Figure 11-1  Serial Data Transmission

Figure 11-2  Standard Serial Data Format
Figure 11-3 UART Block Diagram

Data Bus

RxD

RDR

RSR

Receiver Control

SCSR

TIE RIE

SEL 2 1 0

BClk x8

BAUD Rate Generator

BClk

Ld

TDR

TSR

1

Transmitter Control

Transmitter

TxD

8
Figure 11-4  SM Chart for UART Transmitter

- **IDLE**
  - **TDRE**
    - 1: load TSR, set TDRE
    - 0: SYNCH
  - **SYNCH**
    - 0: BCLK↑
    - 1: clear TSR(0)
      - start bit
      - shift TSR, inc Bct
      - Bct=9
      - clear Bct

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 (8 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;
begnin
TxD <= TSR(0);
setTDRE <= loadTSR;
Bclk_rising <= Bclk and (not Bclk_dlayed); -- indicates the rising edge of bit clock
end
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 <= '1';
    else nextstate <= IDLE; end if;
end case;
    -- Xmit_Control
end process;
Figure 11-5(b) VHDL Code for UART Transmitter

```vhdl
-- synchronize with the bit clock
when SYNCH =>  
    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
            TSR(0) <= '0';
        end if;
        if (shftTSR = '1') then
            TSR <= '1' & TSR(8 downto 1);
        end if;
    end if;
bclk_dlayed <= bclk;
end if;
end process;
end xmit;
```
Operation of UART Receiver

1. When the UART detects a start bit, it reads in the remaining bits serially and shifts them into the RSR.
2. When all of the data bits and the stop bit have been received, the RSR is loaded into the RDR, and the Receive Data Register Full flag (RDRF) 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.

Figure 11-6 Sampling RxD with BclkX8

* Read data at these points
Figure 11-7 SM Chart 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';
Figure 11-8(b) VHDL Code for UART Receiver

```vhdl
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 wait for 8 clock cycles
           nextstate <= RECV_DATA;
        else
           shftRSR <= '1'; inc2 <= '1'; clr1 <= '1';
           nextstate <= RECV_DATA;
        end if;
    elsif (ct2 /= 8) then
        shftRSR <= '1'; inc2 <= '1'; clr1 <= '1';
        nextstate <= RECV_DATA;
    else
        nextstate <= IDLE;
        setRDRF <= '1'; clr1 <= '1'; clr2 <= '1';
        if (RDRF = '1') then setOE <= '1';
        elsif (RxD = '0') then setFE <= '1';
        else loadRDR <= '1'; end if; end if;
  end if;
end case;
end process;
```
Figure 11-8(c) VHDL Code for UART Receiver

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 Synthesized UART receiver
Figure 11-10  Baud Rate Generator

<table>
<thead>
<tr>
<th>Select Bits</th>
<th>BAUD Rate ((Bclk))</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>38462</td>
</tr>
<tr>
<td>001</td>
<td>19231</td>
</tr>
<tr>
<td>010</td>
<td>9615</td>
</tr>
<tr>
<td>011</td>
<td>4808</td>
</tr>
<tr>
<td>100</td>
<td>2404</td>
</tr>
<tr>
<td>101</td>
<td>1202</td>
</tr>
<tr>
<td>110</td>
<td>601</td>
</tr>
<tr>
<td>111</td>
<td>300.5</td>
</tr>
</tbody>
</table>
Figure 11-11(a) VHDL Code for BAUD Rate Generator

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all; -- use '+' operator, CONV_INTEGER 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;
```
Figure 11-11(b) VHDL Code for BAUD Rate Generator

Clkdiv13 <= ctr1(3);  -- divide Sysclk by 13

process (Clkdiv13)
begin
  if (rising_edge(Clkdiv13)) then
    ctr2 <= ctr2 + 1;
  end if;
end process;

BclkX8 <= ctr2(CONV_INTEGER(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;
Figure 11-12(a) VHDL Code for Complete UART

library ieee;
use ieee.std_logic_1164.all;

description UART is
port (SCI_sel, R_W, clk, rst_b, 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 (Sysclk, rst_b: in std_logic;
 Sel: in std_logic_vector(2 downto 0);
 BclkX8: buffer std_logic;
 Bclk: out std_logic);
end component;
Figure 11-12(b) VHDL Code for Complete UART

```vhdl
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);
  end if;
end process;
```

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 ADDR2 = "00") else '0';
loadTDR <= '1' when (SCI_Write = '1' and ADDR2 = "00") else '0';
loadSCCR <= '1' when (SCI_Write = '1' and ADDR2 = "10") else '0';
DBUS <= "ZZZZZZZZZ" when (SCI_Read = '0')  -- tristate bus when not reading
  else RDR when (ADDR2 = "00")  -- write appropriate register to the bus
  else SCSR when (ADDR2 = "01")
  else SCCR;  -- dbus = sccr, if ADDR2 is "10" or "11"
end uart1;
Figure 11-13 Simplified Block Diagram for M68HC05 Microcontroller
Figure 11-14 MC68HC05C4 Programming Model

Accumulator (A)

Index register (X)

Stack pointer (SP)

Program counter (PC)

Condition code register (CCR)

Half-carry flag
Interrupt mask
Negative flag
Zero flag
Carry/borrow flag
### Table 11-1 Opcodes for MC6805

<table>
<thead>
<tr>
<th>Bit Manipulation</th>
<th>Branch</th>
<th>Read/Modify/Write</th>
<th>Control</th>
<th>Register/Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>IMM</td>
<td>Dir</td>
</tr>
<tr>
<td>Hi</td>
<td>Lo</td>
<td></td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*not implemented in this text*
### Table 11-2(a) 6805 Instructions: Register–memory Instructions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Instruction</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>add</td>
<td>A ← A + M</td>
<td>NZ, C ← carry</td>
</tr>
<tr>
<td>ADC</td>
<td>add with carry</td>
<td>A ← A + M + C</td>
<td>NZ, C ← carry</td>
</tr>
<tr>
<td>SUB</td>
<td>subtract</td>
<td>A ← A – M</td>
<td>NZ, C ← borrow</td>
</tr>
<tr>
<td>SBC</td>
<td>subtract with borrow</td>
<td>A ← A – M – C</td>
<td>NZ, C ← borrow</td>
</tr>
<tr>
<td>CMP</td>
<td>compare A</td>
<td>A – M</td>
<td>NZ, C ← borrow</td>
</tr>
<tr>
<td>CPX</td>
<td>compare X</td>
<td>X – M</td>
<td>NZ, C ← borrow</td>
</tr>
<tr>
<td>AND</td>
<td>and</td>
<td>A ← A and M</td>
<td>NZ</td>
</tr>
<tr>
<td>BIT</td>
<td>bit test</td>
<td>A and M</td>
<td>NZ</td>
</tr>
<tr>
<td>ORA</td>
<td>or</td>
<td>A ← A or M</td>
<td>NZ</td>
</tr>
<tr>
<td>EOR</td>
<td>exclusive-or</td>
<td>A ← A xor M</td>
<td>NZ</td>
</tr>
<tr>
<td>LDA</td>
<td>load A</td>
<td>A ← M</td>
<td>NZ</td>
</tr>
<tr>
<td>LDX</td>
<td>load X</td>
<td>X ← M</td>
<td>NZ</td>
</tr>
<tr>
<td>STA</td>
<td>store A</td>
<td>M ← A</td>
<td>NZ</td>
</tr>
<tr>
<td>STX</td>
<td>store X</td>
<td>M ← X</td>
<td>NZ</td>
</tr>
<tr>
<td>JMP</td>
<td>jump</td>
<td>jump to EA</td>
<td></td>
</tr>
<tr>
<td>JSR</td>
<td>jump to subroutine</td>
<td>push PC on stack, jump to EA</td>
<td></td>
</tr>
</tbody>
</table>
Table 11-2(b)  6805 Instructions: Read–modify–write Instructions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Instruction</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG</td>
<td>negate</td>
<td>$R \leftarrow 0 - R$</td>
<td>NZ, $C \leftarrow$ borrow</td>
</tr>
<tr>
<td>COM</td>
<td>complement</td>
<td>$R \leftarrow \text{not } R$</td>
<td>NZ, $C \leftarrow 1$</td>
</tr>
<tr>
<td>TST</td>
<td>test</td>
<td>$0 - R$</td>
<td>NZ</td>
</tr>
<tr>
<td>CLR</td>
<td>clear</td>
<td>$R \leftarrow 0$</td>
<td>NZ</td>
</tr>
<tr>
<td>INC</td>
<td>increment</td>
<td>$R \leftarrow R + 1$</td>
<td>NZ</td>
</tr>
<tr>
<td>DEC</td>
<td>decrement</td>
<td>$R \leftarrow R - 1$</td>
<td>NZ</td>
</tr>
<tr>
<td>LSL</td>
<td>logical shift left</td>
<td>$R \leftarrow R(6 \text{ downto } 0) &amp; '0'$</td>
<td>NZ, $C \leftarrow R(7)$</td>
</tr>
<tr>
<td>ROL</td>
<td>rotate left</td>
<td>$R \leftarrow R(6 \text{ downto } 0) &amp; C$</td>
<td>NZ, $C \leftarrow R(7)$</td>
</tr>
<tr>
<td>ASR</td>
<td>arithmetic shift right</td>
<td>$R \leftarrow R(7) &amp; R(7 \text{ downto } 1)$</td>
<td>NZ, $C \leftarrow R(0)$</td>
</tr>
<tr>
<td>LSR</td>
<td>logical shift right</td>
<td>$R \leftarrow '0' &amp; R(7 \text{ downto } 1)$</td>
<td>NZ, $C \leftarrow R(0)$</td>
</tr>
<tr>
<td>ROR</td>
<td>rotate right</td>
<td>$R \leftarrow C &amp; R(7 \text{ downto } 1)$</td>
<td>NZ, $C \leftarrow R(0)$</td>
</tr>
</tbody>
</table>
Table 11-2(c)  6805 Instructions: Control Instructions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAX</td>
<td>transfer A to X</td>
<td>X ← A</td>
</tr>
<tr>
<td>TXA</td>
<td>transfer X to A</td>
<td>A ← X</td>
</tr>
<tr>
<td>CLC</td>
<td>clear carry</td>
<td>C ← '0'</td>
</tr>
<tr>
<td>SEC</td>
<td>set carry</td>
<td>C ← '1'</td>
</tr>
<tr>
<td>CLI</td>
<td>clear I</td>
<td>I ← '0'</td>
</tr>
<tr>
<td>SEI</td>
<td>set I</td>
<td>I ← '1'</td>
</tr>
<tr>
<td>RSP</td>
<td>reset SP</td>
<td>SP ← &quot;111111&quot;</td>
</tr>
<tr>
<td>NOP</td>
<td>no operation</td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td>return from interrupt</td>
<td>pop CCR, A, X, PCH, PCL; return to address in PC</td>
</tr>
<tr>
<td>RTS</td>
<td>return from subroutine</td>
<td>pop PCH, PCL; return to address in PC</td>
</tr>
<tr>
<td>SWI</td>
<td>software interrupt</td>
<td>push PCL, PCH, X, A, CCR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jump to address from interrupt vector table</td>
</tr>
</tbody>
</table>
### Table 11-2(d)  6805 Instructions: Branch Instructions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Instruction</th>
<th>Branch if</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRA</td>
<td>branch always</td>
<td>always</td>
</tr>
<tr>
<td>BRN</td>
<td>branch never</td>
<td>never</td>
</tr>
<tr>
<td>BHI</td>
<td>branch if higher</td>
<td>(C or Z) = '0'</td>
</tr>
<tr>
<td>BLS</td>
<td>branch of lower or same</td>
<td>(C or Z) = '1'</td>
</tr>
<tr>
<td>BCC</td>
<td>branch if carry clear</td>
<td>C = '0'</td>
</tr>
<tr>
<td>BCS</td>
<td>branch if carry set`</td>
<td>C = '1'</td>
</tr>
<tr>
<td>BNE</td>
<td>branch if not equal</td>
<td>Z = '0'</td>
</tr>
<tr>
<td>BEQ</td>
<td>branch if equal</td>
<td>Z = '1'</td>
</tr>
<tr>
<td>BPL</td>
<td>branch if plus</td>
<td>N = '0'</td>
</tr>
<tr>
<td>BMI</td>
<td>branch if minus</td>
<td>N = '1'</td>
</tr>
<tr>
<td>BMC</td>
<td>branch if int. mask clear</td>
<td>I = '0'</td>
</tr>
<tr>
<td>BMS</td>
<td>branch if int. mask set</td>
<td>I = '1'</td>
</tr>
</tbody>
</table>
## Table 11-3  6805 Addressing Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Name</th>
<th>Bytes</th>
<th>Examples</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm</td>
<td>immediate</td>
<td>2</td>
<td>ADD ii</td>
<td>data = byte 2 of inst.</td>
</tr>
<tr>
<td>dir,</td>
<td>direct</td>
<td>2</td>
<td>ADD dd</td>
<td>EA = dd</td>
</tr>
<tr>
<td>dirm</td>
<td></td>
<td></td>
<td>INC dd</td>
<td></td>
</tr>
<tr>
<td>ext</td>
<td>extended</td>
<td>3</td>
<td>ADD hh ll</td>
<td>EA = hh ll</td>
</tr>
<tr>
<td>ix</td>
<td>indexed,</td>
<td>1</td>
<td>ADD ,X</td>
<td>EA = X</td>
</tr>
<tr>
<td>ixm</td>
<td>no offset</td>
<td></td>
<td>INC ,X</td>
<td></td>
</tr>
<tr>
<td>ix1,</td>
<td>indexed,</td>
<td>2</td>
<td>ADD ff,X</td>
<td>EA = ff + X</td>
</tr>
<tr>
<td>ix1m</td>
<td>1-byte offset</td>
<td></td>
<td>INC ff,X</td>
<td></td>
</tr>
<tr>
<td>ix2</td>
<td>indexed,</td>
<td>3</td>
<td>ADD ee ff,X</td>
<td>EA = (ee ff) + X</td>
</tr>
<tr>
<td></td>
<td>2-byte offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rel</td>
<td>relative</td>
<td>2</td>
<td>BRA rr</td>
<td>EA = PC+2+rr*</td>
</tr>
<tr>
<td>inha,</td>
<td>inherent</td>
<td>1</td>
<td>INCA</td>
<td>data is in A</td>
</tr>
<tr>
<td>inhx</td>
<td></td>
<td></td>
<td>INCX</td>
<td>data is in X</td>
</tr>
<tr>
<td>inh1,</td>
<td>inherent</td>
<td>1</td>
<td>RTI</td>
<td>opcode implies location of operands</td>
</tr>
<tr>
<td>inh2</td>
<td></td>
<td></td>
<td>TAX</td>
<td></td>
</tr>
</tbody>
</table>

*rr is sign-extended before addition
Table 11-4(a) Cycle-by-Cycle Operations for 6805 Instructions

1st cycle is **{fetch}** for all instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>2nd cycle</th>
<th>3rd cycle</th>
<th>4th cycle</th>
<th>5th cycle</th>
<th>6th cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD imm</td>
<td>{addr}</td>
<td>Md ← mem(PC)</td>
<td>(A ← A + Md)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC ← PC + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD dir</td>
<td>{addr}</td>
<td>MARL ← mem(PC)</td>
<td>Md ← mem(MAR)</td>
<td>(A ← A + Md)*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC ← PC + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD ix</td>
<td>{addr}</td>
<td>MARL ← X</td>
<td>{data}</td>
<td>Md ← mem(MAR)</td>
<td>(A ← A + Md)*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD ix1</td>
<td>{addr}</td>
<td>MARL ← mem(PC)</td>
<td>{addx}</td>
<td>MAR ← MAR + X</td>
<td>{data}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC ← PC + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD ext</td>
<td>{addr}</td>
<td>MARH ← mem(PC)</td>
<td>{addr}</td>
<td>MARL ← mem(PC)</td>
<td>{data}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC ← PC + 1</td>
<td></td>
<td>PC ← PC + 1</td>
<td></td>
</tr>
<tr>
<td>ADD ix2</td>
<td>{addr}</td>
<td>MARH ← mem(PC)</td>
<td>{addr}</td>
<td>MARL ← mem(PC)</td>
<td>{data}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC ← PC + 1</td>
<td></td>
<td>PC ← PC + 1</td>
<td></td>
</tr>
<tr>
<td>STA ext</td>
<td>{addr}</td>
<td>MARH ← mem(PC)</td>
<td>{addr}</td>
<td>MARL ← mem(PC)</td>
<td>{data}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC ← PC + 1</td>
<td></td>
<td>PC ← PC + 1</td>
<td></td>
</tr>
<tr>
<td>INC A</td>
<td>{addr}</td>
<td>A ← A + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC dirm</td>
<td>{addr}</td>
<td>MARL ← mem(PC)</td>
<td>{data}</td>
<td>Md ← mem(MAR)</td>
<td>{rd_mod_wr}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC ← PC + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC ixm</td>
<td>{addr}</td>
<td>MARL ← X</td>
<td>{data}</td>
<td>Md ← mem(MAR)</td>
<td>{rd_mod_wr}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC ixm1</td>
<td>{addr}</td>
<td>MARL ← mem(PC)</td>
<td>{addx}</td>
<td>MAR ← MAR + X</td>
<td>{data}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC ← PC + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2nd cycle</td>
<td>3rd cycle</td>
<td>4th cycle</td>
<td>5th cycle</td>
<td>6th cycle</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------------</td>
<td>------------------------------------------------</td>
<td>------------------------------------------------</td>
<td>------------------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>JMP dir</td>
<td>{addr1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCL ← mem(PC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP ix</td>
<td>{addr1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCL ← X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP ix1</td>
<td>{addr1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MARL ← mem(PC)</td>
<td>{addx}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 1</td>
<td>PC ← MAR + X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP ext</td>
<td>{addr1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MARH ← mem(PC)</td>
<td>{addr2}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 1</td>
<td>PCL ← mem(PC)</td>
<td>PCH ← MARH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP ix2</td>
<td>{addr1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MARH ← mem(PC)</td>
<td>{addr2}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 1</td>
<td>MARL ← mem(PC)</td>
<td>PC ← PC + 1</td>
<td>{addx}</td>
<td>PC ← MAR + X</td>
</tr>
<tr>
<td>JSR dir</td>
<td>{addr1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MARL ← mem(PC)</td>
<td>{push1}</td>
<td>mem(SP) ← PCL</td>
<td>{push2}</td>
<td>mem(SP) ← PCH</td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 1</td>
<td>SP ← SP - 1</td>
<td>SP ← SP - 1</td>
<td>PC ← MAR</td>
<td>SP ← SP - 1</td>
</tr>
<tr>
<td>JSR ix</td>
<td>{addr1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MARL ← X</td>
<td>{push1}</td>
<td>mem(SP) ← PCL</td>
<td>{push2}</td>
<td>mem(SP) ← PCH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SP ← SP - 1</td>
<td>SP ← SP - 1</td>
<td>PC ← MAR</td>
<td>SP ← SP - 1</td>
</tr>
<tr>
<td>JSR ix1</td>
<td>{addr1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MARL ← mem(PC)</td>
<td>{addx}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 1</td>
<td>MAR ← MAR + X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSR ext</td>
<td>{addr1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MARH ← mem(PC)</td>
<td>{addr2}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 1</td>
<td>MARL ← mem(PC)</td>
<td>PC ← PC + 1</td>
<td>{push1}</td>
<td>mem(SP) ← PCL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC ← PC + 1</td>
<td>SP ← SP - 1</td>
<td>{push2}</td>
<td>mem(SP) ← PCH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SP ← SP - 1</td>
<td>PC ← MAR</td>
<td>SP ← SP - 1</td>
</tr>
<tr>
<td>JSR ix2</td>
<td>{addr1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MARH ← mem(PC)</td>
<td>{addr2}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC ← PC + 1</td>
<td>MARL ← mem(PC)</td>
<td>PC ← PC + 1</td>
<td>{addx}</td>
<td>MAR ← MAR + X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2nd cycle</td>
<td>3rd cycle</td>
<td>4th cycle</td>
<td>5th cycle</td>
<td>6th cycle</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>RTS</td>
<td><code>{addr1}</code></td>
<td><code>{pop2}</code></td>
<td><code>{pop1}</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SP ← SP + 1</td>
<td>PCH ← mem(SP)</td>
<td>PCL ← mem(SP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRA rel</td>
<td><code>{addr1}</code></td>
<td><code>{BRtest}</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Md ← mem(PC)</td>
<td>PC ← PC + {sign_ext&amp;Md}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWI</td>
<td><code>{addr1}</code></td>
<td><code>{push1}</code></td>
<td><code>{push2}</code></td>
<td><code>{push3}</code></td>
<td><code>{push4}</code></td>
</tr>
<tr>
<td></td>
<td>no action</td>
<td>mem(SP) ← PCL</td>
<td>mem(SP) ← PCH</td>
<td>mem(SP) ← X</td>
<td>mem(SP) ← A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SP ← SP - 1</td>
<td>SP ← SP - 1</td>
<td>SP ← SP - 1</td>
<td>SP ← SP - 1</td>
</tr>
<tr>
<td>SWI continued</td>
<td>(7th cycle)</td>
<td>(8th cycle)</td>
<td>(9th cycle)</td>
<td>(10th cycle)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>{push5}</code></td>
<td><code>{cycle8}</code></td>
<td><code>{cycle9}</code></td>
<td><code>{cycle10}</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mem(SP) ← CCR</td>
<td>MAR ← vector addr.</td>
<td>PCH ← mem(MAR)</td>
<td>PCL ← mem(MAR)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I ← 1</td>
<td>MAR ← MAR + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td><code>{addr1}</code></td>
<td><code>{pop5}</code></td>
<td><code>{pop4}</code></td>
<td><code>{pop3}</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SP ← SP + 1</td>
<td>CCR ← mem(SP)</td>
<td>A ← mem(SP)</td>
<td>X ← mem(SP)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SP ← SP + 1</td>
<td>SP ← SP + 1</td>
<td>SP ← SP + 1</td>
<td></td>
</tr>
</tbody>
</table>

*Completion of ALU operation overlaps fetch of next instruction.*
Table 11-5  Example of Cycle-by-Cycle Instruction Execution

<table>
<thead>
<tr>
<th>PC</th>
<th>MAR</th>
<th>Address Bus</th>
<th>Data Bus</th>
<th>Opcode Reg.</th>
<th>A</th>
<th>Md</th>
</tr>
</thead>
<tbody>
<tr>
<td>0300</td>
<td>xxxx</td>
<td>0300</td>
<td>BB</td>
<td>xx</td>
<td>12</td>
<td>xx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0301</td>
<td>xxxx</td>
<td>0301</td>
<td>43</td>
<td>BB</td>
<td>12</td>
<td>xx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0302</td>
<td>0043</td>
<td>0043</td>
<td>36</td>
<td>BB</td>
<td>12</td>
<td>xx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0302</td>
<td>0043</td>
<td>0302</td>
<td>B7</td>
<td>BB</td>
<td>12</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0303</td>
<td>0043</td>
<td>0303</td>
<td>57</td>
<td>B7</td>
<td>48</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0304</td>
<td>0057</td>
<td>0057</td>
<td>48</td>
<td>B7</td>
<td>48</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0304</td>
<td>0057</td>
<td>0304</td>
<td>4C</td>
<td>B7</td>
<td>48</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 11-15 Partial State Graph for CPU Controller

- **push1**: JSR
- **push2**: hw_int
- **push3**: JSR
- **push4**: SWI
- **push5**: JSR_dir
- **push5**: rst
- **push5**: reset
- **cycle8**: rst
- **cycle9**: rst
- **cycle10**: cycle9
- **fetch**: imm, inha, inhx
- **fetch**: JMP_dir
- **addr1**: push1
- **addr1**: JSR_dir
- **addr1**: rel
- **addr1**: testBR
- **addr2**: ix1, ix2
- **addr2**: dir, ix
- **addr2**: ext, ix2
- **addr2**: RTS
- **addx**: ix2
- **addx**: RTS
- **data**: ext
- **data**: RTI
- **data**: else
- **data**: rd_mod_wr
- **data**: writeback
- **data**: pop1
- **data**: pop2
- **data**: pop3
- **data**: pop4
- **data**: pop5
- **push1**: push3
- **push1**: push4
- **push1**: push5
Main Sections of Behavioral Code for CPU:

1. Signal and control declarations;
2. Procedure \texttt{ALU\_OP}, which performs ALU operations for register-memory instructions including all two-operand instructions;
3. Procedure \texttt{ALU\_1}, which performs ALU and shifting operations for read-modify-write instructions, which have a single operand;
4. Procedure \texttt{fill\_memory}, which fills the memory with instructions and data for test purposes;
5. Process \texttt{cpu\_cycles}, which specifies the state sequence for the CPU controller and the actions which occur in each state.

Examples of 4-bit codes for opcodes and addressing modes:

```vhdl
-- lower 4 bits of opcode (specifies operation)
subtype ot is std_logic_vector(3 downto 0);
constant SUB: ot:="0000"; constant CMP: ot:="0001";
...

-- upper 4 bits of opcode (specifies addressing mode)
constant REL: ot:="0010"; constant DIRM: ot:="0011";
...```

procedure ALU_OP -- perform ALU operation with 2 operands
    (Md : in std_logic_vector(7 downto 0);
    signal A, X : inout std_logic_vector(7 downto 0);
    signal N, Z, C : inout std_logic) is
variable res : std_logic_vector(8 downto 0); -- result of ALU operation
variable updateNZ : Boolean := TRUE; -- update NZ flags by default
begin
    case OP is
    when LDA => res := '0'&Md; A <= res(7 downto 0);
    when LDX => res := '0'&Md; X <= res(7 downto 0);
    when ADD => res := ('0'&A) + ('0'&Md); C <= res(8); A <= res(7 downto 0);
    when ADC => res := ('0'&A) + ('0'&Md) + C; C <= res(8); A <= res(7 downto 0);
    when SUB => res := ('0'&A) - ('0'&Md); C <= res(8); A <= res(7 downto 0);
    when SBC => res := ('0'&A) - ('0'&Md) - C; C <= res(8); A <= res(7 downto 0);
    when CMP => res := ('0'&A) - ('0'&Md); C <= res(8);
    when CPX => res := ('0'&X) - ('0'&Md); C <= res(8);
    when ANDa => res := '0'&(A and Md); A <= res(7 downto 0);
    when BITa => res := '0'&(A or Md);
    when ORA => res := '0'&(A or Md); A <= res(7 downto 0);
    when EOR => res := '0'&(A xor Md); A <= res(7 downto 0);
    when others => updateNZ := FALSE;
    end case;
    if updateNZ then N <= res(7);
        if res(7 downto 0) = "00000000" then Z <= '1'; else Z <= '0'; end if;
    end if;
end ALU_OP;
Figure 11-17(a) Partial Listing of *cpu_cycles* Process

cpu_cycles: process
  variable reg_mem, hw_interrupt, BR: Boolean;
  variable sign_ext: std_logic_vector(4 downto 0);

begin
  reg_mem := (mode = imm) or (mode = dir) or (mode = ext) or (mode = ix) or
  (mode = ix1) or (mode = ix2);
  hw_interrupt := (I = '0') and (IRQ = '1' or SCint = '1');

  wait until rising_edge(CLK);

  if (rst_b = '0') then ST <= reset; fill_memory(mem);
  else
    case ST is
      when reset => SP <= "111111";
      if (rst_b = '1') then ST <= cycle8; end if;
      when fetch =>
        if reg_mem then ALU_OP(Md, A, X, N, Z, C); end if;
        -- complete previous operation
        if hw_interrupt then ST <= push1;
        else Opcode <= mem(CONV_INTEGER(PC)); PC <= PC+1; -- fetch opcode
          ST <= addr1; end if;
      when addr1 =>
        case mode is
          when inha => ALU1(A, N, Z, C); ST <= fetch; -- do operation on A
          when inhx => ALU1(X, N, Z, C); ST <= fetch; -- do operation on X
          when imm => Md <= mem(CONV_INTEGER(PC)); -- get immediate data
            PC <= PC+1; ST <= fetch;
        end case;
    end case;
  end if;
end process;
when inh1 =>
  if OP = SWI then ST <= push1;
  elsif OP = RTS then ST<= pop2; SP <= SP+1;
  elsif OP = RTI then ST <= pop5; SP <= SP+1;
  end if;
when inh2 =>
case OP is
  when TAX =>  X <= A;
  when CLC =>  C <= '0';
  when SEC => C <= '1';
  when CLI =>  I <= '0';
  when SEI => I <= '1';
  when RSP => SP <= "111111";
  when TXA => A <= X;
  when others =>
    assert(false) report "illegal instruction, mode = inh2";
end case;
ST <= fetch;
when dir =>
  if OP = JMP then PC <= zero&mem(CONV_INTEGER(PC)); ST <= fetch;
  else MAR <= zero&mem(CONV_INTEGER(PC)); PC <=PC+1;
  -- get direct address
  if (OP=JSR) then  ST <= push1; else ST <= data; end if;
end if;

...  (remainder of process omitted - see Appendix D)
Part of VHDL code for \{testBR\}:

```vhdl
when testBR =>
  case OP is
    when BRA => BR := TRUE; -- branch always
    when BRN => BR := FALSE;  -- branch never
    when BHI => BR := (C or Z) = '0'; -- branch if higher
    when BLS => BR := (C or Z) = '1'; -- branch if lower or same
    when BCC => BR := C = '0';  -- branch if carry clear
    when BCS => BR := C = '1';  -- branch if carry set
    ...  (other branch instructions omitted here)
  end case;
if Md(7) = '1' then sign_ext := "11111"; else sign_ext := zero; end if;
if BR then PC <= PC + (sign_ext&Md); end if;
ST <= fetch;
```
Figure 11-18 Addressing Unit
Figure 11-19 Data Unit

Data Bus

A
X
Md
CCR

ALU control

ALU

Op1
Op2

alu9

shifout

MUX1
MUX2

A X Md 0
Md 0

shiftout

8
9

CC logic
From Page 406-407

VHDL code for ALU input MUXes:

-- define ALU input operand MUXes
  op1 <= A when selA = '1' -- MUX for op1
  else X when selX = '1'
  else Md when selMd1 = '1'
  else "00000000";
  op2 <= Md when selMd2 = '1' -- MUX for op2
  else "00000000";

VHDL code for the SHIFTER:

  shiftout <= shiftin&alu9(7 downto 1) when rsh='1' -- right shift
  else alu9(6 downto 0)&shiftin when lsh='1' -- left shift
  else alu9(7 downto 0); -- pass through
VHDL code for ALU:

```vhdl
op1_com <= not op1 when com1='1' else op1; -- complementers
op2_com <= not op2 when com2='1' else op2;
alu9 <= '0'&(op1_com and op2_com) when and2 = '1'  
     else '0'&(op1_com or op2_com) when or2 = '1' 
     else '0'&(op1_com xor op2_com) when xor2 = '1' 
     else ('0'&op1_com) + ('0'&op2_com) + Cin;      -- adder
```

Synthesizable VHDL Code for CPU (Appendix E)

CPU1 architecture has the following sections:

1. Declarations of signals and constants

2. Concurrent statements for the bus interface, ALU input MUXes, ALU and shifter operations, address adder, and operation decoder

3. Process `ALU_control` which generates control signals for the ALU and for loading the \( A \), \( X \), and \( Md \) registers

4. Process `CPU_control` which implements the control state machine (Figure 11-15) and generates control signals for loading registers

5. Process `update_reg` which updates the registers on the rising edge of the clock
Data Bus Interface

-- drive the data bus with tri-state buffers
dbus <= A when A2db='1' else hi_Z;
dbus <= X when X2db='1' else hi_Z;
dbus <= Md when Md2db='1' else hi_Z;
dbus <= "000"&PCH when PCH2db='1' else hi_Z;
dbus <= PCL when PCL2db='1' else hi_Z;
dbus <= "1110"&CCR when CCR2db='1' else hi_Z;

4-to-16 line decoder for opcode

type decode_type is array(0 to 15) of bit_vector(15 downto 0);
signal opd : bit_vector(15 downto 0);
constant decode: decode_type :=
    (X"0001", X"0002", X"0004", X"0008", X"0010", X"0020", X"0040", X"0080",
     X"0100", X"0200", X"0400", X"0800", X"1000", X"2000", X"4000", X"8000");
...

opd <= decode(TO_INTEGER(Opcode(3 downto 0)));-- 4-to-16 decoder
Partial VHDL code for ALU_control:

-- set default values of control signals
    
    Cin <= '0'; com1 <= '0'; com2 <= '0'; updateNZ<='1'; updateC<='1';
    ALU2A <= '1'; ALU2X <= '0'; ALU2Md <= '0';
    selA <= '1'; selX <= '0'; selMd1 <= '0'; selMd2 <= '1';

    ... if SUBC='1' then com2<='1'; Cin<= not C; end if;
    if CPX = '1' then selX <= '1'; selA <= '0'; com2 <= '1';
                Cin <= '1'; ALU2A<='0'; end if;
Figure 11-21(a) Partial Listing of CPU_control Process

CPU_control: **process** (ST, rst_b, opd, mode, IRQ, SCint, CCR, MAR, X, PC, Md)

    variable reg_mem, hw_interrupt, BR : boolean;

begin
nST <= reset; BR := FALSE; wr <= '0'; update <= '0';
xadd1 <= (others => '0'); xadd2 <= (others => '0'); va <= "000";
db2A <= '0'; db2X <= '0'; db2Md <= '0'; db2CCR <= '0'; db2opcode <= '0';
... (all control signals are set to '0' here)

reg_mem:= (mode = imm) **or** (mode = dir) **or** (mode = ext) **or** (mode = ix)
    **or** (mode = ix1) **or** (mode = ix2);
hw_interrupt := (I = '0') **and** (IRQ = '1' **or** SCint = '1');

if (rst_b = '0') then nST <= reset;
else
case ST is
when reset =>
    setSP <= '1';
    if (rst_b = '1') then nST <= cycle8; end if;
when fetch =>
    if (reg_mem **and** JMP = '0' **and** JSR = '0')
        then update <= '1'; end if; -- update registers if not JMP or JSR
    if hw_interrupt then nST <= push1;
    else PC2ab<='1'; db2opcode<='1'; incPC<='1'; -- read opcode
        nST <= addr1; end if;
...
Figure 11-21(b) Partial Listing of CPU_control Process

```vhdl
when addr1 =>
case mode is
  when inha | inhx => update <= '1'; nST<= fetch;
  when imm => PC2ab<= '1'; db2Md<='1'; incPC<='1';
     nST <= fetch;
  when inh1 =>
      if SWI = '1' then nST <= push1;
      elsif RTS = '1' then nST <= pop2; incSP<='1';
      elsif RTI = '1' then nST <= pop5; incSP<='1';
      end if;
  when inh2 => update <= '1'; nST <= fetch;
  when dir =>  PC2ab<='1';
      if JMP='1' then db2PCL<='1'; clrPCH<=1'; nST<=fetch;
      else db2MARL<='1'; clrMARH<='1'; incPC <= '1';
          if JSR='1' then nST<=push1; else nST<=data; end if;
      end if;

... (remainder of process omitted -- see Appendix E)
```
From Page 411

Code from update_reg process to update the PC:

    wait until CLK'event and CLK='1';
    if incPC = '1' then PC <= PC + 1; end if;
    if xadd2PC = '1' then PC <= xadd; end if;
    if db2PCH = '1' then PCH <= dbus(4 downto 0); end if;
    if MARH2PCH = '1' then PCH <= MARH; end if;
    if MARL2PCL = '1' then PCL <= MARL; end if;
    if clrPCH = '1' then PCH <= "00000"; end if;
    if db2PCL = '1' then PCL <= dbus; end if;
    if X2PCL = '1' then PCL <= X; end if;

Code for updating A, X, Md, and flags:

    if (update = '1') then
        if (ALU2A = '1') then A <= Shiftout; end if;
        if (ALU2X = '1') then X <= Shiftout; end if;
        if (ALU2Md = '1') then Md <= Shiftout; end if;
        if updateNZ='1' then N <= Shiftout(7);
            if Shiftout = "00000000" then Z <= '1'; else Z <= '0'; end if;
        end if;
        if updateC='1' then C <= newC; end if;
    end if;
Figure 11-22 Parallel Port Block Diagram

- Pin Interface Logic
- Port A data register
- Data Direction Register A
- Data bus
- 8 I/O pins
library ieee;
use ieee.std_logic_1164.all;

entity PORT_A is
port(clk, rst_b, Port_Sel, ADDR0, R_W: in std_logic;
    DBUS:    inout std_logic_vector(7 downto 0);
    PinA:    inout std_logic_vector(7 downto 0));
end PORT_A;

architecture port1 of PORT_A is
signal DDRA, PORTA : std_logic_vector(7 downto 0);
signal loadDDRA, loadPORTA, ReadPORTA, ReadDDRA : std_logic;
Figure 11-23(b)  VHDL Code for Parallel Port

begin
loadPORTA <= '1' when (Port_Sel='1' and ADDR0='0' and R_W='1') else '0';
loadDDRA <= '1' when (Port_Sel='1' and ADDR0='1' and R_W='1') else '0';
ReadPORTA <= '1' when (Port_Sel='1' and ADDR0='0' and R_W='0') else '0';
ReadDDRA <= '1' when (Port_Sel='1' and ADDR0='1' and R_W='0') else '0';

-- pin interface logic
Portbits: for i in 7 downto 0 generate
  PinA(i) <= PORTA(i) when DDRA(i) = '1' else 'Z';  -- set external pin state
  DBUS(i) <= DDRA(i) when (ReadDDRA = '1')  -- read data direction register
    else PinA(i) when (ReadPORTA = '1')
    else 'Z';
end generate;

process (clk, rst_b)  -- this process writes to the port registers
begin
  if (rst_b = '0') then DDRA <= "00000000";  -- set all pins to inputs
    elsif (rising_edge(clk)) then
      if (loadDDRA = '1') then DDRA <= DBUS; end if;
      if (loadPORTA = '1') then PORTA <= DBUS;end if;
    end if;
end process;
end port1;
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);
    cs, 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, cs1, cs2, 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;
    cs2 <= SelHiRam and not clk;
    we <= wr and not clk;
    -- select ram on 2nd half of clock cycle
    -- write enable on 2nd half of clock cycle
Figure 11-24(c)  Top-level VHDL for 6805 Microcontroller

-- 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) = "0000001" else '0';
-- 32 <= addr <= 63
SelHiRam <= '1' when addr_bus(12 downto 5) = "1111111" else '0';
-- addr >= 8160 (1FE0h)
end M6805_64;
Figure 11-25  Multiplexed Data Bus Structure
Summary of steps used to design CPU:

1. Define the register structure, instruction set, and addressing modes.
2. Construct a table which 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.