# Homework

- Reading (linked from my web page)
  - S and S Extracts
  - National Semiconductor UART Data Sheet
- Machine Projects
  - mp2 due at start of class 12
- Labs
  - Continue labs in your assigned section

# Addressing I/O Devices

- Intel I/O devices have addresses assigned in an “orthogonal” space from memory addresses
  - Remember the M/IO# signal that is used with the address bus to select memory versus I/O devices?
- Use I/O instructions for I/O device addresses

```
inw        inb
outw       outb
```

# Addressing I/O Devices

- The “input” instruction – direct addressing

```
inw $0xdd, %ax    #  8 bit address
inb $0xdd, %al    #  8 bit address
```

- The “input” instruction – indirect addressing

```
movw $0x3f8, %dx
inw (%dx), %ax    # 16 bit address
inb (%dx), %al    # 16 bit address
```

- Reads from an I/O device to a register

# Addressing I/O Devices

- The “output” instruction – direct addressing

```
outw %ax, $0xdd  #  8 bit address
outb %al, $0xdd  #  8 bit address
```

- The “output” instruction – indirect addressing

```
movw $0x3f8, %dx
outw %ax, (%dx)  # 16 bit address
outb %al, (%dx)  # 16 bit address
```

- Writes from a register to an I/O device

# Addressing I/O Devices

- In some processor architectures (Motorola 68xxx and Arduino ATMEGA), there are no M/IO# signal(s) in the control bus or special `in` and `out` instructions
- This is called using "memory mapped I/O"
  - I/O device registers are accessed in the same address space as memory locations
  - In assembly, use equivalent of "move" instructions to write or read data to or from I/O device registers like memory
  - In C, dereference pointers to write or read data to or from I/O device registers like memory

# Accessing the Serial Port

- PC specification allows up to four serial ports
  - COM1: base address is 0x3f8
  - COM2: base address is 0x2f8

| | Write | | | | | | | | Read | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0x3f8** | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| **0x3fb** | DLAB | Set Brk | Stk Par | Evn Par | Par Enb | # Stop | Len Sel 1 | Len Sel 0 | Same as Write | | | | | | | |
| **0x3fc** | 0 | 0 | 0 | Loop | Out2 | Out1 | RTS | DTR | Same as Write | | | | | | | |
| **0x3fd** | - - - | | | | | | | | RX ERR | TX EMP | THRE | BRK Int | FRM ERR | PAR ERR | OVRN ERR | Data RDY |
| **0x3fe** | - - - | | | | | | | | DCD | RI | DSR | CTS | DCD CHG | TE RI | DSR CHG | CTS CHG |

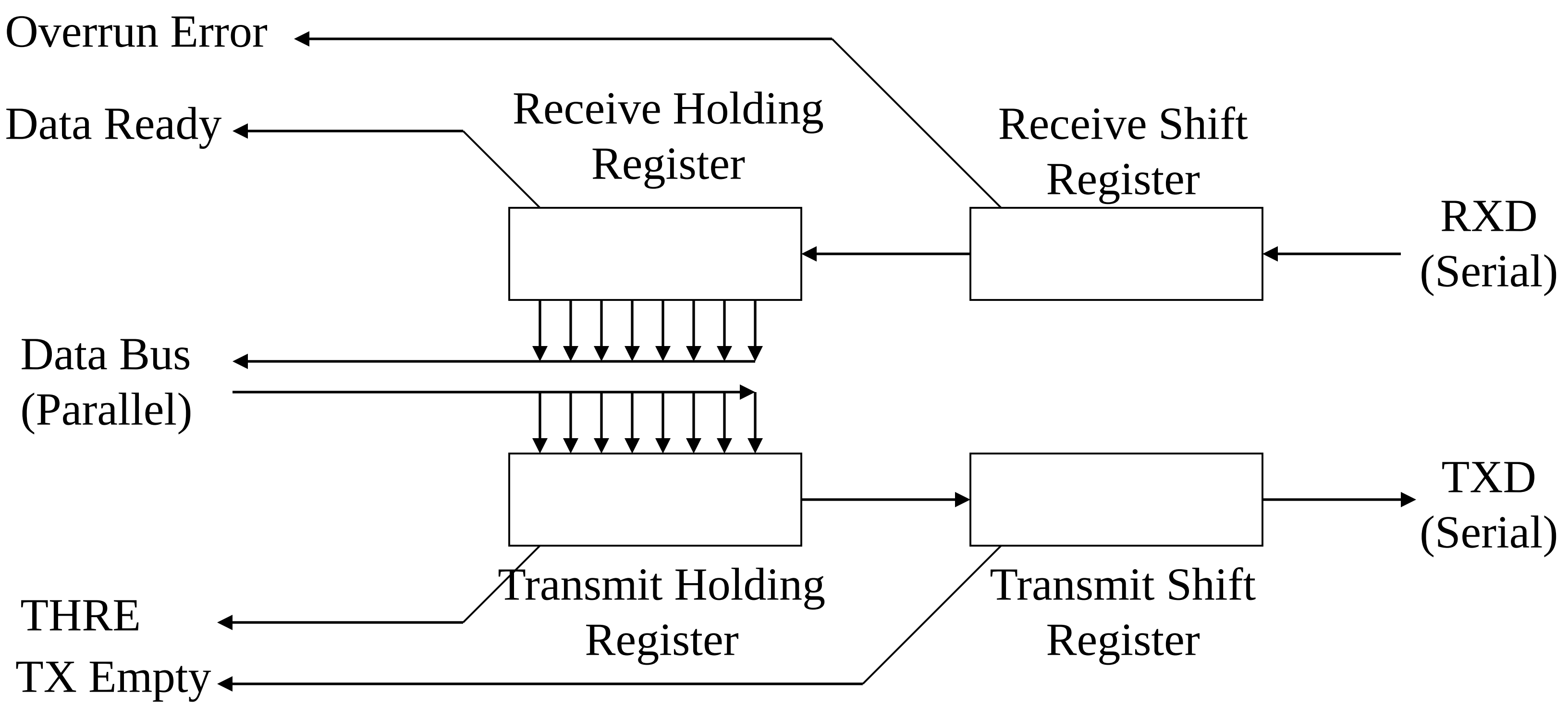
# Accessing the Serial Port

- Don’t want to use hard coded numbers!
- Look at $pcinc/serial.h for symbolic constants

```
#define COM1_BASE   0x3f8
#define COM2_BASE   0x2f8
#define UART_TX     0  /* send data */
#define UART_RX     0  /* recv data */
. . .
#define UART_LCR    3  /* line control */
#define UART_MCR    4  /* modem control */
#define UART_LSR    5  /* line status */
#define UART_MSR    6  /* modem status */
#define UART_SCR    7  /* scratch */
```

# Parallel Serial Conversion

- UART performs double buffered, bidirectional, parallel-to-serial / serial-to-parallel conversion:



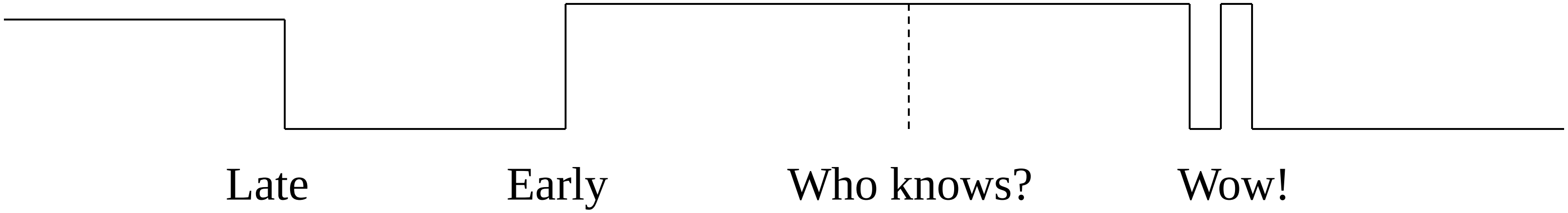
# UART Receiver Sampling

- Characters are sent/received asynchronously
  - Clocks of receiver and transmitter are independent and only nominally at the same rate (+/- 0.01%)
  - Furthermore, the phases of the clocks relative to each other are completely arbitrary
- Receiver strategy:
  - *Synch* on initial edge then “center sample” bits
  - Sample 16 times the baud rate, starting with the eighth clock period after leading edge of start bit

# UART Receiver Sampling

- “Ideal” Serial Data Waveform

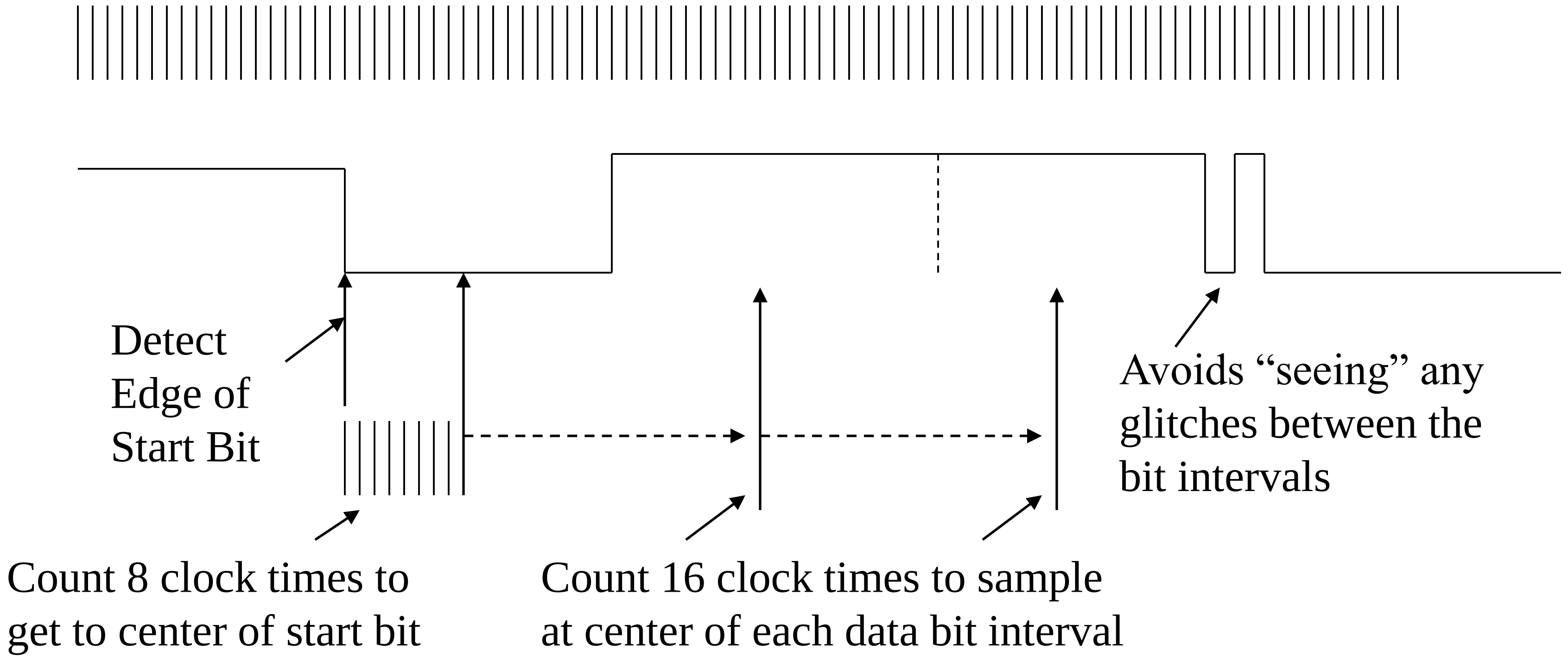

- What the Receiver “sees”




- Therefore receiver “center samples” data bits to get accurate indication of one or zero state

# UART Receiver Sampling

- Receiver runs its clock to check for one or zero state of input RXD signal at 16 times bit rate:



# Strategies for I/O Driver Code

- Two Basic Strategies for I/O Driver Code
  - Status Polling
  - Interrupt Driven
- Status Polling
  - Uses only the port addresses on the I/O device
  - Ties up the entire processor for the duration of I/O
- Interrupt Driven
  - Adds an interrupt line from I/O device to processor
  - Allows processor to do other work during I/O

# Status Polling

- Review the serial port details:
  - Status and Control Registers
- We will look at assembly language driver to send and receive data in “full duplex” mode
  - Simplex – Broadcasting

    (data going only one direction all the time)
  - Half Duplex – Sending or receiving alternately (data going only one direction at a time)
  - Full Duplex – Sending and receiving at same time (data going both directions simultaneously)

# Initializing the UART

- Tutor does this for us on COM1: and COM2:
  - Select speed, data bits, parity, and number of stop bits
  - Turn on DTR and wait for DSR on
- Half duplex mode modem signal handshake:
  - Transmit: Turn on RTS and wait for CTS on
  - Receive: Turn off RTS and wait for DCD on
- Full duplex mode modem signal handshake:
  - Turn on RTS and leave it on
  - Transmit whenever CTS on
  - Receive whenever DCD on

# Status Polling

- Loop on <u>send/receive data to/from</u> COM2:

  (Assume Tutor has initialized bit rate and line control)

  1. Turn on DTR & RTS, wait for DSR, CTS, & DCD
  2. Read data ready (DR)
  3. If data is ready, read a byte of receive data
  4. Read transmit holding register empty (THRE)
  5. If THR is empty, write a byte of transmit data
  6. Jump back to step 2
- Processor loop is much faster than byte transfer rate
- But, hard to do other work while looping on status

# Status Polling Assembly Code

- Step 1a: Turn on DTR and RTS

```
movw  $0x2fc, %dx # modem control
inb   (%dx), %al  # get current
orb   $0x03, %al  # or on 2 lsbs
outb  %al, (%dx)  # set control
```

# Status Polling Assembly Code

- Step 1b: Wait for DSR, CTS, and DCD

```
    movw  $0x2fe, %dx # modem status
loop1:
    inb   (%dx), %al  # get current
    andb  $0xb0, %al  # get 3 signals
    xorb  $0xb0, %al  # check all 3
    jnz   loop1       # some missing
# all 3 are on now
```

# Status Polling Assembly Code

- Step 2: Read Data Ready
- Step 3: If ready, read a byte from receive data

```
loop2:
  movw  $0x2fd, %dx     # line status
  inb   (%dx), %al      # get data ready
  andb  $0x01, %al      # look at dr
  jz    xmit       # if recv data
  movw  $0x2f8, %dx     # i/o data addr
  inb   (%dx), %al      # move rx to %al
  movb  %al, somewhere # save it somewhere
  movw  $0x2fd, %dx     # line status
```

# Status Polling Assembly Code

- Step 4: Read transmit holding register empty
- Step 5: If empty, write a byte to transmit data

```
xmit:
    inb   (%dx), %al       # get thre
    andb  $0x20, %al       # look at thre
    jz    loop2            # if tx hr empty
    movb  somewhere, %al # get data to send
    movw  $0x2f8, %dx      # i/o data addr
    outb  %al, (%dx)       # send it
    jmp   loop2            # and loop
```

# COM Port Driver in C - Receive

```
#include <serial.h>

void unsigned char pollgetc()
{
/* polling loop, waiting for DR bit to go on */
  while ((inpt(COM1_BASE + UART_LSR) & UART_LSR_DR) == 0)
      ;

/* input character  */
  return inpt(COM1_BASE + UART_RX);
}
```

# COM Port Driver in C - Transmit

```
#include <serial.h>

void pollputc(unsigned char ch)
{
/* polling loop, waiting for THRE bit to go on */
  while ((inpt(COM1_BASE + UART_LSR) & UART_LSR_THRE) == 0)
     ;

/* output character  */
  outpt(COM1_BASE + UART_TX, ch);
}
```