Assembly Leaning

cmpq %rbx, %r12 ; %r12 - %rbx
cmp = compare
q = quad word (4*16) = 64 bits
%rbx = source operand
%r12 = destination operand

It is equivalent of %r12 - %rbx but doesn't store the result. Instead, it sets the CPU flags (ZF, SF, CF etc.) based on the result so that a subsequent conditional jump (like je, jne, jl, jg, etc.) can act accordingly.
SF: Sign Flag (for signed numbers)
ZF: Zero Flag
CF: Carry Flag (not negative numbers i.e., unsigned)
OF: Overflow Flag (signed)

These flags are stored inside FLAGS registers (i.e., EFLAGS for 32-bit and RFLAGS for 64 bit). These signed flag are called bit fields.

If destination is greater/equal/smaller than source

je:jump if equal (or jz: jump if zero)
jne: jump if not equal (or jnz)
jg: jump if greater (or jnle: jump if not less or equal) – same thing
jge: jump if greater or equal (jnl: jump if not less)
jl: jump if less (jnge)
jle: jump if less or equal (jng)

ja or jnbe — Jump if above (unsigned >)
jae or jnb — Jump if above or equal (unsigned >=)
jb or jnae — Jump if below (unsigned <)
jbe or jna — Jump if below or equal (unsigned <=)

Signed integers: Represent both positive and negative values using **two's complement. This is why there different instuctions for signed and different for unsigned.**

```
for(j=0;j<100000;j++){
      i = j % 8;
            time2 = rdtsc();
            while(time2 < time3) time2 = rdtsc();
            // sender's loop, changed in send.s, lines 626 to 787
            if(i == 0) {
            for(z = 0; z < 100; z++){

            }
      } else if(i == 1) {
            for(z = 0; z < 100; z++){

            }
      } else if(i == 2) {
            for(z = 0; z < 100; z++){

            }
      }else if(i == 3) {
            for(z = 0; z < 100; z++){

            }
      }else if(i == 4) {
            for(z = 0; z < 100; z++){

            }
      }else if(i == 5) {
            for(z = 0; z < 100; z++){

            }
      }else if(i == 6) {
            for(z = 0; z < 100; z++){

            }
      } else {
            for(z = 0; z < 100; z++){

            }
      }
      time3 += 7000;
      }
```

It checks if i==0, if true then jumps to for(z = 0; z < 100; z++){}, if not true then jumps to else if(i == 1). Same logic continues. In case of true, there always happens 100 iterations.

```
.L39:
      cmpq  %rbx, %r12
      jb    .L40
      cmpl  $0, -164(%rbp); if i==0
      jne   .L41; jump to i==1 if i==0 false


      movl  $0, %eax; eax = 0
      jmp   .L42
      .align     32
.L42:
      cmpl  $329, %eax
      jle   .L43; jump to loop
      jmp   .L44; exit loop
.L43:; if eax<=329, eax starts from 0
      adcl  $1, -180(%rbp)
      addl  $1, %eax
.L44:
      addq  $7000, %rbx  ;time3 += 7000;
      addl  $1, -184(%rbp); increase j
```

Actually, the red above is patched (i.e., modified). If it was non-patched that would be,

```
      movl    $0, -180(%rbp); for loop logic from here; z=0; -180(%rbp) holds z
      jmp     .L42
.L42:
      cmpl  $99, -180(%rbp)          ; compare i with 99
      jle   .L43                     ; if i <= 99, jump back to .L43
      jmp   .L44                     ; else jump to .L44; for loop logic ends here
```

As, you can see above, instead of 100 iterations, it modifed to 330 iterations. Then it does time3 = time3+7000 also increments j. This is done when i==0.

It means when unmodified, there is 180 iterations and time3 = time3+7000 also increments j, but when modified there are 330 iterations and time3 = time3+7000 also increments j.

when i==1,
The modified jumps to .L46 and iterates 160 times and does time3 = time3+7000 also increments j

"time3 = time3+7000 also increments j" is .L44

when i==2, it iterates 400 times, doing nothing inside the loop. In each iteration, it executes 10 nops. And finally does .L44 again.

When i==3, it runs 30 iterations. In each iteration, it swaps a register with stack memory, increments stack memory at offset -180, and increments Z. This is costly operations (so we will have observable side effects)

when i==4, it executes as original code.

when i==4, 5, 6 or else, it behaves original.

It means, in the patched (i.e., modfied) we have exact same behavior when i==0/4/5/6/7, and we have different behaviors when i=2/3/4. It means there are 4 different timing behaviors. This is expected.

4 Unique Timing Behaviors:

1. 100 iterations (for i == 0, i == 4, i == 5, i == 6, i > 6).
2. 180 iterations (for i == 1).
3. 400 iterations with 10 NOPs each (for i == 2).
4. 30 iterations (for i == 3).

The sender can modify its execution behavior to generate different time gaps based on its internal logic, and the receiver can monitor these time differences to infer information.

Receiver detects when i==1, the physical core behaves this timing behavior and so on. This is information leakage.

explicit `.align 32` directives so that the loop bodies always start on cache-line boundaries

They also modified receiver exactly like they did for sender (exact iteration counts for each timing case (100, 180, 400+NOP, 30))

guaranteed NOP padding in the “else” timing bucket
—all of which ensure that you *only* measure the sender’s induced retire-unit contention and not any extra noise from the compiler’s code layout.

Custom Data Storage & Threshold Logic

The patched `receive.s` allocates a big array in `.bss` (`control`, or the `-8000032(%rbp)` buffer you saw) and then:

- uses `rdtsc_begin`/`end` to stamp each message,
- subtracts the timestamps and stores each delta in `-8000032(%rbp,%rax,8)` indexed by the message counter,

- loops exactly 100 000 times and then iterates over that array to `printf` only the values above a chosen threshold.
  You **cannot** write that kind of fixed-offset, pointer-arithmetic, unrolled data-collection loop in portable C without risking the compiler inserting prologue/epilogue code in the middle, reordering instructions, or otherwise disturbing the timing measurements.

Compiler Optimizations Would Break the Channel

If they'd left it in C, the compiler might:

- inline or unroll loops differently based on optimization level,
- introduce extra loads/stores around the timing code,
- move the `rdtsc` calls, or
- mis-align the data buffer,
  making the four distinct timing patterns **undetectable** (or unstable) on the receiver side.