

## If statements

Stuck? Join our Discord for help! <https://discord.gg/wvfe3XJ>

The general format of an if statement is

```
if(condition)
{
    do-stuff-here
}
else if(condition) //this is an optional condition
{
    do-stuff-here
}
Else
{
    do-stuff-here
}
```

If statements use 3 important instructions in assembly:

- *cmpq source2, source1*: it is like computing a-b without setting destination
- *testq source2, source1*: it is like computing a&b without setting destination

Jump instructions are used to transfer control to different instructions, and there are different types of jumps:

| Jump Type | Description        |
|-----------|--------------------|
| jmp       | Unconditional      |
| je        | Equal/Zero         |
| jne       | Not Equal/Not Zero |
| js        | Negative           |

|     |                  |
|-----|------------------|
| jns | Nonnegative      |
| lg  | Greater          |
| jge | Greater or Equal |
| jl  | Less             |
| jle | Less or Equal    |
| ja  | Above(unsigned)  |
| jb  | Below(unsigned)  |
|     |                  |

The last 2 values of the table refer to unsigned integers. Unsigned integers cannot be negative while signed integers represent both positive and negative values. Since the computer needs to differentiate between them, it uses different methods to interpret these values. For signed integers, it uses something called the two's complement representation and for unsigned integers it uses normal binary calculations.

Start r2 with

*r2 -d if1*

**Remember to run**  
***e asm.syntax=att***

And run the following commands

*aaa*

*afi*

*pdf @main*

This analyses the program, lists the functions and disassembles the main function.

```
[0x7f374d371090]> pdf @main
/ (fcn) main 43
int main (int argc, char **argv, char **envp);
; var int32_t var_8h @ rbp-0x8
; var int32_t var_4h @ rbp-0x4
; DATA XREF from entry0 (0x55ae5283650d)
0x55ae528365fa 55          pushq %rbp
0x55ae528365fb 4889e5      movq %rsp, %rbp
0x55ae528365fe c745f8030000. movl $3, var_8h
0x55ae52836605 c745fc040000. movl $4, var_4h
0x55ae5283660c 8b45f8      movl var_8h, %eax
0x55ae5283660f 3b45fc      cmpl var_4h, %eax
; < 0x55ae52836612 7d06       jge 0x55ae5283661a
0x55ae52836614 8345f805   addl $5, var_8h
; == 0x55ae52836618 eb04       jmp 0x55ae5283661e
; -> 0x55ae5283661a 8345fc03   addl $3, var_4h
; ; CODE XREF from main (0x55ae52836618)
; -> 0x55ae5283661e b800000000 movl $0, %eax
0x55ae52836623 5d         popq %rbp
0x55ae52836624 c3         retq
```

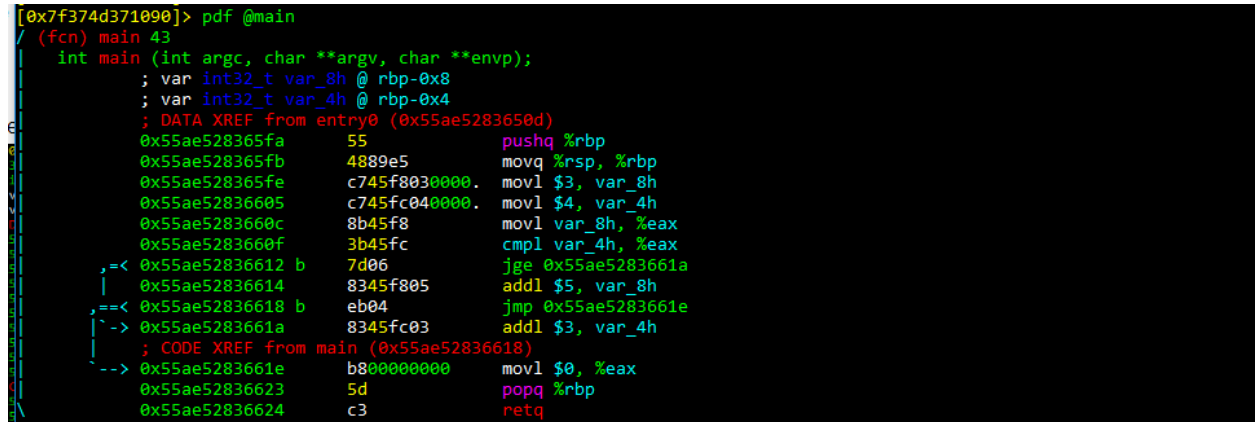
We'll then start by setting a break point on the jge and the jmp instruction by using the command:

`db 0x55ae52836612`(which is the hex address of the jge instruction)

`db 0x55ae52836618`(which is the hex address of the jmp instruction)

We've added breakpoints to stop the execution of the program at those points so we can see the state of the program

Doing so will show the following:



```
[0x7f374d371090]> pdf @main
(fcn) main 43
int main (int argc, char **argv, char **envp);
; var int32_t var_8h @ rbp-0x8
; var int32_t var_4h @ rbp-0x4
; DATA XREF from entry0 (0x55ae5283650d)
0x55ae528365fa 55          pushq %rbp
0x55ae528365fb 4889e5      movq %rsp, %rbp
0x55ae528365fe c745f8030000. movl $3, var_8h
0x55ae52836605 c745fc040000. movl $4, var_4h
0x55ae5283660c 8b45f8      movl var_8h, %eax
0x55ae5283660f 3b45fc      cmpl var_4h, %eax
;=< 0x55ae52836612 b 7d06      jge 0x55ae5283661a
| 0x55ae52836614 8345f805    addl $5, var_8h
;==< 0x55ae52836618 b eb04      jmp 0x55ae5283661e
|'-> 0x55ae5283661a 8345fc03    addl $3, var_4h
; CODE XREF from main (0x55ae52836618)
|'-> 0x55ae5283661e b800000000. movl $0, %eax
0x55ae52836623 5d          popq %rbp
0x55ae52836624 c3          retq
```

We now run `dc` to start execution of the program and the program will start execution and stop at the break point. Let's examine what has happened before hitting the breakpoint:

- The first 2 lines are about pushing the frame pointer onto the stacker and saving it(this is about how functions are called, and will be examined later)
- The next 3 lines are about assigning values 3 and 4 to the local arguments/variables `var_8h` and `var_4h`. It then stores the value in `var_8h` in the `%eax` register.
- The `cmpl` instruction compares the value of `eax` with that of the `var_8h` argument

To view the value of the registers, type in  
`dr`

```
[0x55ae52836612]> dr
rax = 0x00000003
rbx = 0x00000000
rcx = 0x55ae52836630
rdx = 0x7fff92f40058
r8 = 0x7f374d36bd80
r9 = 0x7f374d36bd80
r10 = 0x00000000
r11 = 0x00000000
r12 = 0x55ae528364f0
r13 = 0x7fff92f40040
r14 = 0x00000000
r15 = 0x00000000
rsi = 0x7fff92f40048
rdi = 0x00000001
rsp = 0x7fff92f3ff60
rbp = 0x7fff92f3ff60
rip = 0x55ae52836612
rflags = 0x00000297
orax = 0xffffffffffffffff
```

We can see that the value of rax, which is the 64 bit version of eax contains 3. We saw that the jge instruction is jumping based on whether value of eax is greater than var\_4h. To see what's in var\_4h, we can see that at the top of the main function, it tells us the position of var\_4h. Run the command:

`px @rbp-04x`

And that shows the value of 4.

We know that eax contains 3, and 3 is not greater than 4, so the jump will not execute. Instead it will move to the next instruction. To check this, run the `ds` command which seeks/moves onto the next instruction.

```
(fcn) main 43
int main (int argc, char **argv, char **envp);
; var int32_t var_8h @ rbp-0x8
; var int32_t var_4h @ rbp-0x4
; DATA XREF from entry0 (0x55ae5283650d)
0x55ae528365fa 55          pushq %rbp
0x55ae528365fb 4889e5      movq %rsp, %rbp
0x55ae528365fe c745f8030000. movl $3, var_8h
0x55ae52836605 c745fc040000. movl $4, var_4h
0x55ae5283660c 8b45f8      movl var_8h, %eax
0x55ae5283660f 3b45fc      cmpl var_4h, %eax
;=< 0x55ae52836612 b 7d06      jge 0x55ae5283661a
;-- rip:
| 0x55ae52836614 8345f805    addl $5, var_8h
;=< 0x55ae52836618 b eb04      jmp 0x55ae5283661e
|--> 0x55ae5283661a 8345fc03    addl $3, var_4h
; CODE XREF from main (0x55ae52836618)
--> 0x55ae5283661e b800000000. movl $0, %eax
0x55ae52836623 5d          popq %rbp
0x55ae52836624 c3          retq
```

The rip(which is the current instruction pointer) shows that it moves onto the next instruction - which shows we are correct. The current instruction then adds 5 to var\_8h which is a local argument. To see that this actually happens, first check the value of var\_8h, run `ds` and check the value again. This will show it increments by 5.

```

[0x55ae52836612]> px @rbp-0x8
   offset  0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7fff92f3ff58 0300 0000 0400 0000 3066 8352 ae55 0000 .....0f.R.U..
0x7fff92f3ff68 970b fa4c 377f 0000 0100 0000 0000 ....L7.....
0x7fff92f3ff78 4800 f492 ff7f 0000 0080 0000 0100 0000 H.....
0x7fff92f3ff88 fa65 8352 ae55 0000 0000 0000 0000 .e.R.U.....
0x7fff92f3ff98 976f 608c 8d2f efd6 f064 8352 ae55 0000 .o'.....d.R.U..
0x7fff92f3ffa8 4000 f492 ff7f 0000 0000 0000 0000 @.....
0x7fff92f3ffb8 0000 0000 0000 0000 976f e0be 6caf 4c8b .....o..l.L.
0x7fff92f3ffc8 976f 9e56 7f13 dd8a 0000 0000 ff7f 0000 .o.V.....
0x7fff92f3ffd8 0000 0000 0000 0000 0000 0000 0000 .....
0x7fff92f3ffe8 3307 384d 377f 0000 3866 364d 377f 0000 3.8M7...8f6M7...
0x7fff92f3fff8 3161 0700 0000 0000 0000 0000 0000 0000 1a.....
0x7fff92f40008 0000 0000 0000 0000 0000 0000 0000 .....
0x7fff92f40018 f064 8352 ae55 0000 4000 f492 ff7f 0000 .d.R.U..@.....
0x7fff92f40028 1a65 8352 ae55 0000 3800 f492 ff7f 0000 .e.R.U..8.....
0x7fff92f40038 1c00 0000 0000 0000 0100 0000 0000 .....
0x7fff92f40048 9417 f492 ff7f 0000 0000 0000 0000 .....
[0x55ae52836612]> ds
[0x55ae52836612]> px @rbp-0x8
   offset  0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7fff92f3ff58 0300 0000 0400 0000 3066 8352 ae55 0000 .....0f.R.U..
0x7fff92f3ff68 970b fa4c 377f 0000 0100 0000 0000 ....L7.....
0x7fff92f3ff78 4800 f492 ff7f 0000 0080 0000 0100 0000 H.....
0x7fff92f3ff88 fa65 8352 ae55 0000 0000 0000 0000 .e.R.U.....
0x7fff92f3ff98 976f 608c 8d2f efd6 f064 8352 ae55 0000 .o'.....d.R.U..
0x7fff92f3ffa8 4000 f492 ff7f 0000 0000 0000 0000 @.....
0x7fff92f3ffb8 0000 0000 0000 0000 976f e0be 6caf 4c8b .....o..l.L.
0x7fff92f3ffc8 976f 9e56 7f13 dd8a 0000 0000 ff7f 0000 .o.V.....
0x7fff92f3ffd8 0000 0000 0000 0000 0000 0000 0000 .....
0x7fff92f3ffe8 3307 384d 377f 0000 3866 364d 377f 0000 3.8M7...8f6M7...
0x7fff92f3fff8 3161 0700 0000 0000 0000 0000 0000 0000 1a.....
0x7fff92f40008 0000 0000 0000 0000 0000 0000 0000 .....
0x7fff92f40018 f064 8352 ae55 0000 4000 f492 ff7f 0000 .d.R.U..@.....
0x7fff92f40028 1a65 8352 ae55 0000 3800 f492 ff7f 0000 .e.R.U..8.....
0x7fff92f40038 1c00 0000 0000 0000 0100 0000 0000 .....
0x7fff92f40048 9417 f492 ff7f 0000 0000 0000 0000 .....

```

Note that because we are checking the exact address, we only need to check to 0 offset. The value stored in memory is stored as hex.

The next instruction is an unconditional jump and it just jumps to clearing the eax register. The popq instruction involves popping a value of the stack and reading it, and the return instruction sets this popped value to the current instruction pointer. In this case, it shows the execution of the program has been completed. To understand better about how an if statement work, you can check the corresponding C file in the same folder.