GNU/Linux assembly language

Reviewing the basics of assembly language programming for Intel x86-based Linux systems

Our motivations

- We want to study the new Intel processor technologies for Core-2 Duo & Pentium-D:
  - EM64T (Extended Memory 64-bit technology)
  - VT (Virtualization Technology)
- These capabilities are optionally ‘enabled’ during the system’s ‘startup’ phase, so we will want to execute our own ‘boot loader’ code to have control at the earliest stage

‘boot’ code

- On PC systems the mechanism for doing the IPL (‘Initial Program Load’) requires a ‘boot-loader’ program that can fit within a single 512-byte disk-sector
- High-level programming languages (such C/C++) typically employ compilers which generate object-code files whose size is too large to be stored in a disk-sector

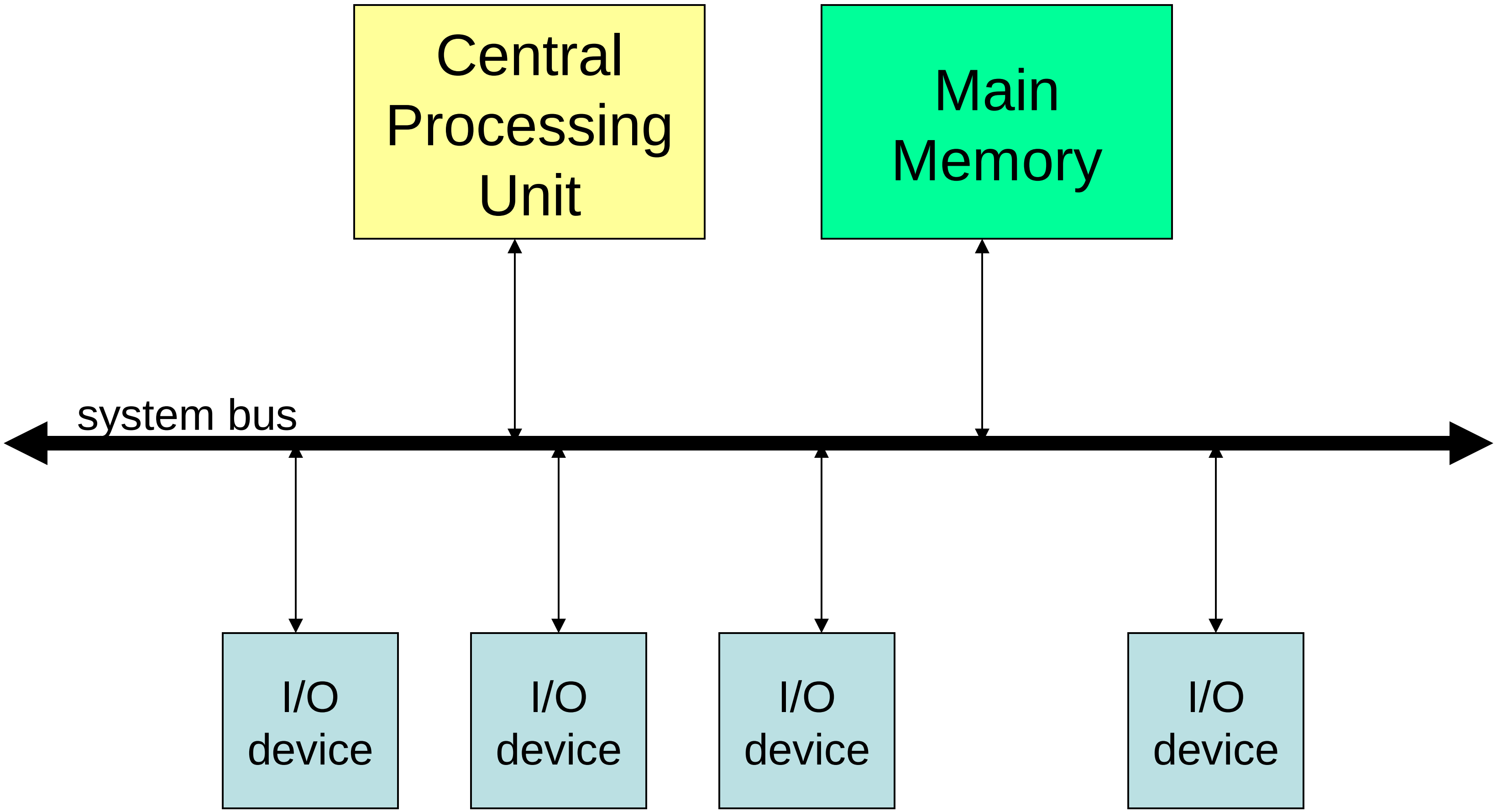
Portability implications

- Also ‘High-Level’ languages typically have platform ‘portability’ as a key design-goal, but they can’t achieve that requirement if architecture-specific language-constructs are employed; thus they usually offer to a systems programmer only those features in the ‘lowest common denominator’ of the various different computer-systems

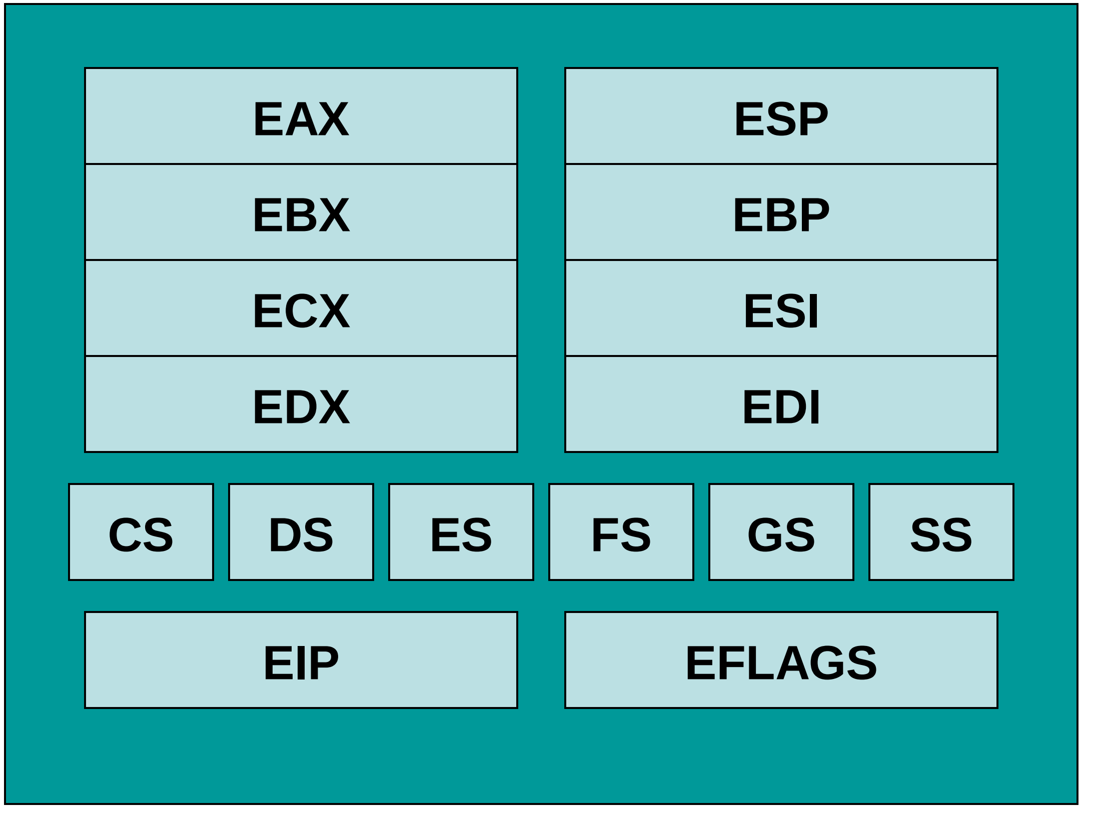
Intel’s VMX instructions

- So when we want to explore Intel’s VM-x technology (Virtualization for x86 CPUs), we will need to use ‘low-level’ computer language (i.e., assembly language)
- The advantage, of course, is that we will thereby acquire total control over the CPU in any Intel-based computer that supports the new so-called VMX instruction-set.

Simplified Block Diagram



The sixteen x86 registers



Intel Pentium processor

The x86 EFLAGS register

31–22	21	20	19	18	17	16	15	14	13-12	11	10	9	8	7	6	5	4	3	2	1	0
0000000000	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Legend:

IOPL = I/O Privilege-Level (0,1,2,3)
AC = Alignment-Check (1=yes, 0=no)
NT = Nested-Task (1=yes, 0=no)
RF = Resume Flag (1=yes, 0=no)
VM = Virtual-8086 Mode (1=yes, 0=no)
VIF = ‘Virtual’ Interrupt-Flag
VIP = ‘Virtual’ Interrupt is Pending
ID = the CPUID-instruction is implemented if this bit can be ‘toggled’

ZF = Zero Flag
SF = Sign Flag
CF = Carry Flag
PF = Parity Flag
OF = Overflow Flag
AF = Auxiliary Flag

= ‘reserved’ bit

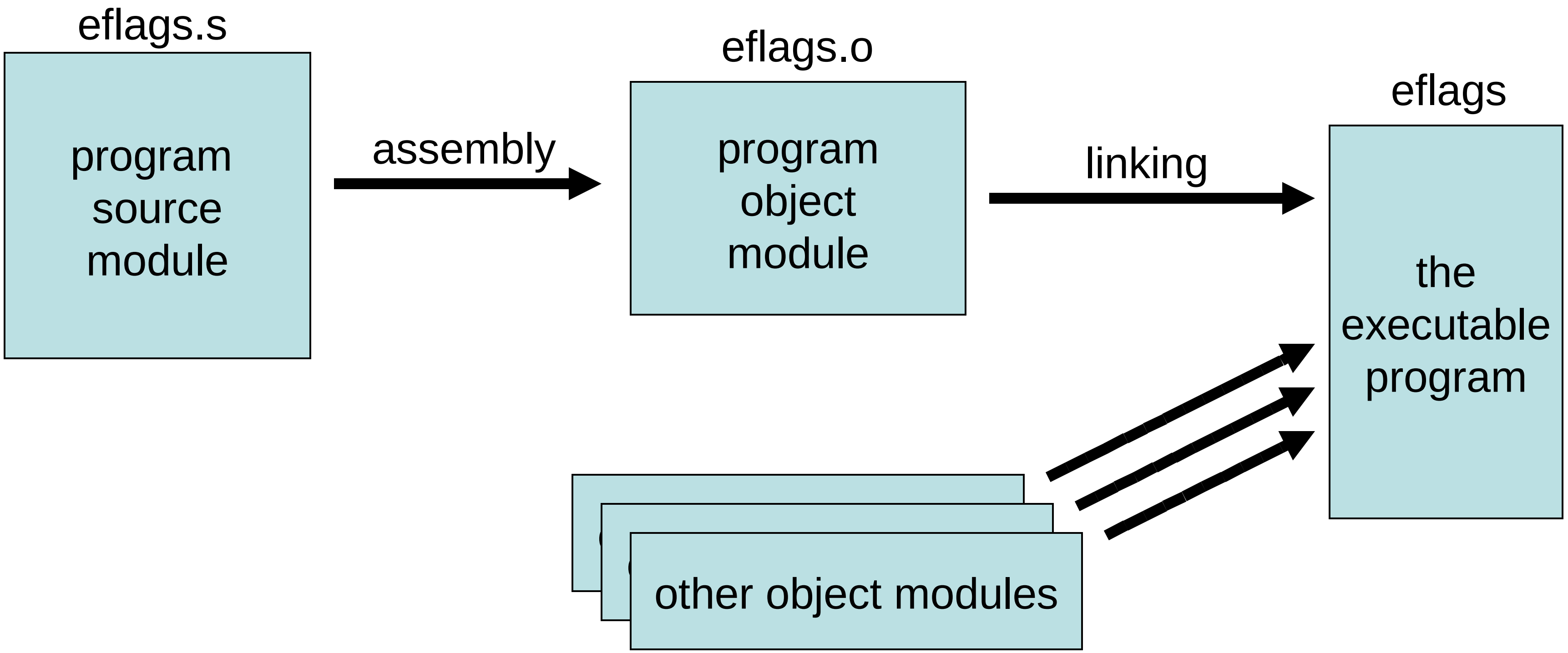
= ‘status’ flag (for application programming)

Our ‘eflags.s’ demo

- This program shows the EFLAGS bits at the moment the program began executing

- It’s instructive to run this program *before* you execute our ‘iopl3’ command, and do so again *after* you execute that command

program translation steps



Not every program requires additional object-modules

program translation commands

- To ‘assemble’ your source-file:

```
$ as  eflags.s  –o eflags.o
```

- To ‘link’ the resulting object-file:

```
$ ld  eflags.o  –o eflags
```

- To ‘run’ your executable program:

```
$ ./eflags
```

Last night (Wed, 24 Jan 2007) Alex Fedosov wrote:
>
> Subject: Re: hardware
>
> I checked #11-#12 serial cable and it seems that it was
> loose. Let me know if it continues to be a problem.
>
> Good news! Your machines are here! We will be building
> them tomorrow.
>
> -a

Our ‘feedback.s’ demo

- We wrote an example-program that shows how you could program the serial UART in the GNU/Linux assembly language
- It uses the x86 ‘in’ and ‘out’ instructions to access the UART’s i/o-ports:
  – 0x03F8: port-address for Divisor-Latch
  – 0x03FB: port-address for Line-Control
  – 0x03FD: port-address for Line-Status
  – 0x03F8: port-address for RxD and TxD

Statement layout

label:	opcode	operand(s)	# comment

Parsing an assembly language statement

- A colon character (‘:’) terminates the label
- A white-space character (e.g., blank or tab) separates the opcode from its operand(s)
- A hash character (‘#’) begins the comment

How ‘outb’ works

- Three-steps to perform ‘outb(data, port);’
  - Step 1: put port-address into DX register
  - Step 2: put data-value into AL register
  - Step 3: output data-value to port-address

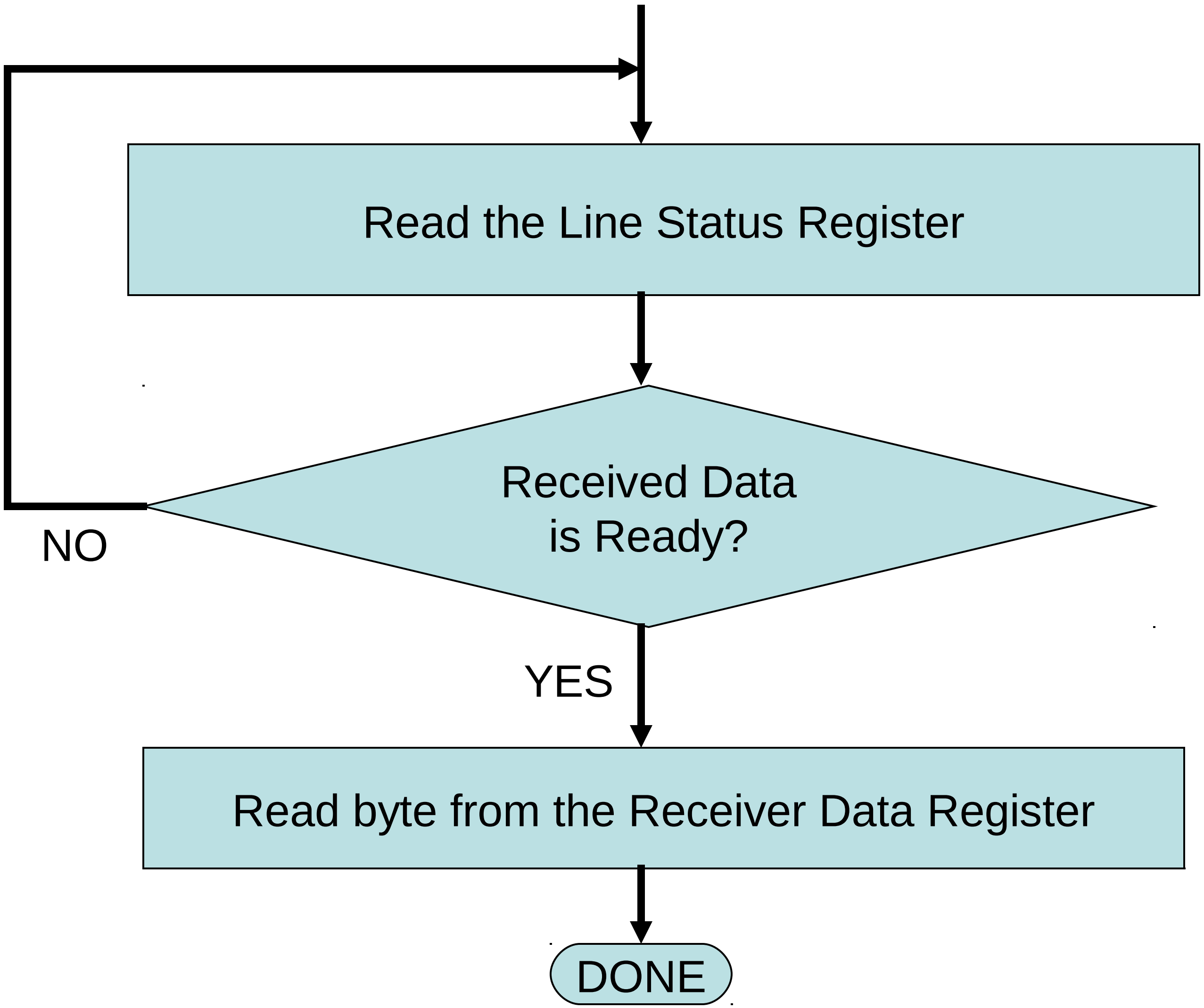
```
mov     $0x03FB, %dx      # UART’s Line_Control register
mov     $0x80, %al        # value for setting the DLAB bit
out     %al, %dx          # output data-byte to the i/o-port
```

How ‘in’ works

- Three steps to perform ‘data = in(port);’
  - Step 1: put port-address into DX register
  - Step 2: input from port-address to AL register
  - Step 3: assign value in AL to location ‘data’

```
mov     $0x03FD, %dx     # UART’s Line_Status register
in      %dx, %al         # input from i/o-port to accumulator
mov     %al, data        # copy accumulator-value to variable
```

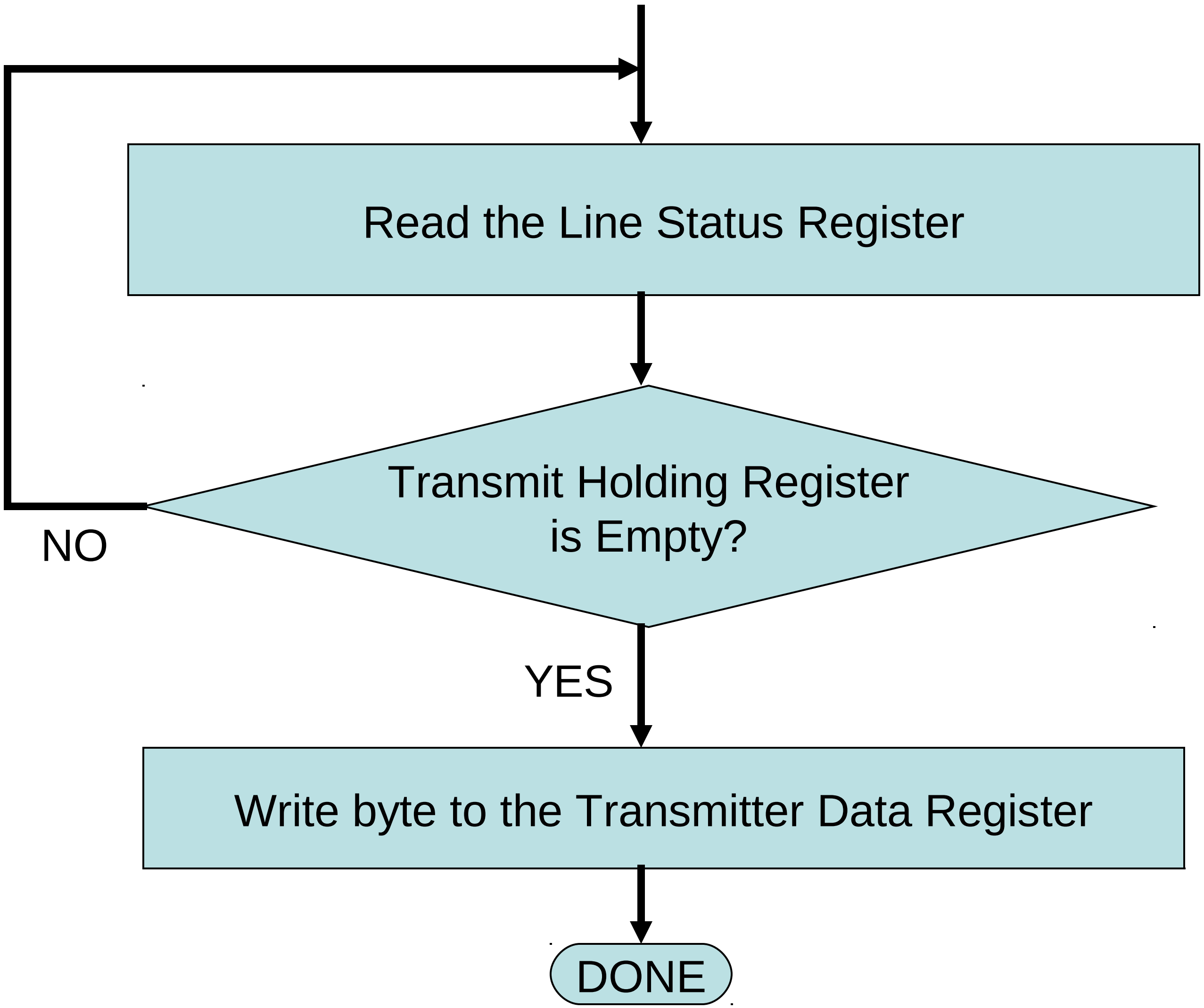
How to receive a byte



Rx implementation

```
# This assembly language code-fragment inputs a byte of data from
# the remote PC, then assigns it to a memory-location labeled ‘inch’

        mov     $0x03FD, %dx    # UART’s Line_Status register
spin1:  in      %dx, %al        # input the current line-status
        test    $0x01, %al      # is the RDR-bit zero?
        jz      spin1           # yes, no new data received

        mov     $0x03F8, %dx    # UART’s RxData register
        in      %dx, %al        # input the new data-byte
        mov     %al, inch       # store the data into ‘inch’
```

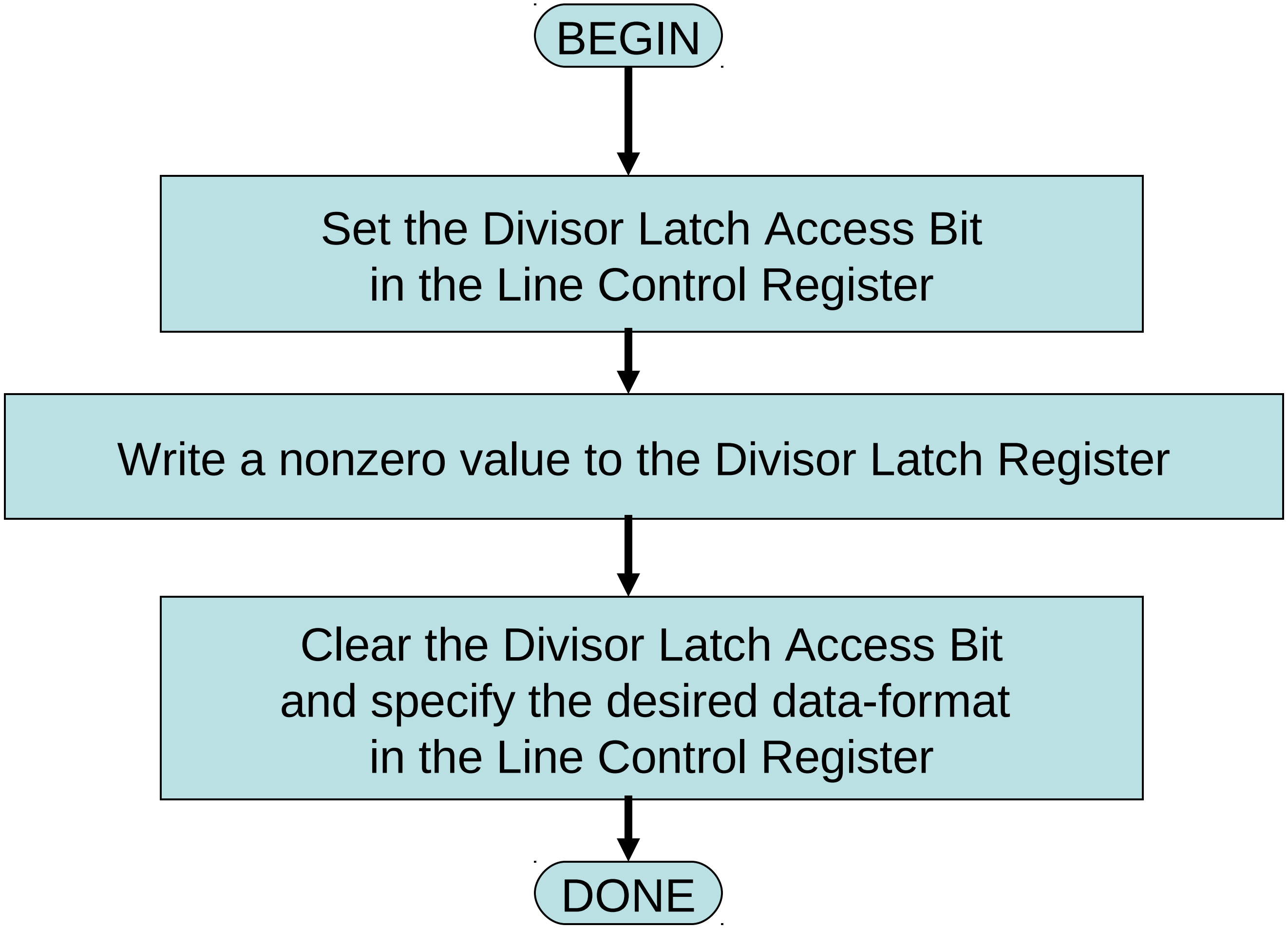
How to transmit a byte



Tx implementation

```
# This assembly language code-fragment fetches a byte of data from
# a memory-location labeled 'outch', then outputs it to the remote PC

        mov     $0x03FD, %dx    # UART's Line_Status register
spin2:  in      %dx, %al        # input the current line-status
        test    $0x20, %al      # is the THRE-bit zero?
        jz      spin2           # yes, transmitter is still busy

        mov     $0x03F8, %dx    # UART's TxData register
        mov     outch, %al      # get the value to transmit
        out     %al, %dx        # output byte to TxD register
```

Initializing the UART



Init-UART implementation

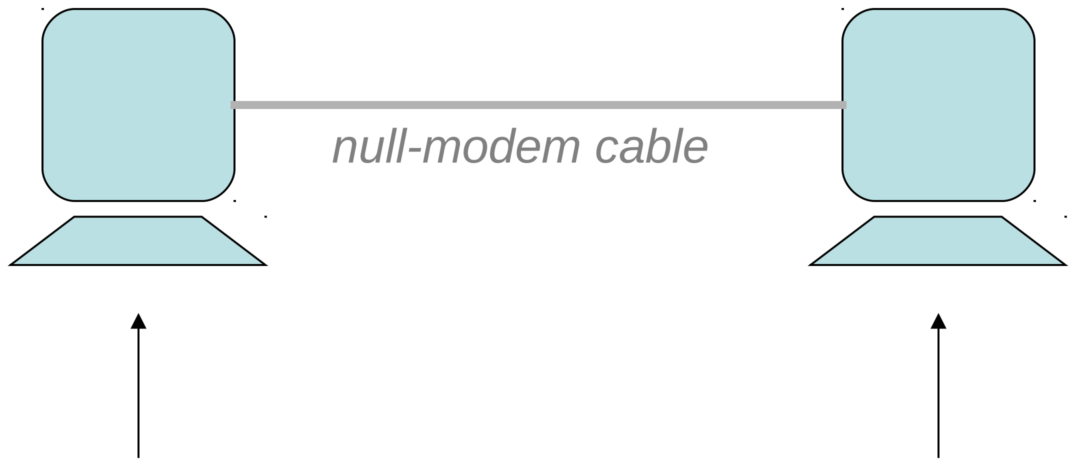
```
# This assembly language code-fragment initializes the UART for
# 'polled-mode' operation at 115200 baud and 8-N-1 data-format

        mov     $0x03FB, %dx    # UART Line_Control register
        mov     $0x80, %al      # set the DLAB-bit (bit 7) to 1
        out     %al, %dx        # for access to Divisor_Latch

        mov     $0x03F8, %dx    # UART Divisor_Latch register
        mov     $0x0001, %ax    # use 1 as the divisor-value
        out     %ax, %dx        # output the 16-bit latch-value

        mov     $0x03FB, %dx    # UART Line_Control register
        mov     $0x03, %al      # set data-format to 8-N-1
        out     %al, %dx        # establish UART data-format
```

In-class exercises

- You can try assembling, linking, and then running our ‘feedback.s’ demo-program (use the ‘trycable’ program to send some characters via the ‘null-modem’ cable)

- Can you modify this ‘feedback.s’ program so that it will continuously display and transmit every character it receives?

Exercise illustration



Start our ‘feedback’ program running on this workstation

Then run our ‘trycable’ program on this other workstation

Now watch what happened on these two screens