Up to this point, we have used simple design examples that illustrate how to use VHDL in the design process. In this chapter, we present some realistic design examples that show how VHDL, together with synthesis tools, can be used to design complex digital systems. We first design a receiver-transmitter for a serial data port, and then we design a simple microcontroller similar to the Motorola M68HC05.

11.1 UART DESIGN

Most computers and microcontrollers have one or more serial data ports used to communicate with serial input/output devices such as keyboards and serial printers. By using a modem (modulator-demodulator) connected to a serial port, serial data can be transmitted to and received from a remote location via telephone lines (see Figure 11-1). The serial communication interface, which receives and transmits serial data, is often called a UART (Universal Asynchronous Receiver-Transmitter). *RxD* is the received serial data signal and *TxD* is the transmitted data signal.

Figure 11-1 Serial Data Transmission

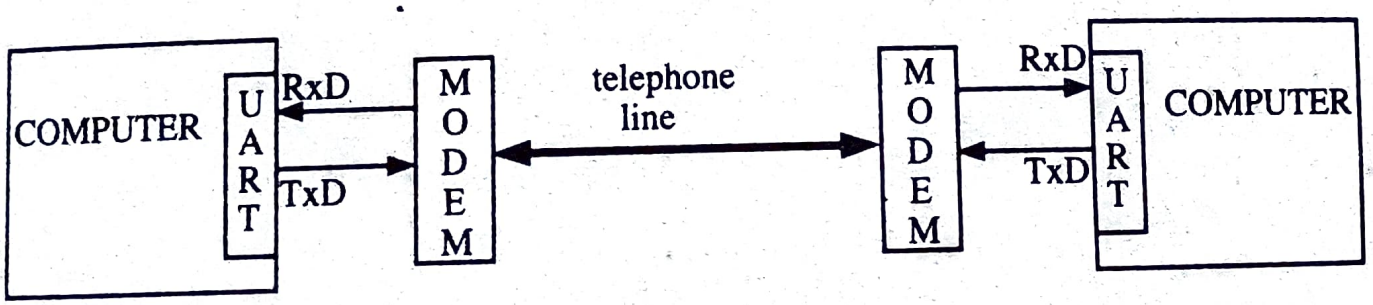
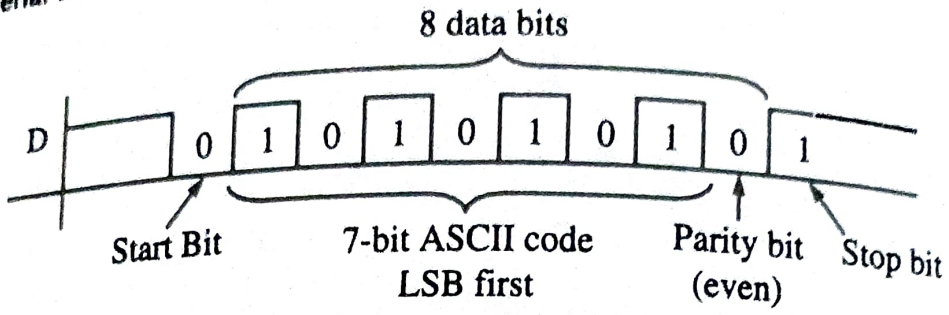


Figure 11-2 shows the standard format for serial data transmission. Since there is no clock line, the data (*D*) is transmitted asynchronously, one byte at a time. When no data is being transmitted, *D* remains high. To mark the start of transmission, *D* goes low for one bit time, which is referred to as the start bit. Then eight data bits are transmitted, least significant bit first. When text is being transmitted, ASCII code is usually used. In ASCII code, each alphanumeric character is represented by a 7-bit code. The eighth bit may be used as a parity check bit. In the example, the letter U, coded as 1010101, is transmitted followed by a 0 parity bit, so that the total number of 1s is even (even parity). After eight

bits are transmitted, *D* must go high for at least one bit time, which is referred to as the stop bit. Then transmission of another character can start at any time. The number of bits transmitted per second is frequently referred to as the *BAUD rate*.

Figure 11-2 Standard Serial Data Format



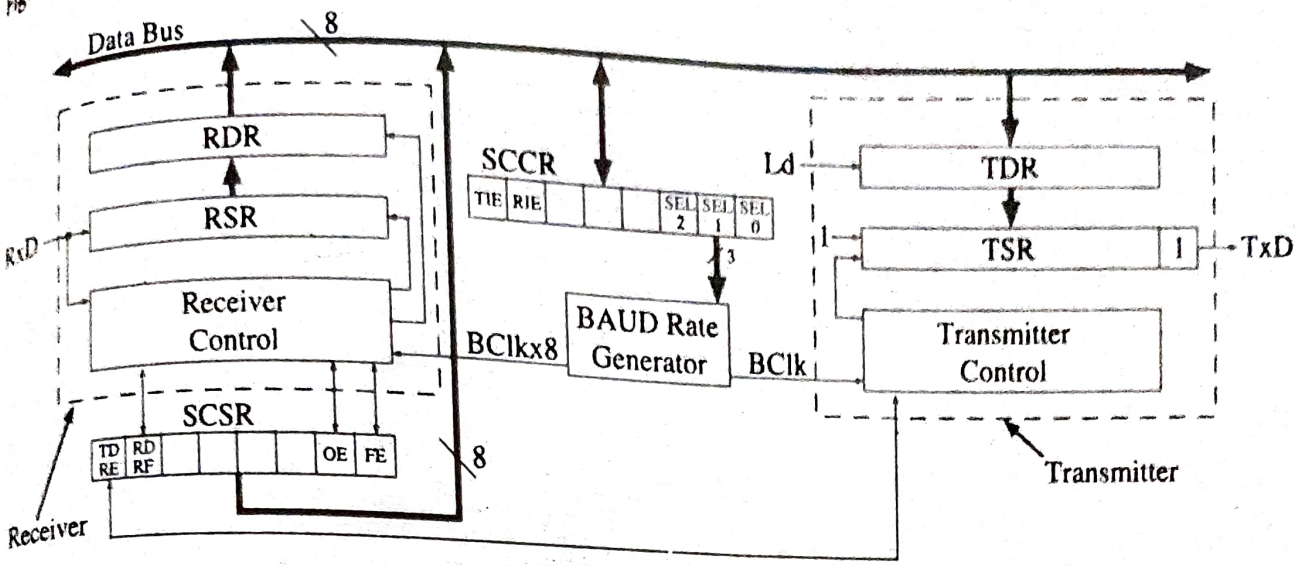
When transmitting, the UART takes eight bits of parallel data and converts the data to a serial bit stream that consists of a start bit (logic '0'), 8 data bits (least significant bit first), and one or more stop bits (logic '1'). When receiving, the UART detects the start bit, receives the 8 data bits, and converts the data to parallel form when it detects the stop bit. Since no clock is transmitted, the UART must synchronize the incoming bit stream with the local clock.

We now design a simplified version of a UART similar to the one used within the MC6805, MC6811, and other microcontrollers. Figure 11-3 shows the UART connected to the 8-bit data bus. The following six 8-bit registers are used:

RSR	Receive shift register
RDR	Receive data register
TDR	Transmit data register
TSR	Transmit shift register
SCCR	Serial communications control register
SCSR	Serial communications status register

The following discussion assumes that the UART is connected to a microcontroller data and address bus so that the CPU can read and write to the registers. *RDR*, *TDR*, *SCCR*, and *SCSR* are memory-mapped; that is, each register is assigned an address in the microcontroller memory space. *RDR*, *SCSR*, and *SCCR* can drive the data bus through tristate buffers; *TDR* and *SCCR* can be loaded from the data bus.

Figure 11-3 UART Block Diagram



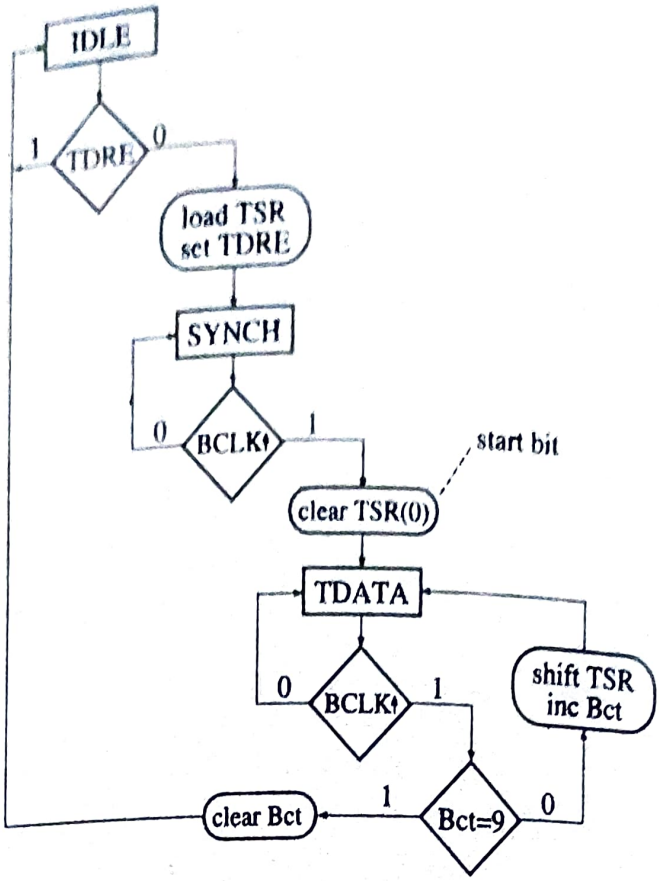
Beside the registers, the three main components of the UART are the BAUD rate generator, the receiver control, and the transmitter control. The BAUD rate generator divides down the system clock to provide the bit clock (*BClk*) with a period equal to one bit time and also *BClkX8*, which has a frequency eight times the *BClk* frequency.

The *TDRE* (transmit data register empty) bit in the *SCSR* is set when *TDR* is empty. When the microcontroller is ready to transmit data, the following occurs:

1. The microcontroller waits until *TDRE* = '1' and then loads a byte of data into *TDR* and clears *TDRE*.
2. The UART transfers data from *TDR* to *TSR* and sets *TDRE*.
3. The UART outputs a start bit ('0') for one bit time and then shifts *TSR* right to transmit the eight data bits followed by a stop bit ('1').

Figure 11-4 shows the SM chart for the transmitter. The corresponding sequential machine (SM) is clocked by the microcontroller system clock (*CLK*). In the IDLE state, the SM waits until *TDR* has been loaded and *TDRE* is cleared. In the SYNCH state, the SM waits for the rising edge of the bit clock (*Bclk*↑) and then clears the low-order bit of *TSR* to transmit a '0' for one bit time. In the TDATA state, each time *Bclk*↑ is detected, *TSR* is shifted right to transmit the next data bit and the bit counter (*Bct*) is incremented. When *Bct* = 9, 8 data bits and a stop bit have transmitted. *Bct* is then cleared and the SM goes back to IDLE.

Figure 11-4 SM Chart for UART Transmitter



The VHDL code for the UART transmitter (Figure 11-5) is based on the SM chart of Figure 11-4. The transmitter contains the *TDR* and *TSR* registers and the transmit control. It interfaces with *TDRE* and the data bus (DBUS). The first process represents the combinational network, which generates the nextstate and control signals. The second process updates the registers on the rising edge of the clock. The signal *Bclk_rising* is '1' for one system clock time following the rising edge of *Bclk*. To generate *Bclk_rising*, *Bclk* is stored in a flip-flop named *Bclk_Dlayed*. Then *Bclk_rising* is '1' if the current value of *Bclk* is '1' and the previous value (stored in *Bclk_Dlayed*) is '0'. Thus,

```
Bclk_rising <= Bclk and not Bclk_Dlayed
```

Figure 11-5 VHDL Code for UART Transmitter

```
library ieee;
use ieee.std_logic_1164.all;

entity UART_Transmitter is
port(Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;
     DBUS:in std_logic_vector(7 downto 0);
     setTDRE, TxD: out std_logic);
end UART_Transmitter;

architecture xmit of UART_Transmitter is
```

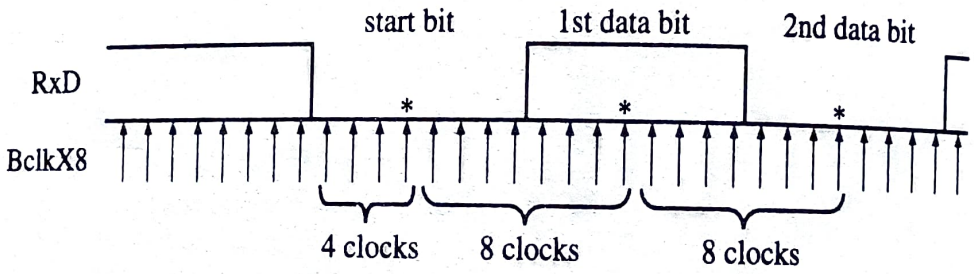
```
type stateType is (IDLE, SYNCH, TDATA);
signal state, nextstate : stateType;
signal TSR : std_logic_vector (7 downto 0);      -- Transmit Shift Register
signal TDR : std_logic_vector(7 downto 0);       -- Transmit Data Register
signal Bct: integer range 0 to 9;                -- counts number of bits sent
signal inc, clr, loadTSR, shftTSR, start: std_logic;
signal Bclk_rising, Bclk_Dlayed: std_logic;

begin
TxD <= TSR(0);
setTDRE <= loadTSR;
Bclk_rising <= Bclk and (not Bclk_Dlayed);
  -- indicates the rising edge of bit clock

Xmit_Control: process(state, TDRE, Bct, Bclk_rising)
begin
  inc <= '0'; clr <= '0'; loadTSR <= '0'; shftTSR <= '0'; start <= '0';
     -- reset control signals
  case state is
  when IDLE => if (TDRE = '0' ) then
     loadTSR <= '1'; nextstate <= SYNCH;
     else nextstate <= IDLE; end if;
  when SYNCH =>                                  -- synchronize with the bit clock
     if (Bclk_rising = '1') then
           start <= '1'; nextstate <= TDATA;
           else nextstate <= SYNCH; end if;
  when TDATA =>
     if (Bclk_rising = '0') then nextstate <= TDATA;
     elsif (Bct /= 9) then
           shftTSR <= '1'; inc <= '1'; nextstate <= TDATA;
     else clr <= '1'; nextstate <= IDLE; end if;
  end case;
end process;

Xmit_update: process (sysclk, rst_b)
begin
  if (rst_b = '0') then
     TSR <= "111111111"; state <= IDLE; Bct <= 0; Bclk_Dlayed <= '0';
  elsif (sysclk'event and sysclk = '1') then
     state <= nextstate;
     if (clr = '1') then Bct <= 0; elsif (inc = '1') then
                         Bct <= Bct + 1; end if;
     if (loadTDR = '1') then TDR <= DBUS; end if;
     if (loadTSR = '1') then TSR <= TDR & '1'; end if;
     if (start = '1') then TSRout <= '0'; end if;
     if (shftTSR = '1') then TSR <= '1' & TSR(8 downto 1); end if;
                  -- shift out one bit
     Bclk_Dlayed <= Bclk;                        -- Bclk delayed by 1 sysclk
  end if;
end process;
end xmit;
```

The operation of the UART receiver is as follows:

1. When the UART detects a start bit, it reads in the remaining bits serially and shifts them into the *RSR*.
2. When all the data bits and the stop bit have been received, the *RSR* is loaded into the *RDR*, and the Receive Data Register Full (*RDRF*) flag in the *SCSR* is set.
3. The microcontroller checks the *RDRF* flag, and if it is set, the *RDR* is read and the flag is cleared.

The bit stream coming in on *RxD* is not synchronized with the local bit clock (*Bclk*). If we attempted to read *RxD* at the rising edge of *Bclk* we would have a problem if *RxD* changed near the clock edge. We could have setup and hold time problems. If the bit rate of the incoming signal differed from *Bclk* by a small amount, we could end up reading some bits at the wrong time. To avoid these problems, we will sample *RxD* eight times during each bit time. (Some systems sample 16 times per bit.) We will sample on the rising edge of *BclkX8*. The arrows in Figure 11-6 indicate the rising edge of *BclkX8*. Ideally, we should read the bit value at the middle of each bit time for maximum reliability. When *RxD* first goes to 0, we will wait for four *BclkX8* periods, and we should be near the middle of the start bit. Then we will wait eight more *BclkX8* periods, which should take us near the middle of the first data bit . We continue reading once every eight *BclkX8* clocks until we have read the stop bit.

Figure 11-6 Sampling RxD with BclkX8



* *Read data at these points.*

Figure 11-7 shows an SM chart for the UART receiver. Two counters are used. *Ct1* counts the number of *BclkX8* clocks. *Ct2* counts the number of bits received after the start bit. In the IDLE state, the SM waits for the start bit (*RxD* = '0') and then goes to the Start Detected state. The SM waits for the rising edge of *BclkX8* (*BclkX8*↑) and then samples *RxD* again. Since the start bit should be '0' for eight *BclkX8* clocks, we should read '0'. *Ct1* is still 0, so *Ct1* is incremented and the SM waits for *BclkX8*↑. If *RxD* = '1', this is an error condition and the SM clears *Ct1* and resets to the IDLE state. Otherwise, the SM keeps looping. When *RxD* is '0' for the fourth time, *Ct1* = 3, so *Ct1* is cleared and the state goes to Receive Data. In this state, the SM increments *Ct1* after every rising edge of *BclkX8*. After the eighth clock, *Ct1* = 7 and *Ct2* is checked. If it is not 8, the current value of *RxD* is shifted into *RSR*, *Ct2* is incremented, and *Ct1* is cleared. If *Ct2* = 8, all 8 bits have been read and we should be in the middle of the stop bit. If *RDRF* = 1, the microcontroller has not yet read the previously received data byte, and an overrun error has occurred, in which

case the *OE* flag in the status register is set and the new data is ignored. If *RxD* = '0', the stop bit has not been detected properly, and the framing error (*FE*) flag in the status register is set. If no errors have occurred, *RDR* is loaded from *RSR*. In all cases, *RDRF* is set to indicate that the receive operation is completed and the counters are cleared.

The VHDL code for the UART receiver (Figure 11-8) is based on the SM chart of Figure 11-7. The receiver contains the *RDR* and *RSR* registers and the receive control. The control interfaces with *SCSR*, and *RDR* can drive data onto the data bus. The first process represents the combinational network, which generates the nextstate and control signals. The second process updates the registers on the rising edge of the clock. The signal *BclkX8_rising* is '1' for one system clock time following the rising edge of *BclkX8*. *BclkX8_rising* is generated the same manner as *Bclk_rising*.

Figure 11-7 SM Chart for UART Receiver

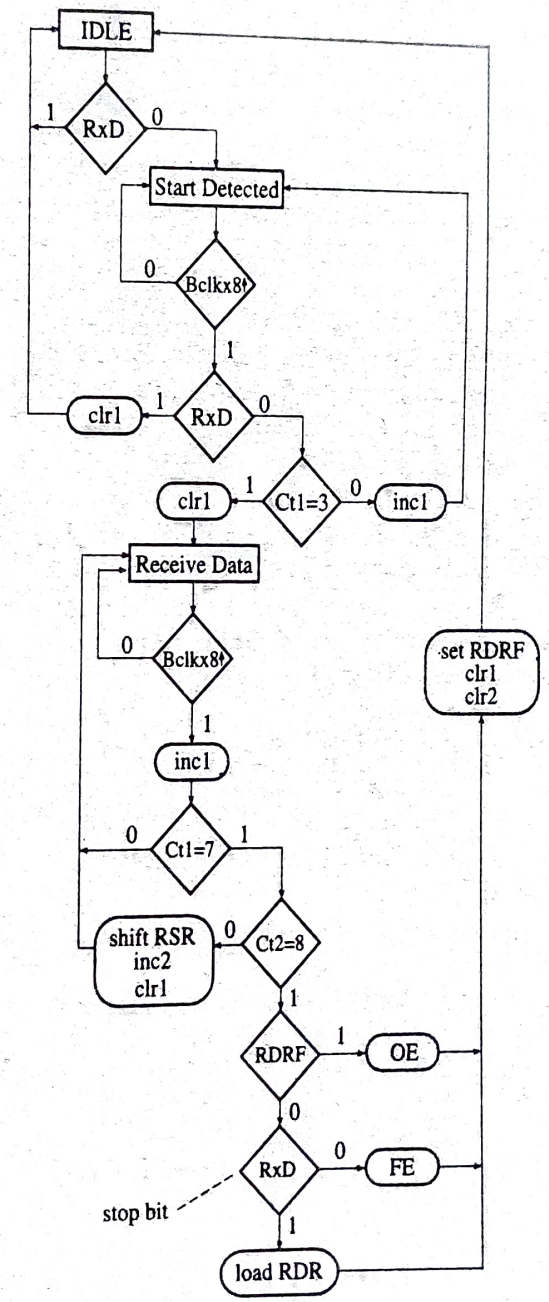


Figure 11-8 VHDL Code for UART Receiver

```
library ieee;
use ieee.std_logic_1164.all;

entity UART_Receiver is
port(RxD, BclkX8, sysclk, rst_b, RDRF: in std_logic;
     RDR: out std_logic_vector(7 downto 0);
     setRDRF, setOE, setFE: out std_logic);
end UART_Receiver;

architecture rcvr of UART_Receiver is

type stateType is (IDLE, START_DETECTED, RECV_DATA);
signal state, nextstate: stateType;

signal RSR: std_logic_vector (7 downto 0);     -- receive shift register
signal ct1 : integer range 0 to 7; -- indicates when to read the RxD input
signal ct2 : integer range 0 to 8;      -- counts number of bits read
signal inc1, inc2, clr1, clr2, shftRSR, loadRDR : std_logic;
signal BclkX8_Dlayed, BclkX8_rising : std_logic;

begin
BclkX8_rising <= BclkX8 and (not BclkX8_Dlayed);
     -- indicates the rising edge of bitX8 clock
Rcvr_Control:      process(state, RxD, RDRF, ct1, ct2, BclkX8_rising)
begin
     -- reset control signals
  inc1 <= '0'; inc2 <= '0'; clr1 <= '0'; clr2 <= '0';
  shftRSR <= '0'; loadRDR <= '0'; setRDRF <= '0'; setOE <= '0'; setFE <=
      '0';
  case state is
     when IDLE => if (RxD = '0' ) then nextstate <= START_DETECTED;
       else nextstate <= IDLE; end if;
     when START_DETECTED =>
       if (BclkX8_rising = '0') then nextstate <= START_DETECTED;
       elsif (RxD = '1') then clr1 <= '1'; nextstate <= IDLE;
       elsif (ct1 = 3) then clr1 <= '1'; nextstate <= RECV_DATA;
       else inc1 <= '1'; nextstate <= START_DETECTED; end if;
     when RECV_DATA =>
       if (BclkX8_rising = '0') then nextstate <= RECV_DATA;
       else inc1 <= '1';
         if (ct1 /= 7) then nextstate <= RECV_DATA;
            -- wait for 8 clock cycles
         elsif (ct2 /= 8) then
           shftRSR <= '1'; inc2 <= '1'; clr1 <= '1'; -- read next data bit
           nextstate <= RECV_DATA;
```

```
      else
        nextstate <= IDLE;
        setRDRF <= '1'; clr1 <= '1'; clr2 <= '1';
        if (RDRF = '1') then setOE <= '1';    -- overrun error
        elsif (RxD = '0') then setFE <= '1'; -- framing error
        else loadRDR <= '1'; end if;          -- load recv data register
      end if;
    end if;
  end case;
end process;

rcvr_update: process (sysclk, rst_b)
begin
  if (rst_b = '0') then state <= IDLE; BclkX8_Dlayed <= '0';
    ct1 <= 0; ct2 <= 0;
  elsif (sysclk'event and sysclk = '1') then
    state <= nextstate;
    if (clr1 = '1') then ct1 <= 0; elsif (inc1 = '1') then
      ct1 <= ct1 + 1; end if;
    if (clr2 = '1') then ct2 <= 0; elsif (inc2 = '1') then
      ct2 <= ct2 + 1; end if;
    if (shftRSR = '1') then RSR <= RxD & RSR(7 downto 1); end if;
        -- update shift reg.
    if (loadRDR = '1') then RDR <= RSR; end if;
    BclkX8_Dlayed <= BclkX8;    -- BclkX8 delayed by 1 sysclk
  end if;
end process;
end rcvr;
```

Figure 11-9 shows the result of synthesizing the UART receiver using the Xilinx 4000 series as a target. The resulting implementation requires 26 flip-flops and 18 function generators.

Next we will design a programmable BAUD rate generator. Three bits in the *SCCR* are used to select any one of eight BAUD rates. We will assume that the system clock is 8 MHz and we want BAUD rates 300, 600, 1200, 2400, 4800, 9600, 19200, and 38400. The maximum *BclkX8* frequency needed is 38400 × 8 = 307200. To get this frequency, we should divide 8 MHz by 26.04. Since we can divide only by an integer, we need to either accept a small error in the BAUD rate or adjust the system clock frequency downward to 7.9877 MHz to compensate.

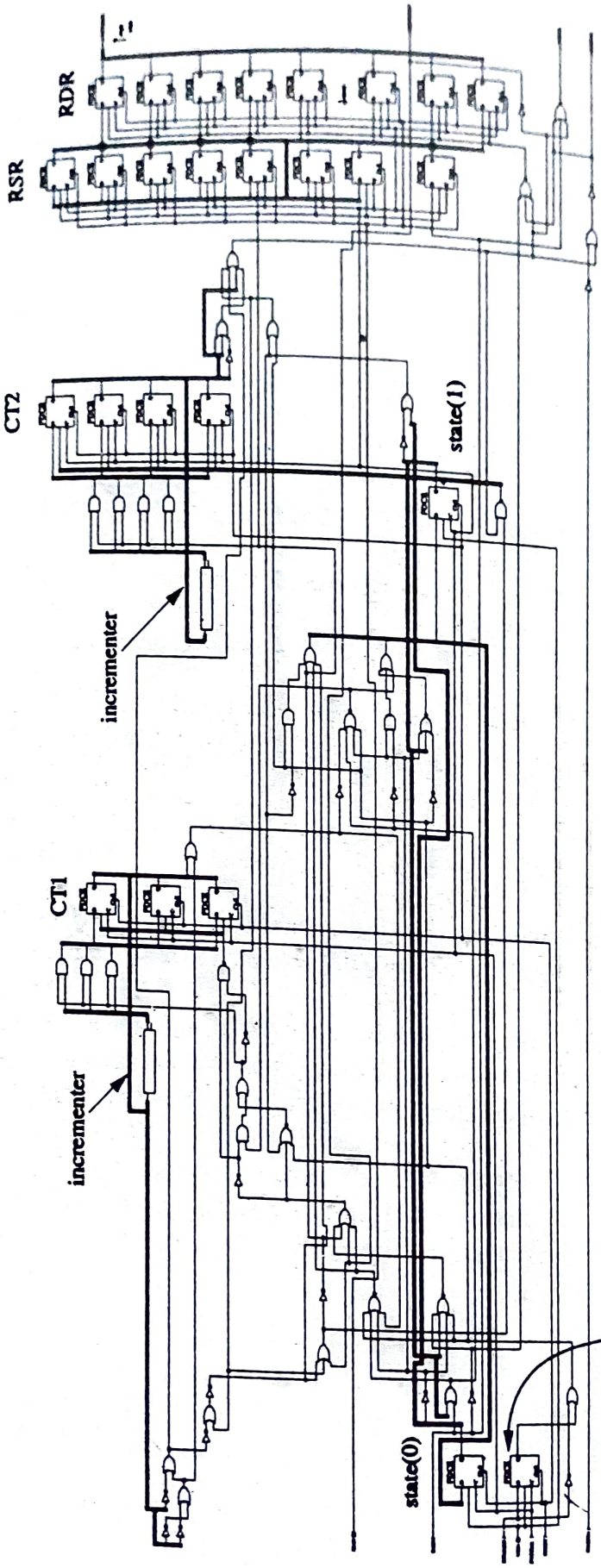
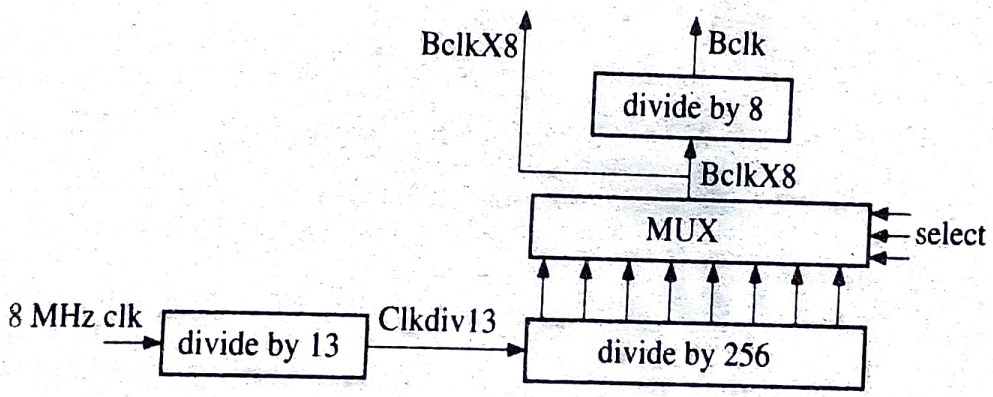


Figure 11-9 Synthesized UART Receiver

Figure 11-10 shows a block diagram for the BAUD rate generator. The 8-MHz system clock is first divided by 13 using a counter. This counter output goes to an 8-bit binary counter. The outputs of the flip-flops in this counter correspond to divide by 2, divide by 4, . . ., and divide by 256. One of these outputs is selected by a multiplexer. The MUX select inputs come from the lower 3 bits of the *SCCR*. The MUX output corresponds to *BclkX8*, which is further divided by 8 to give *Bclk*. Assuming an 8-MHz clock, the frequencies generated are given by the following table:

Select Bits	BAUD Rate (*Bclk*)
000	38462
001	19231
010	9615
011	4808
100	2404
101	1202
110	601
111	300.5

Figure 11-10 Baud Rate Generator



The VHDL code for the BAUD rate generator is given in Figure 11-11. The first process increments the divide-by-13 counter on the rising edge of the system clock. The second process increments the divide-by-256 counter on the rising edge of *Clkdiv13*. A concurrent statement generates the MUX output, *BclkX8*. The third process increments the divide-by-8 counter on the rising edge of *BclkX8* to generate *Bclk*.

Figure 11-11 VHDL Code for BAUD Rate Generator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;   -- use '+' operator, CONVER_INT func.

entity clk_divider is
port(Sysclk, rst_b: in std_logic;
  Sel: in std_logic_vector(2 downto 0);
  BclkX8: buffer std_logic;
  Bclk: out std_logic);
end clk_divider;

architecture baudgen of clk_divider is
signal ctr1: std_logic_vector (3 downto 0):= "0000";      -- divide by 13
                                                             counter
signal ctr2: std_logic_vector (7 downto 0):= "00000000"; -- div by 256 ctr
signal ctr3: std_logic_vector (2 downto 0):= "000";       -- divide by 8
                                                             counter

signal Clkdiv13: std_logic;

begin
process (Sysclk)                          -- first divide system clock by 13
begin
  if (Sysclk'event and Sysclk = '1') then
    if (ctr1 = "1100") then ctr1 <= "0000";
  else       ctr1 <= ctr1 + 1; end if;
  end if;
end process;
Clkdiv13 <= ctr1(3);                      -- divide Sysclk by 13

process (Clkdiv13)                        -- clk_divdr is an 8-bit counter
begin
  if (rising_edge(Clkdiv13)) then
      ctr2 <= ctr2 + 1;
  end if;
end process;

BclkX8 <= ctr2(CONVERT_INT(sel));  -- select baud rate
process (BclkX8)
begin
  if (rising_edge(BclkX8)) then
      ctr3 <= ctr3 + 1;
  end if;
end process;
Bclk <= ctr3(2);
                                          -- Bclk is BclkX8 divided by 8
end baudgen;
```

To complete the UART design, we need to interconnect the three components we have designed, connect them to the control and status registers, and add the interrupt generation logic and the bus interface. Figure 11-12 gives the VHDL code for the complete UART.

SCI_IRQ is an interrupt signal that interrupts the CPU when the UART receiver or transmitter needs attention. When the *RIE* (receive interrupt enable) is set in *SCCR*, *SCI_IRQ* is generated whenever *RDRE* or *OE* is '1'. When *TIE* (transmit interrupt enable) is set in *SCCR*, *SCI_IRQ* is generated whenever *TDRE* is '1'.

The UART is interfaced to a microcontroller address and data busses so that the CPU can read and write to the UART registers when the UART is selected by *SCIsel* = '1'. The last two bits of the address (*ADDR2*), together with the *R_W* signal, are used for register selection as follows:

ADDR2	*R_W*	Action
00	1	DBUS ← RDR
00	0	TDR ← DBUS
01	1	DBUS ← SCSR
01	0	DBUS ← hi-Z
1–	1	DBUS ← SCCR
1–	0	SCCR ← DBUS

When the UART is not selected for reading, the data bus is driven to high-Z.

Figure 11-12 VHDL Code for Complete UART

```
library ieee;
use ieee.std_logic_1164.all;

entity UART is
port(SCI_sel, R_W, clk, RxD : in std_logic;
  ADDR2: in std_logic_vector(1 downto 0);
  DBUS : inout std_logic_vector(7 downto 0);
  SCI_IRQ, TxD : out std_logic);
end UART;

architecture uart1 of UART is
  component UART_Receiver
  port(RxD, BclkX8, sysclk, rst_b, RDRF: in std_logic;
                  RDR: out std_logic_vector(7 downto 0);
                  setRDRF, setOE, setFE: out std_logic);
  end component;
  component UART_Transmitter
  port(Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;
           DBUS: in std_logic_vector(7 downto 0);
           setTDRE, TxD: out std_logic);
  end component;
```

```
  component clk_divider
  port(clk, rst_b: in std_logic;
       Sel: in std_logic_vector(2 downto 0);
       BclkX8, Bclk: out std_logic);
  end component;

  signal RDR : std_logic_vector(7 downto 0);     -- Receive Data Register
  signal SCSR : std_logic_vector(7 downto 0); -- Status Register
  signal SCCR : std_logic_vector(7 downto 0); -- Control Register
  signal TDRE, RDRF, OE, FE, TIE, RIE : std_logic;
  signal BaudSel : std_logic_vector(2 downto 0);

  signal setTDRE, setRDRF, setOE, setFE, loadTDR, loadSCCR : std_logic;
  signal clrRDRF, Bclk, BclkX8, SCI_Read, SCI_Write : std_logic;

begin

RCVR: UART_Receiver port map(RxD, BclkX8, clk, rst_b, RDRF, RDR, setRDRF,
                             setOE, setFE);
XMIT: UART_Transmitter port map(Bclk, clk, rst_b, TDRE, loadTDR, DBUS,
                             setTDRE, TxD);
CLKDIV: clk_divider port map(clk, rst_b, BaudSel, BclkX8, Bclk);

-- This process updates the control and status registers
process (clk, rst_b)
begin
  if (rst_b = '0') then
      TDRE <= '1'; RDRF <= '0'; OE<= '0'; FE <= '0';
      TIE <= '0'; RIE <= '0';
  elsif (rising_edge(clk)) then
      TDRE <= (setTDRE and not TDRE) or (not loadTDR and TDRE);
      RDRF <= (setRDRF and not RDRF) or (not clrRDRF and RDRF);
      OE <= (setOE and not OE) or (not clrRDRF and OE);
      FE <= (setFE and not FE) or (not clrRDRF and FE);
  if (loadSCCR = '1') then TIE <= DBUS(7); RIE <= DBUS(6);
      BaudSel <= DBUS(2 downto 0); end if;
  end if;
end process;

-- IRQ generation logic
  SCI_IRQ <= '1' when  ((RIE = '1' and (RDRF = '1' or OE = '1'))
                    or (TIE = '1' and TDRE = '1'))
             else '0';
-- Bus Interface
SCSR <= TDRE & RDRF & "0000" & OE & FE;
SCCR <= TIE & RIE & "000" & BaudSel;
SCI_Read <= '1' when (SCI_sel = '1' and R_W = '0') else '0';
SCI_Write <= '1' when (SCI_sel = '1' and R_W = '1') else '0';
```

```
clrRDRF <= '1' when (SCI_Read = '1' and ADDR = "00") else '0';
loadTDR <= '1' when (SCI_Write = '1' and ADDR = "00") else '0';
loadSCCR <= '1' when (SCI_Write = '1' and ADDR = "10") else '0';
DBUS <= "ZZZZZZZZ" when (SCI_Read = '0')-- tristate bus when not reading
  else RDR when (ADDR = "00")     -- write appropriate register to the bus

  else SCSR when (ADDR = "01")
  else SCCR;                      -- dbus = sccr, if addr is "10" or "11"
end uart1;
```

The VHDL code in Figure 11-12 was synthesized using the *Xilinx 4000 series* as a target. The resulting implementation required 90 FG function generators and 74 flip-flops and fits into an XC4003. When synthesized using the *ALTERA 7000E series* as a target, 120 logic cells and 74 flip-flops were required.

Figure 11-24 Top-level VHDL for 6805 Microcontrolier

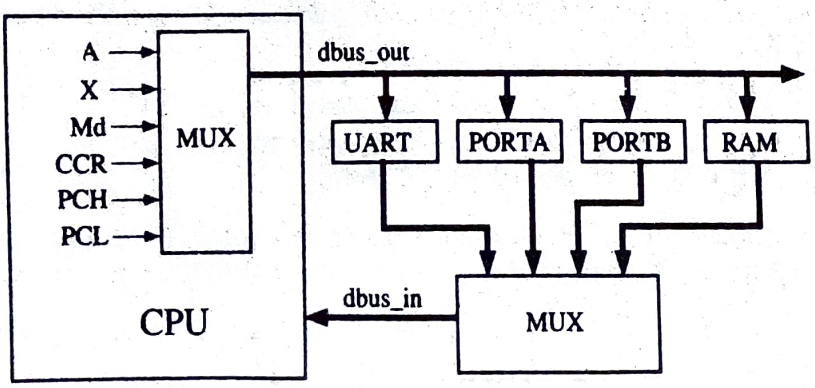
```
library ieee;
use ieee.std_logic_1164.all;

entity m68hc05 is
port(clk, rst_b, irq, RxD : in std_logic;
     PortA, PortB : inout std_logic_vector(7 downto 0);
     TxD : out std_logic);
end m68hc05;

architecture M6805_64 of m68hc05 is
component cpu6805
port(clk, rst_b, IRQ, SCint: in std_logic;
  dbus : inout std_logic_vector(7 downto 0);
  abus : out std_logic_vector(12 downto 0);
  wr: out std_logic);
end component;

component ram32X8_io
port (addr_bus: in std_logic_vector(4 downto 0);
            data_bus: inout std_logic_vector(7 downto 0);
            cpu_wr: in  std_logic);
end component;

component PORT_A
port(clk, rst_b, Port_Sel, ADDR, R_W : in std_logic;
     DBUS : inout std_logic_vector(7 downto 0);
     PinA : inout std_logic_vector(7 downto 0));
end component;

component UART
port(SCI_sel, R_W, clk, rst_b, RxD : in std_logic;
  ADDR : in std_logic_vector(1 downto 0);
  DBUS : inout std_logic_vector(7 downto 0);
  SCI_IRQ, TxD : out std_logic);
end component ;

signal SCint, wr, cs, we: std_logic;
signal SelLowRam, SelHiRAM, SelPA, SelPB, SelSC : std_logic;
signal addr_bus: std_logic_vector(12 downto 0) := (others => '0');
signal data_bus: std_logic_vector(7 downto 0) := (others => '0');
```

```
begin
CPU: cpu6805 port map (clk, rst_b, irq, SCint, data_bus, addr_bus, wr);
PA: PORT_A port map (clk, rst_b, SelPA, addr_bus(0), wr, data_bus, PortA);
PB: PORT_A port map (clk, rst_b, SelPB, addr_bus(0), wr, data_bus, PortB);
Uart1: UART port map (SelSC, wr, clk, rst_b, RxD, addr_bus(1 downto 0),
                      data_bus, SCint, TxD);

LowRAM: ram32X8_io port map (addr_bus(4 downto 0), data_bus, cs1, we);
HiRAM:  ram32X8_io port map (addr_bus(4 downto 0), data_bus, cs2, we);

-- memory interface
cs1 <= SelLowRam and not clk;   -- select ram on 2nd half of clock cycle
cs2 <= SelHiRam and not clk;
we <= wr and not clk;           -- write enable on 2nd half of clock cycle

-- address decoder
SelPA <= '1' when addr_bus(12 downto 1) = "000000000000" else '0';
SelPB <= '1' when addr_bus(12 downto 1) = "000000000001" else '0';
SelSC <= '1' when addr_bus(12 downto 2) = "00000000001"  else '0';
SelLowRam <= '1' when addr_bus(12 downto 5) = "00000001" else '0';
--           32 <= addr <= 63
SelHiRam <= '1'  when addr_bus(12 downto 5) = "11111111" else '0';
--           addr >= 8160 (1FE0h)
end M6805_64;
```

When we synthesized our 6805 microcontroller VHDL code with the XC4020E as a target, the resulting implementation required 692 *F* and *G* function generators and 178 flip-flops, plus 16 CLBs for the RAM. We simulated the system to verify correct execution of the instructions, interrupts, and I/O interfaces. We then downloaded the bit file to a 4020 so that we could test the hardware. The exercises at the end of this chapter suggest ways in which the design can be improved and expanded.

In order to synthesize the 6805 microcontroller for the FLEX 10K20, a number of changes in the VHDL code were required. Since the 10K20 does not support internal tristate bidirectional busses, we changed the CPU data bus structure and used multiplexers to select the data going into and out of the CPU, as shown in Figure 11-25. We also changed the memory components and used four of the 10K20 EABs to implement a 1024 × 8 RAM. Except for the EABs, our design requires less than 50% of the resources on the 10K20.

Figure 11-25 Multiplexed Data Bus Structure



In this chapter we designed and implemented a UART and a microcontroller using VHDL and synthesis tools. We used the following steps in designing the microcontroller CPU:

1. Define the register structure, instruction set, and addressing modes.
2. Construct a table that shows the register transfers that take place during each clock cycle.
3. Design the control state machine.
4. Write behavioral VHDL code based on (1), (2), and (3). Simulate execution of the instructions to verify that specifications are met.
5. Work out block diagrams for the major components of the CPU and determine the needed control signals.
6. Rewrite the VHDL based on (5). Again, simulate execution of the instructions.
7. Synthesize the CPU from the VHDL code. Make changes in the VHDL code as needed to improve the synthesis results.
8. Download the bit file to the actual hardware and verify the operation.

Once we have written and debugged our VHDL code, use of synthesis tools makes it easy to develop a hardware prototype. After we have evaluated the prototype, it is relatively easy to change the design at the VHDL level and then resynthesize it. Although we targeted our design for a specific PLD, retargeting the design for different components is usually straightforward, although changes in the VHDL code may be required.

Problems

11.1 Make necessary changes in the UART receiver VHDL code so that it uses a 16X bit clock instead of an 8X bit clock. Using a faster sampling clock can improve the noise immunity of the receiver.

11.2

(a) Write a VHDL test bench for the UART. Include cases to test overrun error, framing error, noise causing a false start, change of BAUD rate, etc. Simulate the VHDL code.

(b) If suitable hardware is available, write a simpler test bench to allow a loop-back test with *TxD* externally connected to *RxD*. Synthesize the test bench along with the UART, download to the target device, and verify correct operation of the hardware.

11.3 Make necessary changes to the VHDL code to add a parity option to the UART described in Section 11.1. Add two bits (P_1P_0) to the SCCR that select the parity mode as follows:

$P_1P_0 = 00$ 8 data bits, no parity bit
$P_1P_0 = 01$ 7 data bits, 8th bit makes parity even
$P_1P_0 = 10$ 7 data bits, 8th bit makes parity odd
$P_1P_0 = 11$ 7 data bits, 8th bit is always '0'

The transmitter should generate the even, odd, or '0' parity bit as specified. The receiver should check the parity bit to verify that it is correct. If not, it should set a PE (parity error) flag in the SCSR.