

Figure 11-1 Serial Data Transmission

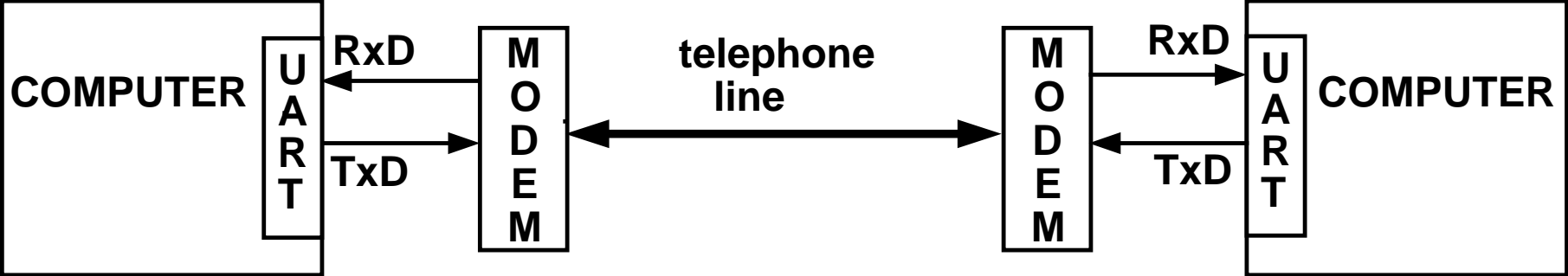


Figure 11-2 Standard Serial Data Format

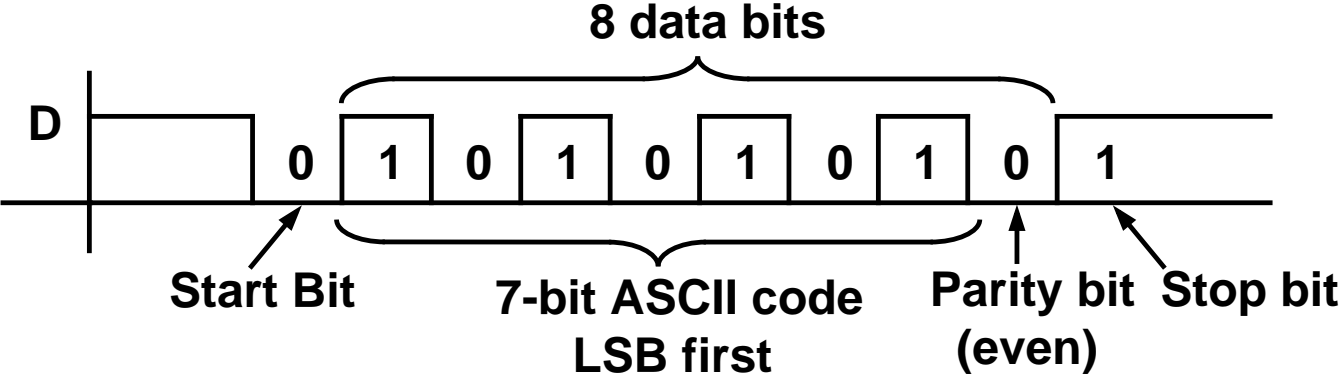


Figure 11-3 UART Block Diagram

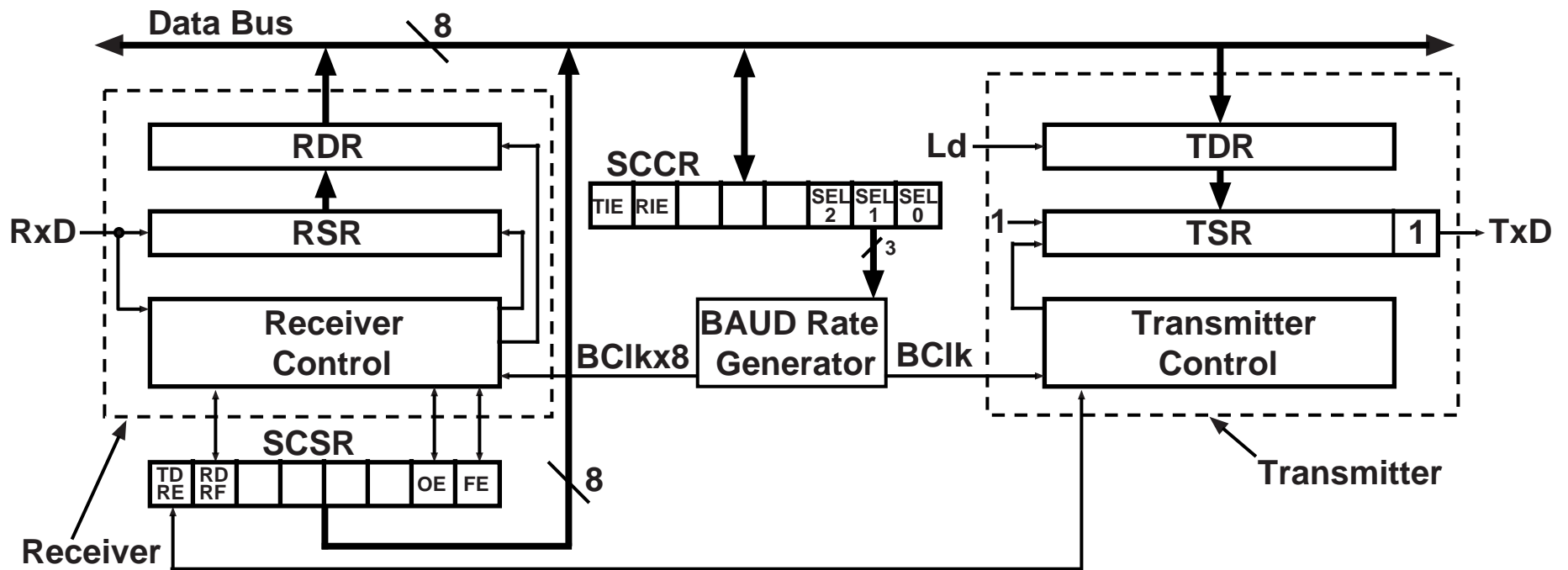


Figure 11-4 SM Chart for UART Transmitter

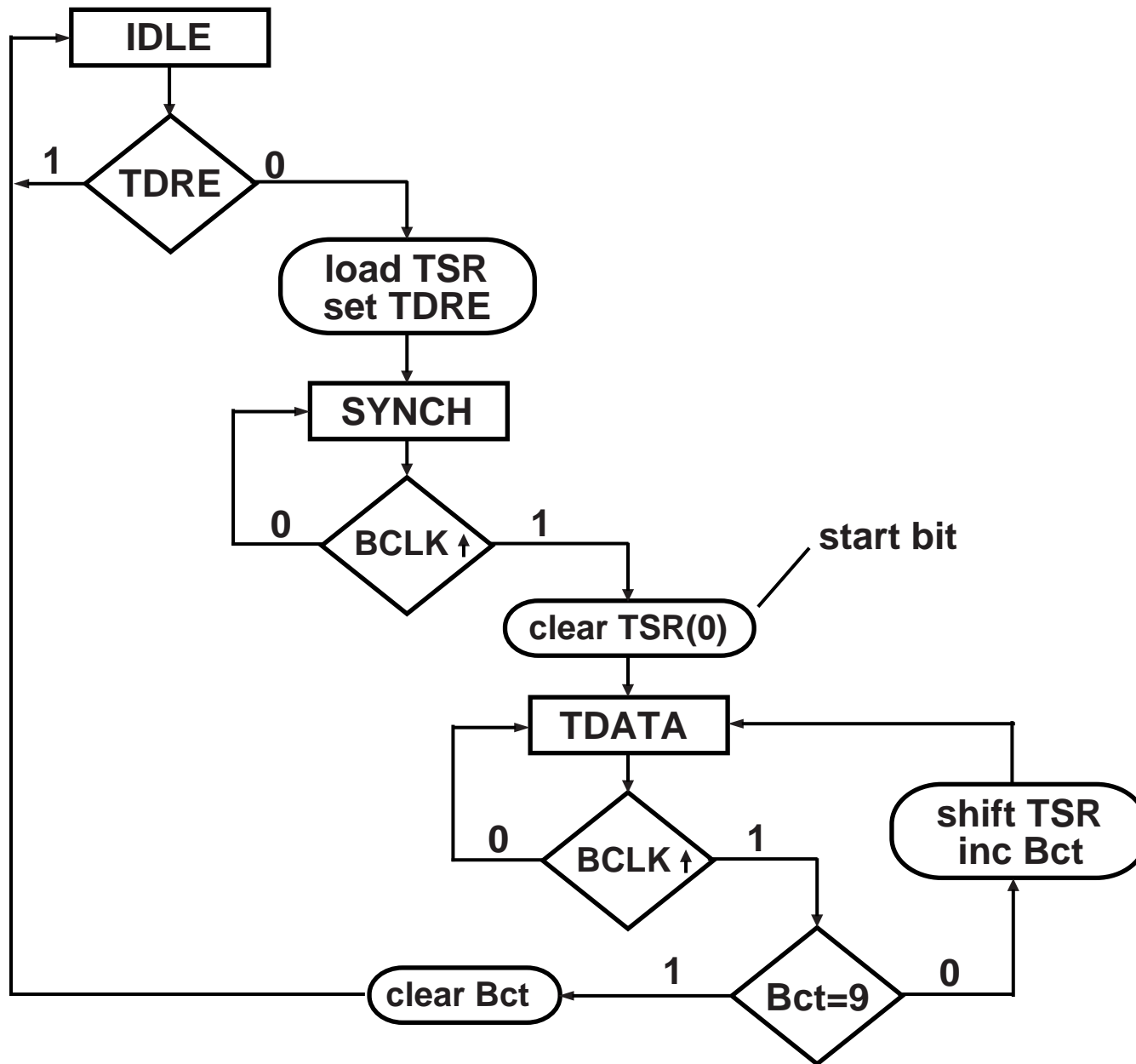


Figure 11-5(a) VHDL Code for UART Transmitter

```
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 (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;
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;
```

Figure 11-5(b) VHDL Code for UART Transmitter

```
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;

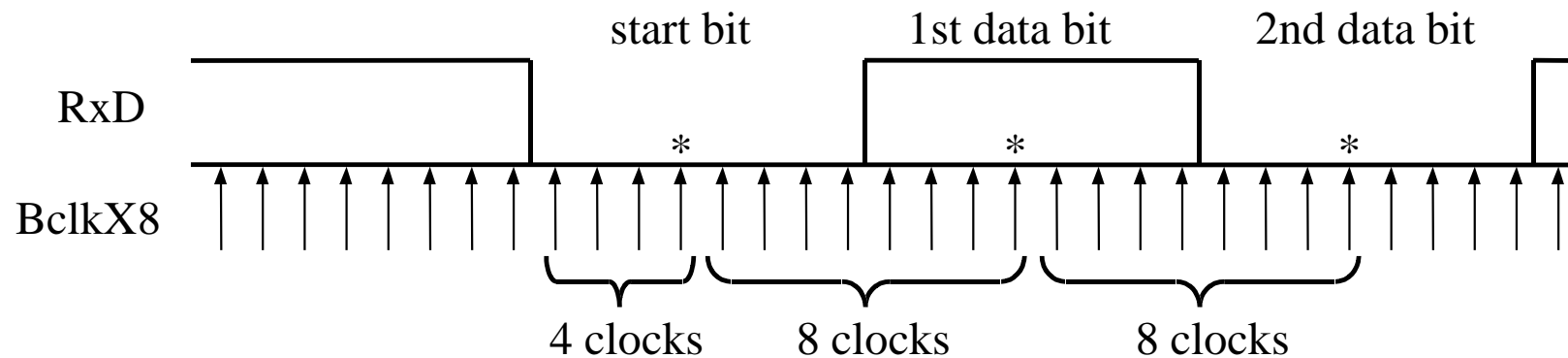
Xmit_update: process (sysclk, rst_b)
begin
    if (rst_b = '0') then
        TSR <= "111111111"; state <= IDLE; Bct <= 0; Bclk_dlayed <= '0';
    elsif (sysclk'event and sysclk = '1') then
        state <= nextstate;
        if (clr = '1') then Bct <= 0; elsif (inc = '1') then
            Bct <= Bct + 1; end if;
        if (loadTDR = '1') then TDR <= DBUS; end if;
        if (loadTSR = '1') then TSR <= TDR & '1'; end if;
        if (start = '1') then TSR(0) <= '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;
```

From Page 378

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

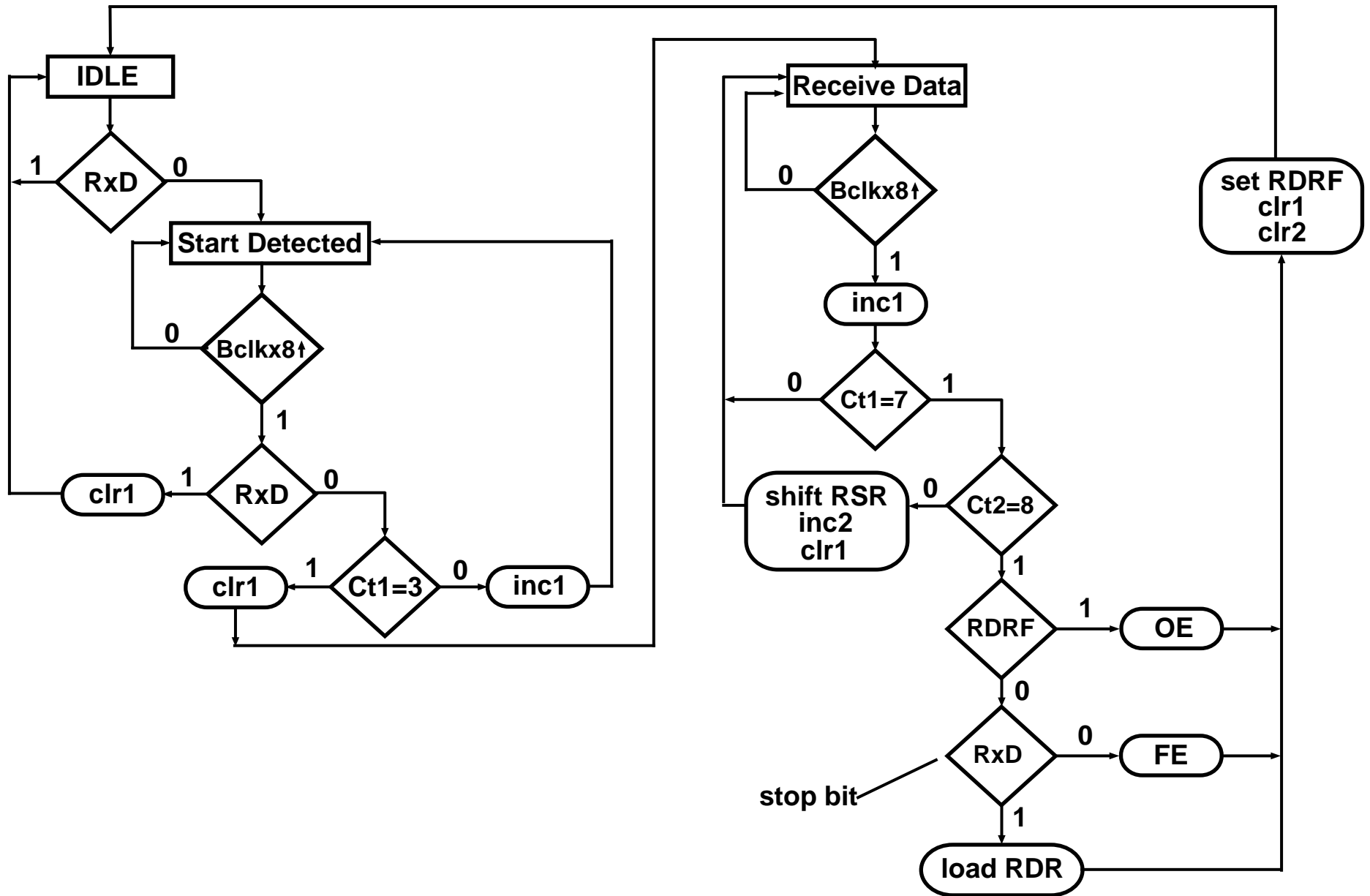


Figure 11-8(a) VHDL Code for UART Receiver

```
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_Dlaged, BclkX8_rising : std_logic;

begin
BclkX8_rising <= BclkX8 and (not BclkX8_Dlaged);
      -- 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

```
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;
```

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

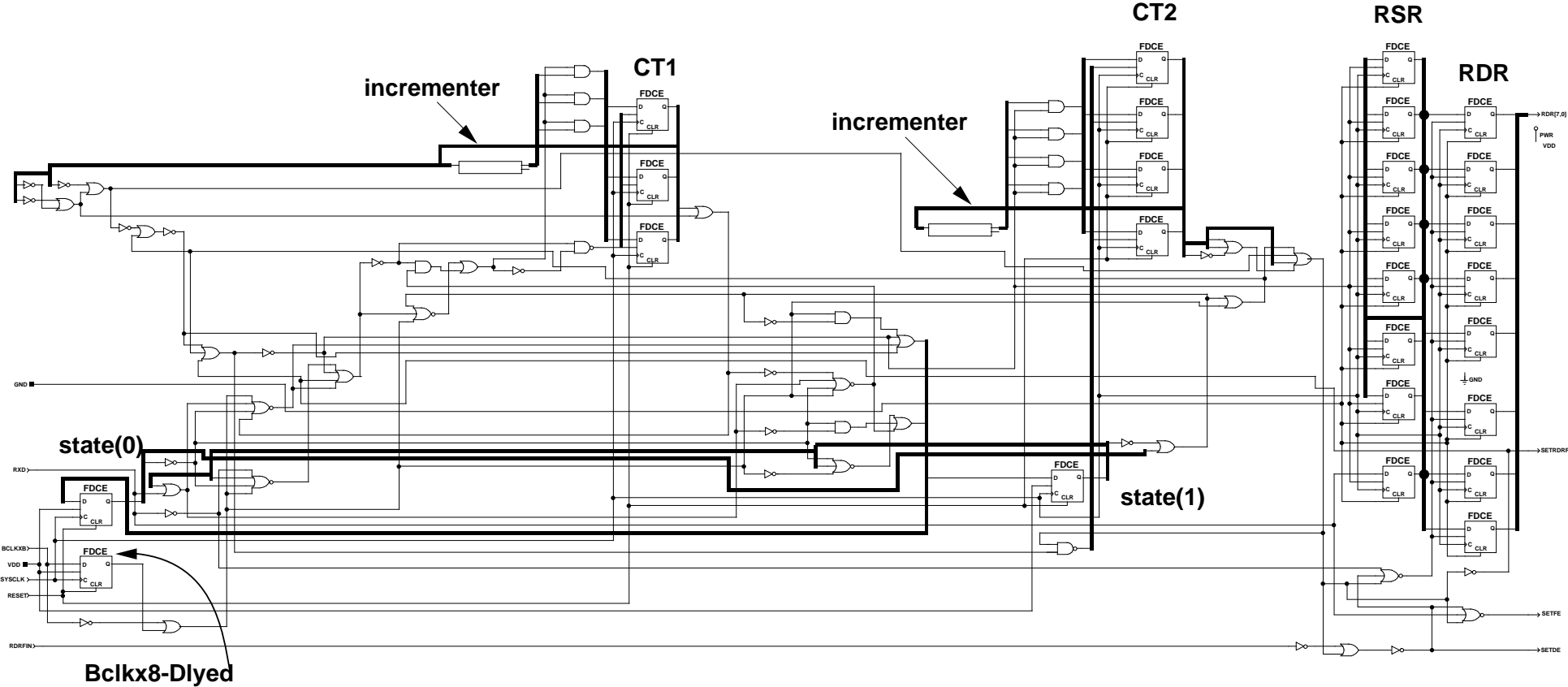
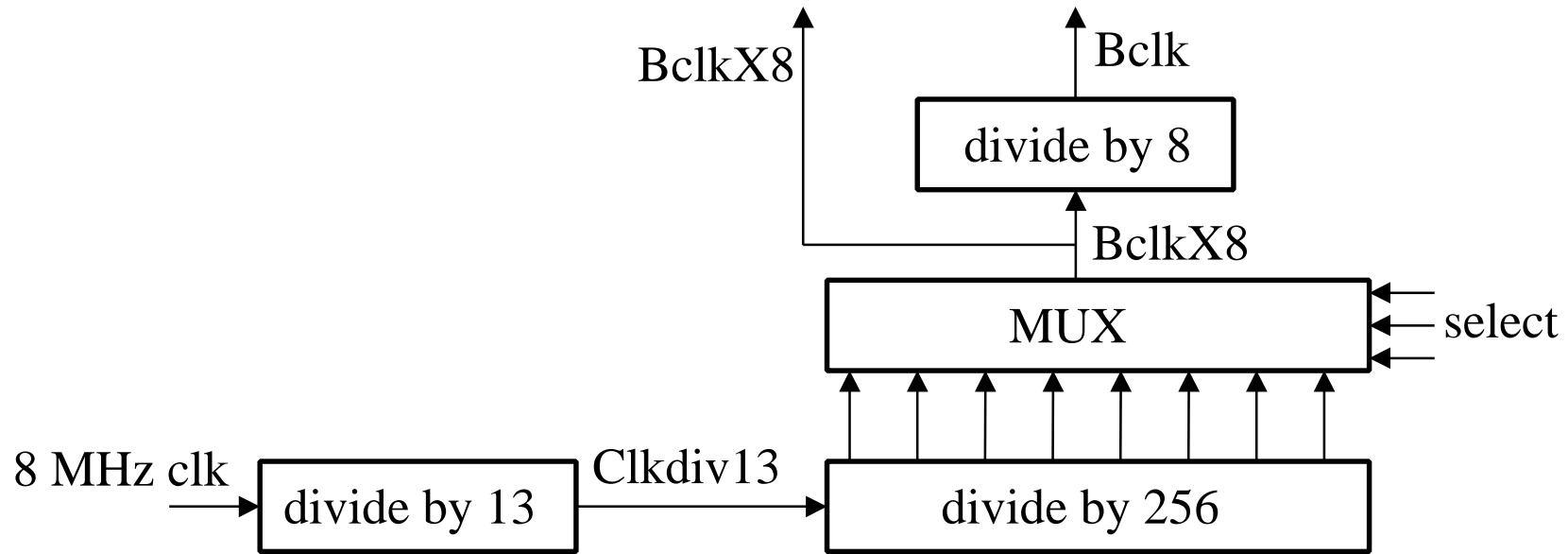


Figure 11-10 Baud Rate Generator



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

Figure 11-11(a) VHDL Code for BAUD Rate Generator

```
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) -- clk_divdr is an 8-bit counter
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;  
  
entity 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

```
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);
```

Figure 11-12(c) VHDL Code for Complete UART

```
        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 <= "ZZZZZZZZ" 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

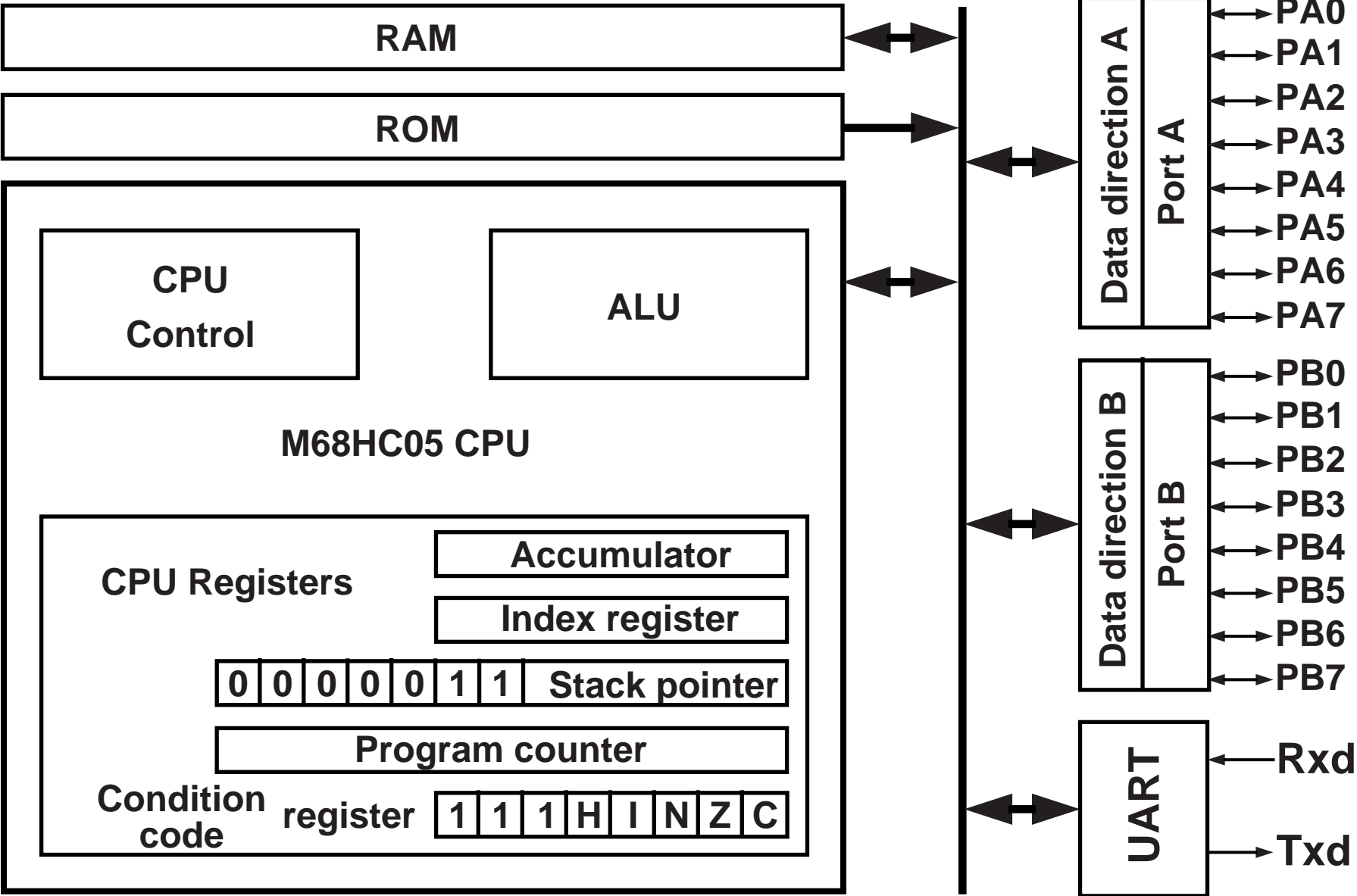


Figure 11-14 MC68HC05C4 Programming Model

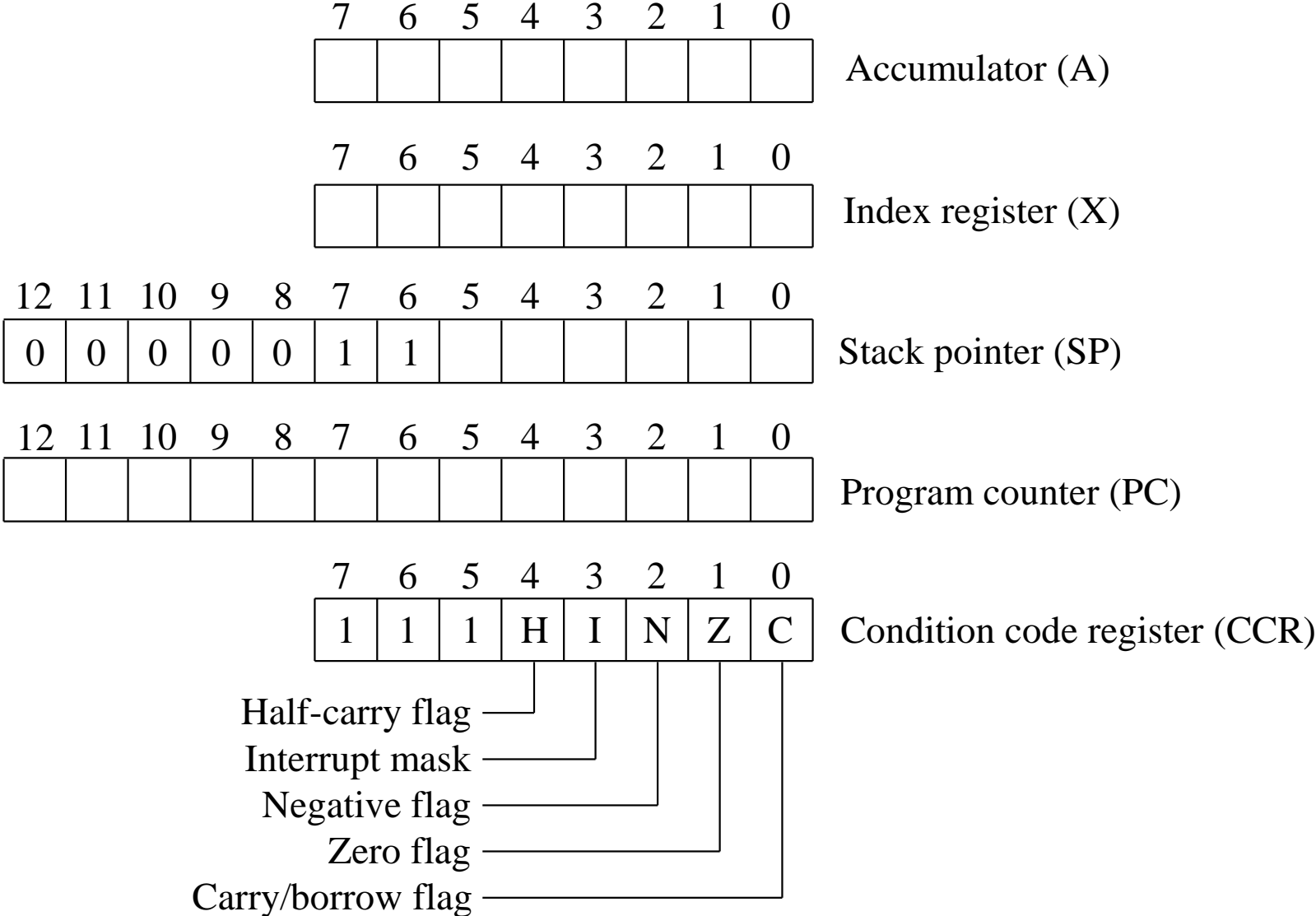


Table 11-1 Opcodes for MC6805

		Bit Manipulation		Branch	Read/Modify/Write				Control		Register/Memory						
		BTB	BSC	REL	DIRM	INHA	INHX	IX1M	IXM	INH1	INH2	IMM	DIR	EXT	IX2	IX1	IX
Hi	Lo	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		BRSET0*	BSET0*	BRA	NEG	NEG	NEG	NEG	NEG	RTI		SUB	SUB	SUB	SUB	SUB	SUB
1		BRCLR0*	BCLR0*	BRN						RTS		CMP	CMP	CMP	CMP	CMP	CMP
2		BRSET1*	BSET1*	BHI		MUL*						SBC	SBC	SBC	SBC	SBC	SBC
3		BRCLR1*	BCLR1*	BLS	COM	COM	COM	COM	COM	SWI		CPX	CPX	CPX	CPX	CPX	CPX
4		BRSET2*	BSET2*	BCC	LSR	LSR	LSR	LSR	LSR			AND	AND	AND	AND	AND	AND
5		BRCLR2*	BCLR2*	BCS								BIT	BIT	BIT	BIT	BIT	BIT
6		BRSET3*	BSET3*	BNE	ROR	ROR	ROR	ROR	ROR			LDA	LDA	LDA	LDA	LDA	LDA
7		BRCLR3*	BCLR3*	BEQ	ASR	ASR	ASR	ASR	ASR		TAX		STA	STA	STA	STA	STA
8		BRSET4*	BSET4*	BHCC*	LSL	LSL	LSL	LSL	LSL		CLC	EOR	EOR	EOR	EOR	EOR	EOR
9		BRCLR4*	BCLR4*	BHCS*	ROL	ROL	ROL	ROL	ROL		SEC	ADC	ADC	ADC	ADC	ADC	ADC
A		BRSET5*	BSET5*	BPL	DEC	DEC	DEC	DEC	DEC		CLI	ORA	ORA	ORA	ORA	ORA	ORA
B		BRCLR5*	BCLR5*	BMI							SEI	ADD	ADD	ADD	ADD	ADD	ADD
C		BRSET6*	BSET6*	BMC	INC	INC	INC	INC	INC		RSP		JMP	JMP	JMP	JMP	JMP
D		BRCLR6*	BCLR6*	BMS	TST	TST	TST	TST	TST		NOP	BSR*	JSR	JSR	JSR	JSR	JSR
E		BRSET7*	BSET7*	BIL*						STOP*		LDX	LDX	LDX	LDX	LDX	LDX
F		BRCLR7*	BCLR7*	BIH*	CLR	CLR	CLR	CLR	CLR	WAIT*	TXA		STX	STX	STX	STX	STX

*not implemented in this text

Table 11-2(a) 6805 Instructions: Register–memory Instructions

Symbol	Instruction	Operation	Flags
ADD	add	$A \leftarrow A + M$	NZ, C \leftarrow carry
ADC	add with carry	$A \leftarrow A + M + C$	NZ, C \leftarrow carry
SUB	subtract	$A \leftarrow A - M$	NZ, C \leftarrow borrow
SBC	subtract with borrow	$A \leftarrow A - M - C$	NZ, C \leftarrow borrow
CMP	compare A	$A - M$	NZ, C \leftarrow borrow
CPX	compare X	$X - M$	NZ, C \leftarrow borrow
AND	and	$A \leftarrow A \text{ and } M$	NZ
BIT	bit test	$A \text{ and } M$	NZ
ORA	or	$A \leftarrow A \text{ or } M$	NZ
EOR	exclusive-or	$A \leftarrow A \text{ xor } M$	NZ
LDA	load A	$A \leftarrow M$	NZ
LDX	load X	$X \leftarrow M$	NZ
STA	store A	$M \leftarrow A$	NZ
STX	store X	$M \leftarrow X$	NZ
JMP	jump	jump to EA	
JSR	jump to subroutine	push PC on stack, jump to EA	

Table 11-2(b) 6805 Instructions: Read–modify–write Instructions

Symbol	Instruction	Operation	Flags
NEG	negate	$R \leftarrow 0 - R$	NZ, C \leftarrow borrow
COM	complement	$R \leftarrow \text{not } R$	NZ, C \leftarrow 1
TST	test	$0 - R$	NZ
CLR	clear	$R \leftarrow 0$	NZ
INC	increment	$R \leftarrow R + 1$	NZ
DEC	decrement	$R \leftarrow R - 1$	NZ
LSL	logical shift left	$R \leftarrow R(6 \text{ downto } 0) \& '0'$	NZ, C \leftarrow R(7)
ROL	rotate left	$R \leftarrow R(6 \text{ downto } 0) \& C$	NZ, C \leftarrow R(7)
ASR	arithmetic shift right	$R \leftarrow R(7) \& R(7 \text{ downto } 1)$	NZ, C \leftarrow R(0)
LSR	logical shift right	$R \leftarrow '0' \& R(7 \text{ downto } 1)$	NZ, C \leftarrow R(0)
ROR	rotate right	$R \leftarrow C \& R(7 \text{ downto } 1)$	NZ, C \leftarrow R(0)

Table 11-2(c) 6805 Instructions: Control Instructions

Symbol	Instruction	Operation
TAX	transfer A to X	$X \leftarrow A$
TXA	transfer X to A	$A \leftarrow X$
CLC	clear carry	$C \leftarrow '0'$
SEC	set carry	$C \leftarrow '1'$
CLI	clear I	$I \leftarrow '0'$
SEI	set I	$I \leftarrow '1'$
RSP	reset SP	$SP \leftarrow "111111"$
NOP	no operation	
RTI	return from interrupt	pop CCR, A, X, PCH, PCL; return to address in PC
RTS	return from subroutine	pop PCH, PCL; return to address in PC
SWI	software interrupt	push PCL, PCH, X, A, CCR jump to address from interrupt vector table

Table 11-2(d) 6805 Instructions: Branch Instructions

Symbol	Instruction	Branch if
BRA	branch always	always
BRN	branch never	never
BHI	branch if higher	(C or Z) = '0'
BLS	branch of lower or same	(C or Z) = '1'
BCC	branch if carry clear	C = '0'
BCS	branch if carry set`	C = '1'
BNE	branch if not equal	Z = '0'
BEQ	branch if equal	Z = '1'
BPL	branch if plus	N = '0'
BMI	branch if minus	N = '1'
BMC	branch if int. <u>m</u> ask clear	I = '0'
BMS	branch if int. <u>m</u> ask set	I = '1'

Table 11-3 6805 Addressing Modes

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

*rr is sign-extended before addition

Table 11-4(a) Cycle-by-Cycle Operations for 6805 Instructions

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

	2nd cycle	3rd cycle	4th cycle	5th cycle	6th cycle
ADD imm	{addr1} Md ← mem(PC) PC ← PC + 1	(A ← A + Md)*			
ADD dir	{addr1} MARL ← mem(PC) PC ← PC + 1	{data} Md ← mem(MAR)	(A ← A + Md)*		
ADD ix	{addr1} MARL ← X	{data} Md ← mem(MAR)	(A ← A + Md)*		
ADD ix1	{addr1} MARL ← mem(PC) PC ← PC + 1	{addrx} MAR ← MAR + X	{data} Md ← mem(MAR)	(A ← A + Md)*	
ADD ext	{addr1} MARH ← mem(PC) PC ← PC + 1	{addr2} MARL ← mem(PC) PC ← PC + 1	{data} Md ← mem(MAR)	(A ← A + Md)*	
ADD ix2	{addr1} MARH ← mem(PC) PC ← PC + 1	{addr2} MARL ← mem(PC) PC ← PC + 1	{addrx} MAR ← MAR + X	{data} Md ← mem(MAR)	(A ← A + Md)*
STA ext	{addr1} MARH ← mem(PC) PC ← PC + 1	{addr2} MARL ← mem(PC) PC ← PC + 1	{data} mem(MAR) ← A		
INC A	{addr1} A ← A + 1				
INC dirm	{addr1} MARL ← mem(PC) PC ← PC + 1	{data} Md ← mem(MAR)	{rd_mod_wr} Md ← Md + 1	{writeback} mem(MAR) ← Md	
INC ixm	{addr1} MARL ← X	{data} Md ← mem(MAR)	{rd_mod_wr} Md ← Md + 1	{writeback} mem(MAR) ← Md	
INC ixm1	{addr1} MARL ← mem(PC) PC ← PC + 1	{addrx} MAR ← MAR + X	{data} Md ← mem(MAR)	{rd_mod_wr} Md ← Md + 1	{writeback} mem(MAR) ← Md

Table 11-4(b) Cycle-by-Cycle Operations for 6805 Instructions

	2nd cycle	3rd cycle	4th cycle	5th cycle	6th cycle
JMP dir	{addr1} PCL ← mem(PC)				
JMP ix	{addr1} PCL ← X				
JMP ix1	{addr1} MARL ← mem(PC) PC ← PC + 1	{addrx} PC ← MAR + X			
JMP ext	{addr1} MARH ← mem(PC) PC ← PC + 1	{addr2} PCL ← mem(PC) PCH ← MARH			
JMP ix2	{addr1} MARH ← mem(PC) PC ← PC + 1	{addr2} MARL ← mem(PC) PC ← PC + 1	{addrx} PC ← MAR + X		
JSR dir	{addr1} MARL ← mem(PC) PC ← PC + 1	{push1} mem(SP) ← PCL SP ← SP - 1	{push2} mem(SP) ← PCH SP ← SP - 1 PC ← MAR		
JSR ix	{addr1} MARL ← X	{push1} mem(SP) ← PCL SP ← SP - 1	{push2} mem(SP) ← PCH SP ← SP - 1 PC ← MAR		
JSR ix1	{addr1} MARL ← mem(PC) PC ← PC + 1	{addrx} MAR ← MAR + X	{push1} mem(SP) ← PCL SP ← SP - 1	{push2} mem(SP) ← PCH SP ← SP - 1 PC ← MAR	
JSR ext	{addr1} MARH ← mem(PC) PC ← PC + 1	{addr2} MARL ← mem(PC) PC ← PC + 1	{push1} mem(SP) ← PCL SP ← SP - 1	{push2} mem(SP) ← PCH SP ← SP - 1 PC ← MAR	
JSR ix2	{addr1} MARH ← mem(PC) PC ← PC + 1	{addr2} MARL ← mem(PC) PC ← PC + 1	{addrx} MAR ← MAR + X	{push1} mem(SP) ← PCL SP ← SP - 1	{push2} mem(SP) ← PCH SP ← SP - 1 PC ← MAR

Table 11-4(c) Cycle-by-Cycle Operations for 6805 Instructions

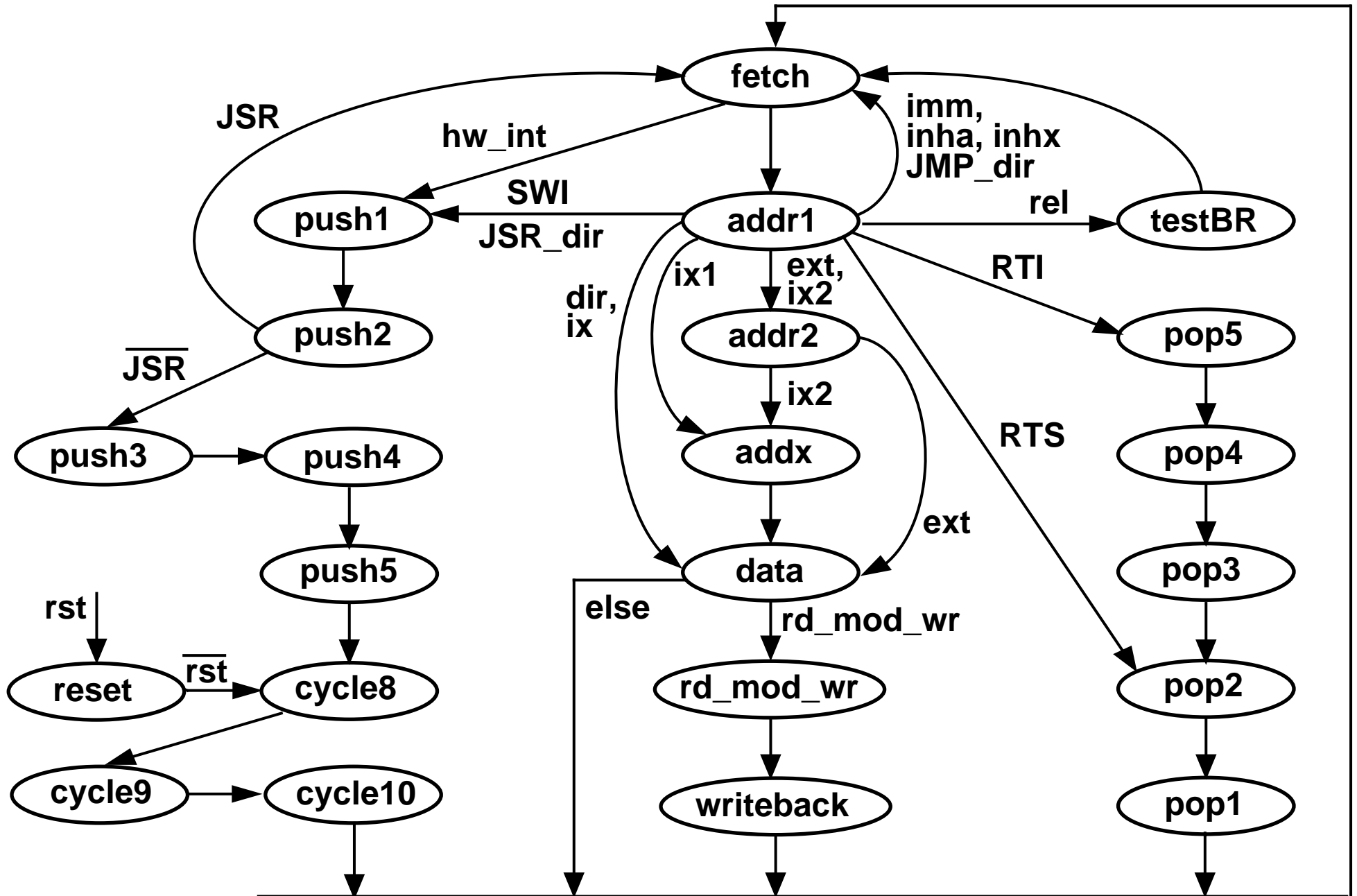
	2nd cycle	3rd cycle	4th cycle	5th cycle	6th cycle
RTS	{addr1} SP ← SP + 1	{pop2} PCH ← mem(SP) SP ← SP + 1	{pop1} PCL ← mem(SP)		
BRA rel	{addr1} Md ← mem(PC) PC ← PC + 1	{BRtest} PC ← PC + sign_ext&Md			
SWI	{addr1} no action	{push1} mem(SP) ← PCL SP ← SP - 1	{push2} mem(SP) ← PCH SP ← SP - 1	{push3} mem(SP) ← X SP ← SP - 1	{push4} mem(SP) ← A SP ← SP - 1
SWI continued	(7th cycle) {push5} mem(SP) ← CCR SP ← SP - 1	(8th cycle) {cycle8} MAR ← vector addr. I ← 1	(9th cycle) {cycle9} PCH ← mem(MAR) MAR ← MAR + 1	(10th cycle) {cycle10} PCL ← mem(MAR)	
RTI	{addr1} SP ← SP + 1	{pop5} CCR ← mem(SP) SP ← SP + 1	{pop4} A ← mem(SP) SP ← SP + 1	{pop3} X ← mem(SP) SP ← SP + 1	6th and 7th cycles same as 3rd and 4th cycles of RTS

*Completion of ALU operation overlaps fetch of next instruction.

Table 11-5 Example of Cycle-by-Cycle Instruction Execution

<i>PC</i>	<i>MAR</i>	Address Bus	Data Bus	Opcode Reg.	<i>A</i>	<i>Md</i>	
0300	xxxx	0300	BB	xx	12	xx	fetch opcode
0301	xxxx	0301	43	BB	12	xx	get direct address
0302	0043	0043	36	BB	12	xx	read memory data
0302	0043	0302	B7	BB	12	36	do addition, fetch next opcode
0303	0043	0303	57	B7	48	36	get direct address
0304	0057	0057	48	B7	48	36	store data in memory
0304	0057	0304	4C	B7	48	36	fetch next opcode

Figure 11-15 Partial State Graph for CPU Controller



From Page 400

Main Sections of Behavioral Code for CPU:

1. Signal and control declarations;
2. Procedure *ALU_OP*, which performs ALU operations for register-memory instructions including all two-operand instructions;
3. Procedure *ALU_1*, which performs ALU and shifting operations for read-modify-write instructions, which have a single operand;
4. Procedure *fill_memory*, which fills the memory with instructions and data for test purposes;
5. Process *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:

```
-- 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";
...
```

Figure 11-16 ALU Operations for Register-Memory Instructions

```
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 and 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;

Figure 11-17(b) Partial Listing of *cpu_cycles* 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)
```

From Page 403

Part of VHDL code for {testBR}:

```
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

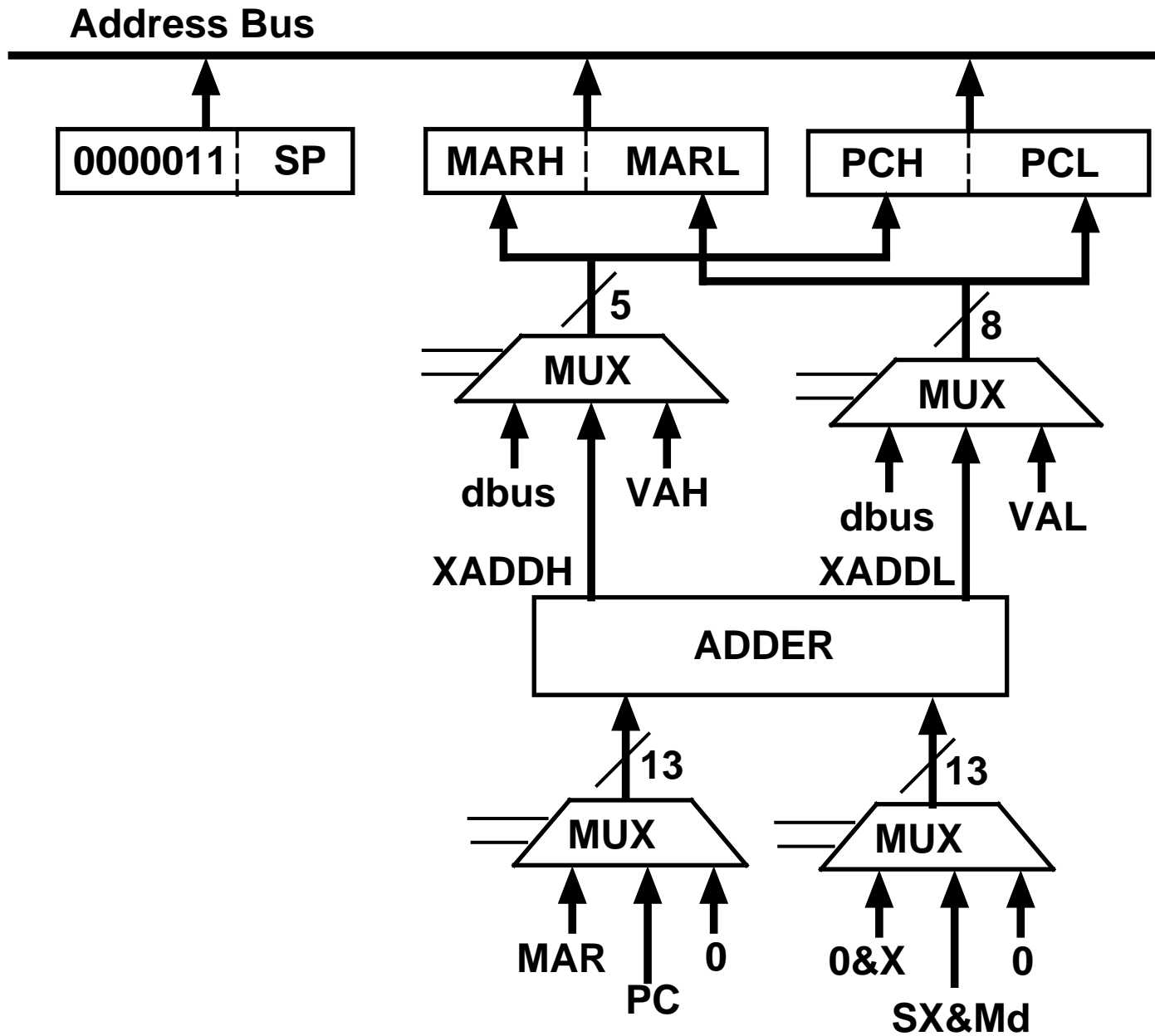
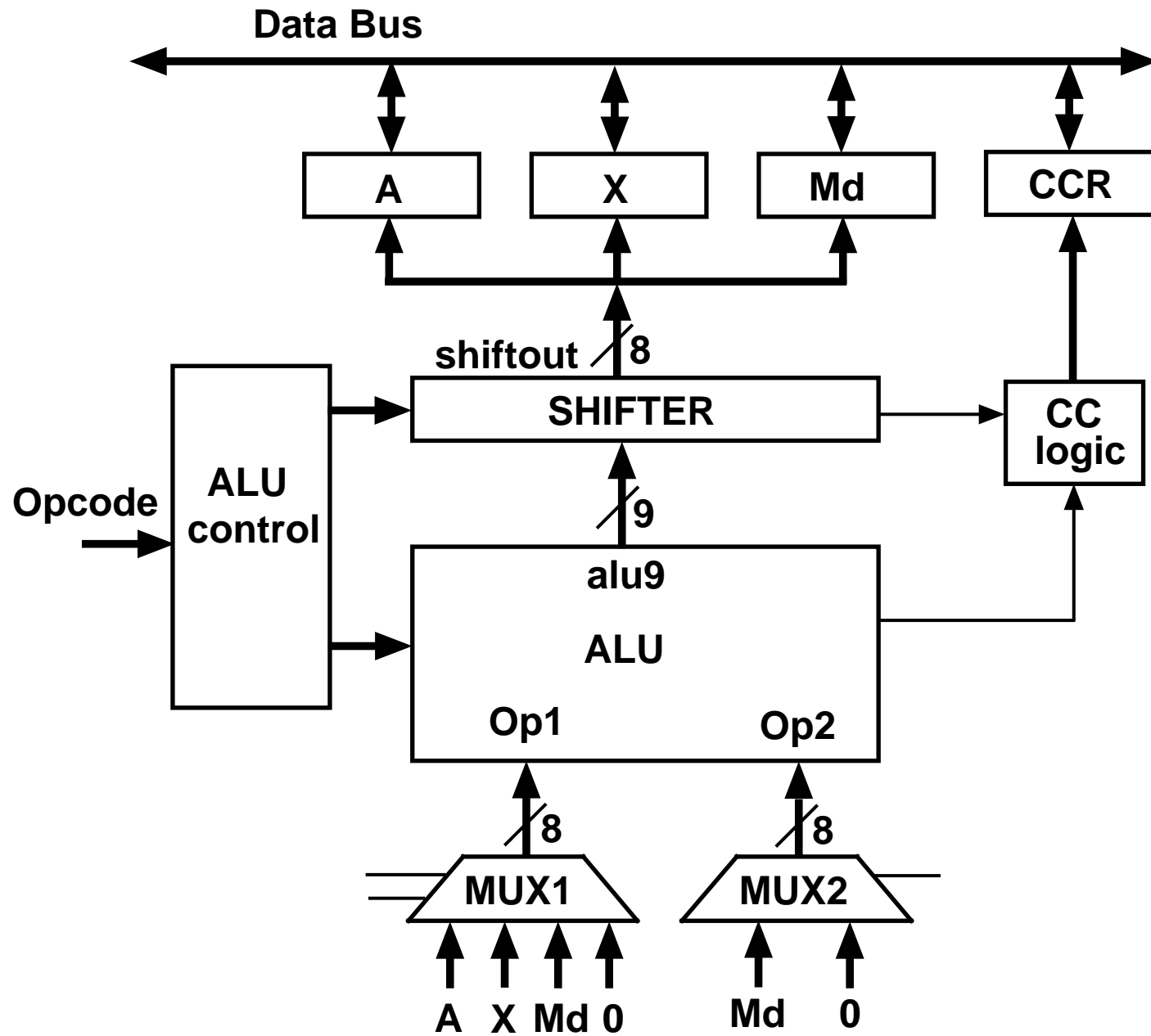


Figure 11-19 Data Unit



From Page 406-407

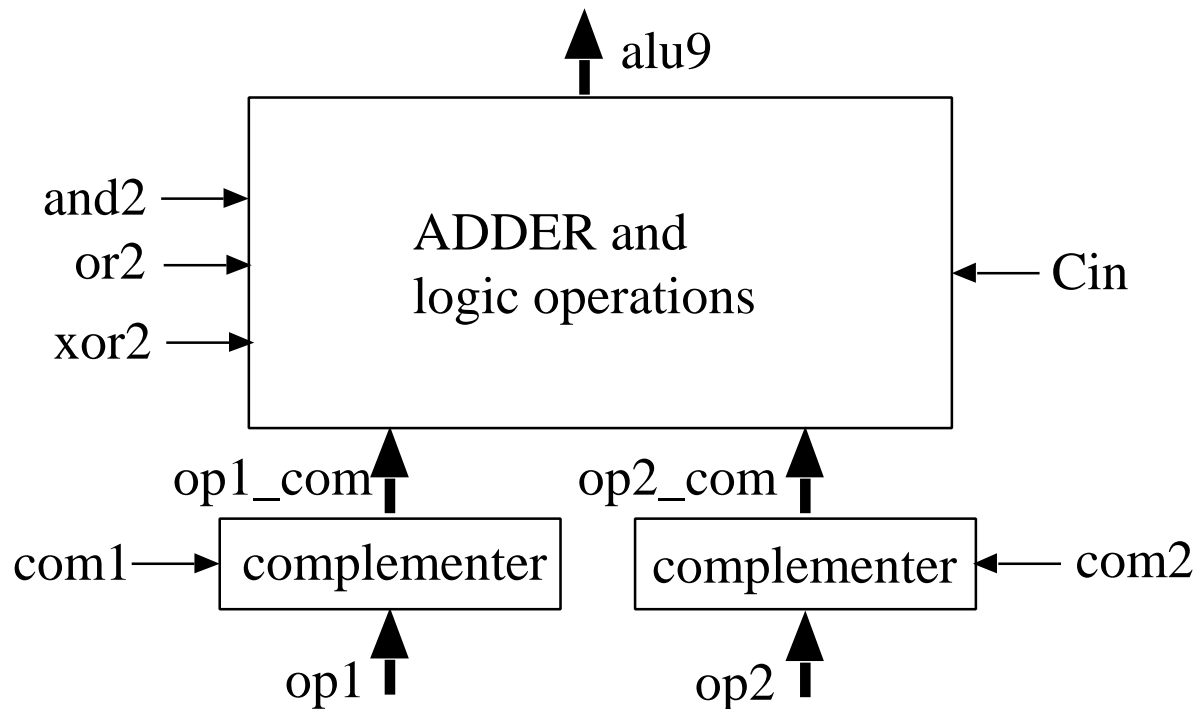
VHDL code for ALU input MUXes:

```
-- 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
```

Figure 11-20 ALU Detail



VHDL code for ALU:

```

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'    -- logic operations
      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

```

From Page 407

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

From Page 408

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
```

From Page 409

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

```
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

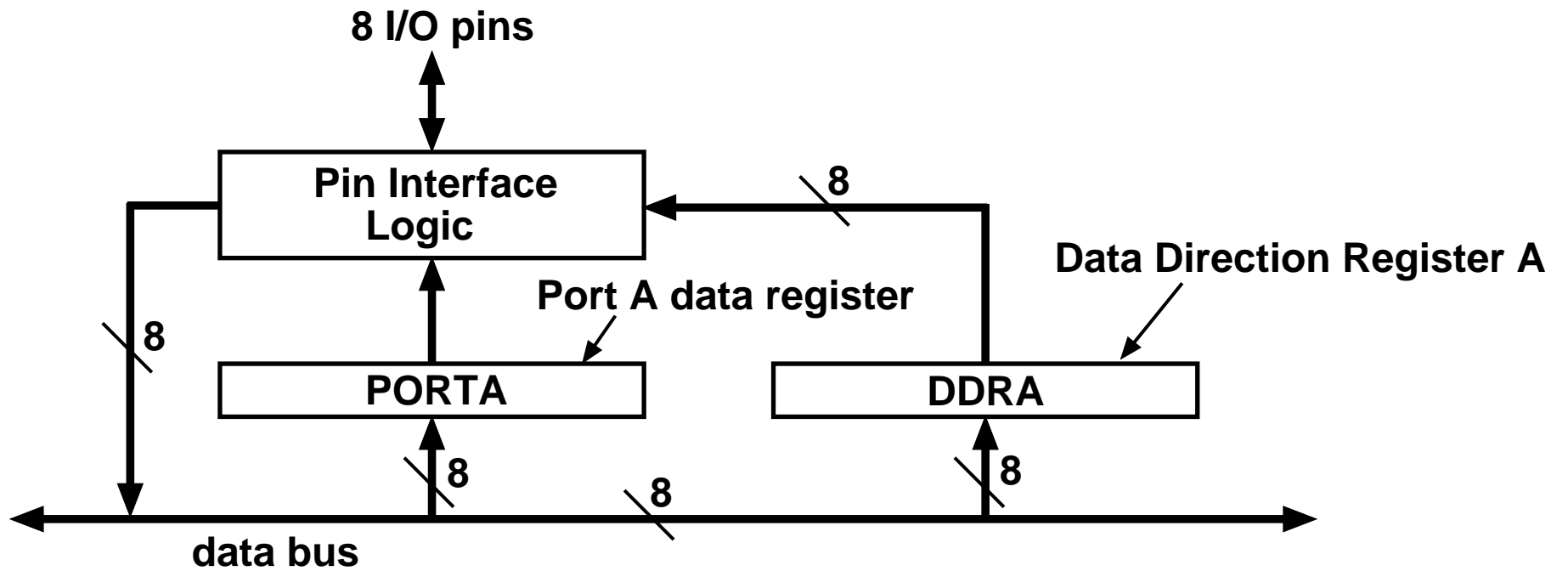


Figure 11-23(a) VHDL Code for Parallel Port

```
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;

Figure 11-24(a) Top-level VHDL for 6805 Microcontroller

```
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;
```

Figure 11-24(b) Top-level VHDL for 6805 Microcontroller

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;

-- select ram on 2nd half of clock cycle

cs2 <= SelHiRam **and not** clk;

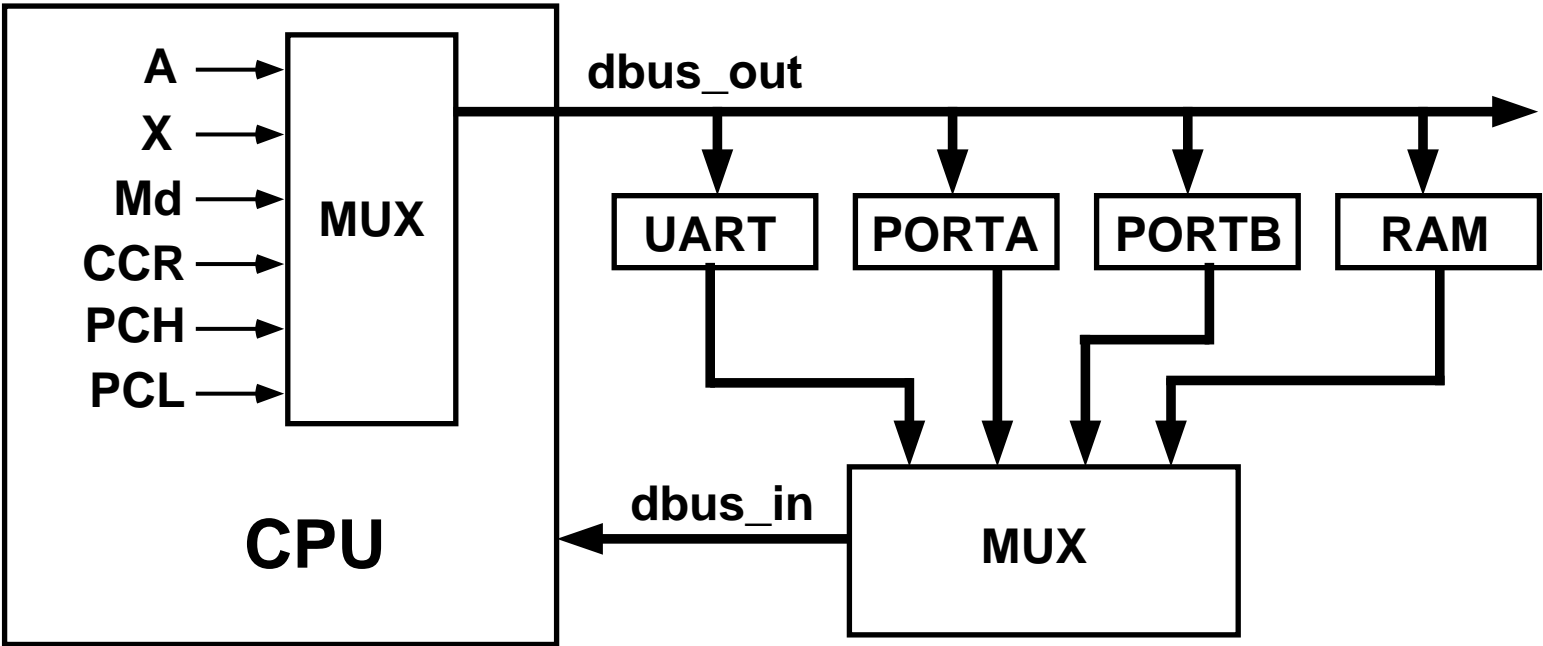
we <= wr **and not** clk;

-- write enable on 2nd half of clock cycle

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) = "00000001" else '0';
--      32 <= addr <= 63
SelHiRam <= '1' when addr_bus(12 downto 5) = "11111111" else '0';
--      addr >= 8160 (1FE0h)
end M6805_64;
```

Figure 11-25 Multiplexed Data Bus Structure



From Page 416:

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.