Sequential Circuit Design: Practice

Topics

- Poor design practice
- More counters
- Register as fast temporary storage
- Pipelining

Synchronous design is the most important for designing large, complex systems

In the past, some non-synchronous design practices were used to save chips/area

- Misuse of asynchronous reset
- Misuse of gated clock
- Misuse of derived clock

Misuse of asynchronous reset

- Rule: you should **never** use reset to clear register during normal operation

  Here's an example of a poorly designed *mod-10* counter which clears the register immediately after the counter reaches "1010"

Poor Sequential Circuit Design Practice

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

entity mod10_counter is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(3 downto 0)
   );
end mod10_counter;

architecture poor_async_arch of mod10_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   signal async_clr: std_logic;
   begin
```

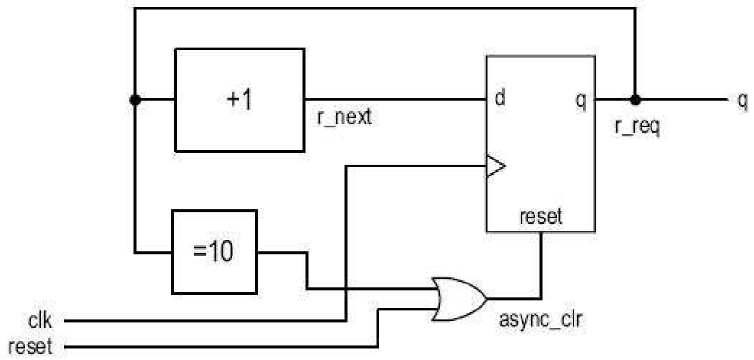
Poor Sequential Circuit Design Practice

```
      -- register
      process(clk, async_clr)
         begin
         if (async_clr = '1') then
            r_reg <= (others => '0');
         elsif (clk'event and clk = '1') then
            r_reg <= r_next;
         end if;
      end process;

   -- asynchronous clear
      async_clr <= '1' when (reset = '1' or r_reg = "1010")
                else '0';

   -- next state and output logic
      r_next <= r_reg + 1;
      q <= std_logic_vector(r_reg);
   end poor_async_arch;
```

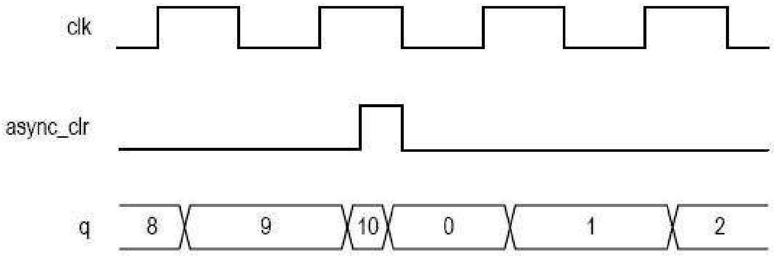
Poor Sequential Circuit Design Practice

Problem

- Transition from "1001" to "0000" goes through "1010" state (see timing diag.)
- Any glitches in combo logic driving *aync_clr* can reset the counter
- Can NOT apply timing analysis we did in last chapter to determine max. clk. freq.



(a) Block diagram



(b) Timing diagram

Figure 9.1 Decade counter using asynchronous reset

Asynchronous reset should only be used for power-on initialization

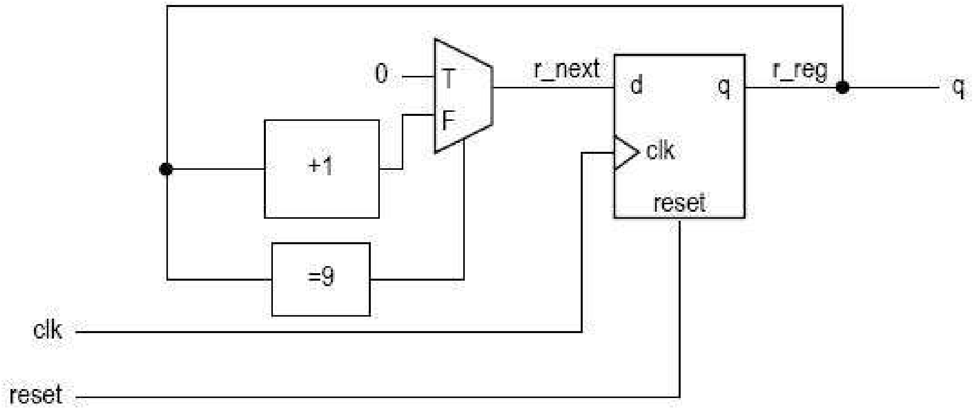
Poor Sequential Circuit Design Practice

Remedy: load "0000" synchronously -- looked at this in last chapter

```
architecture two_seg_arch of mod10_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   begin

   -- register
   process(clk, reset)
      begin
      if (reset = '1') then
         r_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         r_reg <= r_next;
      end if;
   end process;
```

Poor Sequential Circuit Design Practice

```
        -- next-state logic
        r_next <= (others => '0') when r_reg = 9 else
                  r_reg + 1;

        -- output logic
        q <= std_logic_vector(r_reg);
    end two_seg_arch;
```



Misuse of gated clock

Rule: you should **not** insert logic, e.g., an AND gate, to stop the clock from clocking a new value into a register

Poor Sequential Circuit Design Practice

The clock tree is a specially designed structure (b/c it needs to drive potentially thousands of FFs in the design) and should not be interfered with

Consider a counter with an enable signal

One may attempt to implement the enable by AND’ing the clk with it

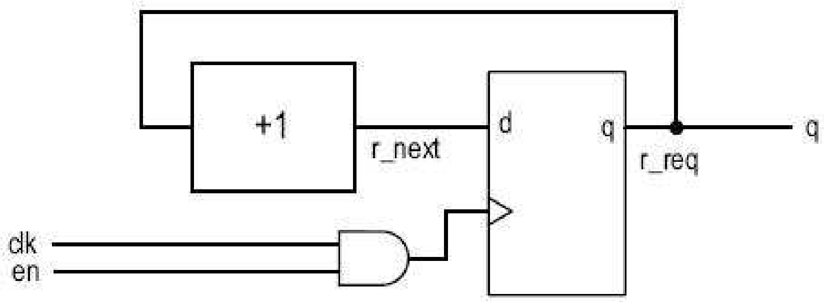


Figure 9.2 Disabling FF with gated clock

There are several problems

- *en* does not change with clk, potentially narrowing the actual clk pulse to the FF
- If *en* is not glitch-free, counter may ’count’ more often then it is supposed to
- With the AND in the clock path, it interferes with construction and analysis of clock distribution tree

Poor Sequential Circuit Design Practice

A POOR approach to solving this problem

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

entity binary_counter is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(3 downto 0)
   );
end binary_counter;

architecture gated_clk_arch of binary_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   signal gated_clk: std_logic;
   begin
```

Poor Sequential Circuit Design Practice

```
      -- register
      process(gated_clk, reset)
         begin
         if (reset = '1') then
            r_reg <= (others => '0');
         elsif (gated_clk'event and gated_clk = '1') then
            r_reg <= r_next;
         end if;
      end process;

      -- gated clock -- poor design practice
      gated_clk <= clk and en;

      -- next-state and output logic
      r_next <= r_reg + 1;
      q <= std_logic_vector(r_reg);
   end gated_clk_arch;
```

Poor Sequential Circuit Design Practice

A BETTER approach

```
architecture two_seg_arch of binary_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   begin

   -- register
   process(clk, reset)
      begin
      if (reset = '1') then
         r_reg <= (others =>'0');
      elsif (clk'event and clk = '1') then
         r_reg <= r_next;
      end if;
   end process;
```

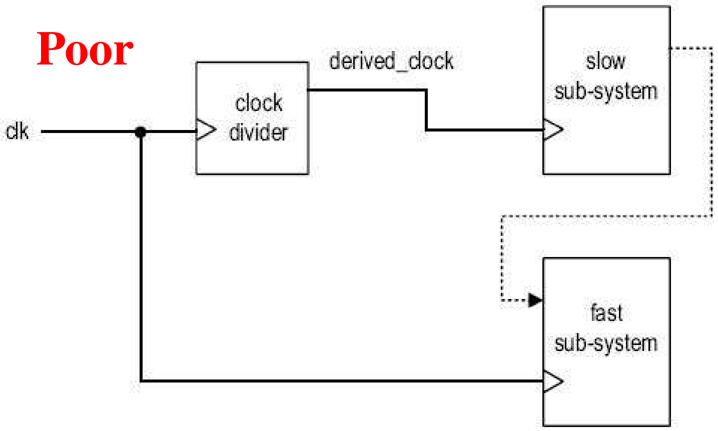
Poor Sequential Circuit Design Practice

```
     -- next-state logic
     r_next <= r_reg + 1 when en = '1' else
               r_reg;

     -- output logic
     q <= std_logic_vector(r_reg);
  end two_seg_arch;
```

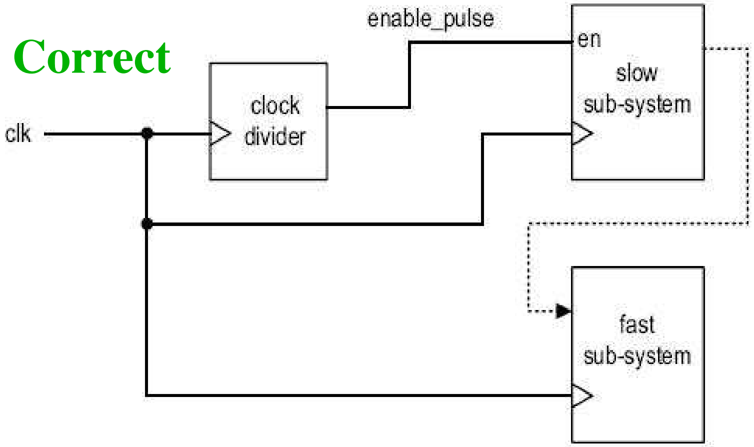
Misuse of derived clock

- Subsystems may run at different clock rates
- Rule: do **not** use a derived slow clock for the slower subsystems

Poor



Correct



Poor Sequential Circuit Design Practice

The basic problem with the diagram on the left is that the system is **no longer** synchronous

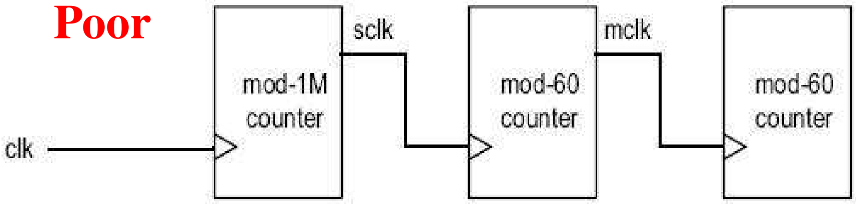
This complicates timing analysis, i.e., we can not use the simple method we looked at earlier

- We must treat this as a **two clock system** with different frequencies and phases

Consider a design that implements a "second and minutes counter"

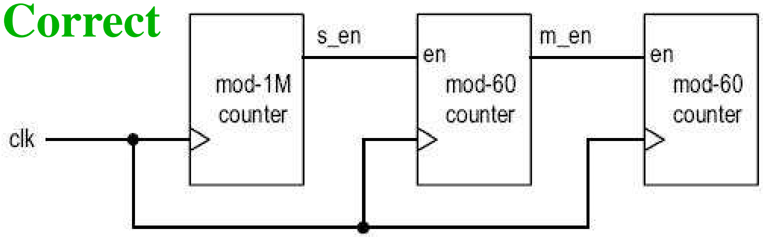
- Assume the input clk rate is 1 MHz clock

Poor



(a) Design with derived clock

Correct



(b) Design with a single synchronous clock

An example of a **POOR** design that uses derived clocks is as follows

```
library ieee;
use ieee.std_logic_1164.cb;
```

Poor Sequential Circuit Design Practice

```
use ieee.numeric_std.all;

entity timer is
   port(
      clk, reset: in std_logic;
      sec,min: out std_logic_vector(5 downto 0)
   );
end timer;

architecture multi_clock_arch of timer is
   signal r_reg: unsigned(19 downto 0);
   signal r_next: unsigned(19 downto 0);
   signal s_reg, m_reg: unsigned(5 downto 0);
   signal s_next, m_next: unsigned(5 downto 0);
   signal sclk, mclk: std_logic;
   begin
```

Poor Sequential Circuit Design Practice

```
-- register
process(clk, reset)
   begin
   if (reset = '1') then
      r_reg <= (others => '0');
   elsif (clk'event and clk = '1') then
      r_reg <= r_next;
   end if;
end process;

-- next-state logic
r_next <= (others => '0') when r_reg = 999999 else
          r_reg + 1;

-- output logic -- clock has 50% duty cycle
sclk <= '0' when r_reg < 500000 else
        '1';
```

Poor Sequential Circuit Design Practice

```
-- second divider
process(sclk, reset)
   begin
   if (reset = '1') then
      s_reg <= (others =>'0');
   elsif (sclk'event and sclk='1') then
      s_reg <= s_next;
   end if;
end process;

-- next-state logic
s_next <= (others => '0') when s_reg = 59 else
          s_reg + 1;

-- output logic (50% duty cycle)
mclk <= '0' when s_reg < 30 else
        '1';
sec <= std_logic_vector(s_reg);
```

Poor Sequential Circuit Design Practice

```
        -- minute divider
        process(mclk, reset)
           begin
           if (reset = '1') then
              m_reg <= (others => '0');
           elsif (mclk'event and mclk = '1') then
              m_reg <= m_next;
           end if;
        end process;

        -- next-state logic
        m_next <= (others => '0') when m_reg = 59 else
                  m_reg + 1;

        -- output logic
        min <= std_logic_vector(m_reg);
     end multi_clock_arch;
```

Proper Sequential Circuit Design Practice

A BETTER approach is to use a *synchronous 1-clock pulse*

```
architecture single_clock_arch of timer is
   signal r_reg: unsigned(19 downto 0);
   signal r_next: unsigned(19 downto 0);
   signal s_reg, m_reg: unsigned(5 downto 0);
   signal s_next, m_next: unsigned(5 downto 0);
   signal s_en, m_en: std_logic;
   begin

   -- register
   process(clk, reset)
      begin
      if (reset = '1') then
         r_reg <= (others => '0');
         s_reg <= (others => '0');
         m_reg <= (others => '0');
```

Proper Sequential Circuit Design Practice

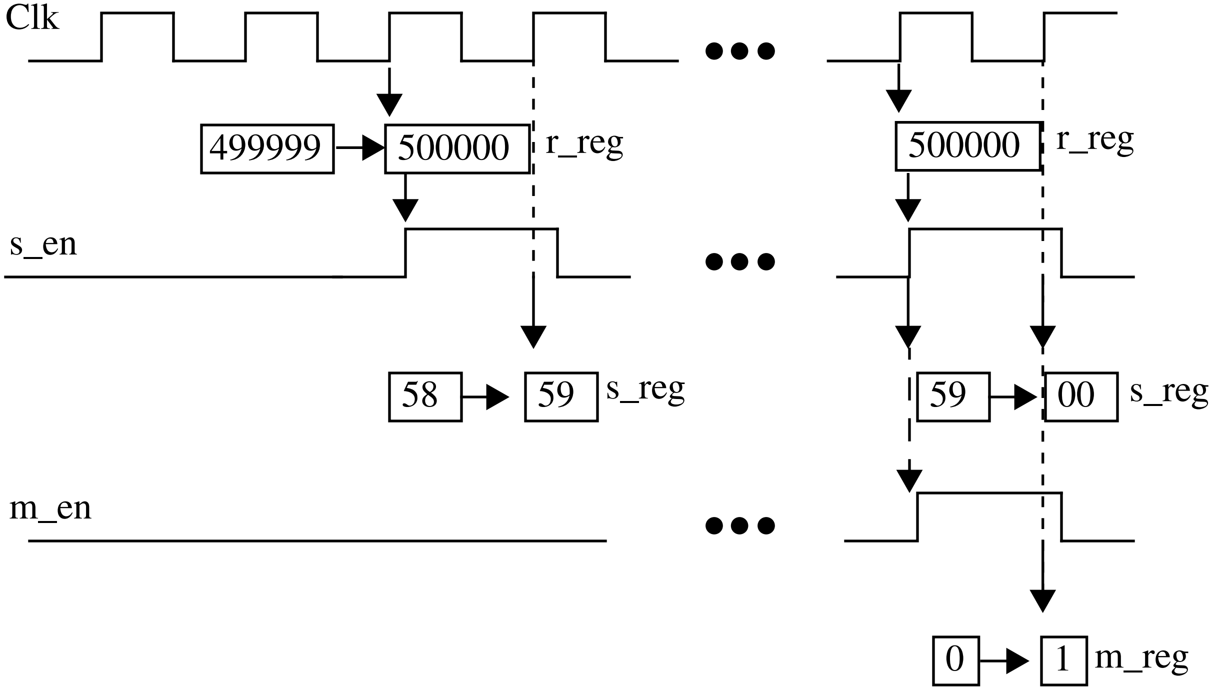
```
            elsif (clk'event and clk = '1') then
               r_reg <= r_next;
               s_reg <= s_next;
               m_reg <= m_next;
            end if;
         end process;

         -- next-state/output logic for mod-1000000 counter
         r_next <= (others => '0') when r_reg = 999999 else
                    r_reg + 1;

         s_en <= '1' when r_reg = 500000 else
                  '0';
```

Proper Sequential Circuit Design Practice

```
      -- next state logic/output logic for second divider
      s_next <= (others => '0') when
                (s_reg = 59 and s_en = '1') else
                s_reg + 1 when s_en = '1' else
                s_reg;
      m_en <= '1' when s_reg = 59 and s_en = '1' else
              '0';

      -- next-state logic for minute divider
      m_next <= (others => '0') when
                (m_reg = 59 and m_en = '1') else
                m_reg + 1 when m_en = '1' else
                m_reg;

      -- output logic
      sec <= std_logic_vector(s_reg);
      min <= std_logic_vector(m_reg);
   end single_clock_arch;
```

Proper Sequential Circuit Design Practice

Timing Diagram



Power Concerns

Power is now a major design criteria

In CMOS technology

High clock rate implies high switching frequencies and *dynamic power* is **proportional** to the switching frequency

Clock manipulation can reduce switching frequency but this should NOT be done at RT level

The proper flow is

- Design/synthesize/verify the regular synchronous subsystems

- Use special circuit (PLL etc.) to obtain derived clocks

- Use "power optimization" software tools to add *gated clocks* to some of the registers

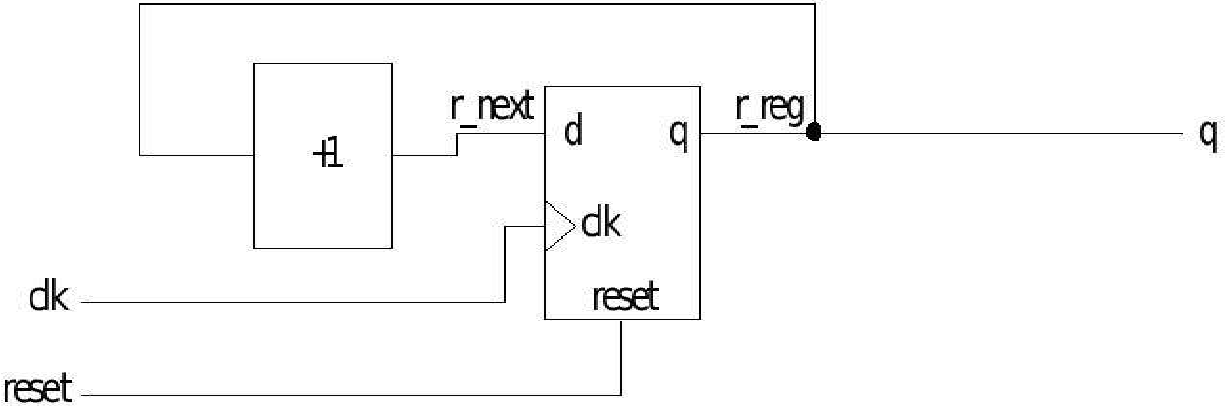
Counters

A counter circulates its internal state through a set of patterns

- Binary
- Gray counter
- Ring counter
- Linear Feedback Shift Register (LFSR)
- BCD counter

Binary counter

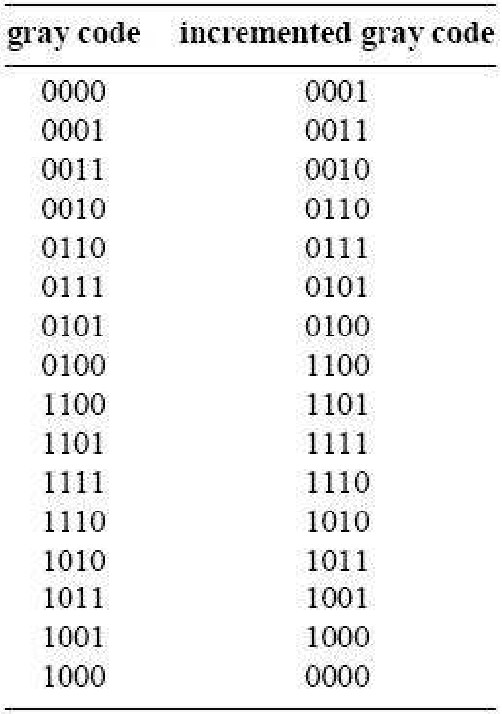
- State follows binary counting sequence
- Use an incrementer for the next-state logic



Counters

Gray counter

- State changes one-bit at a time
- Use a Gray incrementer



gray code	incremented gray code
0000	0001
0001	0011
0011	0010
0010	0110
0110	0111
0111	0101
0101	0100
0100	1100
1100	1101
1101	1111
1111	1110
1110	1010
1010	1011
1011	1001
1001	1000
1000	0000

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

Counters

```
entity gray_counter4 is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(3 downto 0)
   );
end gray_counter4;

architecture arch of gray_counter4 is
   constant WIDTH: natural := 4;
   signal g_reg: unsigned(WIDTH-1 downto 0);
   signal g_next, b, b1: unsigned(WIDTH-1 downto 0);
   begin

   -- register
   process(clk, reset)
      begin
```

Counters

```
            if (reset = '1') then
               g_reg <= (others => '0');
            elsif (clk'event and clk = '1') cb
               g_reg <= g_next;
            end if;
         end process;

         -- next-state logic -- gray to binary
         b <= g_reg xor ('0' & b(WIDTH-1 downto 1));

         b1 <= b+1; -- increment

         -- binary to gray
         g_next <= b1 xor ('0' & b1(WIDTH-1 downto 1));

         -- output logic
         q <= std_logic_vector(g_reg);
      end arch;
```

Counters

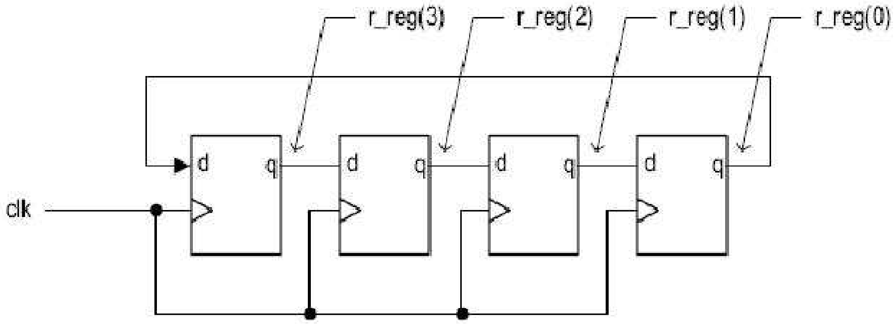
Ring counter

- Circulates a single 1, e.g., in a 4-bit ring counter:

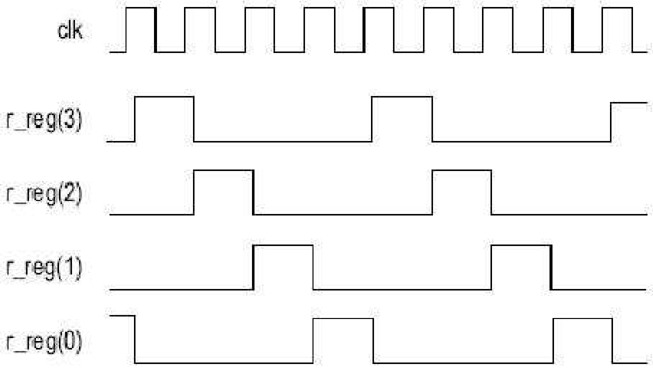
  "1000", "0100", "0010", "0001"

There are *n* patterns for n-bit register where the output appears as an n-phase signal

In the **non self-correcting** design, "0001" is inserted at initialization and that’s it



(a) Conceptual block diagram



Counters

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

entity ring_counter is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(3 downto 0)
   );
end ring_counter;

architecture reset_arch of ring_counter is
   constant WIDTH: natural := 4;
   signal r_reg: std_logic_vector(WIDTH-1 downto 0);
   signal r_next: std_logic_vector(WIDTH-1 downto 0);
   begin
```

Counters

```
      -- register
      process(clk, reset)
         begin
         if (reset = '1') then
            r_reg <= (0 => '1', others => '0');
         elsif (clk'event and clk = '1') then
            r_reg <= r_next;
         end if;
      end process;

      -- next-state logic
      r_next <= r_reg(0) & r_reg(WIDTH-1 downto 1);

      -- output logic
      q <= r_reg;
   end reset_arch;
```

This simple design makes this a very fast counter (much faster than a binary cnter)

Counters

A self-correcting design ensures at a ’1’ is always circulating in the ring

This is accomplished by inspecting the 3 MSBs -- if "000", then the combo. logic inserts a ’1’ into the low order bit

```
architecture self_correct_arch of ring_counter is
   constant WIDTH: natural := 4;
   signal r_reg, r_next:
      std_logic_vector(WIDTH-1 downto 0);
   signal s_in: std_logic;
   begin

   -- register
   process(clk, reset)
      begin
      if (reset = ’1’) then
-- no special input pattern is needed in this version
-- since the ’1’ is not circulated - its generated
         r_reg <= (others => ’0’);
```

Counters

```
         elsif (clk'event and clk = '1') then
            r_reg <= r_next;
         end if;
      end process;

      -- next-state logic
      s_in <= '1' when r_reg(WIDTH-1 downto 1) = "000" else
              '0';
      r_next <= s_in & r_reg(WIDTH-1 downto 1);

      -- output logic
      q <= r_reg;
   end self_correct_arch;
```

Counters

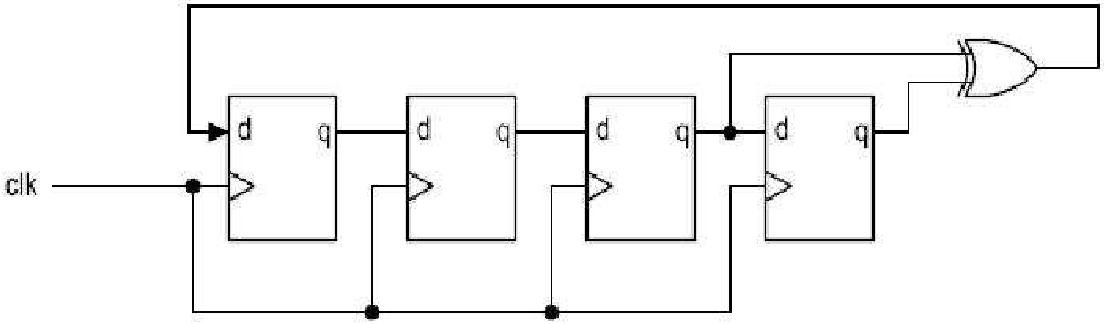
Linear Feedback Shift Register (LFSR)

An LFSR is a shifter register that contains an **XOR** feedback network that determines the next serial input value

Only a subset of the register bits are used in the **XOR** operation

By carefully selecting the bits, an LFSR can be designed to circulate through all 2^n-1 states for an *n*-bit register

Consider a 4-bit LFSR



"1000", "0100", "0010", "1001", "1100", "0110", "1011", "0101", "1010", "1101", "1110", "1111", "0111", "0011", "0001".

Note that the state "0000" is excluded -- if it ever shows up, the LFSR becomes stuck

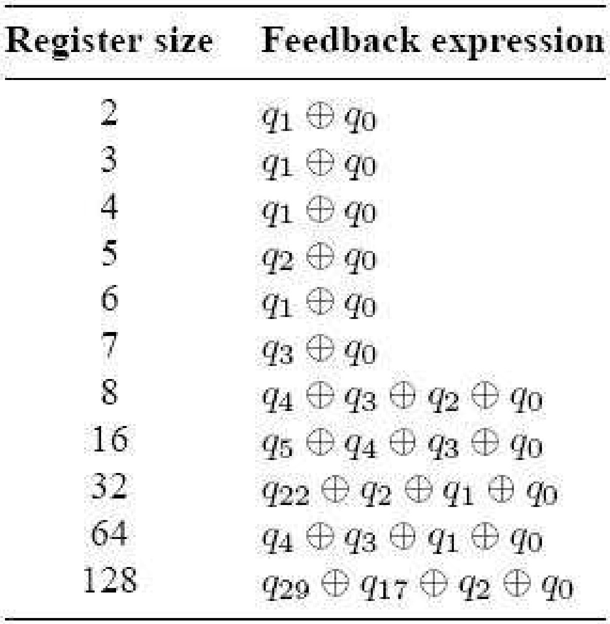
Counters

The properties of an LFSR are derived from the theory of **finite fields**

The term *linear* is used because the feedback expression is described using AND and XOR operations, which define a linear system in algebra

In addition to the '2^n -1 states' properties, the following are also true

- The feedback circuit to generate a maximal number of states exists for any *n*
- The output sequence is pseudorandom, i.e., it exhibits certain statistical properties and appears random



Register size	Feedback expression
2	$q_1 \oplus q_0$
3	$q_1 \oplus q_0$
4	$q_1 \oplus q_0$
5	$q_2 \oplus q_0$
6	$q_1 \oplus q_0$
7	$q_3 \oplus q_0$
8	$q_4 \oplus q_3 \oplus q_2 \oplus q_0$
16	$q_5 \oplus q_4 \oplus q_3 \oplus q_0$
32	$q_{22} \oplus q_2 \oplus q_1 \oplus q_0$
64	$q_4 \oplus q_3 \oplus q_1 \oplus q_0$
128	$q_{29} \oplus q_{17} \oplus q_2 \oplus q_0$

The 'taps' for the XOR gates are defined using primitive polynomials

These examples show that very little logic is needed in the feedback circuit - only between 1 and 3 XOR gates

Counters

Applications of LFSRs

- Pseudorandom: used in testing, data encryption/decryption
- A counter with simple next-state logic

For example, a 128-bit LFSR using 3 XOR gates will circulate 2^{128}-1 patterns -- which takes 10^{12} years for a 100 GHz system

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

entity lfsr4 is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(3 downto 0)
   );
end lfsr4;
```

Counters

```
architecture no_zero_arch of lfsr4 is
   signal r_reg, r_next: std_logic_vector(3 downto 0);
   signal fb: std_logic;
   constant SEED: std_logic_vector(3 downto 0):="0001";
   begin

   -- register
   process(clk, reset)
      begin
      if (reset = '1') then
         r_reg <= SEED;
      elsif (clk'event and clk = '1') then
         r_reg <= r_next;
      end if;
   end process;
```

Counters

```
      -- next-state logic
      fb <= r_reg(1) xor r_reg(0);
      r_next <= fb & r_reg(3 downto 1);

      -- output logic
      q <= r_reg;
   end no_zero_arch;
```

Text covers design that includes "00..00" state

Text covers BCD counter, which is similar in design to the second/minute counter

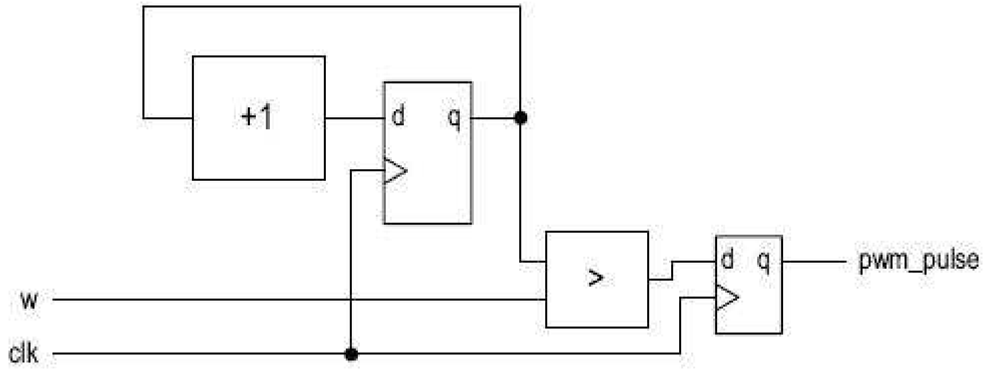
Pulse Width Modulation (PWM)

- Duty cycle: percentage of time that the signal is asserted

- PWM uses a signal, *w*, to specify the duty cycle
  Duty cycle is *w*/16 if *w* is not "0000"
  Duty cycle is 16/16 if *w* is "0000"

Counters

Implemented by a binary counter with a special output circuit



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pwm is
   port(
      clk, reset: in std_logic;
      w: in std_logic_vector(3 downto 0);
      pwm_pulse: out std_logic
   );
end pwm;
```

Counters

```
architecture two_seg_arch of pwm is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   signal buf_reg: std_logic;
   signal buf_next: std_logic;
   begin

   -- register & output buffer
   process(clk, reset)
      begin
      if (reset = '1') then
         r_reg <= (others => '0');
         buf_reg <= '0';
      elsif (clk'event and clk = '1') then
         r_reg <= r_next;
         buf_reg <= buf_next;
      end if;
   end process;
```

Counters

```
-- next-state logic
   r_next <= r_reg + 1;

-- output logic
   buf_next <=
       '1' when (r_reg<unsigned(w)) or (w="0000") else
       '0';
-- buffered to remove glitches
   pwm_pulse <= buf_reg;
end two_seg_arch;
```

Register as Fast Temporary Storage

Registers are too large to serve as mass storage -- RAMs are better b/c they are smaller (they are designed at transistor level and use minimal area)

Registers are usually used to construct *small*, **fast** temporal storage in digital systems, for example, as

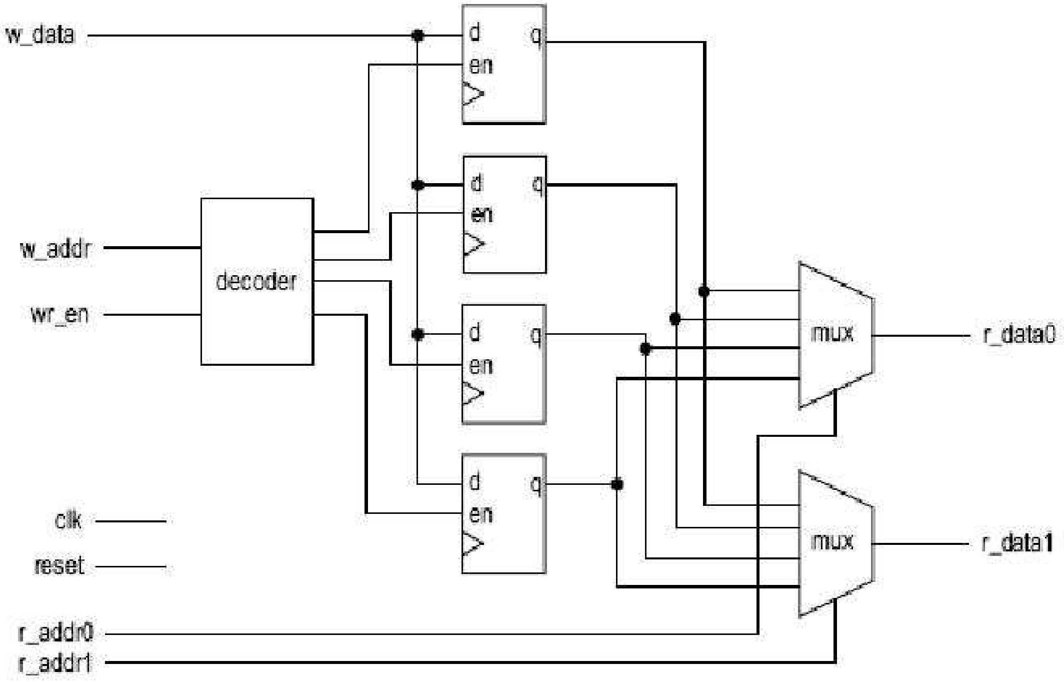
Register file, fast FIFO, Fast CAM (content addressable memory)

Register File

Register file

- Registers arranged as a 1-D array
- Each register is identified with an address
- Normally has 1 write port (with write enable signal) and two or more read ports

For example, a 4-word register file with 1 write port and two read ports



Decoder used to route the write enable signal, MUXs used to create ports

Register File

Write decoding circuit behaves as follows

- Outputs "0000" if *wr_en* is '0'
- Asserts one bit according to *w_addr* if *wr_en* is '1'

A 2-D data type is needed here

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

entity reg_file is
   port(
      clk, reset: in std_logic;
      wr_en: in std_logic;
      w_addr: in std_logic_vector(1 downto 0);
      w_data: in std_logic_vector(15 downto 0);
      r_addr0, r_addr1: in std_logic_vector(1 downto 0);
      r_data0, r_data1: out
         std_logic_vector(15 downto 0));
end reg_file;
```

Register File

```
architecture no_loop_arch of reg_file is
   constant W: natural := 2; -- # of bits in address
   constant B: natural := 16; -- # of bits in data
   type reg_file_type is array (2**W-1 downto 0) of
        std_logic_vector(B-1 downto 0);
   signal array_reg: reg_file_type;
   signal array_next: reg_file_type;
   signal en: std_logic_vector(2**W-1 downto 0);
   begin

   -- register
   process(clk, reset)
      begin
      if (reset = '1') then
         array_reg(3) <= (others => '0');
         array_reg(2) <= (others => '0');
         array_reg(1) <= (others => '0');
         array_reg(0) <= (others => '0');
```

Register File

```
            elsif (clk'event and clk = '1') then
               array_reg(3) <= array_next(3);
               array_reg(2) <= array_next(2);
               array_reg(1) <= array_next(1);
               array_reg(0) <= array_next(0);
            end if;
         end process;

         -- enable logic for register
         process(array_reg, en, w_data)
            begin
            array_next(3) <= array_reg(3);
            array_next(2) <= array_reg(2);
            array_next(1) <= array_reg(1);
            array_next(0) <= array_reg(0);
            if (en(3) = '1') then
               array_next(3) <= w_data;
            end if;
```

Register File

```
         if (en(2) = '1') then
            array_next(2) <= w_data;
         end if;
         if (en(1) = '1') then
            array_next(1) <= w_data;
         end if;
         if (en(0) = '1') then
            array_next(0) <= w_data;
         end if;
      end process;

      -- decoding for write address
      process(wr_en, w_addr)
         begin
         if (wr_en = '0') then
            en <= (others => '0');
         else
            case w_addr is
```

Register File

```
                 when "00" =>   en <= "0001";
                 when "01" =>   en <= "0010";
                 when "10" =>   en <= "0100";
                 when others => en <= "1000";
              end case;
           end if;
        end process;

        -- read multiplexing
        with r_addr0 select
           r_data0 <=  array_reg(0) when "00",
                       array_reg(1) when "01",
                       array_reg(2) when "10",
                       array_reg(3) when others;
```

Register File

```
        with r_addr1 select
           r_data1 <=  array_reg(0) when "00",
                       array_reg(1) when "01",
                       array_reg(2) when "10",
                       array_reg(3) when others;
    end no_loop_arch;
```

FIFO Buffer

- A first-in-first out buffer acts as "Elastic" storage between two subsystems

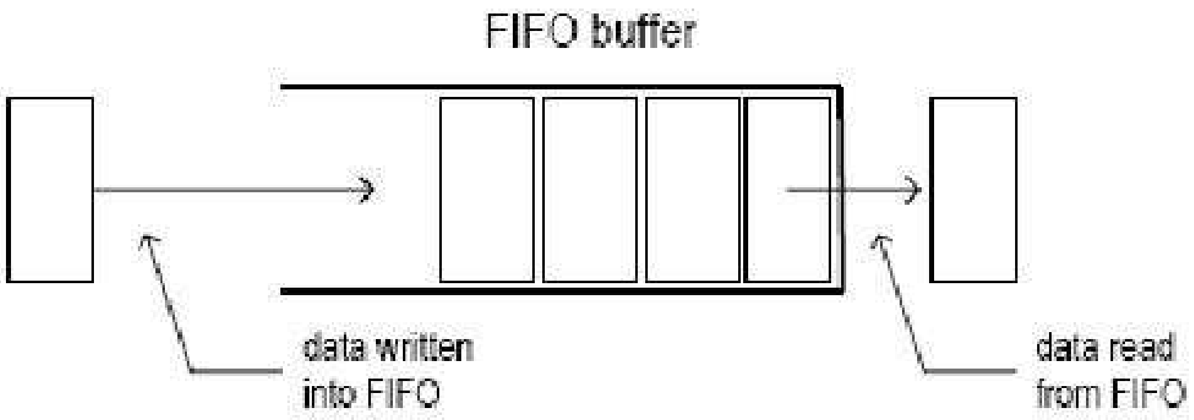


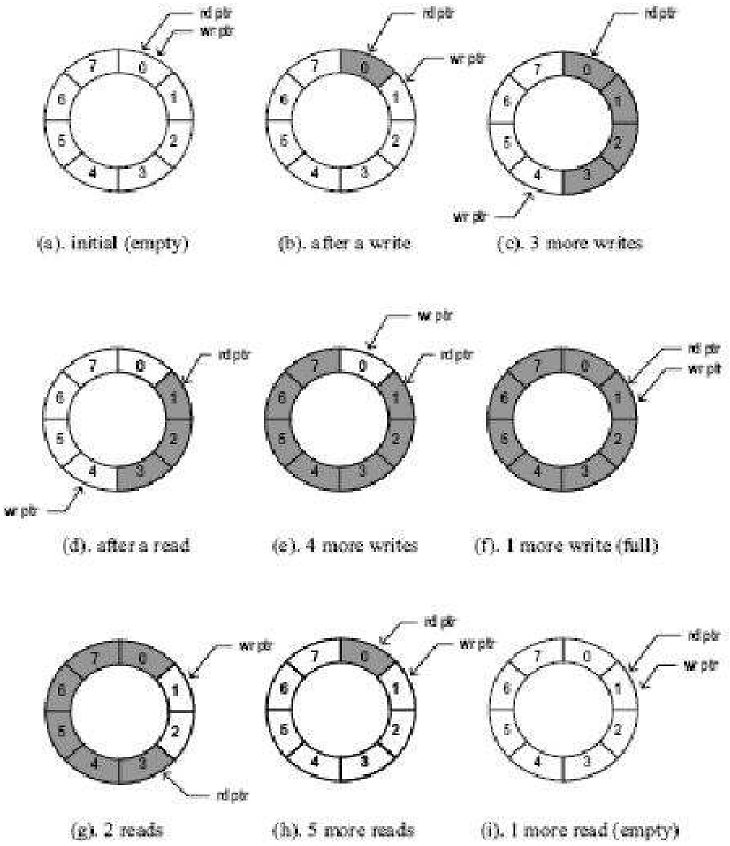
Figure 9.11 Conceptual diagram of a FIFO buffer.

FIFO Buffer

- Circular queue implementation
- Use two pointers and a "generic storage"

  Write pointer: points to the empty slot **before** the head of the queue

  Read pointer: points to the first element at the tail of the queue



(a). initial (empty) (b). after a write (c). 3 more writes

(d). after a read (e). 4 more writes (f). 1 more write (full)

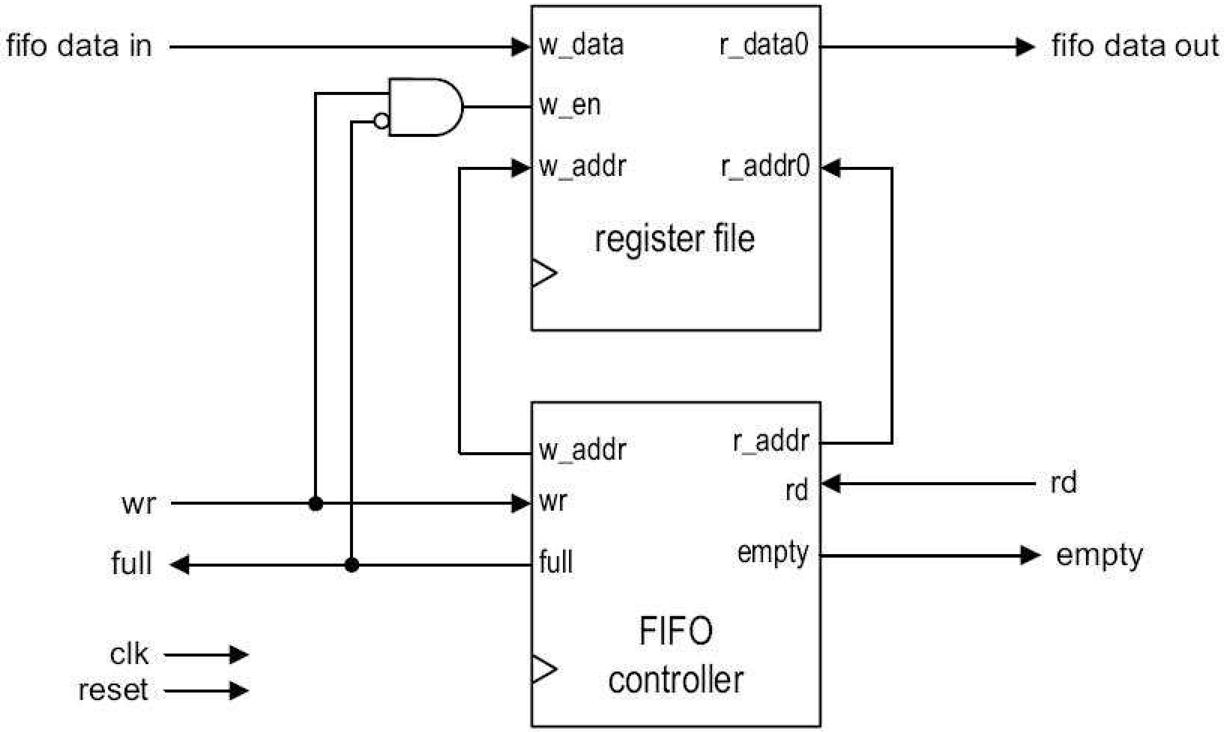
(g). 2 reads (h). 5 more reads (i). 1 more read (empty)

FIFO Buffer

FIFO controller

- The read and write pointers are defined using 2 counters
- Tricky part is distinguishing between *full* and *empty* status because in both cases, the pointers are equal

Design 1: Augmented binary counter



FIFO Buffer

Augmented binary counter:

- Increase the size of the counter by 1 bit
- Use LSBs for as register address
- Use MSB to distinguish full or empty

Write pointer	Read pointer	Operation	Status
0 000	0 000	initialization	empty
0 111	0 000	after 7 writes	
1 000	0 000	after 1 write	full
1 000	0 100	after 4 reads	
1 100	0 100	after 4 writes	full
1 100	1 011	after 7 reads	
1 100	1 100	after 1 read	empty
0 011	1 100	after 7 writes	
0 100	1 100	after 1 write	full
0 100	0 100	after 8 reads	empty

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

FIFO Buffer

```
entity fifo_sync_ctrl4 is
   port(
      clk, reset: in std_logic;
      wr, rd: in std_logic;
      full, empty: out std_logic;
      w_addr, r_addr: out std_logic_vector(1 downto 0)
   );
end fifo_sync_ctrl4;

-- merge this code with register file code to complete,
-- use component instantiation

architecture enlarged_bin_arch of fifo_sync_ctrl4 is
   constant N: natural:=2;
   signal w_ptr_reg, w_ptr_next: unsigned(N downto 0);
   signal r_ptr_reg, r_ptr_next: unsigned(N downto 0);
   signal full_flag, empty_flag: std_logic;
   begin
```

FIFO Buffer

```
       -- register
       process(clk, reset)
          begin
          if (reset = '1') then
             w_ptr_reg <= (others => '0');
             r_ptr_reg <= (others => '0');
          elsif (clk'event and clk = '1') then
             w_ptr_reg <= w_ptr_next;
             r_ptr_reg <= r_ptr_next;
          end if;
       end process;

    -- write pointer next-state logic. Nothing special is
    -- done here, just add 1, MSB is set automatically
       w_ptr_next <=
          w_ptr_reg + 1 when wr='1' and full_flag='0' else
          w_ptr_reg;
```

FIFO Buffer

```
-- compare MSBs, when different and addresses are the
-- same, then FIFO is full
   full_flag <=
      '1' when r_ptr_reg(N) /= w_ptr_reg(N) and
               r_ptr_reg(N-1 downto 0) =
                      w_ptr_reg(N-1 downto 0)
          else '0';

-- write port output
   w_addr <= std_logic_vector(w_ptr_reg(N-1 downto 0));
   full <= full_flag;

-- read pointer next-state logic
   r_ptr_next <=
      r_ptr_reg + 1 when rd='1' and empty_flag='0' else
      r_ptr_reg;
```

FIFO Buffer

```
-- FIFO is empty when MSBs are equal and address bits
-- are the same
   empty_flag <= '1' when r_ptr_reg = w_ptr_reg else
                 '0';

-- read port output
   r_addr <= std_logic_vector(r_ptr_reg(N-1 downto 0));
   empty <= empty_flag;
end enlarged_bin_arch;
```

Design 2: Use 2 extra **status** FFs

- *full_reg*/*empty_reg* track the status of the FIFO and are initialized to '0' and '1'
- The are updated according to the current request given by the *wr* and *rd* signals:
  - "00": no change
  - "11": advance both read and write ptrs -- no change to full/empty status
  - "10": advance write ptr; de-assert *empty* -- assert *full* when *write_ptr=read_ptr*
  - "01": advance read ptr; de-assert *full* -- assert *empty* when *write_ptr=read_ptr*

FIFO Buffer

```
architecture lookahead_bin_arch of fifo_sync_ctrl4 is
   constant N: natural := 2;
   signal w_ptr_reg, w_ptr_next: unsigned(N-1 downto 0);
   signal w_ptr_succ: unsigned(N-1 downto 0);
   signal r_ptr_reg, r_ptr_next: unsigned(N-1 downto 0);
   signal r_ptr_succ: unsigned(N-1 downto 0);
   signal full_reg, empty_reg: std_logic;
   signal full_next, empty_next: std_logic;
   signal wr_op: std_logic_vector(1 downto 0);
   begin

   -- register
   process(clk, reset)
      begin
      if (reset = '1') then
         w_ptr_reg <= (others => '0');
         r_ptr_reg <= (others => '0');
```

FIFO Buffer

```
            elsif (clk'event and clk = '1') then
               w_ptr_reg <= w_ptr_next;
               r_ptr_reg <= r_ptr_next;
            end if;
         end process;

         -- status FF
         process(clk, reset)
            begin
            if (reset = '1') then
               full_reg <= '0';
               empty_reg <= '1';
            elsif (clk'event and clk = '1') then
               full_reg <= full_next;
               empty_reg <= empty_next;
            end if;
         end process;
```

FIFO Buffer

```
   -- next values of the write and read pointers
      w_ptr_succ <= w_ptr_reg + 1;
      r_ptr_succ <= r_ptr_reg + 1;

   -- next-state logic
      wr_op <= wr & rd;
      process(w_ptr_reg, w_ptr_succ, r_ptr_reg,
         r_ptr_succ, wr_op, empty_reg, full_reg)
         begin
         w_ptr_next <= w_ptr_reg;
         r_ptr_next <= r_ptr_reg;
         full_next <= full_reg;
         empty_next <= empty_reg;

         case wr_op is
            when "00" => -- no change
```

FIFO Buffer

```
        when "10" => -- write
           if (full_reg /= '1') then -- not full
              w_ptr_next <= w_ptr_succ;
              empty_next <= '0';
              if (w_ptr_succ = r_ptr_reg) then
                 full_next <='1';
              end if;
           end if;

        when "01" => -- read
           if (empty_reg /= '1') then -- not empty
              r_ptr_next <= r_ptr_succ;
              full_next <= '0';
              if (r_ptr_succ = w_ptr_reg) then
                 empty_next <='1';
              end if;
           end if;
```

FIFO Buffer

```
    -- write/read -- status not affected
               when others =>
                  w_ptr_next <= w_ptr_succ;
                  r_ptr_next <= r_ptr_succ;
            end case;
         end process;
    -- write port output
         w_addr <= std_logic_vector(w_ptr_reg);
         full <= full_reg;
         r_addr <= std_logic_vector(r_ptr_reg);
         empty <= empty_reg;
    end lookahead_bin_arch;
```

Can also use an LFSR, works b/c *write_ptr* and *read_ptr* follow the same pat.

```
 w_ptr_succ <= (w_ptr_reg(1) xor w_ptr_reg(0)) &
    w_ptr_reg(3 downto 1);
 r_ptr_succ <= (r_ptr_reg(1) xor r_ptr_reg(0)) &
    r_ptr_reg(3 downto 1);
```

Pipelines

Pipelines are used to increase system performance by overlapping operations

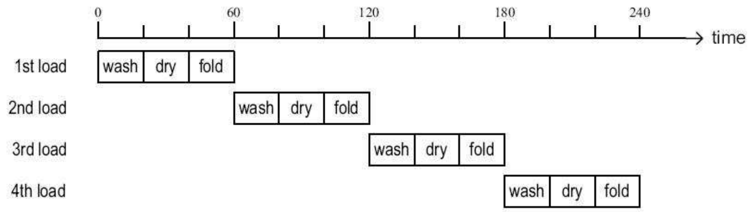
Systems performance can be measured using two metrics

Delay: required time to complete one task

Throughput: number of tasks completed per unit time

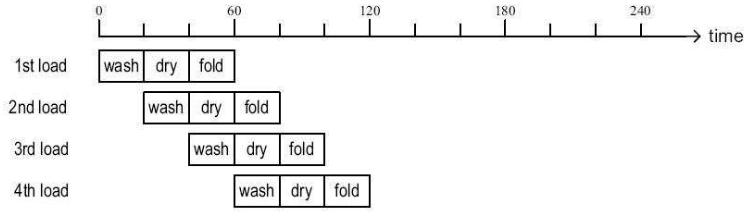
Pipelining increases **throughput** by overlapping operations

Basic idea is to divide the combinational logic into a set of stages, with buffers (registers or latches) inserted between each stage



(a) Non-pipelined sequence

Sequential laundry



(b) Pipelined sequence

Pipelined laundry

Pipelines

Non-pipelined: Delay: 60 min, Throughput 1/60 load per min

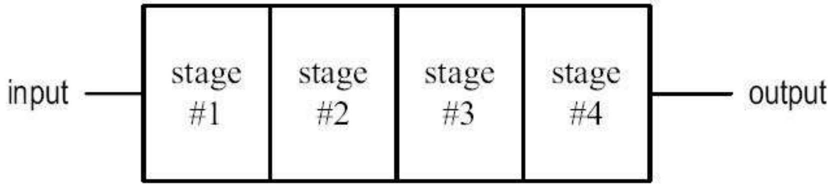
Pipelined: Delay: 60 min, Throughput of *4* loads is *4*/(40 + *4**20) loads per min

where 40 is the time to load the first two stages

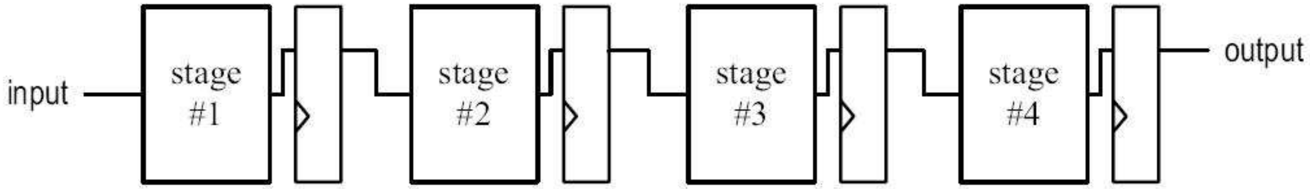
This yields 2/60, twice the throughput

In practice, stages are often **not** of equal length

Clock cycle time set by longest stage in a pipelined combinational circuit



(a) Original combinational circuit



(b) Pipelined circuit

Pipelines

Given a pipeline with stage delays of T_1, T_2, T_3 and T_4, clock cycle time is bounded by the

$$T_{max} = \max(T_1, T_2, T_3, T_4)$$

AND the setup and clock-to-q delays of the pipeline registers

$$T_c = T_{max} + T_{setup} + T_{cq}$$

In non-pipelined version, delay to process one item is

$$T_{comb} = T_1 + T_2 + T_3 + T_4$$

For the pipelined version, its actually **longer**

$$T_{pipe} = 4T_c = 4*T_{max} + 4*(T_{setup} + T_{cq})$$

The *win* is actually w.r.t. the **throughput** metric

$$TP_{comb} = 1/T_{comb}$$

For pipelined version, it takes $3*T_c$ time to fill the pipeline and the time to process k items is $3*T_c + kT_c$ yielding $TP = k/(3*T_c + kT_c)$ which approaches $1/T_c$

Pipelines

Not all circuits are amenable to pipelines -- if the circuit meets the following criteria, then it is a candidate for a pipeline

Data is always available for the pipelined circuit's inputs

System **throughput** is an important performance characteristic

Combinational circuit can be *divided into stages* with similar propagation delays

Propagation delay of a stage is **much larger** than the T_{setup} and T_{cq} of the register

Procedure to Add a Pipeline

- Derive the block diagram of the original combinational circuit and arrange the circuit as a cascading chain
- Identify the major components and estimate the relative propagation delays of these components
- Divide the chain into stages of similar propagation delays
- Identify the signals that cross the boundary of the chain and insert registers

Pipelines

Consider a pipelined combinational multiplier

$$
\begin{array}{cccccccccl}
 & & & & & a_3 & a_2 & a_1 & a_0 & \text{multiplicand} \\
\times & & & & & b_3 & b_2 & b_1 & b_0 & \text{multiplier} \\
\hline
 & & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & \\
 & & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & \\
 & & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & \\
+ & & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & & \\
\hline
 & y_7 & y_6 & y_5 & y_4 & y_3 & y_2 & y_1 & y_0 & \text{product}
\end{array}
$$

The two major components are the *adder* and *bit-product* generation circuit

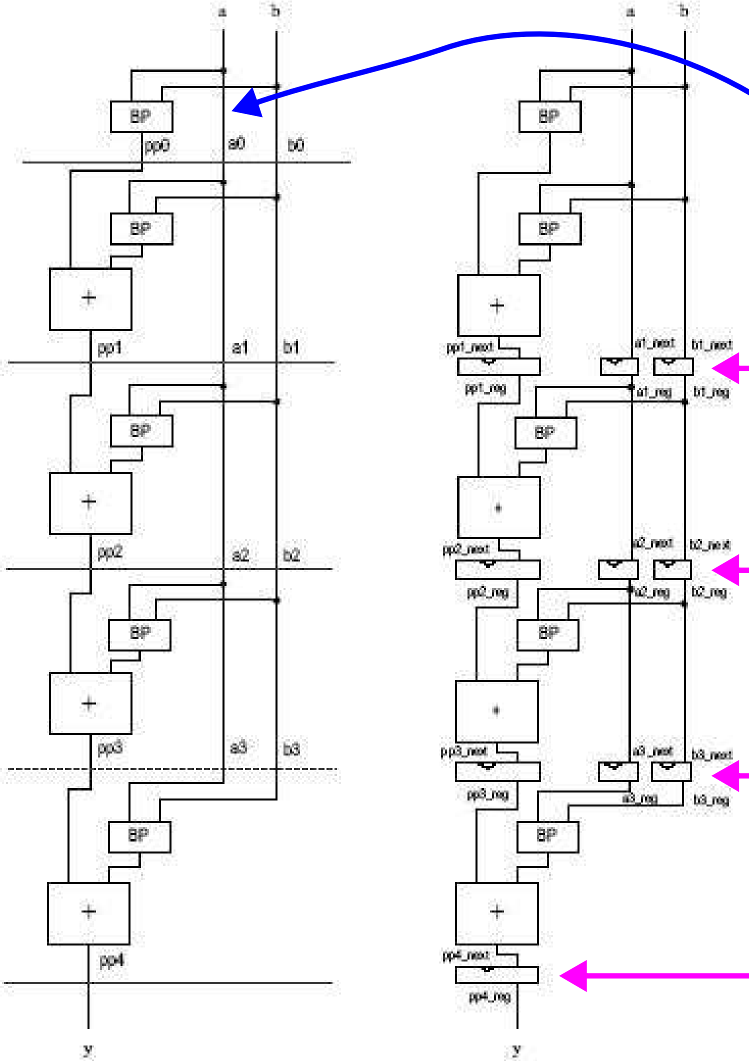
Arrange this components in cascade as shown on the next slide, with the bit-product labeled as BP

The bit-product circuit is simply an AND operation and therefore has a small delay

We combine it with the adder to define a **stage**

The horizontal lines in the combinational version on the next slide define the stages

Pipelines



(a). Non-pipelined design

(b). Pipelined design

Since no addition occurs
in the zeroth stage, we will
merge it with the first stage

Pipeline registers

Pipeline registers

Pipeline registers

Pipeline registers

Pipelines

There are two types of pipeline registers

- One type to accommodate the computation flow and to store the intermediate results (*partial products* pp_1, ... pp_4)
- Second type to preserve the info needed in each stage, i.e., a_1, a_2, a_3, b_1, b_2, and b_3

Since there are different multiplications occurring in each stage, the operands for any given multiplication must move along with the partial products

NON-pipelined version

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

entity mult5 is
   port(
      clk, reset: in std_logic;
      a, b: in std_logic_vector(4 downto 0);
      y: out std_logic_vector(9 downto 0));
end mult5;
```

Pipelines

```
    architecture comb_arch of mult5 is
       constant WIDTH: integer:=5;
       signal a0, a1, a2, a3:
          std_logic_vector(WIDTH-1 downto 0);
       signal b0, b1, b2, b3:
          std_logic_vector(WIDTH-1 downto 0);
       signal bv0, bv1, bv2, bv3, bv4:
          std_logic_vector(WIDTH-1 downto 0);
       signal bp0, bp1, bp2, bp3, bp4:
          unsigned(2*WIDTH-1 downto 0);
       signal pp0, pp1, pp2, pp3, pp4:
          unsigned(2*WIDTH-1 downto 0);
       begin

    -- stage 0 (signal names are for use later when we
    -- show the pipelined version
       bv0 <= (others => b(0)); -- a * MSB of b
       bp0 <=unsigned("00000" & (bv0 and a));
```

Pipelines

```
        pp0 <= bp0;
        a0 <= a; -- not needed here but this is what we'll
        b0 <= b; -- end up doing in the pipelined version

     -- stage 1
        bv1 <= (others => b0(1));
        bp1 <=unsigned("0000" & (bv1 and a0) & "0");
        pp1 <= pp0 + bp1;
        a1 <= a0;
        b1 <= b0;

     -- stage 2
        bv2 <= (others => b1(2));
        bp2 <=unsigned("000" & (bv2 and a1) & "00");
        pp2 <= pp1 + bp2;
        a2 <= a1;
        b2 <= b1;
```

Pipelines

```
   -- stage 3
      bv3 <= (others => b2(3));
      bp3 <=unsigned("00" & (bv3 and a2) & "000");
      pp3 <= pp2 + bp3;
      a3 <= a2;
      b3 <= b2;

   -- stage 4
      bv4 <= (others => b3(4));
      bp4 <=unsigned("0" & (bv4 and a3) & "0000");
      pp4 <= pp3 + bp4;

   -- output
      y <= std_logic_vector(pp4);
   end comb_arch;
```

Pipelines

To implement the pipeline, we replace

```
pp2 <= pp1 + bp2; -- stage 2
pp3 <= pp2 + bp3; -- stage 3
```

with a pipeline register so these values are stored

```
-- register
   if (reset = '1') then
      pp2_reg <= (others => '0');
   elsif (clk'event and clk='1') then
      pp2_reg <= pp2_next;
   end if;

-- stage 2
   pp2_next <= pp1_reg + bp2;
-- stage 3
   pp3_next <= pp2_reg + bp3;
```

The complete code is given below

Pipelines

```
architecture four_stage_pipe_arch of mult5 is
   constant WIDTH: integer:=5;
   signal a1_reg, a2_reg, a3_reg:
      std_logic_vector(WIDTH-1 downto 0);
   signal a0, a1_next, a2_next, a3_next:
      std_logic_vector(WIDTH-1 downto 0);
   signal b1_reg, b2_reg, b3_reg:
      std_logic_vector(WIDTH-1 downto 0);
   signal b0, b1_next, b2_next, b3_next:
      std_logic_vector(WIDTH-1 downto 0);
   signal bv0, bv1, bv2, bv3, bv4:
      std_logic_vector(WIDTH-1 downto 0);
   signal bp0, bp1, bp2, bp3, bp4:
      unsigned(2*WIDTH-1 downto 0);
   signal pp1_reg, pp2_reg, pp3_reg, pp4_reg:
      unsigned(2*WIDTH-1 downto 0);
   signal pp0, pp1_next, pp2_next, pp3_next, pp4_next:
      unsigned(2*WIDTH-1 downto 0);
```

Pipelines

```
     begin

  -- pipeline registers (buffers)
     process(clk, reset)
        begin
        if (reset = '1') then
           pp1_reg <= (others => '0');
           pp2_reg <= (others => '0');
           pp3_reg <= (others => '0');
           pp4_reg <= (others => '0');
           a1_reg <= (others => '0');
           a2_reg <= (others => '0');
           a3_reg <= (others => '0');
           b1_reg <= (others => '0');
           b2_reg <= (others => '0');
           b3_reg <= (others => '0');
```

Pipelines

```
            elsif (clk'event and clk = '1') then
               pp1_reg <= pp1_next;
               pp2_reg <= pp2_next;
               pp3_reg <= pp3_next;
               pp4_reg <= pp4_next;
               a1_reg <= a1_next;
               a2_reg <= a2_next;
               a3_reg <= a3_next;
               b1_reg <= b1_next;
               b2_reg <= b2_next;
               b3_reg <= b3_next;
            end if;
         end process;

      -- merged stage 0 & 1 for pipeline
         bv0 <= (others => b(0));
         bp0 <=unsigned("00000" & (bv0 and a));
         pp0 <= bp0;
```

Pipelines

```
a0 <= a;
b0 <= b;

-- merged with above
bv1 <= (others => b0(1));
bp1 <=unsigned("0000" & (bv1 and a0) & "0");
pp1_next <= pp0 + bp1;
a1_next <= a0;
b1_next <= b0;

-- stage 2
bv2 <= (others => b1_reg(2));
bp2 <=unsigned("000" & (bv2 and a1_reg) & "00");
pp2_next <= pp1_reg + bp2;
a2_next <= a1_reg;
b2_next <= b1_reg;
```

Pipelines

```
        -- stage 3
        bv3 <= (others => b2_reg(3));
        bp3 <=unsigned("00" & (bv3 and a2_reg) & "000");
        pp3_next <= pp2_reg + bp3;
        a3_next <= a2_reg;
        b3_next <= b2_reg;

        -- stage 4
        bv4 <= (others => b3_reg(4));
        bp4 <=unsigned("0" & (bv4 and a3_reg) & "0000");
        pp4_next <= pp3_reg + bp4;

        -- output
        y <= std_logic_vector(pp4_reg);
     end four_stage_pipe_arch;
```

Shizzam -- your first pipeline!

Pipelines

There are several improvements we can make

- We can use a smaller *(n+1)*-bit adder to replace the *2n*-bit adder
- We can reduce the size of the partial-product register b/c the LSBs actually grow from *n+1* bits to *2n* bits

  Therefore, the MSBs of the initial partial products are wasted (they are always '0')

  For example, we can use a *5*-bit register for the initial partial product (pp_0 signal) and increase the size by 1 in each stage.
- We can reduce the size of the registers that hold the *b* signal since only the *ith* bit of *b* is needed in the *ith* stage

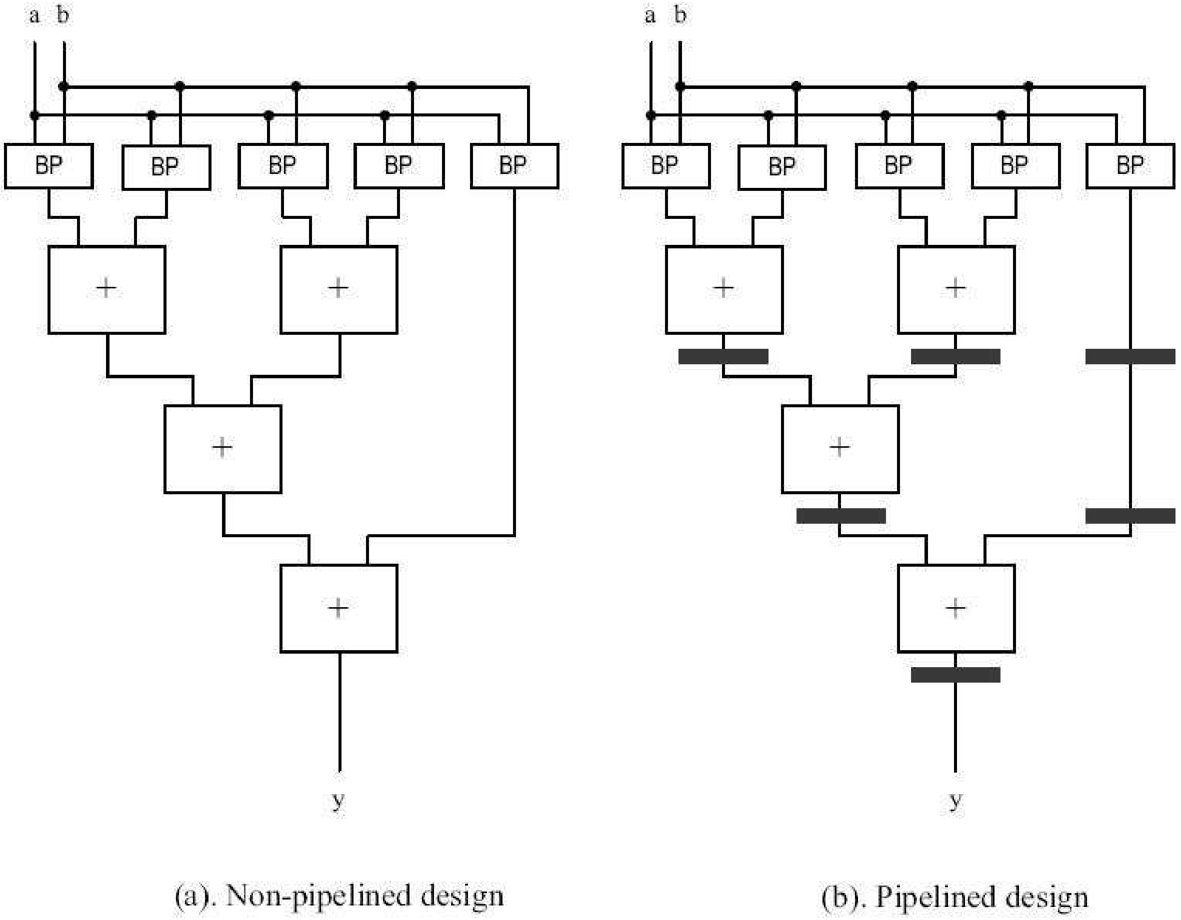
See text for VHDL code

You can also reduce the delay of the *n*-bit combinational multiplier from *n-1* adders to $ceiling(\log_2 n)$ using a tree-shaped network

This also works for the pipelined version by computing the bit-products in parallel and feeding them into tree-shaped network

Pipelines

Non-pipelined and pipelined version of tree adder network (see text for VHDL)



(a). Non-pipelined design

(b). Pipelined design

Figure 9.21 Block diagram of a tree-shaped pipelined multiplication circuit.