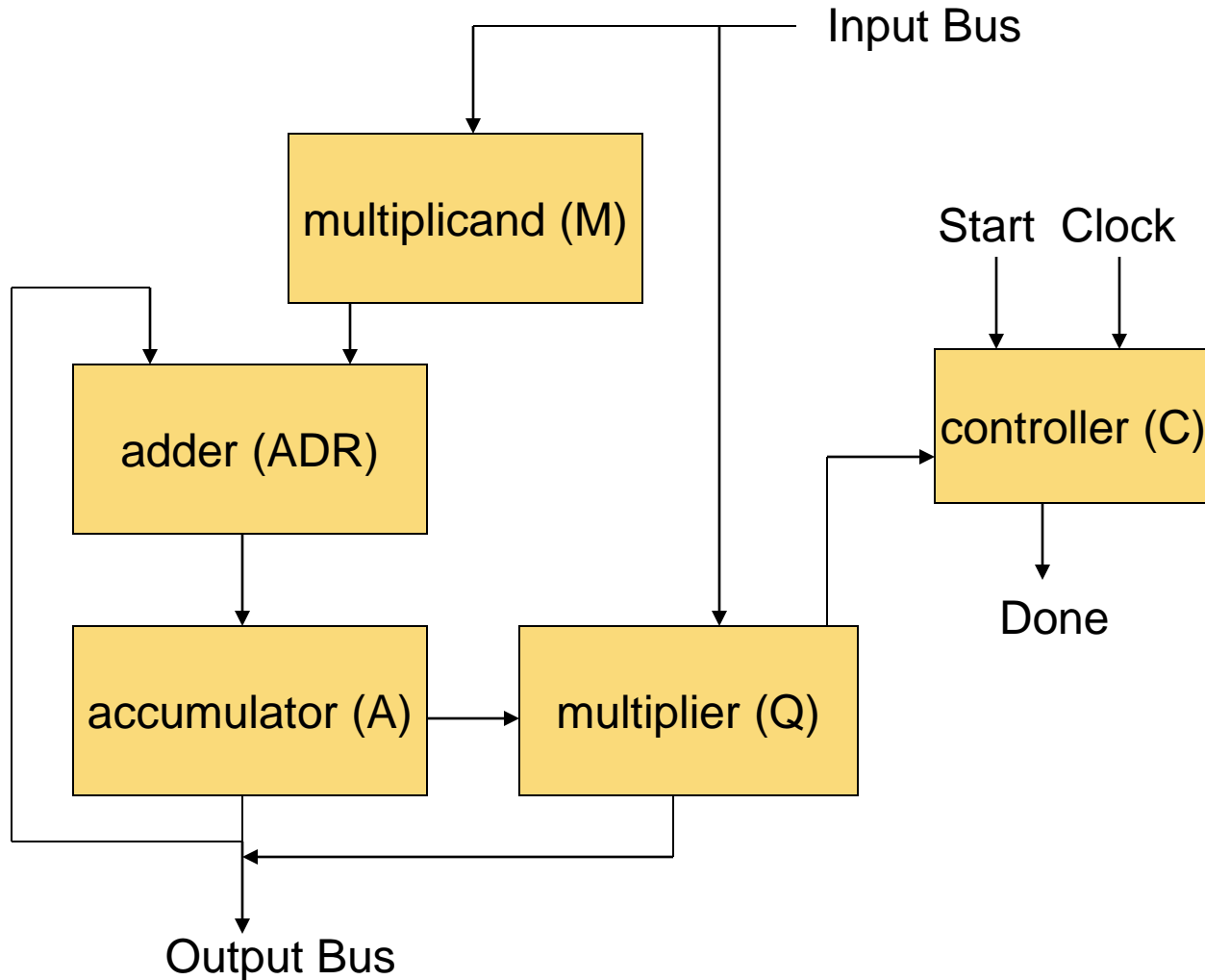
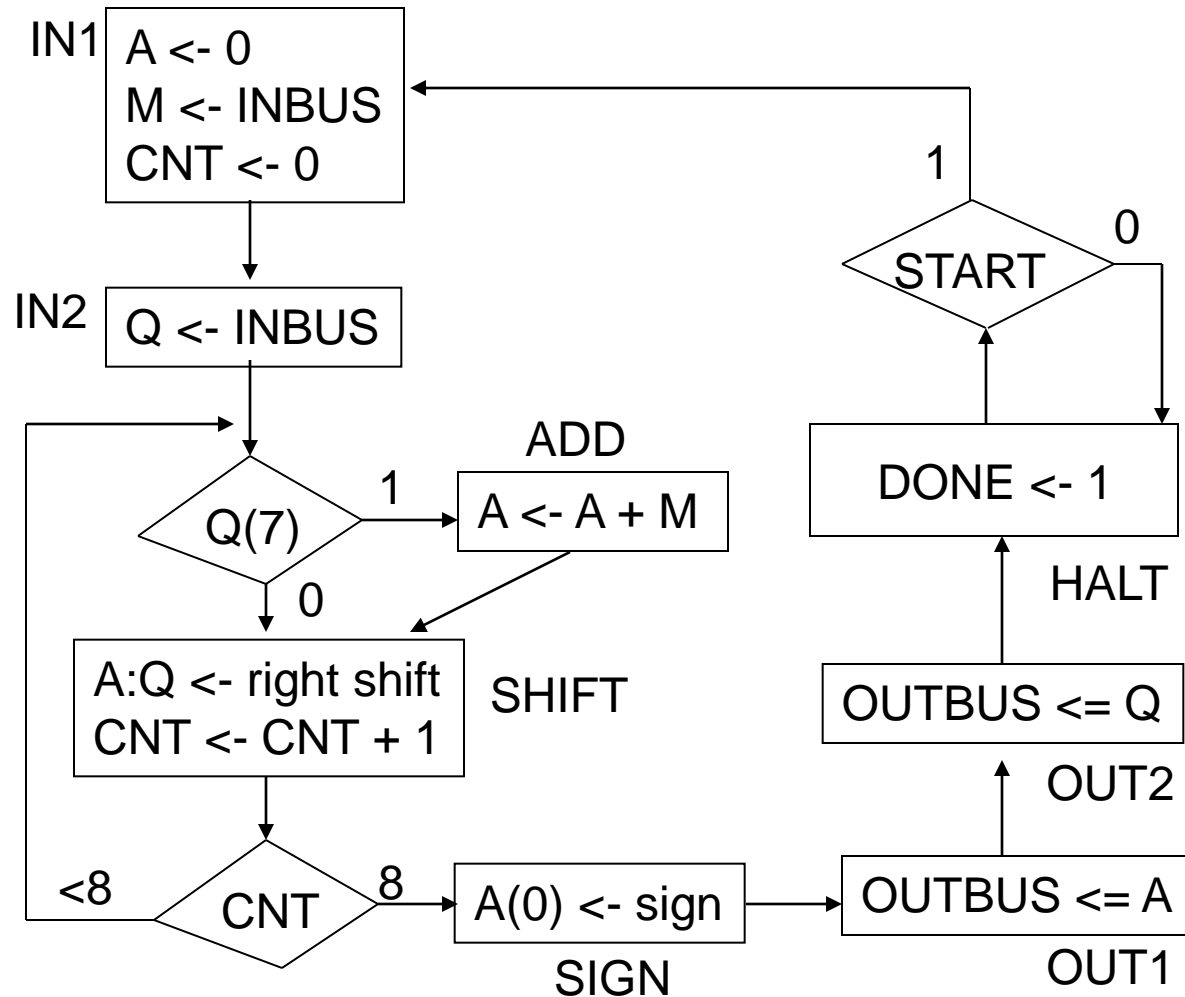


System Example: 8x8 multiplier



Multiply Algorithm



Multiplier – Top Level

entity multiplier is

```
port (INBUS:   in bit_vector(0 to 7);  
      OUTBUS: out bit_vector(0 to 7);  
      CLOCK:   in bit;  
      START:   in bit;  
      DONE:    out bit);
```

end multiplier;

architecture structure of multiplier is

[component declarations go here]

-- internal signals to interconnect components

```
signal AR, MR, QR, AD, Ain: bit_vector(0 to 7);  
signal AMload, AMadd, Qload, AQshift, AQoutEn, AQoutSel: bit;  
signal SignLd: bit;
```



Multiplier – Top Level (continued)

begin

-- Output multiplexer

```
OUTBUS <= AR when AQoutEn = '1' and AQoutSel = '0' else  
        QR when AQoutEn = '1' and AQoutSel = '1' else  
        "00000000";
```

```
Ain(0) <= AD(0) when AMadd = '1' else MR(0) xor QR(7);
```

```
Ain(1 to 7) <= AD(1 to 7);
```

```
M:  mreg  port map (INBUS, MR, AMload);
```

```
Q:  Qreg  port map (INBUS, QR, AR(7), Qload, SignLd, AQshift);
```

```
A:  areg  port map (Ain, AR, AMadd, SignLd, AQshift, AMload);
```

```
ADR: adder port map (AR, MR, AD);
```

```
C:  mctrl port map (START, CLOCK, QR(7), AMload, AMadd, Qload, AQshift,  
                  SignLd, AQoutEn, AQoutSel, DONE);
```

end;



Multiplicand Register (mreg)

-- simple parallel-load register

entity mreg is

```
port (Min: in bit_vector(0 to 7);
      Mout: out bit_vector(0 to 7);
      Load: in bit);
```

end mreg;

architecture comp of mreg is

begin

```
process (Load)          -- wait for change in Load
```

```
begin
```

```
    if Load = '1' then
```

```
        Mout <= Min;    -- parallel load
```

```
    end if;
```

```
end process;
```

```
end;
```



Accumulator Register (areg)

-- shift register with clear and parallel load

entity Areg is

```
port (Ain:  in bit_vector(0 to 7);
      Aout: out bit_vector(0 to 7);
      Load: in bit;                -- load entire register
      Load0: in bit;              -- load a0 only
      Shift: in bit;              -- shift right
      Clear: in bit);             -- clear register
```

end Areg;

architecture comp of areg is

```
    signal A: bit_vector(0 to 7);    -- internal state
```

(continue next slide)



Accumulator Register (areg)

```
begin
  Aout <= A;                                -- internal value to outputs

  process (Clear, Load, Load0, Shift) -- wait for event
  begin
    if Clear = '1' then
      A <= "00000000";                      -- clear register
    elsif Load = '1' then
      A <= Ain;                             -- parallel load
    elsif Shift = '1' then
      A <= '0' & A(0 to 6);                 -- right shift
    elsif Load0 = '1' then
      A(0) <= Ain(0);                      -- load A(0) only
    end if;
  end process;
end;
```



Multiplier/Product Register (Qreg)

-- shift register with parallel load

entity Qreg is

port (Qin: in bit_vector(0 to 7);

Qout: out bit_vector(0 to 7);

SerIn: in bit; -- serial input for shift

Load: in bit; -- parallel load

Clear7: in bit; -- clear bit 7

Shift: in bit); -- right shift

end Qreg;

architecture comp of qreg is

signal Q: bit_vector(0 to 7); -- internal storage

(continue next slide)



Multiplier/Product Register (Qreg)

```
begin
```

```
    Qout <= Q;    -- drive output from internal storage
```

```
    process (Load, Shift, Clear7)    -- wait for event
```

```
    begin
```

```
        if Load = '1' then
```

```
            Q <= Qin;    -- load Q
```

```
        elsif Shift = '1' then
```

```
            Q <= SerIn & Q(0 to 6);    -- shift Q right
```

```
        elsif Clear7 = '1' then
```

```
            Q(7) <= '0';    -- clear bit Q(7)
```

```
        end if;
```

```
    end process;
```

```
end;
```



8-bit adder (behavioral)

use work.qsim_logic.all; -- contains bit_vector addition

entity adder is

```
    port( X,Y:in  bit_vector(0 to 7);  
          Z: out bit_vector(0 to 7));
```

end adder;

architecture comp of adder is

```
    signal temp: bit_vector(0 to 8);
```

begin

```
    temp <= ("00" & X(1 to 7)) + ("00" & Y(1 to 7));  
    Z <= temp (1 to 8);
```

end;



Multiplier Controller

entity mctrl is

```
port (Start:           in bit;           -- start pulse
      Clock:           in bit;           -- clock input
      Q7:              in bit;           -- LSB of multiplier
      AMload:          out bit;          -- load M & A registers
      AMadd:           out bit;          -- load adder result into A
      Qload:           out bit;          -- Load Q register
      AQshift:         out bit;          -- shift A & Q registers
      SignLd:          out bit;          -- load sign into A(0)
      AQoutEn:         out bit;          -- enable output
      AQoutSel:        out bit;          -- select A or Q for output
      DONE:            out bit);        -- external DONE signal
```

end mctrl;



Multiplier Controller - Architecture

architecture comp of mctrl is

```
type states is (Halt,In1,In2,Add,Shift,Sign,Out1,Out2);
```

```
signal State: States := Halt;           -- state of the controller
```

```
begin
```

```
  -- decode state variable for outputs
```

```
  AMload  <= '1' when State = In1 else '0';
```

```
  Qload   <= '1' when State = In2 else '0';
```

```
  AMadd   <= '1' when State = Add and Q7 = '1' else '0';
```

```
  AQshift <= '1' when State = Shift else '0';
```

```
  AQoutSel <= '1' when State = Out2 else '0';
```

```
  SignLd  <= '1' when State = Sign else '0';
```

```
  AQoutEn <= '1' when State = Out1 or State = Out2 else '0';
```

```
  DONE    <= '1' when State = Halt else '0';
```



Controller – State transition process

process (Clock) -- implement state machine state transitions

```
    variable Count: integer;
begin
    if Clock = '1' then
        case State is
            when Halt => if Start = '1' then    -- wait for start pulse
                           State <= In1;
                           Count := 0;
                           end if;
            when In1  => State <= In2;    -- Read 1st operand
            when In2  => State <= Add;    -- Read 2nd operand
            when Add  => State <= Shift;  -- Add multiplicand to accumulator
                           Count := Count + 1;
            when Shift => if Count = 7 then    -- Shift accumulator/multiplier
                           State <= Sign;
                           else
                               State <= Add;
                           end if;
            when Sign  => State <= Out1;  -- Set sign of result
            when Out1  => State <= Out2;  -- Output lower half of product
            when Out2  => State <= Halt;  -- Output upper half of product
        end case;
    end if;
end process;
```

