

Intel's 'cmpxchg' instruction

Intel's documentation

- You can find out what any of the Intel x86 instructions does by consulting the official software developer's manual, online at:

<http://www.intel.com/products/processor/manuals/index.htm>

- Our course-webpage has a link to this site that you can just click (under '**Resources**')
- The instruction-set reference is two parts:
 - Volume 2A: for opcodes A through M
 - Volume 2B: for opcodes N through Z

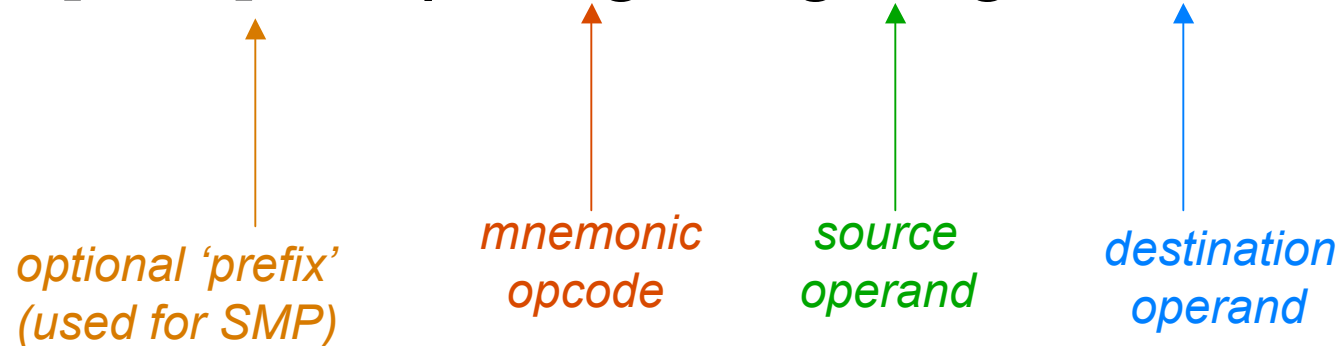
Example: 'cmpxchg'

- Operation of the 'cmpxchg' instruction is described (on 3 pages) in Volume 2A
- There's an English-sentence description, and also a description in 'pseudo-code'
- You probably do not want to print out this complete volume (.pdf) – over 700 pages!
- (You could order a printed copy from Intel)

Instruction format

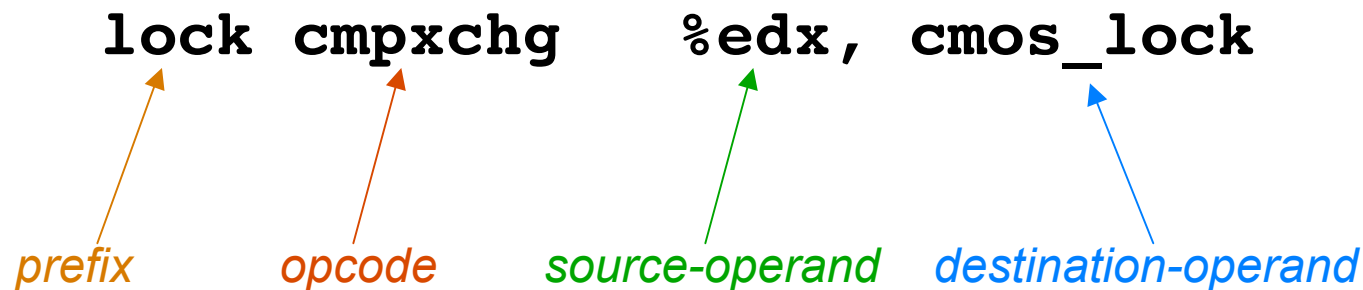
- Intel's assembly language syntax differs from the GNU/Linux syntax (known as 'AT&T syntax' with roots in UNIX history)
- When AT&T syntax is used, the 'cmpxchg' instruction has this layout:

[lock] cmpxchg reg, reg/mem



An instruction-instance

- In our recent disassembly of Linux's kernel function 'rtc_cmos_read()', this 'cmpxchg' instruction-instance was used:



Note: Keep in mind that the accumulator `%eax` will affect what happens!
So we need to consider this instruction within its surrounding context

'effects' and 'affects'

- According to Intel's manual, the 'cmpxchg' instruction also uses two '**implicit**' operands (i.e., operands not mentioned in the instruction)
 - The CPU's accumulator register
 - The CPU's EFLAGS register
- The accumulator-register (EAX) is both a source-operand and a destination-operand
- The six status-bits in the EFLAGS register will get modified, as a 'side-effect' this instruction

'cmpxchg' description

- This instruction compares the accumulator with the destination-operand (so the ZF-bit in EFLAGS gets assigned accordingly)
- Then:
 - If (accumulator == destination)
{ ZF \leftarrow 1; destination \leftarrow source; }
 - If (accumulator != destination)
{ ZF \leftarrow 0; accumulator \leftarrow destination; }

The 'busy-wait' loop

```
# Here is a 'busy-wait' loop, used to wait for the CMOS access to be 'unlocked'  
spin:  mov     cmos_lock, %eax           # copy lock-variable to accumulator  
      test   %eax, %eax               # was CMOS access 'unlocked'?  
      jnz   spin                     # if it wasn't, then check it again
```

```
# A CPU will fall through to here if 'unlocked' access was detected,  
# and that CPU will now attempt to set the 'lock' – in other words, it  
# will try to assign a non-zero value to the 'cmos_lock' variable.
```

```
# But there's a potential 'race' here – the 'cmos_lock' might have been  
# zero when it was copied, but it could have been changed by now...  
# ... and that's why we need to execute 'lock cmpxchg' at this point
```


Busy-waiting will be brief

```
spin:    # see if the lock-variable is clear
         mov     cmos_lock, %eax
         test    %eax, %eax
         jnz     spin

         # ok, now we try to grab the lock
         lock cmpxchg %edx, cmos_lock

         # did another CPU grab it first?
         test    %eax, %eax
         jnz     spin
```

If our CPU wins the 'race', the (non-zero) value from source-operand EDX will have been stored into the (previously zero) 'cmos_lock' memory-location, but the (previously zero) accumulator EAX will not have been modified; hence our CPU will not jump back, but will fall through and execute the 'critical section' of code (just a few instructions), then will promptly clear the 'cmos_lock' variable.

The 'less likely' case

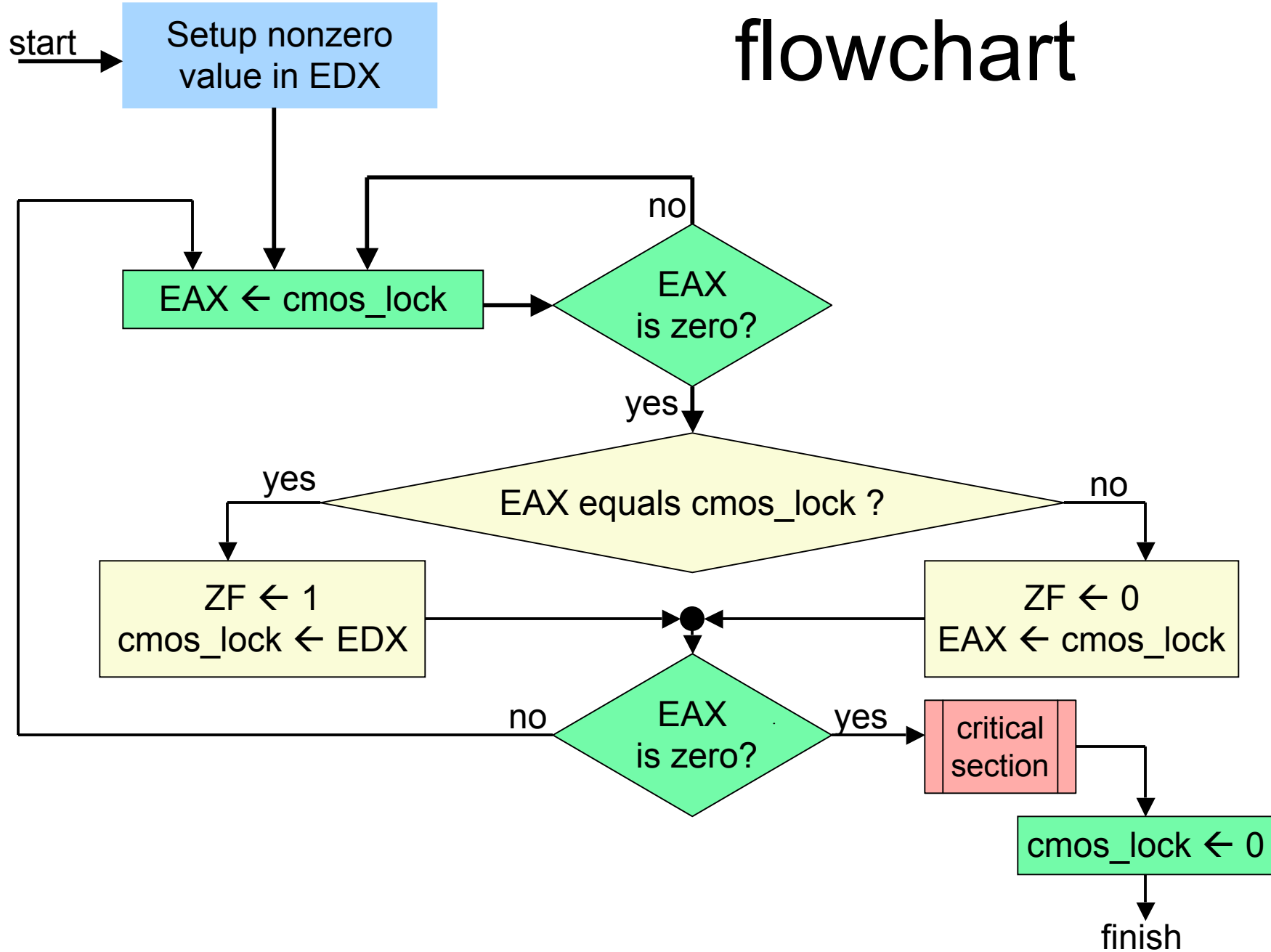
```
spin:    # see if the lock-variable is clear
         mov     cmos_lock, %eax
         test   %eax, %eax
         jnz    spin

         # ok, now we try to grab the lock
         lock  cmpxchg %edx, cmos_lock

         # did another CPU grab it first?
         test   %eax, %eax
         jnz    spin
```

If our CPU loses the 'race', because another CPU changed 'cmos_lock' to some non-zero value after we had fetched our copy of it, then the (now non-zero) value from the 'cmos_lock' destination-operand will have been copied into EAX, and so the final conditional-jump shown above will take our CPU back into the spin-loop, where it will resume busy-waiting until the 'winner' of the race clears 'cmos_lock'.

flowchart




The 'cmos_lock' variable

- This global variable is initialized to zero, meaning that access to CMOS memory locations is not currently 'locked'
- If some CPU stores a non-zero value in this variable's memory-location, it means that access to CMOS memory is 'locked'
- The kernel needs to insure that only one CPU at a time can set this 'lock'

How often is 'cmpxchg' used?

```
$ cat vmlinux.asm | grep cmpxchg
```

```
c01046de:      f0 0f b1 15 3c 99 30      lock cmpxchg %edx,0xc030993c
c0105591:      f0 0f b1 15 3c 99 30      lock cmpxchg %edx,0xc030993c
c01055d9:      f0 0f b1 15 3c 99 30      lock cmpxchg %edx,0xc030993c
c010b895:      f0 0f b1 11                lock cmpxchg %edx,(%ecx)
c010b949:      f0 0f b1 0b                lock cmpxchg %ecx,(%ebx)
c0129a9f:      f0 0f b1 0b                lock cmpxchg %ecx,(%ebx)
c0129acf:      f0 0f b1 0b                lock cmpxchg %ecx,(%ebx)
c012d377:      f0 0f b1 0e                lock cmpxchg %ecx,(%esi)
c012d41a:      f0 0f b1 0e                lock cmpxchg %ecx,(%esi)
c012d968:      f0 0f b1 16                lock cmpxchg %edx,(%esi)
c012e568:      f0 0f b1 2e                lock cmpxchg %ebp,(%esi)
c012e57a:      f0 0f b1 2e                lock cmpxchg %ebp,(%esi)
c012e58a:      f0 0f b1 2e                lock cmpxchg %ebp,(%esi)
c012e83f:      f0 0f b1 13                lock cmpxchg %edx,(%ebx)
c012e931:      f0 0f b1 0a                lock cmpxchg %ecx,(%edx)
c012ea94:      f0 0f b1 11                lock cmpxchg %edx,(%ecx)
c012ecf4:      f0 0f b1 13                lock cmpxchg %edx,(%ebx)
c012f08e:      f0 0f b1 4b 18            lock cmpxchg %ecx,0x18(%ebx)
c012f163:      f0 0f b1 11                lock cmpxchg %edx,(%ecx)
c013cb60:      f0 0f b1 0e                lock cmpxchg %ecx,(%esi)
c0148b3c:      f0 0f b1 29                lock cmpxchg %ebp,(%ecx)
c0150d0f:      f0 0f b1 3b                lock cmpxchg %edi,(%ebx)
c0150d87:      f0 0f b1 31                lock cmpxchg %esi,(%ecx)
c0199c5e:      f0 0f b1 0b                lock cmpxchg %ecx,(%ebx)
c024b06f:      f0 0f b1 0b                lock cmpxchg %ecx,(%ebx)
c024b2fe:      f0 0f b1 51 18            lock cmpxchg %edx,0x18(%ecx)
c024b321:      f0 0f b1 51 18            lock cmpxchg %edx,0x18(%ecx)
c024b34b:      f0 0f b1 4b 18            lock cmpxchg %ecx,0x18(%ebx)
c024b960:      f0 0f b1 53 18            lock cmpxchg %edx,0x18(%ebx)
```

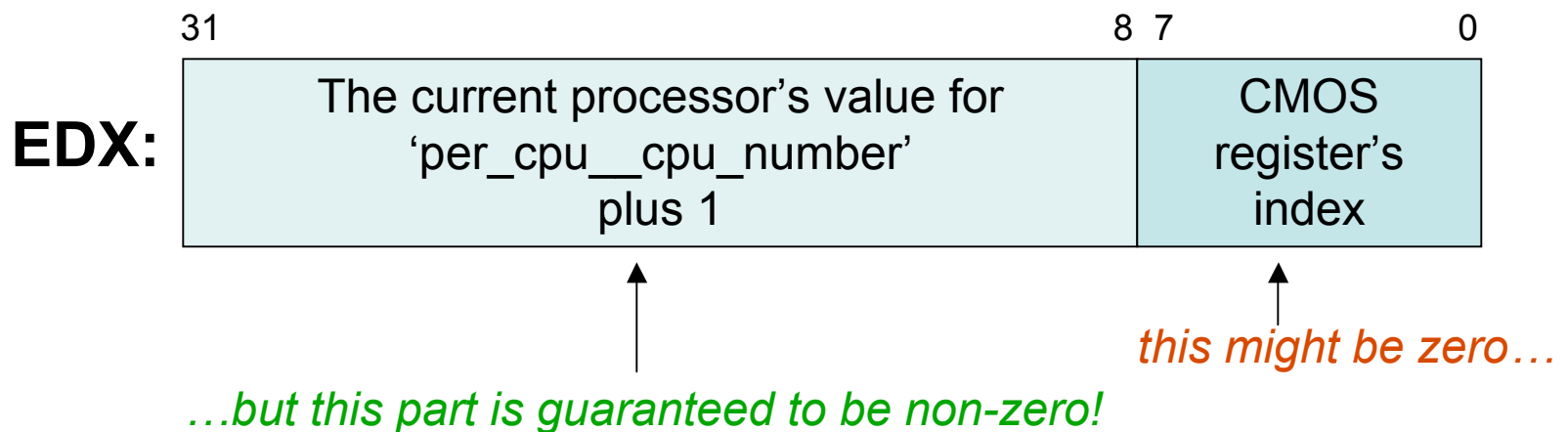


*Here's the occurrence
that we studied in the
'rtc_cmos_read()'
kernel-function...*

...plus 28 other times!

The 'preparation' steps

- The instructions that precede 'cmpxchg' will setup register EDX (source operand) and register EAX (the x86 'accumulator')
- Several instructions are used to set up a value in EDX, and result in this layout:



The 'most likely' senario

- One of the CPUs wishes to access CMOS memory – so it needs to test 'cmos_lock' to be sure that access is now 'unlocked' (i.e., `cmos_lock == 0` is true)
- The CPU copies the 'cmos_lock' variable into the EAX, where it can then be tested using the 'test %eax, %eax' instruction
- A conditional-jump follows the test

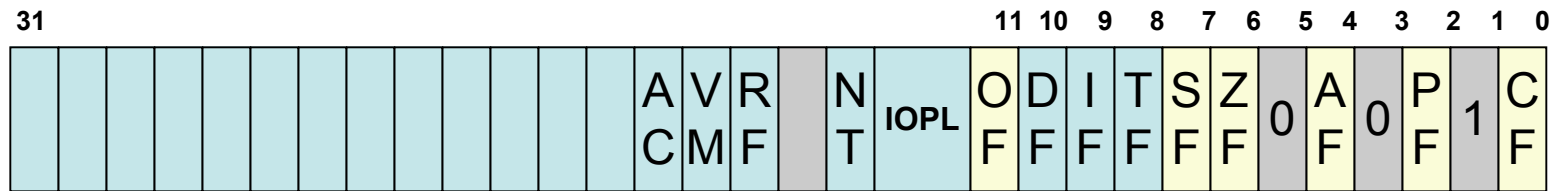
'btr'/'bts' versus 'cmpxchg'

- In an earlier lesson we used the 'btr'/'bts' instructions to achieve 'mutual exclusion', whereas Linux uses 'cmpxchg' to do that
- We think 'btr'/'bts' is easier to understand, so why do you think the Linux developers would prefer to use 'cmpxchg' instead?

<allow some class discussion here>

EFLAGS

- The Intel documentation does not state precisely how other EFLAGS status-bits (besides ZF) are affected by 'cmpxchg', only that they reflect the comparison of 'accumulator' and 'destination' operands



- Usually the CPU implements comparison-of-operands by performing a subtraction