 bit register as a destination and add a length qualifier to the instruction. There is one surprise - a separate instruction to move and sign extend a double word: movsxd. Here are some examples:

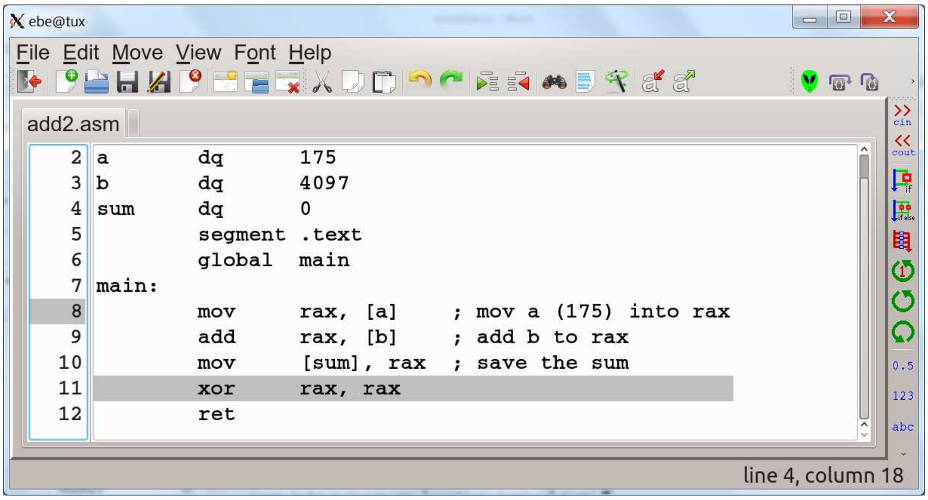
```
        movsx  rax, byte [data]  ; move byte, sign extend
        movzx  rbx, word [sum]   ; move word, zero extend
        movsxd rcx, dword [count]; move dword, sign extend
```

5.4 Moving values from a register to memory

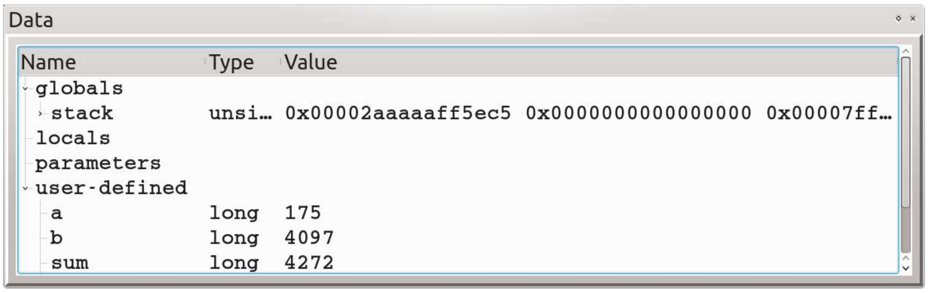
Moving data from a register to memory is very similar to moving from memory to a register - you simply swap the operands so that the memory address is on the left (destination).

```
        mov     [sum], rax
```

Below is a program which adds 2 numbers from memory and stores the sum into a memory location named `sum`:



The source window shows line 11 highlighted which means that the `mov` instruction saving the sum has been executed. You can see that there is a breakpoint on line 8 and clearly the “Run” button was used to start the program and “Next” was clicked 3 times. Below is the data for the program:

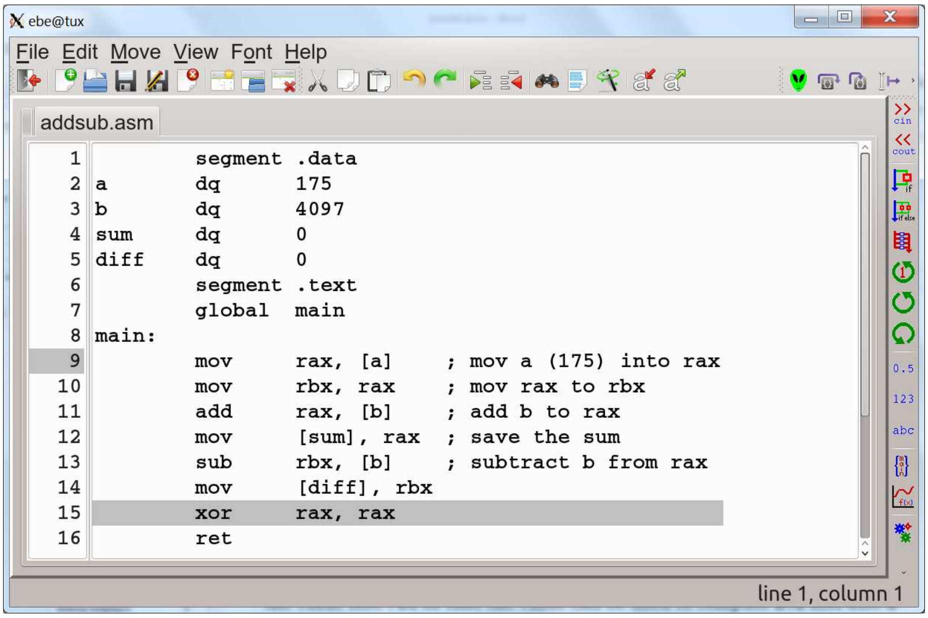


5.5 Moving data from one register to another

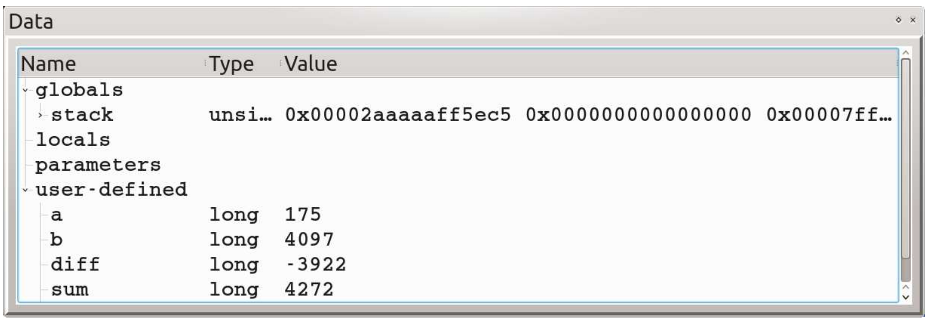
Moving data from one register to another is done as you might expect - simply place 2 register names as operands to the `mov` instruction.

```
        mov     rbx, rax    ; move value in rax to rbx
```

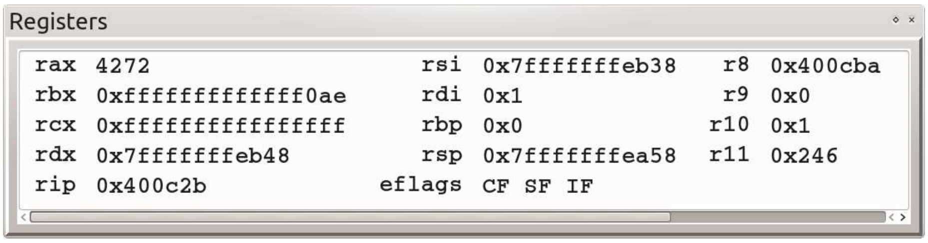
Below is a program which moves the value of `a` into `rax` and then moves the value into `rbx` so that the value can be used to compute `a+b` and also `a-b`.



You can see that there is a breakpoint on line 8 and that line 15 is the next to be executed. This program introduces the `sub` instruction which subtracts one value from another. In this case it subtracts the value from memory location `b` from `rbx` with the difference being placed in `rbx`.



It might be a little interesting to note the value of `eflags` shown in the registers for the addition and subtraction program. You will see `SF` in the flag values which stands for “sign flag” and indicates that the last instruction which modified the flags, `sub`, resulted in a negative value.



Exercises

1. Write an assembly program to define 4 integers in the `.data` section. Give two of these integers positive values and 2 negative values. Define one of your positive numbers using hexadecimal notation. Write instructions to load the 4 integers into 4 different registers and add them with the sum being left in a register. Use gdb or ebe to single-step through your program and inspect each register as it is modified.

2. Write an assembly program to define 4 integers - one each of length 1, 2, 4 and 8 bytes. Load the 4 integers into 4 registers using sign extension for the shorter values. Add the values and store the sum in a memory location.

3. Write an assembly program to define 3 integers of 2 bytes each. Name these `a`, `b` and `c`. Compute and save into 4 memory locations `a+b`, `a-b`, `a+c` and `a-c`.

Chapter 6
A little bit of math

So far the only mathematical operations we have discussed are integer addition and subtraction. With negation, addition, subtraction, multiplication and division it is possible to write some interesting programs. For now we will stick with integer arithmetic.

6.1 Negation

The `neg` instruction performs the two's complement of its operand, which can be either a general purpose register or a memory reference. You can precede a memory reference with a size specifier from the following table:

Specifier	Size in bytes
byte	1
word	2
dword	4
qword	8

The `neg` instruction sets the sign flag (`SF`) if the result is negative and the zero flag (`ZF`) if the result is 0, so it is possible to do conditional operations afterwards.

The following code snippet illustrates a few variations of `neg`:

```
neg   rax         ; negate the value in rax
neg   dword [x] ; negate 4 byte int at x
neg   byte [x]  ; negate byte at x
```

6.2 Addition

Integer addition is performed using the add instruction. This instruction has 2 operands: a destination and a source. As is typical for the x86-64 instructions, the destination operand is first and the source operand is second. It adds the contents of the source and the destination and stores the result in the destination.

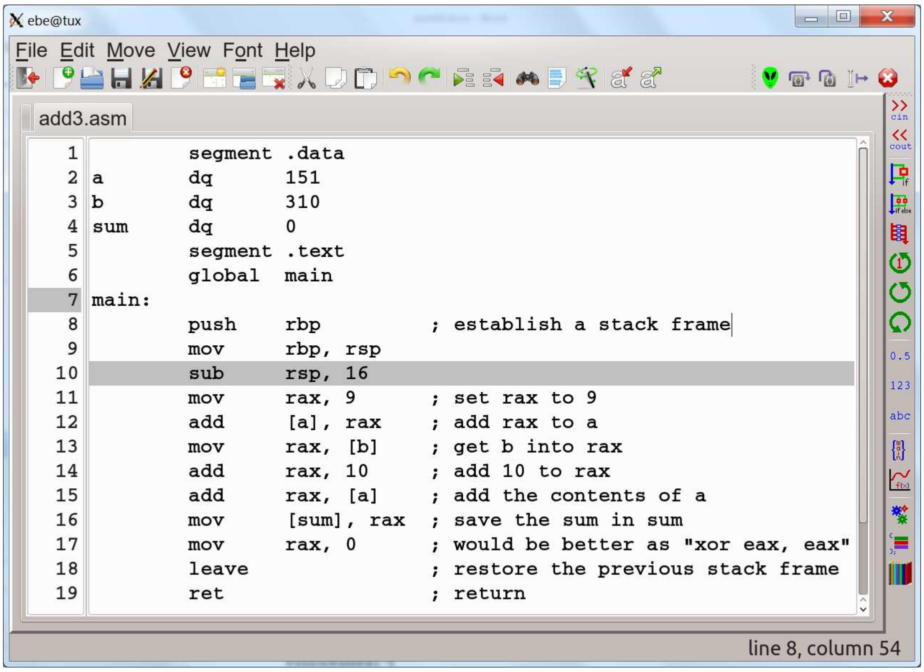
The source operand can be an immediate value (constant) of 32 bits, a memory reference or a register. The destination can be either a memory reference or a register. Only one of the operands may be a memory reference. This restriction to at most one memory operand is another typical pattern for the x86-64 instruction set.

The `add` instruction sets or clears several flags in the `rflags` register based on the results of the operation. These flags can be used in conditional statements following the `add`. The overflow flag (`OF`) is set if the addition overflows. The sign flag (`SF`) is set to the sign bit of the result. The zero flag (`ZF`) is set if the result is 0. Some other flags are set related to performing binary-coded-decimal arithmetic.

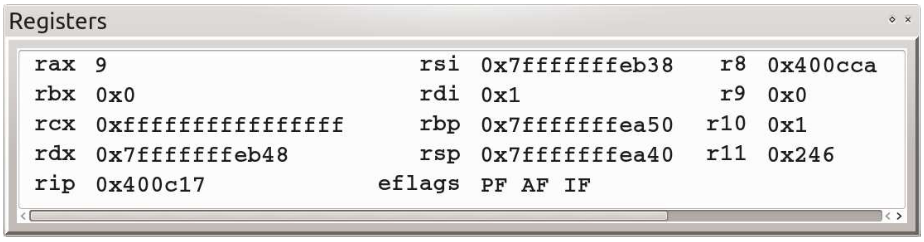
There is no special `add` for signed numbers versus unsigned numbers since the operations are the same. The same is true for subtraction, though there are special signed and unsigned instructions for division and multiplication.

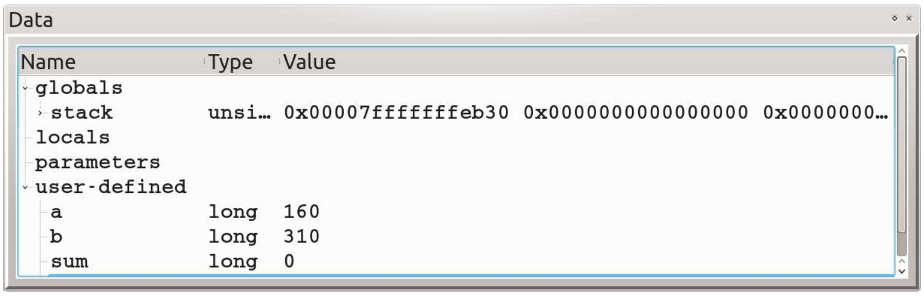
There is a special increment instruction (`inc`), which can be used to add 1 to either a register or a memory location.

Below is a sample program with some `add` instructions. You can see that there is a breakpoint on line 8. It is a little surprising that when you click on the "Run" button, the next line to execute is line 10. The reason for this is that lines 8 and 9 establish a "stack frame" which is used in gdb to keep track of which function is being executed. Gdb doesn't want to break in the middle of the instructions which establish the stack frame for a function, since commands related to local variables would be incorrect until the frame is complete. Line 10 is a fairly common operation to prepare some space for local variables on the stack. These 3 instructions are so common that there is a `leave` instruction which can undo the effect of them to prepare for returning from a function.



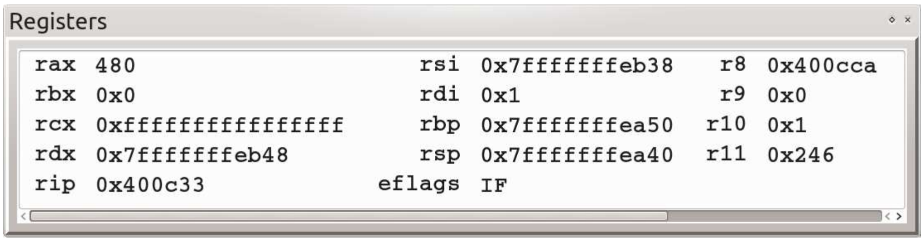
Next we see the registers and data for the program after executing lines 10 through 12.

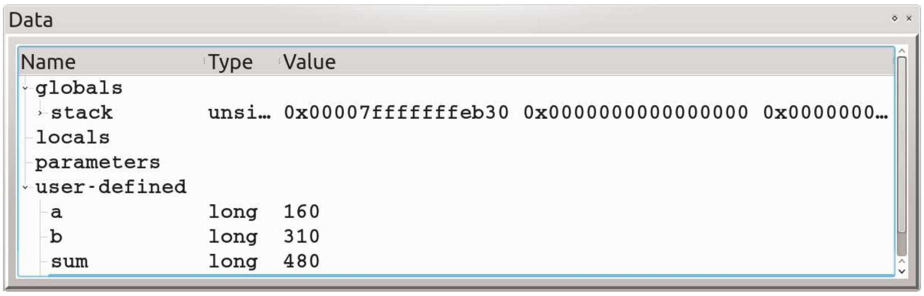




You can see that the sum computed on line 12 has been stored in memory in location `a`.

Below we see the registers and data after executing lines 13 through 16. This starts by moving `b` (310) into `rax`. Then it adds 10 to `rax` to get 320. After adding `a` (160) we get 480 which is stored in `sum`.





Below is a gdb session illustrating this program.

```
(gdb) b 11
Breakpoint 1 at 0x4004c8: file add3.asm,line 11
(gdb) run
Starting program: /home/seyfarth/asm/add3

Breakpoint 1, main ( at add3.asm:11
11        mov     rax, 9     ; set rax to 9
(gdb) ni
12        add     [a], rax   ; add rax to a
(gdb) p $rax
$1 = 9
(gdb) ni
13        mov     rax, [b]   ; get b into rax
(gdb) p a
$2 = 160
(gdb) ni
14        add     rax, 10    ; add 10 to rax
(gdb) p $rax
$3 = 310
(gdb) ni
15        add     rax, [a]   ; add contents of a
(gdb) p $rax
$4 = 320
(gdb) ni
16        mov     [sum], rax ; save sum in sum
(gdb) p $rax
$5 = 480
(gdb) ni
17            mov     rax, 0
(gdb) p sum
$6 = 480
```

6.3 Subtraction

Integer subtraction is performed using the `sub` instruction. This instruction has 2 operands: a destination and a source. It subtracts the contents of the source from the destination and stores the result in the destination.

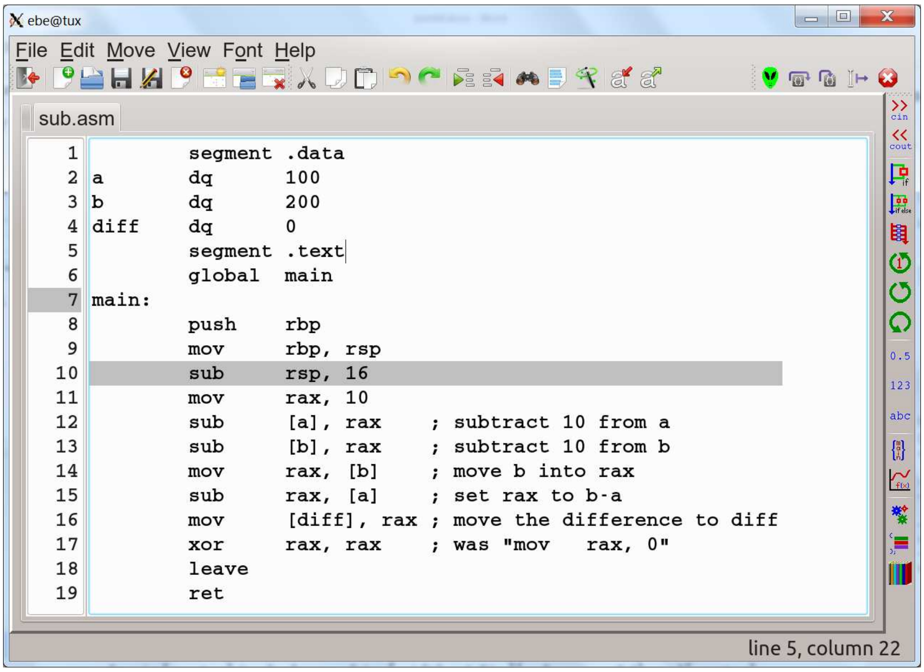
The operand choices follow the same pattern as `add`. The source operand can be an immediate value (constant) of 32 bits, a memory reference or a register. The destination can be either a memory reference or a register. Only one of the operands can be a memory reference.

The `sub` instruction sets or clears the overflow flag (`OF`), the sign flag (`SF`), and the zero flag (`ZF`) like `add`. Some other flags are set related to performing binary-coded-decimal arithmetic.

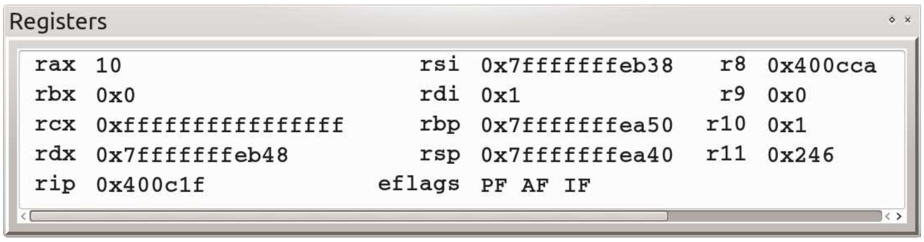
As with addition there is no special subtract for signed numbers versus unsigned numbers.

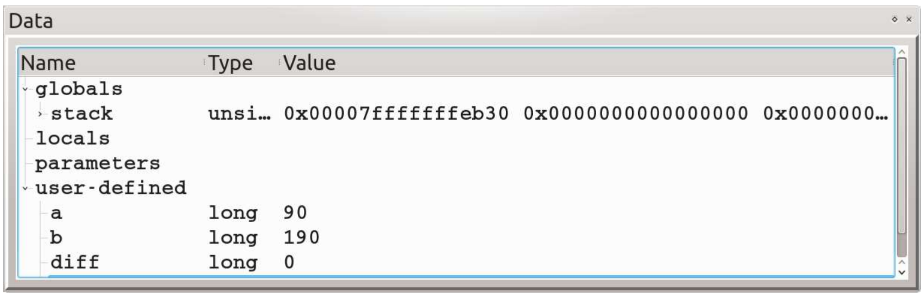
There is a decrement instruction (`dec`) which can be used to decrement either a register or a value in memory.

Below is a program with some `sub` instructions. You can see that the program has a breakpoint on line 8 and that gdb has stopped execution just after establishing the stack frame. Near the end this program uses “`xor rax, rax`” as an alternative method for setting `rax` (the return value for the function) to 0. This instruction is a 3 byte instruction. The same result can be obtained using “`xor eax, eax`” using 2 bytes which can reduce memory using. Both alternatives will execute in 1 cycle, but using fewer bytes may be faster due to using fewer bytes of instruction cache.

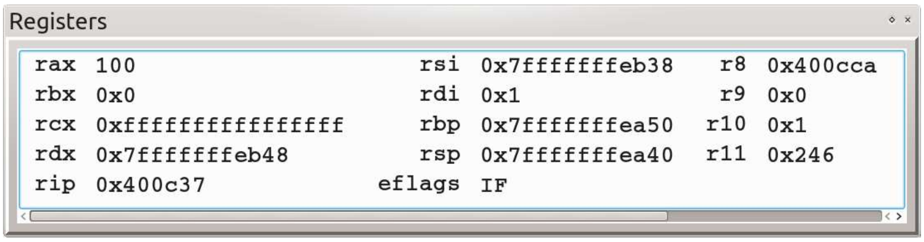


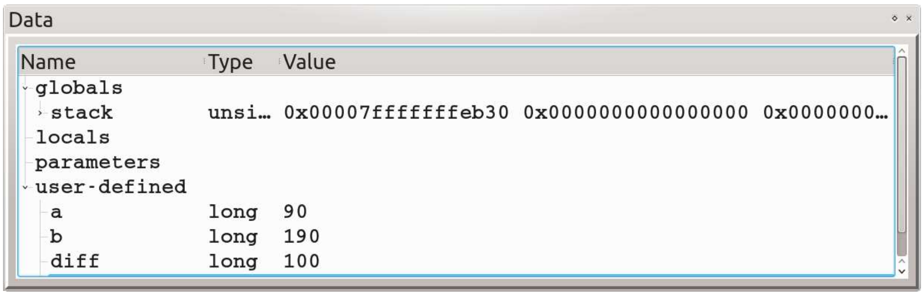
The next two figures show the registers and data for the program after executing lines 11 through 13 which subtract 10 from memory locations a and b.





Next we see the results of executing lines 14 through 16, which stores `b-a` in `diff`.





Here is a gdb session illustrating the same program:

```
(gdb) b 11
Breakpoint 1 at 0x4004c8: file sub.asm, line 11
(gdb) run
Starting program: /home/seyfarth/asm/sub

Breakpoint 1, main ( at sub.asm:11
11        mov    rax, 10
(gdb) ni
12        sub    [a], rax    ; subtract 10 from a
(gdb) p $rax
$1 = 10
(gdb) ni
13        sub    [b], rax    ; subtract 10 from b
(gdb) p a
$2 = 90
(gdb) ni
14        mov    rax, [b]    ; move b into rax
(gdb) p b
$3 = 190
(gdb) ni
15        sub    rax, [a]    ; set rax to b-a
(gdb) p $rax
$4 = 190
(gdb) ni
16        mov    [diff], rax ; move difference to diff
(gdb) p $rax
$5 = 100
(gdb) ni
17        mov     rax, 0
(gdb) p diff
$6 = 100
```

6.4 Multiplication

Multiplication of unsigned integers is performed using the `mul` instruction, while multiplication of signed integers is done using `imul`. The `mul` instruction is fairly simple, but we will skip it in favor of `imul`.

The `imul` instruction, unlike `add` and `sub`, has 3 different forms. One form has 1 operand (the source operand), a second has 2 operands (source and destination) and the third form has 3 operands (destination and 2 source operands).

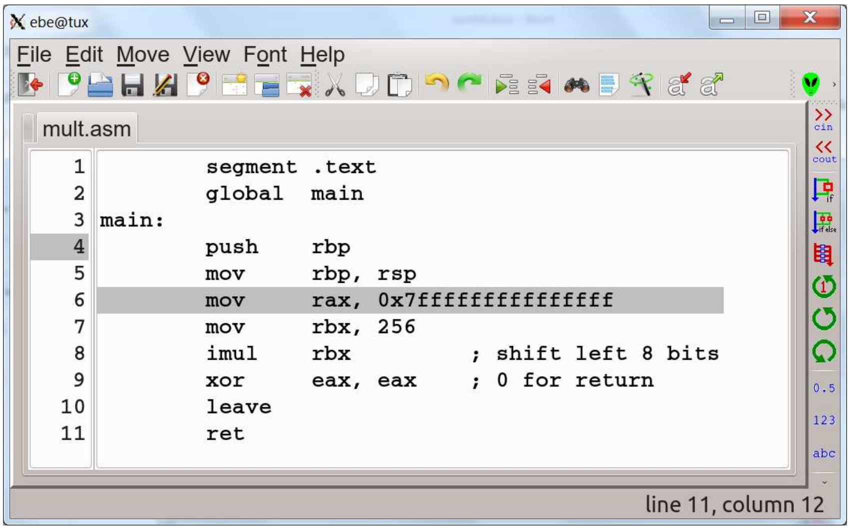
One operand `imul`

The 1 operand version multiples the value in rax by the source operand and stores the result in `rdx:rax`. The source could be a register or a memory reference. The reason for using 2 registers is that multiplying two 64 bit integers yields a 128 bit result. Perhaps you are using large 64 bit integers and need all 128 bits of the product. Then you need this instruction. The low order bits of the answer are in `rax` and the high order bits are in `rdx`.

```
imul   qword [data]; multiply rax by data
mov    [high], rdx ; store top of product
mov    [low], rax  ; store bottom of product
```

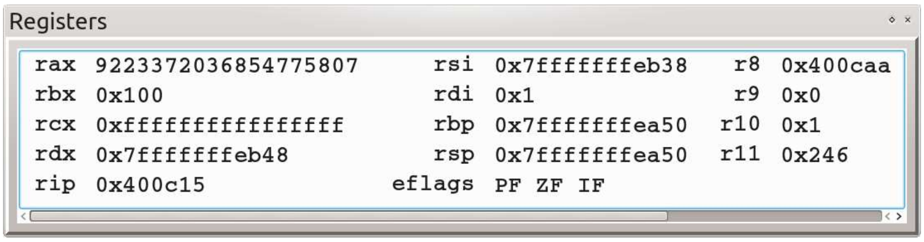
Note that yasm requires the quad-word attribute for the source for the single operand version which uses memory. It issued a warning during testing, but did the correct operation.

Here is a sample program which uses the single operand version of `imul` to illustrate a product which requires both `rax` and `rdx`.

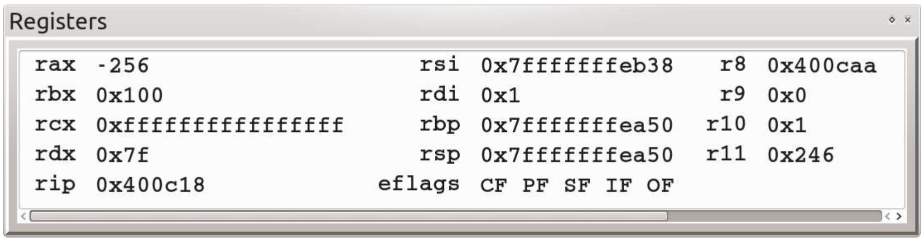


The `mov` in line 6 fills rax with a number composed of 63 bits equal to 1 and a 0 for the sign bit. This is the largest 64 bit signed integer, $2^{63} - 1$. The `imul` instruction in line 8 will multiply this large number by 256. Note that multiplying by a power of 2 is the same as shifting the bits to the left, in this case by 8 bits. This will cause the top 8 bits of `rax` to be placed in `rdx` and 8 zero bits will be introduced in the right of `rax`.

Here are the registers before `imul`:



and then after `imul`:



Two and three operand `imul`

Quite commonly 64 bit products are sufficient and either of the other forms will allow selecting any of the general purpose registers as the destination register.

The two-operand form allows specifying the source operand as a register, a memory reference or an immediate value. The source is multiplied times the destination register and the result is placed in the destination.

```
imul    rax, 100   ; multiply rax by 100
imul    r8, [x]    ; multiply r8 by x
imul    r9, r10    ; multiply r9 by r10
```

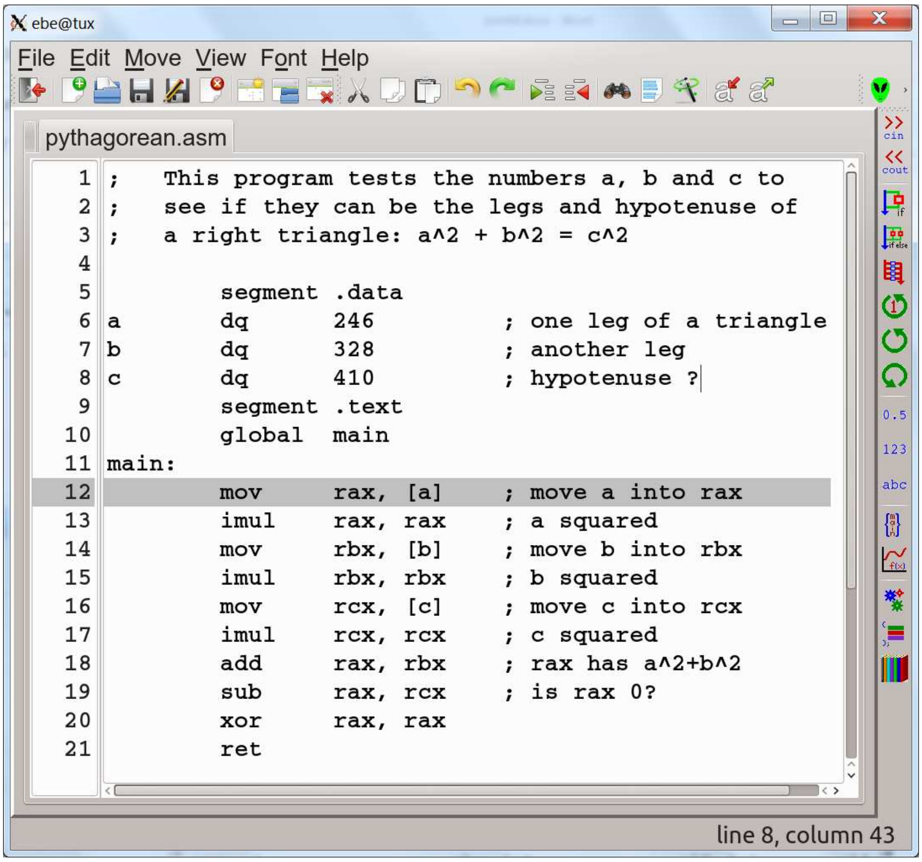
The three-operand form is the only form where the destination register is not one of the factors in the product. Instead the second operand, which is either a register or a memory reference, is multiplied by the third operand which must be an immediate value.

```
imul    rbx, [x], 100 ; store 100*x in rbx
imul    rdx, rbx, 50  ; store 50*rbx in rdx
```

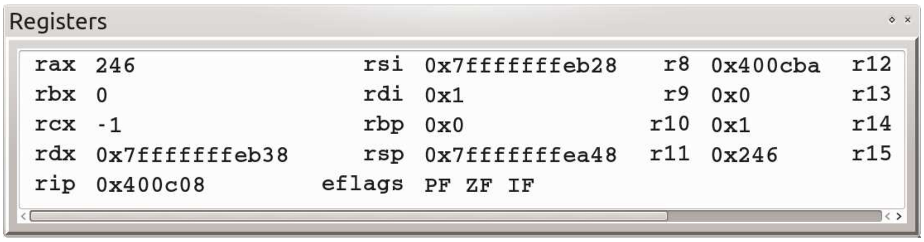
The carry flag (CF) and the overflow flag (OF) are set when the product exceeds 64 bits (unless you explicitly request a smaller multiply). The zero flag and sign flags are undefined, so testing for a zero, positive or negative result requires an additional operation.

Testing for a Pythagorean triple

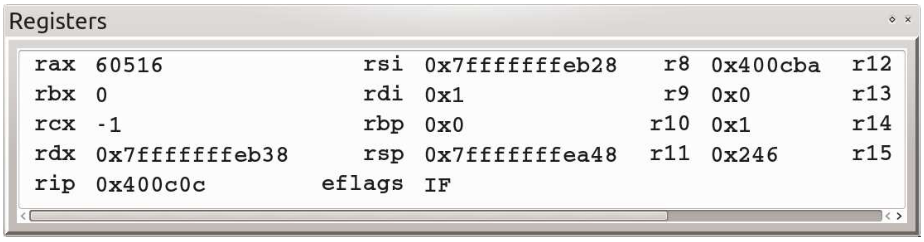
Below is shown a program which uses `imul`, `add` and `sub` to test whether 3 integers, a, b, and c, form a Pythagorean triple. If so, then $a^2 + b^2 = c^2$.



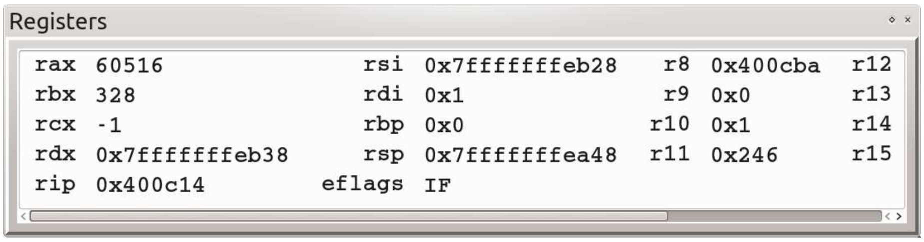
You can see that there is a breakpoint on line 12 and the next line to execute is 12. After clicking on “Next” line 12 will be executed and you can see that the value of `a` is placed in `rax`.



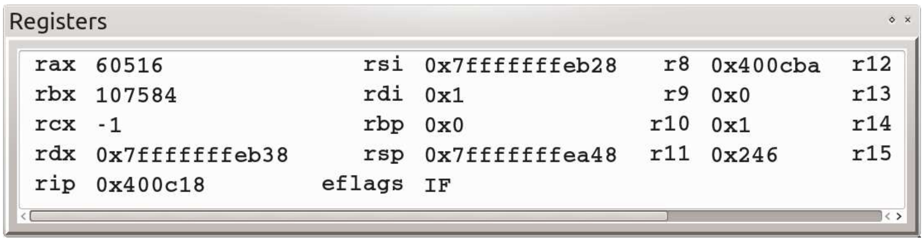
Next rax is multiplied by itself to get a^2 in rax.



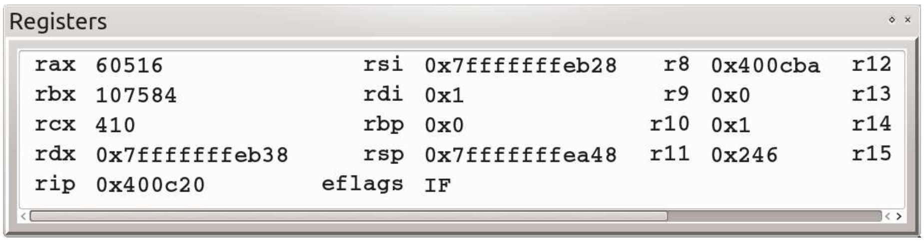
Line 14 moves the value of b into rbx.



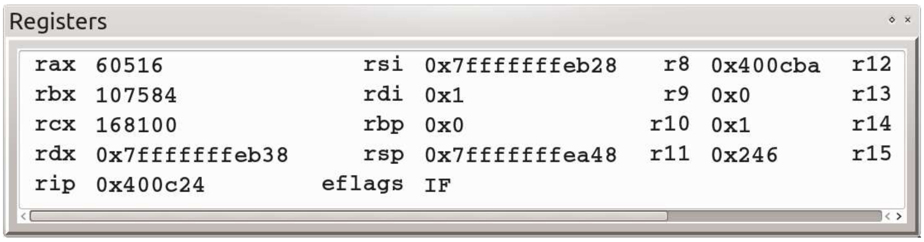
Then rbx is multiplied by itself to get b^2 in rbx.



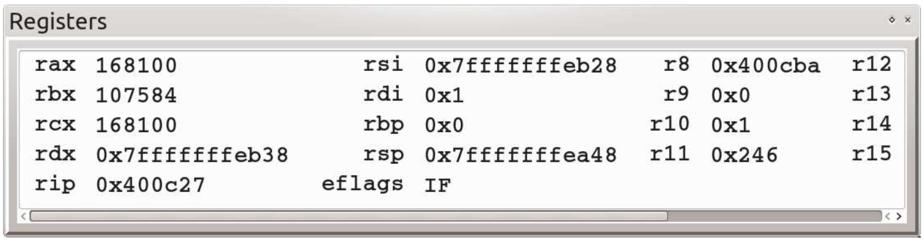
Line 16 moves the value of c into rcx.



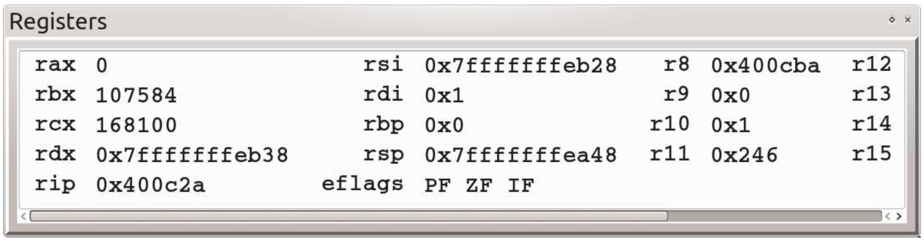
Then rcx is multiplied by itself to get c^2 in rcx.



Line 18 adds `rbx` to `rax` so `rax` holds $a^2 + b^2$.



Finally line 19 subtracts `rcx` from `rax`. After this `rax` holds $a^2 + b^2 - c^2$. If the 3 numbers form a Pythagorean triple then `rax` must be 0. You can see that `rax` is 0 and also that the zero flag (ZF) is set in `eflags`.



If we used a few more instructions we could test to see if ZF were set and print a success message.

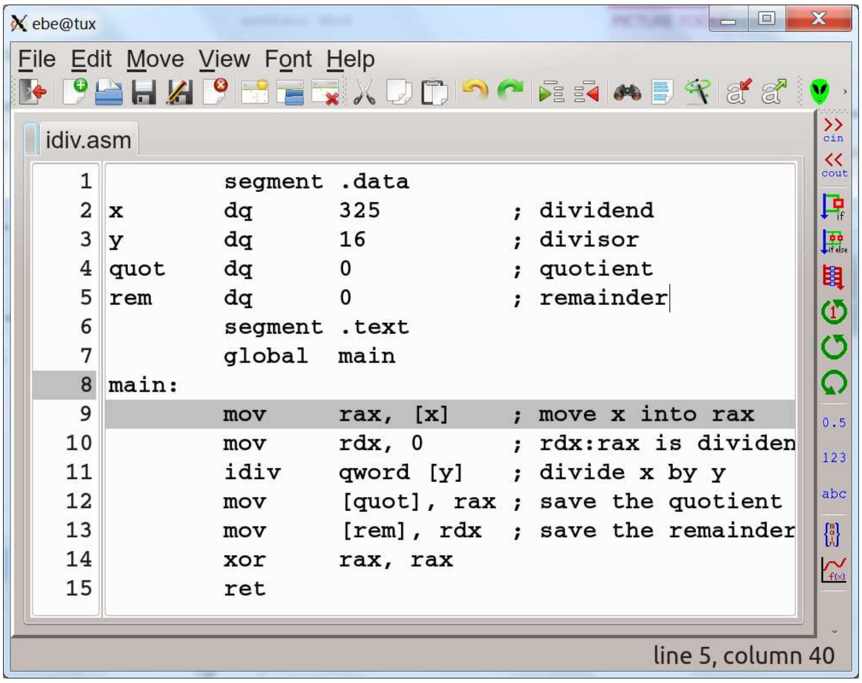
6.5 Division

Division is different from the other mathematics operations in that it returns 2 results: a quotient and a remainder. The `idiv` instruction behaves a little like the inverse of the single operand `imul` instruction in that it uses `rdx:rax` for the 128 bit dividend.

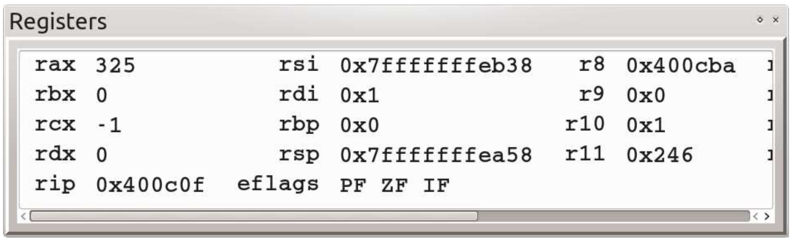
The `idiv` instruction uses a single source operand which can be either a register or a memory reference. The unsigned division instruction `div` operates similarly on unsigned numbers. The dividend is the two registers `rdx` and `rax` with `rdx` holding the most significant bits. The quotient is stored in `rax` and the remainder is stored in `rdx`.

The `idiv` instruction does not set any status flags, so testing the results must be done separately.

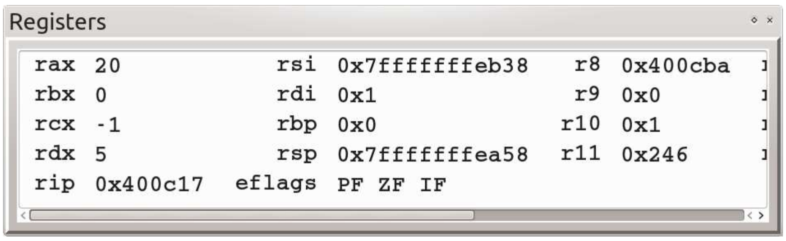
Below is a program which illustrates the `idiv` instruction. You can see that a breakpoint was placed on line 8 and the program was started using the “Run” button.



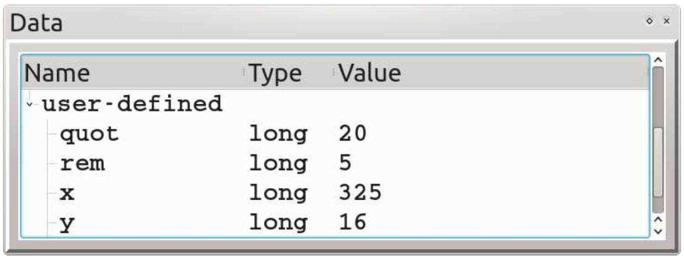
Next we see the registers after loading `x` into `rax` and zeroing out `rdx`.



The next display shows the changes to `rax` and `rdx` from executing the `idiv` instruction. The quotient is 20 and the remainder is 5 since $325 = 20 * 16 + 5$.



The final display shows the variables after executing lines 12 and 13.



6.6 Conditional move instructions

There are a collection of conditional move instructions which can be used profitably rather than using branching. Branching causes the CPU to perform branch prediction which will be correct sometimes and incorrect other times. Incorrect predictions slow down the CPU dramatically by interrupting the instruction pipeline, so it is worthwhile to learn to use conditional move instructions to avoid branching in simple cases.

The conditional move instructions have operands much like the `mov` instruction. There are a variety of them which all have the same 2 operands as `mov`, except that there is no provision for immediate operands.

instruction	effect
`cmovz`	move if result was zero
`cmovnz`	move if result was not zero
`cmovl`	move if result was negative
`cmovle`	move if result was negative or zero
`cmovg`	move if result was positive
`cmovge`	move if result was positive or zero

There are lot more symbolic patterns which have essentially the same meaning, but these are an adequate collection. These all operate by testing for combinations of the sign flag (`SF`) and the zero flag (`ZF`).

The following code snippet converts the value in rax to its absolute value:

```
mov     rbx, rax  ; save original value
neg     rax       ; negate rax
cmovl   rax, rbx  ; replace rax if negative
```

The code below loads a number from memory, subtracts 100 and replaces the difference with 0 if the difference is negative:

```
mov     rbx, 0    ; set rbx to 0
mov     rax, [x]  ; get x from memory
```

```
sub     rax, 100  ; subtract 100 from x
cmovl   rax, rbx  ; set rax to 0 if x-100 was negative
```

6.7 Why move to a register?

Both the add and sub instructions can operate on values stored in memory. Alternatively you could explicitly move the value into a register, perform the operation and then move the result back to the memory location. In this case it is 1 instruction versus 3. It seems obvious that 1 instruction is better.

Now if the value from memory is used in more than 1 operation, it might be faster to move it into a register first. This is a simple optimization which is fairly natural. It has the disadvantage of requiring the programmer to keep track of which variables are in which registers. If this code is not going to be executed billions of times, then the time required will probably not matter. In that case don't overwhelm yourself with optimization tricks. Also if the 2 uses are more than a few instructions apart, then keep it simple.

Exercises

1. Write an assembly language program to compute the distance squared between 2 points in the plane identified as 2 integer coordinates each, stored in memory.

   Remembe the Pythagoran Theorem!

2. If we could do floating point division, this exercise would have you compute the slope of the line segment connecting 2 points. Instead you are to store the difference in x coordinates in 1 memory location and the difference in y coordinates in another. The input points are integers stored in memory. Leave register `rax` with the value 1 if the line segment is vertical (infinite or undefined slope) and 0 if it is not. You should use a conditional move to set the value of `rax`.

3. Write an assembly language program to compute the average of 4 grades. Use memory locations for the 4 grades. Make the grades all different numbers from 0 to 100. Store the average of the 4 grades in memory and also store the remainder from the division in memory.

Chapter 7
Bit operations

A computer is a machine to process bits. So far we have discussed using bits to represent numbers. In this chapter we will learn about a handful of computer instructions which operate on bits without any implied meaning for the bits like signed or unsigned integers.

Individual bits have the values 0 and 1 and are frequently interpreted as false for 0 and true for 1. Individual bits could have other interpretations. A bit might mean male or female or any assignment of an entity to one of 2 mutually exclusive sets. A bit could represent an individual cell in Conway's game of Life.

Sometimes data occurs as numbers with limited range. Suppose you need to process billions of numbers in the range of 0 to 15. Then each number could be stored in 4 bits. Is it worth the trouble to store your numbers in 4 bits when 8 bit bytes are readily available in a language like C++? Perhaps not if you have access to a machine with sufficient memory. Still it might be nice to store the numbers on disk in half the space. So you might need to operate on bit fields.

7.1 Not operation

The not operation is a unary operation, meaning that it has only 1 operand. The everyday interpretation of not is the opposite of a logical statement. In assembly language we apply not to all the bits of a word. C has two versions of not, "`!`" and "~". "`!`" is used for the opposite of a true or false value, while "~" applies to all the bits of a word. It is common to distinguish the two nots by referring to "`!`" as the "logical" not and "~" as the "bit-wise" not. We will use "~" since the assembly language `not` instruction inverts each bit of a word. Here are some examples, illustrating the meaning of not (pretending the length of each value is as shown).

```
~0 == 1
~1 == 0
~10101010b == 01010101b
~0xff00 == 0x00ff
```

The `not` instruction has a single operand which serves as both the source and the destination. It can be applied to bytes, words, double words and quad-words in registers or in memory. Here is a code snippet illustrating its use.

```
mov   rax, 0
not   rax        ; rax == 0xffffffffffffffff
mov   rdx, 0     ; preparing for divide
mov   rbx, 15    ; will divide by 15 (0xf)
div   rbx        ; unsigned divide
                 ; rax == 0x1111111111111111
not   rax        ; rax == 0xeeeeeeeeeeeeeeee
```

Let's assume that you need to manage a set of 64 items. You can associate each possible member of the set with 1 bit of a quad-word. Using `not` will give you the complement of the set.

7.3 And operation

The and operation is also applied in programming in 2 contexts. First it is common to test for both of 2 conditions being true - && in C. Secondly you can do an and operation of each pair of bits in 2 variables - & in C. We will stick with the single & notation, since the assembly language `and` instruction matches the C bit-wise and operation.

Here is a truth table for the and operation:

&	0	1
0	0	0
1	0	1

Applied to some bit fields we get:

```
11001100b & 00001111b == 00001100b
11001100b & 11110000b == 11000000b
0xabcdefab & 0xff == 0xab
0x0123456789 & 0xff00ff00ff == 0x0100450089
```

You might notice that the examples illustrate using & as a bit field selector. Wherever the right operand has a 1 bit, the operation selected the bit from the left operand. You could say the same thing about the left operand, but in these examples the right operand has more obvious "masks" used to select bits.

Below is a code snippet illustrating the use of the `and` instruction:

```
mov   rax, 0x12345678
mov   rbx, rax
and   rbx, 0xf       ; rbx has nibble 0x8
mov   rdx, 0         ; prepare to divide
mov   rcx, 16        ; by 16
idiv  rcx            ; rax has 0x1234567
and   rax, 0xf       ; rax has nibble 0x7
```

It is a little sad to use a divide just to shift the number 4 bits to the right, but shift operations have not been discussed yet.

Using sets of 64 items you can use `and` to form the intersection of 2 sets. Also you can use `and` and `not` to form the difference of 2 sets, since $A - B = A \cap \bar{B}$.

7.3 Or operation

The or operation is the final bit operation with logical and bit-wise `or` meanings. First it is common to test for either (or both) of 2 conditions being true - `||` in C. Secondly you can do an `or` operation of each pair of bits in 2 variables - `|` in C. We will stick with the single `|` notation, since the assembly language `or` instruction matches the bit-wise or operation.

You need to be aware that the "or" of everyday speech is commonly used to mean 1 or the other but not both. When someone asks you if you want of cup of "decaf" or "regular", you probably should not answer "Yes". The "or" of programming means one or the other or both.

Here is a truth table for the or operation:

		0	1
0	0	1
1	1	1

Applied to some bit fields we get:

```
11001100b | 00001111b == 11001111b
11001100b | 11110000b == 11111100b
0xabcdefab | 0xff == 0xabcdefff
0x0123456789 | 0xff00ff00ff == 0xff23ff67ff
```

You might notice that the examples illustrate using `|` as a bit setter. Wherever the right operand has a 1 bit, the operation sets the corresponding bit of the left operand. Again, since or is commutative, we could say the same thing about the left operand, but the right operands have more obvious masks.

Here is a code snippet using the `or` instruction to set some bits:

```
mov     rax, 0x1000
or      rax, 1        ; make the number odd
or      rax, 0xff00   ; set bits 15-8 to 1
```

Using sets of 64 items you can use `or` to form the union of 2 sets.

7.4 Exclusive or operation

The final bit-wise operation is exclusive-or. This operation matches the everyday concept of 1 or the other but not both. The C exclusive-or operator is "^".

Here is a truth table for the exclusive-or operation:

^	0	1
0	0	1
1	1	0

From examining the truth table you can see that exclusive-or could also be called "not equals". In my terminology exclusive-or is a "bit-flipper". Consider the right operand as a mask which selects which bits to flip in the left operand. Consider these examples:

```
00010001b ^ 00000001b == 00010000b
01010101b ^ 11111111b == 10101010b
01110111b ^ 00001111b == 01111000b
0xaaaaaaaa ^ 0xffffffff == 0x55555555
0x12345678 ^ 0x12345678 == 0x00000000
```

The x86-64 exclusive-or instruction is named `xor`. The most common use of `xor` is as an idiom for setting a register to 0. This is done because moving 0 into a register requires 7 bytes for a 64 bit register, while `xor` requires 3 bytes. You can get the same result using the 32 bit version of the intended register which requires only 2 bytes for the instruction.

Observe some uses of `xor`:

```
mov   rax, 0x1234567812345678
xor   eax, eax           ; set to 0
mov   rax, 0x1234
xor   rax, 0xf           ; change to 0x123b
```

You can use `xor` to form the symmetric difference of 2 sets. The symmetric difference of 2 sets is the the elements which are in one of the 2 sets but not both. If you don't like exclusive-or, another way to compute this would be using $A \Delta B = (A \cup B) \cap \overline{A \cap B}$. Surely you like exclusive-or.

7.5 Shift operations

In the code example for the `and` instruction I divided by 16 to achieve the effect of converting `0x12345678` into `0x1234567`. This effect could have been obtained more simply by shifting the register's contents to the right

4 bits. Shifting is an excellent tool for extracting bit fields and for building values with bit fields.

In the x86-64 architecture there are 4 varieties of shift instructions: shift left (`shl`), shift arithmetic left (`sal`), shift right (`shr`), and shift arithmetic right (`sar`). The `shl` and `sal` shift left instructions are actually the same instruction. The `sar` instruction propagates the sign bit into the newly vacated positions on the left which preserves the sign of the number, while `shr` introduces 0 bits from the left.

15															0
1	0	1	0	1	1	0	0	1	0	1	1	0	1	1	0
0	1	0	1	0	1	1	0	0	1	0	1	1	0	1	1
0	0	1	0	1	0	1	1	0	0	1	0	1	1	0	1
0	0	0	1	0	1	0	1	1	0	0	1	0	1	1	0

There are 2 operands for a shift instruction. The first operand is the register or memory location to shift and the second is the number of bits to shift. The number to shift can be 8, 16, 32 or 64 bits in length. The number of bits can be an immediate value or the `cl` register. There are no other choices for the number of bits to shift.

C contains a shift left operator (<<) and a shift right operator (>>). The decision of logical or arithmetic shift right in C depends on the data type being shifted. Shifting a signed integer right uses an arithmetic shift.

Here are some examples of shifting:

```
10101010b >> 2 == 00101010b
10011001b << 4 == 100110010000b
0x12345678 >> 4 == 0x01234567
0x1234567 << 4 == 0x12345670
0xabcd >> 8 == 0x00ab
```

To extract a bit field from a word, you first shift the word right until the right most bit of the field is in the least significant bit position (bit 0) and then "and" the word with a value having a string of 1 bits in bit 0 through n-1 where n is the number of bits in the field to extract. For example to extract bits 4-7, shift right four bits, and then and with `0xf`.

To place some bits into position, you first need to clear the bits and then "or" the new field into the value. The first step is to build the mask with the proper number of 1's for the field width starting at bit 0. Then shift the mask left to align the mask with the value to hold the new field. Negate the mask to form an inverted mask. And the value with the inverted mask to clear out the bits. Then shift the new value left the proper number of bits and or this with the value.

Now consider the following program which extracts a bit field and then replaces a bit field.

ebe@tux

File Edit Move View Font Help

bits.asm

```
 1          segment .text
 2          global  main
 3 main:
 4          push    rbp
 5          mov     rbp, rsp
 6          mov     rax, 0x12345678
 7          shr     rax, 8          ; I want bits 8-15
 8          and     rax, 0xff       ; rax now holds 0x56
 9          mov     rax, 0x12345678 ; I want to replace bits 8-15
10          mov     rdx, 0xaa       ; rdx holds replacement field
11          mov     rbx, 0xff       ; I need an 8 bit mask
12          shl     rbx, 8          ; Shift mask to align @ bit 8
13          not     rbx             ; rbx is the inverted mask
14          and     rax, rbx        ; Now bits 8-15 are all 0
15          shl     rdx, 8          ; Shift the new bits to align
16          or      rax, rdx        ; rax now has 0x1234aa78
17          xor     rax, rax
18          leave
19          ret
```

line 1, column 1

The program was started with a breakpoint on line 4 and I used “Next” until line 6 was executed which placed 0x12345678 into rax.

Registers

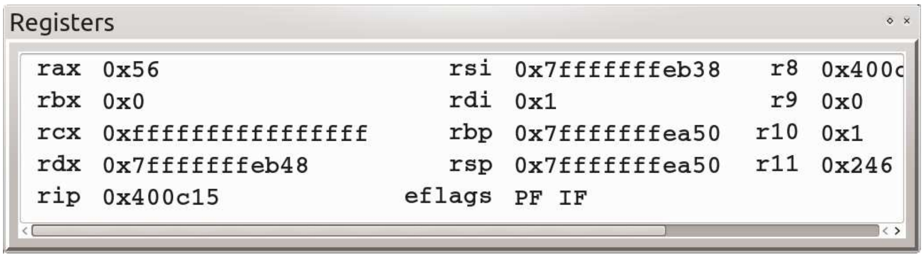
rax	0x12345678	rsi	0x7fffffffeb38	r8	0x400cca
rbx	0x0	rdi	0x1	r9	0x0
rcx	0xffffffffffff…	rbp	0x7fffffffea50	r10	0x1
rdx	0x7fffffffeb48	rsp	0x7fffffffea50	r11	0x246
rip	0x400c0b	eflags	PF ZF IF		

The first goal is to extract bits 8-15. We start by shifting right 8 bits. This leave the target bits in bits 0-7 of rax.

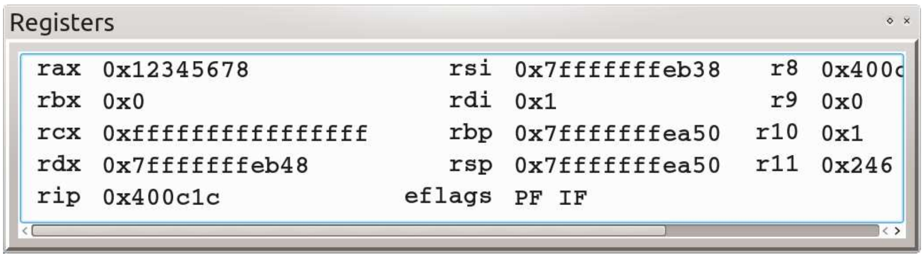
Registers

rax	0x123456	rsi	0x7fffffffeb38	r8	0x400c
rbx	0x0	rdi	0x1	r9	0x0
rcx	0xffffffffffffffff	rbp	0x7fffffffea50	r10	0x1
rdx	0x7fffffffeb48	rsp	0x7fffffffea50	r11	0x246
rip	0x400c0f	eflags	PF IF		

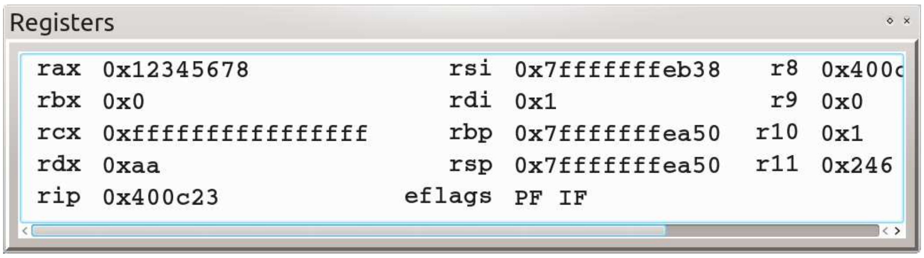
Next we must get rid of bits 8-63. The easiest way to do this is to and with 0xff.



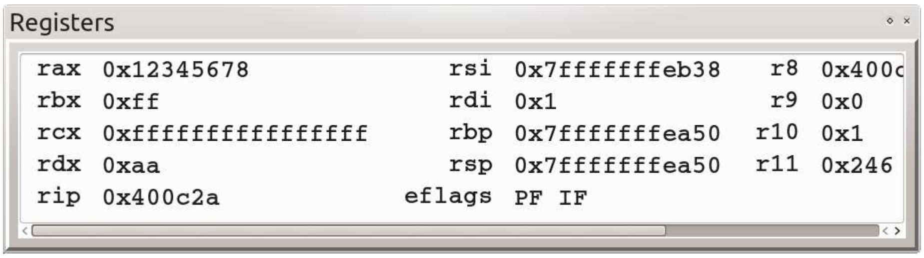
The next goal is to replace bits 8-15 of `0x12345678` with `0xaa` yielding `0x1234aa78`. We start by moving `0x12345678` into `rax`.



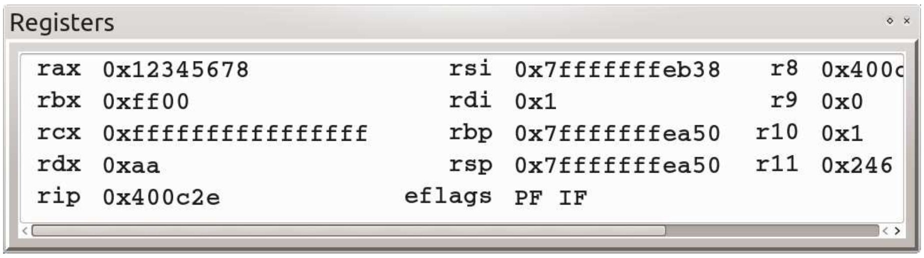
The second step is to get the value `0xaa` into `rdx`.



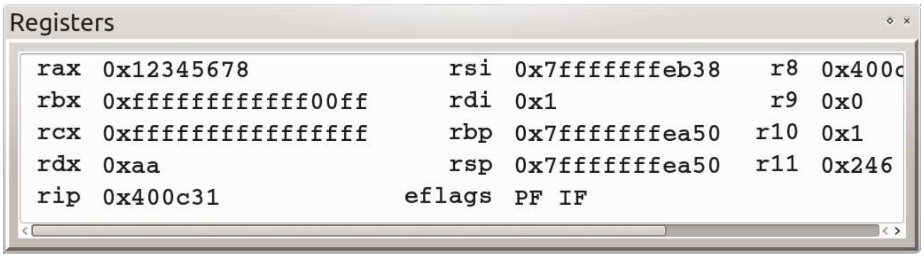
We need a mask to clear out bits 8-15. We start building the mask by placing `0xff` into `rbx`.



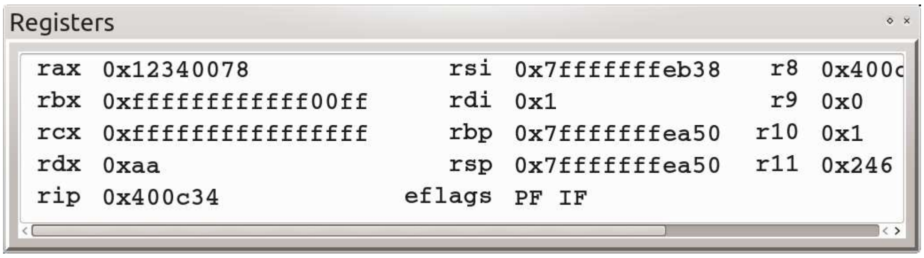
Then we shift `rbx` left 8 positions to align the mask with bits 8-15. We could have started with `0xff00`.



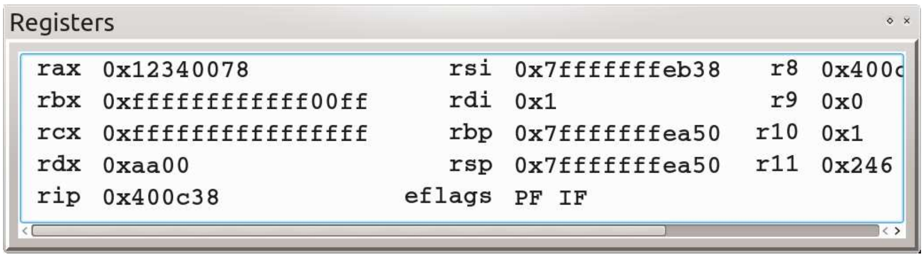
The final preparation of the mask is to complement all the bits with `not`. We could have started with `0xffffffffffff00ff`, but that would require some counting and is not as generally useful.



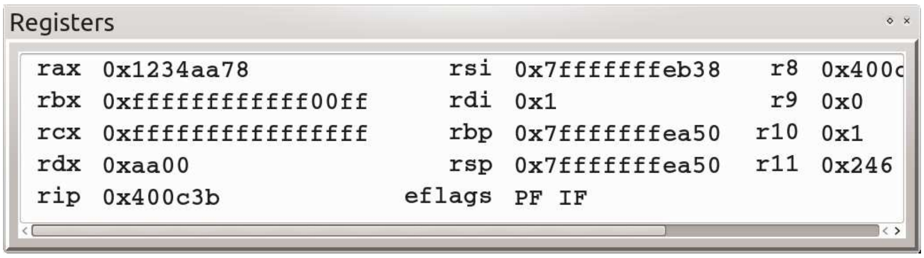
Using `and` as a bit selector we select each bit of `rax` which has a corresponding 1 bit in `rbx`.



Now we can shift `0xaa` left 8 positions to align with bits 8-15.

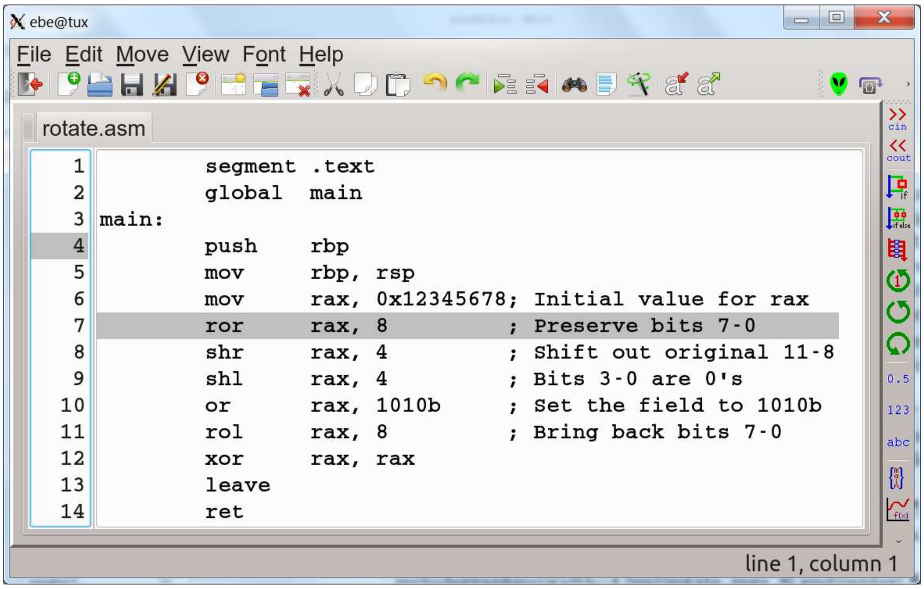


Having cleared out bits 8-15 of `rax`, we now complete the task by `or`'ing `rax` and `rdx`.

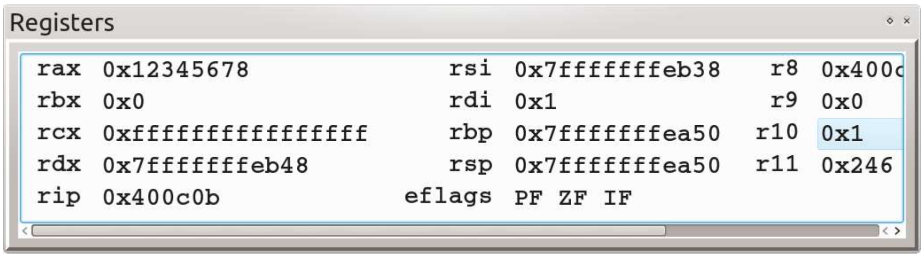


The x86-64 instruction set also includes rotate left (`rol`) and rotate right (`ror`) instructions. These could be used to shift particular parts of a bit string into proper position for testing while preserving the bits. After rotating the proper number of bits in the opposite direction, the original bit string will be left in the register or memory location.

The rotate instructions offer a nice way to clear out some bits. The code below clears out bits 11-8 of `rax` and replaces these bits with `1010b`.



Observe that a breakpoint has been placed on line 4 and the program run and stepped to line 7. In the register display below we see that `0x12345678` has been placed in `rax`.



Executing the rotate instruction on line 7 moves the `0x78` byte in `rax` to the upper part of the register.

Registers

rax	0x7800000000123456	rsi	0x7fffffffeb38	r8	0x400c
rbx	0x0	rdi	0x1	r9	0x0
rcx	0xffffffffffffffff	rbp	0x7fffffffea50	r10	0x1
rdx	0x7fffffffeb48	rsp	0x7fffffffea50	r11	0x246
rip	0x400c0f	eflags	PF ZF IF		

Next the shift instruction on line 8 wipes out bits 3-0 (original 11-8).

Registers

rax	0x780000000012345	rsi	0x7fffffffeb38	r8	0x400c
rbx	0x0	rdi	0x1	r9	0x0
rcx	0xffffffffffffffff	rbp	0x7fffffffea50	r10	0x1
rdx	0x7fffffffeb48	rsp	0x7fffffffea50	r11	0x246
rip	0x400c13	eflags	IF		

The shift instruction on line 9 introduces four 0 bits into `rax`.

Registers

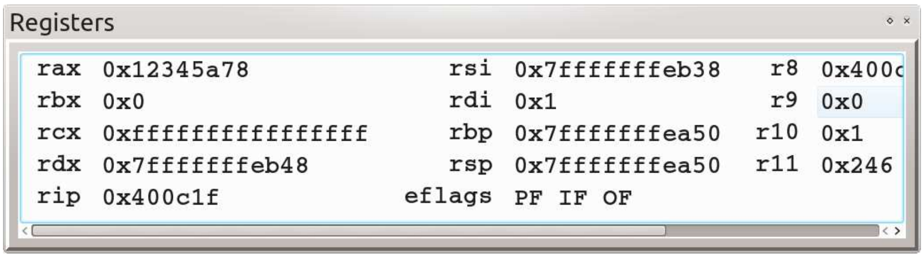
rax	0x7800000000123450	rsi	0x7fffffffeb38	r8	0x400c
rbx	0x0	rdi	0x1	r9	0x0
rcx	0xffffffffffffffff	rbp	0x7fffffffea50	r10	0x1
rdx	0x7fffffffeb48	rsp	0x7fffffffea50	r11	0x246
rip	0x400c17	eflags	PF IF		

Now the `or` instruction at line 10 places `1010b` into bits 3-0.

Registers

rax	0x780000000012345a	rsi	0x7fffffffeb38	r8	0x400c
rbx	0x0	rdi	0x1	r9	0x0
rcx	0xffffffffffffffff	rbp	0x7fffffffea50	r10	0x1
rdx	0x7fffffffeb48	rsp	0x7fffffffea50	r11	0x246
rip	0x400c1b	eflags	PF IF		

Finally the rotate left instruction at line 11 realigns all the bits as they were originally.



Interestingly C provides shift left (<<) and shift right (>>) operations, but does not provide a rotate operation. So a program which does a large amount of bit field manipulations might be better done in assembly. On the other hand a C struct can have bit fields in it and thus the compiler can possibly use rotate instructions with explicit bit fields.

7.6 Bit testing and setting

It takes several instructions to extract or insert a bit field. Sometimes you need to extract or insert a single bit. This can be done using masking and shifting as just illustrated. However it can be simpler and quicker to use the bit test instruction (`bt`) and either the bit test and set instruction (`bts`) or the bit test and reset instruction (`btr`).

The `bt` instruction has 2 operands. The first operand is a 16, 32 or 64 bit word in memory or a register which contains the bit to test. The second operand is the bit number from 0 to the number of bits minus 1 for the word size which is either an immediate value or a value in a register. The `bt` instructions set the carry flag (`CF`) to the value of the bit being tested.

The `bts` and `btr` instructions operate somewhat similarly. Both instructions test the current bit in the same fashion as `bt`. They differ in that `bts` sets the bit to 1 and `btr` resets (or clears) the bit to 0.

One particular possibility for using these instructions is to implement a set of fairly large size where the members of the set are integers from 0 to $n-1$ where n is the universe size. A membership test translates into determining a word and bit number in memory and testing the correct bit in the word. Following the `bt` instruction the `setc` instruction can be used to store the value of the carry flag into an 8 bit register. There are `setCC` instructions for each of the condition flags in the `eflags` register. Insertion into the set translates into determining the word and bit number and using `bts` to set the correct bit. Removal of an element of the set translates into using `btr` to clear the correct bit in memory.

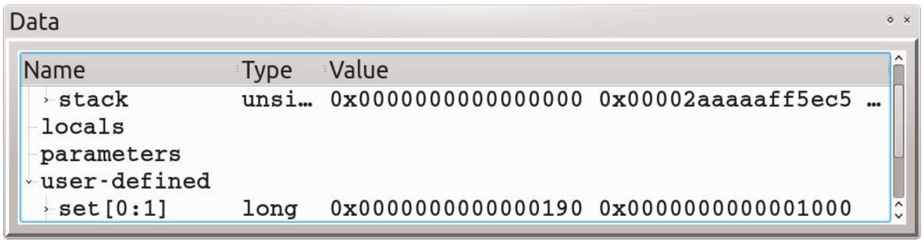
In the code below we assume that the memory for the set is at a memory location named `set` and that the bit number to work on is in register `rax`. The code preserves `rax` and performs testing, insertion and removal.

```
 1          segment .bss
 2 set      resq    10
 3          segment .text
 4          global  main
 5 main:
 6          push    rbp
 7          mov     rbp, rsp
 8          bts     qword [set], 4      ; set bit 4 of set
 9          bts     qword [set], 7      ; set bit 7 of set
10          bts     qword [set], 8      ; set bit 8 of set
11          bts     qword [set+8], 12   ; set bit 76 of set
12          mov     rax, 76             ; test bits 4, 76, 77
13          mov     rbx, rax            ; copy bit number to rbx
14          shr     rbx, 6              ; qword number of set to test
15          mov     rcx, rax            ; copy bit number to rcx
16          and     rcx, 0x3f           ; extract rightmost 6 bits
17          xor     edx, edx            ; set rdx to 0
18          bt      [set+8*rbx], rcx    ; test bit
19          setc    dl                  ; edx equals the tested bit
20          bts     [set+8*rbx], rcx    ; set the bit, insert into set
21          btr     [set+8*rbx], rcx    ; clear the bit, remove
22          xor     rax, rax
23          leave
24          ret
```

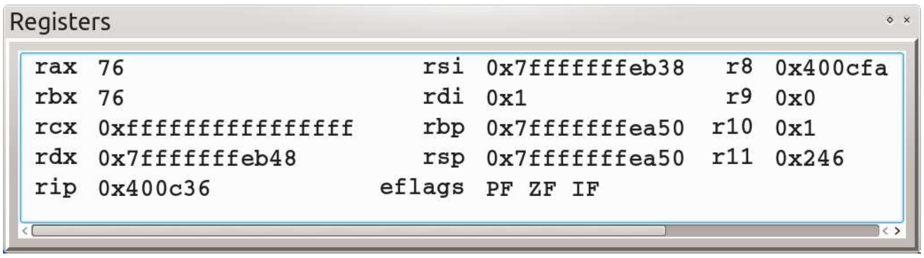
Lines 8 through 11 set bits 4, 7, 8 and 76 in the array `set`. To set bit 76, we use `[set+8]` in the instruction to reference the second quad-word of the array. You will also notice the use of `set+8*rbx` in lines 18, 20 and 21. Previously we have used a variable name in brackets. Now we are using a variable name plus a constant or plus a register times 8. The use of a register times 8 allows indexing an array of 8 byte quantities. The instruction format includes options for multiplying an index register by 2, 4 or 8 to be added to the address specified by `set`. Use 2 for a word array, 4 for a double word array and 8 for a quad-word array. Register `rbx` holds the quad-word index into the `set` array.

Operating on the quad-word of the set in memory as opposed to moving to a register is likely to be the fastest choice, since in real code we will not need to test, insert and then remove in 1 function call. We would do only one of these operations.

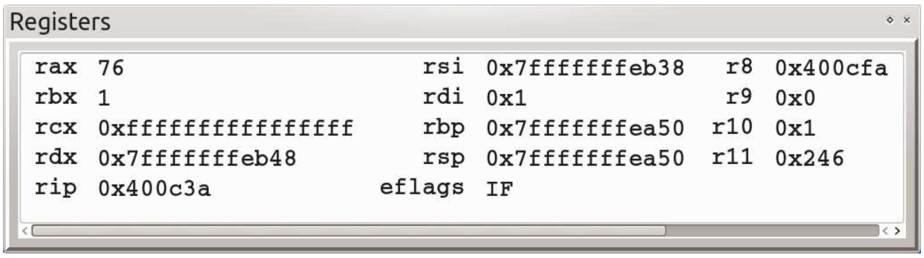
Here we trace through the execution of this program. We start by observing the `set` array in hexadecimal at the breakpoint on line 12. We set bits 4, 7, 8 and 76. Setting bit 4 yields `0x10`, setting bit 7 yields `0x90` and setting bit 8 yields `0x190`. Bit 76 is bit 12 of the second quad-word in the array and yields `0x1000`.



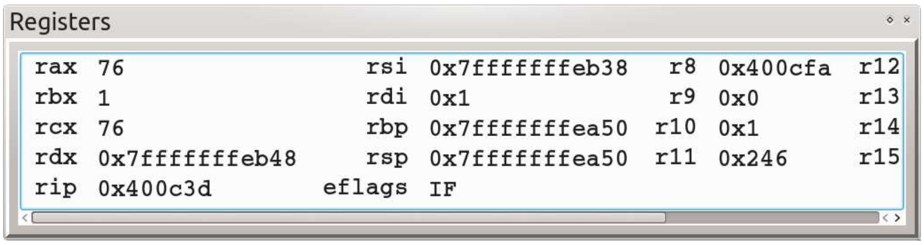
Next lines 12 and 13 move 76 into `rax` and `rbx`.



Shifting the bit number (76) right 6 bits will yield the quad-word number of the array. This works since $2^6 = 64$ and quad-words hold 64 bits. This shift leaves a 1 in `rbx`.



We make another copy of the bit number in `rcx`.



The bit number and'ed with `0x3f` will extract the rightmost 6 bits of the bit number. This will be the bit number of the quad-word containing the bit.

```
Registers
rax  76                    rsi  0x7fffffffeb38   r8  0x400cfa   r12
rbx  1                     rdi  0x1              r9  0x0        r13
rcx  12                    rbp  0x7fffffffea50  r10  0x1        r14
rdx  0x7fffffffeb48        rsp  0x7fffffffea50  r11  0x246      r15
rip  0x400c41           eflags  PF IF
```

Next we use `xor` to zero out `rdx`.

```
Registers
rax  76             rsi  0x7fffffffeb38   r8  0x400cfa   r12
rbx  1              rdi  0x1              r9  0x0        r13
rcx  12             rbp  0x7fffffffea50  r10  0x1        r14
rdx  0              rsp  0x7fffffffea50  r11  0x246      r15
rip  0x400c43    eflags  PF ZF IF
```

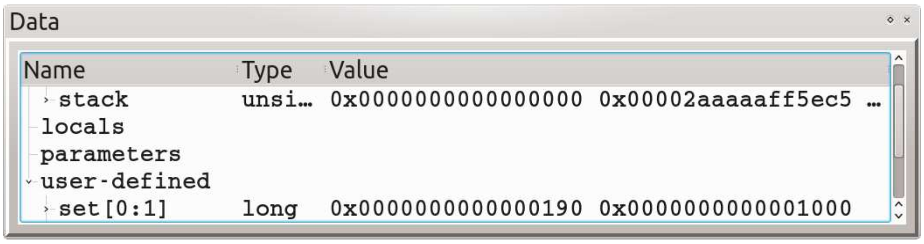
Line 18 tests the bit we wish to test from the array. You will notice that the carry flag (`CF`) is set.

```
Registers
rax  76             rsi  0x7fffffffeb38   r8  0x400cfa   r12
rbx  1              rdi  0x1              r9  0x0        r13
rcx  12             rbp  0x7fffffffea50  r10  0x1        r14
rdx  0              rsp  0x7fffffffea50  r11  0x246      r15
rip  0x400c4c    eflags  CF PF ZF IF
```

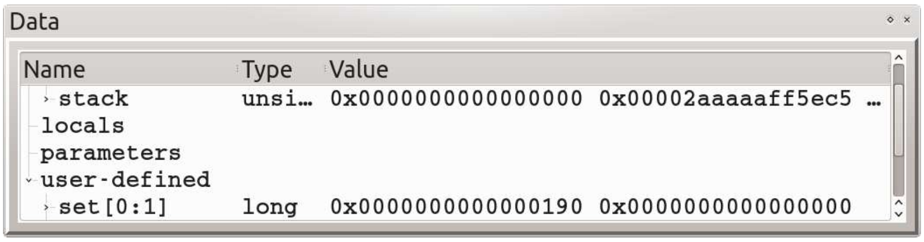
Line 19 uses the `setc` instruction to set `dl` which is now a 1 since 76 was in the set.

```
Registers
rax  76             rsi  0x7fffffffeb38   r8  0x400cfa   r12
rbx  1              rdi  0x1              r9  0x0        r13
rcx  12             rbp  0x7fffffffea50  r10  0x1        r14
rdx  1              rsp  0x7fffffffea50  r11  0x246      r15
rip  0x400c4f    eflags  CF PF ZF IF
```

Line 20 sets the bit in the set.



Line 21 clears the bit (reset), effectively removing 76 from the set.



7.7 Extracting and filling a bit field

To extract a bit field you need to shift the field so that its least significant bit is in position 0 and then bit mask the field with an `and` operation with the appropriate mask. Let's suppose we need to extract bits 51-23 from a quad-word stored in a memory location. Then, after loading the quad-word, we need to shift it right 23 bits to get the least significant bit into the proper position. The bit field is of length 29. The simplest way to get a proper mask (29 bits all 1) is using the value `0x1fffffff`. Seven `f`'s is 28 bits and the 1 gives a total of 29 bits. Here is the code to do the work:

```
mov   rax, [sample]   ; move qword into rax
shr   rax, 23         ; align bit 23 at 0
and   rax, 0x1fffffff ; select 29 low bits
mov   [field], rax    ; save the field
```

Of course it could be that the field width is not a constant. In that case you need an alternative. One possibility is to generate a string of 1 bits based on knowing that $2^n - 1$ is a string of n 1 bits. You can generate 2^n by shifting 1 to the left n times or use `bts`. Then you can subtract 1 using `dec`.

Another way to extract a bit field is to first shift left enough bits to clear out the bits to the left of the field and then shift right enough bits to wipe out the bits to the right of the field. This will be simpler when the field position and width are variable. To extract bits 51-23, we start by shifting left 12 bits. Then we need to shift right 35 bits. In general if the field is bits m through n where m is the higher bit number, we shift left $63 - m$ and then shift right $n + (63 - m)$.

```
mov  rax, [sample] ; move qword into rax
shl  rax, 12       ; wipe out higher bits
shr  rax, 35       ; align the bit field
mov  [field], rax  ; save the field
```

Now suppose we wish to fill in bits 51-23 of `sample` with the bits in `field`. The easy method is to rotate the value to align the field, shift right and then left to clear 29 bits, or in the field, and then rotate the register to get the field back into bits 23-51. Here is the code:

```
mov  rax, [sample] ; move qword into rax
ror  rax, 23       ; align bit 23 at 0
shr  rax, 29       ; wipe out 29 bits
shl  rax, 29       ; align bits again
or   rax, [field]  ; trust field is 29 bits
rol  rax, 23       ; realign the bit fields
mov  [sample], rax ; store fields in memory
```

Exercises

1. Write an assembly program to count all the 1 bits in a byte stored in memory. Use repeated code rather than a loop.

2. Write an assembly program to swap 2 quad-words in memory using `xor`. Use the following algorithm:

```
a = a ^ b
b = a ^ b
a = a ^ b
```

3. Write an assembly program to use 3 quad-words in memory to represent 3 sets: `A`, `B` and `C`. Each set will allow storing set values 0-63 in the corresponding bits of the quad-word. Perform these steps:

```
insert 0 into A
insert 1 into A
insert 7 into A
insert 13 into A
insert 1 into B
insert 3 into B
insert 12 into B
store A union B into C
store A intersect B into C
store A - B into C
remove 7 from C
```

4. Write an assembly program to move a quad-word stored in memory into a register and then compute the exclusive-or of the 8 bytes of the word. Use either `ror` or `rol` to manipulate the bits of the register so that the original value is retained.

5. Write an assembly program to dissect a double stored in memory. This is a 64 bit floating point value. Store the sign bit in one memory location. Store the exponent after subtracting the bias value into a second memory location. Store the fraction field with the implicit 1 bit at the front of the bit string into a third memory location.

6. Write an assembly program to perform a product of 2 float values using integer arithmetic and bit operations. Start with 2 float values in memory and store the product in memory.

Chapter 8
Branching and looping

So far we have not used any branching statements in our code. Using the conditional move instructions added a little flexibility to the code while preserving the CPU's pipeline contents. We have seen that it can be tedious to repeat instructions to process each byte in a quad-word or each bit in a byte. In the next chapter we will work with arrays. It would be fool-hardy to process an array of 1 million elements by repeating the instructions. It might be possible to do this, but it would be painful coping with variable sized arrays. We need loops.

In many programs you will need to test for a condition and perform one of 2 actions based on the results. The conditional move is efficient if the 2 actions are fairly trivial. If each action is several instructions long, then we need a conditional jump statement to branch to one alternative while allowing the CPU to handle the second alternative by not branching. After completing the second alternative we will typically need to branch around the code for the first alternative. We need conditional and unconditional branch statements.

8.1 Unconditional jump

The unconditional jump instruction (`jmp`) is the assembly version of the `goto` statement. However there is clearly no shame in using `jmp`. It is a necessity in assembly language, while `goto` can be avoided in higher level languages.

The basic form of the `jmp` instruction is

```
        jmp     label
```

where `label` is a label in the program's text segment. The assembler will generate a `rip` relative jump instruction, meaning that the flow of control will transfer to a location relative to the current value of the instruction pointer. The simplest relative jump uses an 8 bit signed immediate value and is encoded in 2 bytes. This allows jumping forwards or backwards about 127 bytes. The next variety of relative jump in 64 bit mode uses a 32 bit signed immediate value and requires a total of 5 bytes. Fortunately the assembler figures out which variety it can use and chooses the shorter form. The programmer simply specifies a label.

The effect of the `jmp` statement is that the CPU transfers control to the instruction at the labeled address. This is generally not too exciting except when used with a conditional jump. However, the `jmp` instruction can jump to an address contained in a register or memory location. Using a conditional move one could manage to use an unconditional jump to an address contained in a register to implement a conditional jump. This isn't sensible, since there are conditional jump statements which handle this more efficiently.

There is one more possibility which is more interesting - implementing a switch statement. Suppose you have a variable `i` which is known to contain a value from 0 to 2. Then you can form an array of instruction addresses and use a `jmp` instruction to jump to the correct section of code based on the value of `i`. Here is an example:

```
    segment .data
switch:
    dq    main.case0
    dq    main.case1
    dq    main.case2
i:  dq    2
    segment .text
    global  main
main:
    mov   rax, [i]        ; move i to rax
    jmp   [switch+rax*8] ; switch ( i )
.case0:
    mov   rbx, 100       ; go here if i == 0
    jmp   .end
.case1:
    mov   rbx, 101       ; go here if i == 1
    jmp   .end
.case2:
    mov   rbx, 102       ; go here if i == 2
.end:
    xor   eax, eax
    ret
```

In this code we have used a new form of label with a dot prefix. These labels are referred to as "local" labels. They are defined within the range of enclosing regular labels. Basically the local labels could be used for all labels inside a function and this would allow using the same local labels in multiple functions. Also we used `main.case0` outside of main to refer to the `.case0` label inside main.

From this example we see that an unconditional jump instruction can be used to implement some forms of conditional jumps. Though conditional jumps are more direct and less confusing, in larger switch statements it might be advantageous to build an array of locations to jump to.

8.2 Conditional jump

To use a conditional jump we need an instruction which can set some flags. This could be an arithmetic or bit operation. However doing a subtraction just to learn whether 2 numbers are equal might wipe out a needed value in a register. The x86-64 CPU provides a compare instruction (`cmp`) which subtracts its second operand from its first and sets flags without storing the difference.

There are quite a few conditional jump instructions with the general pattern:

```
jCC     label  ; jump to location
```

The `CC` part of the instruction name represents any of a wide variety of condition codes. The condition codes are based on specific flags in `eflags` such as the zero flag, the sign flag, and the carry flag. Below are some useful conditional jump instructions.

instruction	meaning	aliases	flags
`jz`	jump if zero	`je`	ZF=1
`jnz`	jump if not zero	`jne`	ZF=0
`jg`	jump if > 0	`jnle`	ZF=0 and SF=0
`jge`	jump if ≥ 0	`jnl`	SF=0
`jl`	jump if > 0	`jnge js`	SF=1
`jle`	jump if ≤ 0	`jng`	ZF=1 or SF=1
`jc`	jump if carry	`jb jnae`	CF=1
`jnc`	jump if not carry	`jnb jae`	

It is possible to generate "spaghetti" code using jumps and conditional jumps. It is probably best to stick with high level coding structures translated to assembly language. The general strategy is to start with C code and translate it to assembly. The rest of the conditional jump section discusses how to implement C if statements.

Simple if statement

Let's consider how to implement the equivalent of a C simple `if` statement. Suppose we are implementing the following C code:

```
if ( a < b ) {
    temp = a;
    a = b;
```

```
        b = temp;
    }
```

Then the direct translation to assembly language would be

```
;       if ( a < b ) {
        mov   rax, [a]
        mov   rbx, [b]
        cmp   rax, rbx
        jge   in_order
;           temp = a;
            mov   [temp],  rax
;           a = b;
            mov   [a], rbx
;           b = temp
            mov   [b], rax
;       }
in_order:
```

The most obvious pattern in this code is the inclusion of C code as comments. It can be hard to focus on the purpose of individual assembly statements. Starting with C code which is known to work makes sense. Make each C statement an assembly comment and add assembly statements to achieve each C statement after the C statement. Indenting might help a little though the indentation pattern might seem a little strange.

You will notice that the if condition was less than, but the conditional jump used greater than or equal to. Perhaps it would appeal to you more to use `jnl` rather than `jge`. The effect is identical but the less than mnemonic is part of the assembly instruction (with not). You should select the instruction name which makes the most sense to you.

If/else statement

It is fairly common to do 2 separate actions based on a test. Here is a simple C `if` statement with an `else` clause:

```
        if ( a < b ) {
            max = b;
        } else {
            max = a;
        }
```

This code is simple enough that a conditional move statement is likely to be a faster solution, but nevertheless here is the direct translation to assembly language:

```
;       if ( a < b ) {
        mov   rax, [a]
        mov   rbx, [b]
        cmp   rax, rbx
        jnl   else ;
            max = b;
            mov   [max], rbx
```

```
          jmp   endif
;     } else {
else:
;         max = a;
          mov   [max], rax
;     }
endif:
```

If/else-if/else statement

Just as in C/C++ you can have an `if` statement for the `else` clause, you can continue to do tests in the `else` clause of assembly code conditional statements. Here is a short `if/else-if/else` statement in C:

```
if ( a < b ) {
    result = 1;
} else if ( a > c ) {
    result = 2;
} else {
    result = 3;
}
```

This code is possibly a good candidate for 2 conditional move statements, but simplicity is bliss. Here is the assembly code for this:

```
;     if ( a < b ) {
      mov   rax, [a]
      mov   rbx, [b]
      cmp   rax, rbx
      jnl   else_if
;         result = 1;
          mov   qword [result], 1
          jmp   endif
;     } else if ( a > c ) {
else_if:
      mov   rcx, [c]
      cmp   rax, rcx
      jng   else
;         result = 2;
          mov   qword [result], 2
          jmp   endif
;     } else {
else:
;         result = 3;
          mov   qword [result], 3
;     }
endif:
```

It should be clear that an arbitrary sequence of tests can be used to simulate multiple `else-if` clauses in C.

8.3 Looping with conditional jumps

The jumps and conditional jumps introduced so far have been jumping forward. By jumping backwards, it is possible to produce a variety of loops. In this section we discuss `while` loops, `do-while` loops and counting loops. We also discuss how to implement the effects of C's `continue` and `break` statements with loops.

While loops

The most basic type of loop is possibly the `while` loop. It generally looks like this in C:

```
while ( condition ) {
   statements;
}
```

C `while` loops support the `break` statement which gets out of the loop and the `continue` statement which immediately goes back to the top of the loop. Structured programming favors avoiding `break` and `continue`. However they can be effective solutions to some problems and, used carefully, are frequently clearer than alternatives based on setting condition variables. They are substantially easier to implement in assembly than using condition variables and faster.

Counting 1 bits in a memory quad-word

The general strategy is to shift the bits of a quad-word 1 bit at a time and add bit 0 of the value at each iteration of a loop to the sum of the 1 bits. This loop needs to be done 64 times. Here is the C code for the loop:

```
sum = 0;
i = 0;
while ( i < 64 ) {
    sum += data & 1;
    data = data >> 1;
    i++;
}
```

The program below implements this loop with only the minor change that values are in registers during the execution of the loop. It would be pointless to store these values in memory during the loop. The C code is shown as comments which help explain the assembly code.

```
      segment .data ; long data;
data  dq      0xfedcba9876543210 ; long sum;
sum   dq      0
      segment .text
      global  main
```

```
; int main() ; {
main:
      push    rbp
      mov     rbp, rsp
      sub     rsp, 16

;     int i;          in register rcx
;     Register usage
;
;     rax: bits being examined
;     rbx: carry bit after bt, setc
;     rcx: loop counter i, 0-63
;     rdx: sum of 1 bits

      mov     rax, [data]
      xor     ebx, ebx
;     i = 0;
      xor     ecx, ecx
;     sum = 0;
      xor     edx, edx
;     while ( i < 64 ) {
while:
      cmp     rcx, 64
      jnl     end_while
;         sum += data & 1;
          bt      rax, 0
          setc    bl
          add     edx, ebx
;         data >>= 1;
          shr     rax, 1
;         i++;
          inc     rcx
;     }
      jmp     while
end_while:
      mov     [sum], rdx
      xor     eax, eax
      leave
      ret
```

The first instruction of the loop is `cmp` which is comparing `i` (`rcx`) versus 64. The conditional jump selected, `jnl`, matches the inverse of the C condition. Hopefully this is less confusing than using `jge`. The last instruction of the loop is a jump to the first statement of the loop. This is the typical translation of a `while` loop.

Coding this in C and running

```
gcc -O3 -S countbits.c
```

yields an assembly language file named `countbits.s` which is unfortunately not quite matching our yasm syntax. The assembler for gcc, `gas`, uses the AT&T syntax which differs from the Intel syntax used by yasm. Primarily the source and destination operands are reversed and some slight changes are made to instruction mnemonics. You can also use

```
gcc -O3 -S -masm=intel countbits.c
```

to request that gcc create an assembly file in Intel format which is very close to the code in this book. Here is the loop portion of the program produced by gcc;

```
        mov     rax, QWORD PTR data[rip]
        mov     ecx, 64
        xor     edx, edx
.L2:
        mov rsi, rax
        sar     rax
        and     esi, 1
        add rdx, rsi
        sub     ecx, 1
        jne     .L2
```

You will notice that the compiler eliminated one jump instruction by shifting the test to the end of the loop. Also the compiler did not do a compare instruction. In fact it discovered that the counting up to 64 of `i` was not important. Only the number of iterations mattered, so it decremented down from 64 to 0. Thus it was possible to do a conditional jump after the decrement. Overall the compiler generated a loop with 6 instructions, while the hand-written assembly loop used 8 instructions. As stated in the introduction a good compiler is hard to beat. You can learn a lot from studying the compiler's generated code. If you are interested in efficiency you may be able to do better than the compiler. You could certainly copy the generated code and do exactly the same, but if you can't improve on the compiler's code then you should stick with C.

There is one additional compiler option, `-funroll-all-loops` which tends to speed up code considerably. In this case the compiler used more registers and did 8 iterations of a loop which added up 8 bits in each iteration. The compiler did 8 bits in 24 instructions where before it did 1 bit in 6 instructions. This is about twice as fast. In addition the instruction pipeline is used more effectively in the unrolled version, so perhaps this is 3 times as fast.

Optimization issues like loop unrolling are highly dependent on the CPU architecture. Using the CPU in 64 bit mode gives 16 general-purpose registers while 32 bit mode gives only 8 registers. Loop unrolling is much easier with more registers. Other details like the Intel Core i series processors' use of a queue of micro-opcodes might eliminate most of the effect of loops interrupting the CPU pipeline. Testing is required to see what works best on a particular CPU.

Do-while loops

We saw in the last section that the compiler converted a `while` loop into a `do-while` loop. The `while` structure translates directly into a conditional jump at the top of the loop and an unconditional jump at the bottom of the loop. It is always possible to convert a loop to use a conditional jump at the bottom.

A C `do-while` loop looks like

```
do {
    statements;
} while ( condition );
```

A `do-while` always executes the body of the loop at least once.

Let's look at a program implementing a search in a character array, terminated by a 0 byte. We will do an explicit test before the loop to not execute the loop if the first character is 0. Here is the C code for the loop:

```
i = 0;
c = data[i];
if ( c != 0 ) {
    do {
        if ( c == x ) break;
        i++;
        c = data[i];
    } while ( c != 0 );
}
n = c == 0 ? -1 : i;
```

Here's an assembly implementation of this code:

```
      SECTION .data
data  db      "hello world", 0
n     dq      0
needle:
      db      'w'
      SECTION .text
      global  main
main:
      push    rbp
      mov     rbp, rsp
      sub     rsp, 16

;     Register usage
;
;     rax: c, byte of data array
;     bl:  x, byte to search for
;     rcx: i, loop counter, 0-63

      mov     bl, [needle]
;     i = 0;
      xor     ecx, ecx
;     c = data[i];
      mov     al, [data+rcx]
;     if ( c != 0 ) {
      cmp     al, 0
      jz      end_if
;         do {
do_while:
;             if ( c == x ) break;
              cmp     al, bl
              je      found
;             i++;
              inc     rcx
```

```
;               c = data[i];
                mov     al, [data+rcx];
;           } while ( c != 0 );
            cmp     al, 0
            jnz     do_while
;       }
end_if:
;       n = c == 0 ? -1 : i;
        mov     rcx, -1 ; c == 0 if you reach here
found:
        mov     [n], rcx
;       return 0;
        xor     eax, eax
        leave
        ret
```

The assembly code (if stripped of the C comments) looks simpler than the C code. The C code would look better with a `while` loop. The conditional operator in C was not necessary in the assembly code, since the conditional jump on finding the proper character jumps past the movement of -1 to `rcx`.

It might seem rational to try to use more structured techniques, but the only reasons to use assembly are to improve efficiency or to do something which can't be done in a high level language. Bearing that in mind, we should try to strike a balance between structure and efficiency.

Counting loops

The normal counting loop in C is the `for` loop, which can be used to implement any type of loop. Let's assume that we wish to do array addition. In C we might use

```
for ( i = 0; i < n; i++ ) {
    c[i] = a[i] + b[i];
}
```

Translated into assembly language this loop might be

```
        mov     rdx, [n]
        xor     ecx, ecx
for:    cmp     rcx, rdx
        je      end_for
        mov     rax, [a+rcx*8]
        add     rax, [b+rcx*8]
        mov     [c+rcx*8], rax
        inc     rcx
        jmp     for
end_for:
```

Once again it is possible to do a test on `rdx` being 0 before executing the loop. This could allow the compare and conditional jump statements to be placed at the end of the loop. However it might be easier to simply translate C statements without worrying about optimizations until you improve your assembly skills. Perhaps you are taking an assembly class. If so, does performance affect your grade? If not, then keep it simple.

8.4 Loop instructions

There is a `loop` instruction along with a couple of variants which operate by decrementing the `rcx` register and branching until the register reaches 0. Unfortunately, it is about 4 times faster to subtract 1 explicitly from `rcx` and use `jnz` to perform the conditional jump. This speed difference is CPU specific and only true for a trivial loop. Generally a loop will have other work which will take more time than the loop instruction. Furthermore the `loop` instruction is limited to branching to a 8 bit immediate field, meaning that it can branch backwards or forwards about 127 bytes. All in all, it doesn't seem to be worth using.

Despite the forgoing tale of gloom, perhaps you still wish to use `loop`. Consider the following code which looks in an array for the right-most occurrence of a specific character:

```
        mov     ecx, [n]
more:   cmp     [data+rcx-1],al
        je      found
        loop    more
found:  sub     ecx, 1
        mov     [loc], ecx
```

8.5 Repeat string (array) instructions

The x86-64 repeat instruction (`rep`) repeats a string instruction the number of times specified in the count register (`rcx`). There are a handful of variants which allow early termination based on conditions which may occur during the execution of the loop. The repeat instructions allow setting array elements to a specified value, copying one array to another, and finding a specific value in an array.

String instructions

There are a handful of string instructions. The ones which step through arrays are suffixed with `b`, `w`, `d` or `q` to indicate the size of the array elements (1, 2, 4 or 8 bytes).

The string instructions use registers `rax`, `rsi` and `rdi` for special purposes. Register `rax` or its sub-registers `eax`, `ax` and `al` are used to hold a specific value. Resister rsi is the source address register and rdi is the destination address. None of the string instructions need operands.

All of the string operations working with 1, 2 or 4 byte quantities are encoded in 1 byte, while the 8 byte variants are encoded as 2 bytes. Combined with a 1 byte repeat instruction, this effectively encodes some fairly simple loops in 2 or 3 bytes. It is hard to beat a repeat.

The string operations update the source and/or destination registers after each use. This updating is managed by the direction flag (DF). If DF is 0 then the registers are increased by the size of the data item after each use. If DF is 1 then the registers are decreased after each use.

Move

The movsb instruction moves bytes from the address specified by rsi to the address specified by rdi. The other movs instructions move 2, 4 or 8 byte data elements from [rsi] to [rdi]. The data moved is not stored in a register and no flags are affected. After each data item is moved, the rdi and rsi registers are advanced 1, 2, 4 or 8 bytes depending on the size of the data item.

Below is some code to move 100000 bytes from one array to another:

```
     lea     rsi, [source]
     lea     rdi, [destination]
     mov     rcx, 100000
     rep     movsb
```

Store

The stosb instruction moves the byte in register al to the address specified by rdi. The other variants move data from ax, eax or rax to memory. No flags are affected. A repeated store can fill an array with a single value. You could also use stosb in non-repeat loops taking advantage of the automatic destination register updating.

Here is some code to fill an array with 1000000 double words all equal to 1:

```
     mov     eax, 1
     mov     ecx, 1000000
     lea     rdi, [destination]
     rep     stosd
```

Load

The lodsb instruction moves the byte from the address specified by rsi to the al register. The other variants move more bytes of data into ax, eax or rax. No flags are affected. Repeated loading seems to be of little use. However you can use lods instructions in other loops taking advantage of the automatic source register updating.

Here is a loop which copies data from 1 array to another removing characters equal to 13:

```
       lea     rsi, [source]
       lea     rdi, [destination]
       mov     ecx, 1000000
more:  lodsb
       cmp     al, 13
       je      skip
       stosb
skip:  sub     ecx, 1
       jnz     more
end
```

Scan

The `scasb` instruction searches through an array looking for a byte matching the byte in `al`. It uses the `rdi` register. Here is an implementation of the C `strlen` function:

```
       segment .text
       global  strlen
strlen:
       cld              ; prepare to increment rdi
       mov     rcx, -1  ; maximum iterations
       xor     al, al   ; will scan for 0
       repne   scasb    ; repeatedly scan for 0
       mov     rax, -2  ; start at -1
                        ; end 1 past the end
       sub     rax, rcx
       ret
```

The function starts by setting `rcx` to -1, which would allow quite a long repeat loop since the code uses `repne` to loop. It would decrement `rcx` about 2^{64} times in order to reach 0. Memory would run out first.

It just so happens that the Linux C ABI places the first parameter to a function in `rdi`, so `strlen` starts with the proper address set for the scan. The standard way to return a value is to place it in `rax`, so we place the length there.

Compare

The `cmpsb` instruction compares values of 2 arrays. Typically it is used with `repe` which will continue to compare values until either the count in `ecx` reaches 0 or two different values are located. At this point the comparison is complete.

This is almost good enough to write a version of the C `strcmp` function, but `strcmp` expects strings terminated by 0 and lengths are not usually known for C strings. It is good enough for `memcmp`:

```
        segment .text
        global  memcmp
memcmp: mov     rcx, rdx
        repe    cmpsb    ; compare until end or difference
        cmp     rcx, 0
        jz      equal    ; reached the end
        movzx   eax, byte [rdi-1]
        movzx   ecx, byte [rsi-1]
```

```
        sub     rax, rcx
        ret
equal:  xor     eax, eax
        ret
```

In the memcmp function the repeat loop advances the rdi and rsi registers one too many times. Thus there is a -1 in the move and zero extend instructions to get the 2 bytes. Subtraction is sufficient since memcmp returns 0, a positive or a negative value. It was designed to be implemented with a subtraction yielding the return value. The first 2 parameters to memcmp are rdi and rsi with the proper order.

Set/clear direction

The clear direction cld instruction clears the direction flag to 0, which means to process increasing addresses with the string operations. The set direction std instruction sets the direction flag to 1. Programmers are supposed to clear the direction flag before exiting any function which sets it.