Sequential Circuit Design: Practice

Outline

1. Poor design practice and remedy

- Synchronous design is the most important methodology
- Poor practice in the past (to save chips)
  - Misuse of asynchronous reset
  - Misuse of gated clock
  - Misuse of derived clock

Misuse of asynchronous reset

- Poor design: use reset to clear register in normal operation.
- e.g., a poorly mod-10 counter
  - Clear register immediately after the counter reaches 1010

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

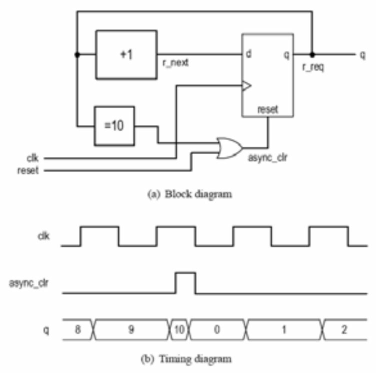
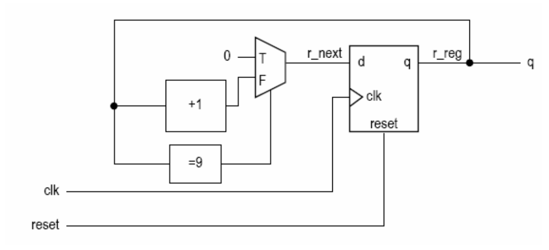


Figure 9.1 Decade counter using asynchronous reset

- Problem
  - Glitches in transition 1001 (9) => 0000 (0)
  - Glitches in aync_clr can reset the counter
  - How about timing analysis? (maximal clock rate)
- Asynchronous reset should only be used for power-on initialization

- Remedy: load "0000" synchronously

```
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;
   -- 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

- Poor design: use a and gate to disable the clock to stop the register to get new value
- E.g., a counter with an enable signal

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity binary_counter is
   port(
      clk, reset: in std_logic;
      en: 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
   -- register
   process (gated_clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (gated_clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- gated clock
   gated_clk <= clk and en;
   -- next state logic
   r_next <= r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
end gated_clk_arch;
```

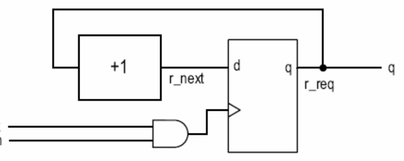
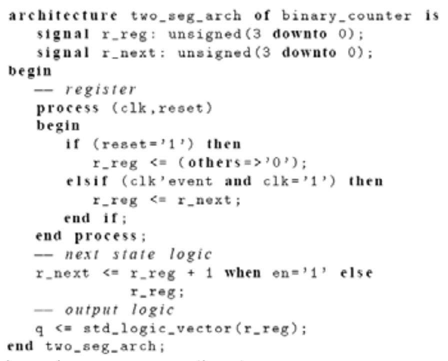


Figure 9.2 Disabling FF with gated clock

- Problem
  - Gated clock width can be narrow
  - Gated clock may pass glitches of en
  - Difficult to design the clock distribution network

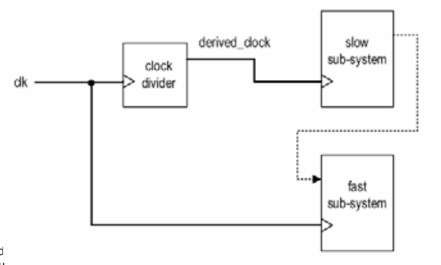
- Remedy: use a synchronous enable



```
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;
   -- 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;
```

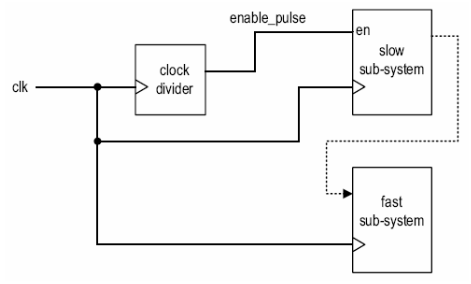
Misuse of derived clock

- Subsystems may run at different clock rate
- Poor design: use a derived slow clock for slow subsystem

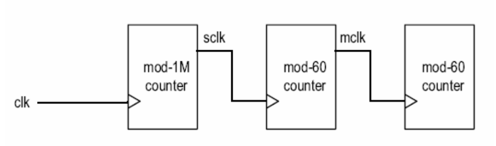


- Problem
  - Multiple clock distribution network
  - How about timing analysis? (maximal clock rate)

- Better use a synchronous one-clock enable pulse

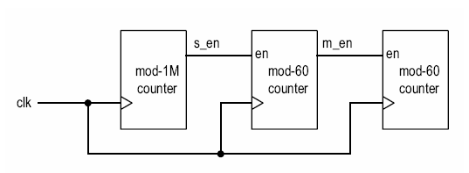


- E.g., second and minutes counter
  - Input: 1 MHz clock
  - Poor design:



(a) Design with derived clock

– Better design



(b) Design with a single synchronous clock

- VHDL code of poor design

```
library ieee;
use ieee.std_logic_1164.all;
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(3 downto 0);
   signal r_next: unsigned(3 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
```

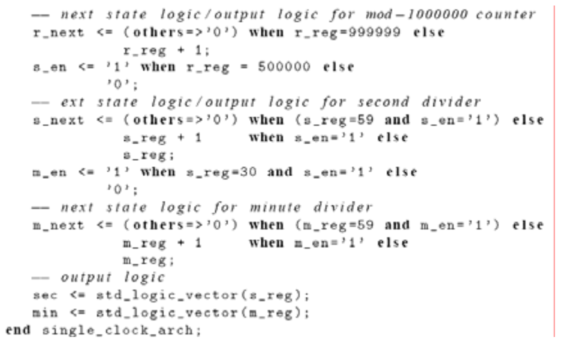
```
-- register
process (clk, reset)
begin
   if (reset='1') then
      r_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      r_reg <= r_next;
   end if;
end process;
-- next state logic
r_next <= (others=>'0') when r_reg=99999 else
          r_reg + 1;
-- output logic
sclk <= '0' when r_reg < 500000 else
        '1';
```

```
-- 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
mclk <= '0' when s_reg < 30 else
        '1';
sec <= std_logic_vector(s_reg);
```

```
   -- 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;
```

- Remedy: use a synchronous 1-clock pulse

```
architecture single_clock_arch of timer is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 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');
      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 logic/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';
  -- ext 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=30 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;
```

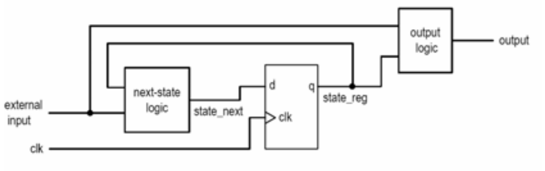
A word about power

- Power is a major design criteria now
- In CMOS technology
  - Dynamic power is proportional to the switching frequency of transistors
  - High clock rate implies high switching freq
- Clock manipulation
  - Can reduce switching frequency
  - But should not be done at RT level

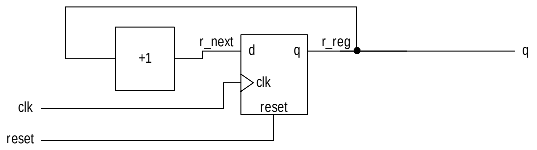
- Development flow:
  1. Design/synthesize/verify a regular synchronous subsystems
  2(a). Derived clock: use special circuit (PLL etc.) to obtain derived clocks
  2(b). Gated clock: use “power optimization” software tool to convert some register into gated clock

2. More counters

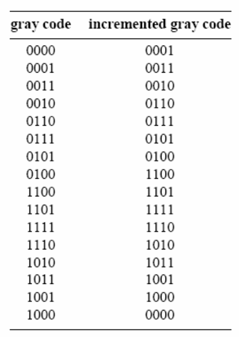
- Counter circulates a set of specific patterns
- Counter:
  – Binary
  – Gray counter
  – Ring counter
  – Linear Feedback Shift Register (LFSR)
  – BCD counter



- Binary counter:
  – State follows binary counting sequence
  – Use an incrementor for the next-state logic



- Gray counter:
  – State changes one-bit at a time
  – Use a Gray incrementor



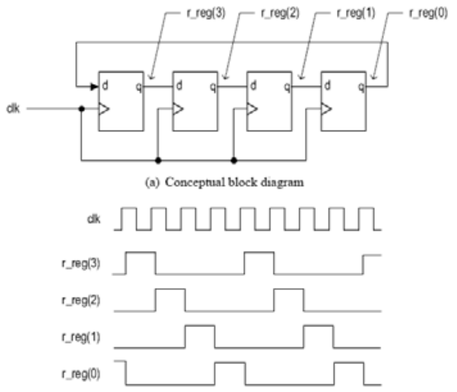
gray code	incremented gray code
0000	0001
0001	0011
0011	0010
0010	0110
0110	0111
0111	0101
0101	0100
0100	1100
1100	1101
1101	1111
1111	1110
1110	1010
1010	1011
1011	1001
1001	1000
1000	0000

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
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
      if (reset='1') then
         g_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         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;
```

Ring counter

- Circulate a single 1
- E.g., 4-bit ring counter:
  1000, 0100, 0010, 0001
- *n* patterns for *n*-bit register
- Output appears as an *n*-phase signal
- Non self-correcting design
  - Insert "0001" at initialization and circulate the pattern in normal operation
  - Fastest counter



```
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 d
   signal r_next: std_logic_vector(WIDTH-1
begin
   -- 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;
 end reset_arch; logic
   r_next <= r_reg(0) & r_reg(WIDTH-1 downt
   -- output logic
   q <= r_reg;
```

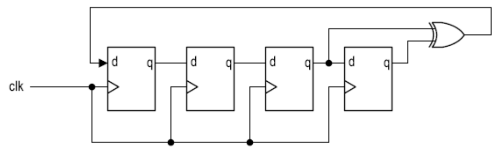
- Self-correcting design:
  shifting in a '1' only when 3 MSBs are 000

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

LFSR (Linear Feedback Shift Reg)

- A sifter reg with a special feedback circuit to generate the serial input
- The feedback circuit performs xor operation over specific bits
- Can circulate through 2^n-1 states for an n-bit register

- E.g, 4-bit LFSR



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

- Property of LFSR
  - N-bit LFSR can cycle through 2^n-1 states
  - The feedback circuit always exists
  - The sequence is pseudorandom

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	$q_{29} \oplus q_{17} \oplus q_2 \oplus q_0$

- Application of LFSR
  - Pseudorandom: used in testing, data encryption/decryption
  - A counter with simple next-state logic e.g., 128-bit LFSR using 3 xor gates to circulate 2^{128}-1 patterns (takes 10^{12} years for a 100 GHz system)

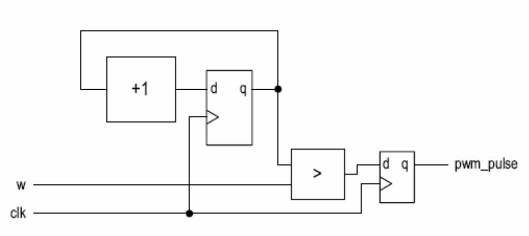
```
use ieee.std_logic_1164.all;
entity lfsr4 is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(3 downto 0));
end lfsr4;

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";
```

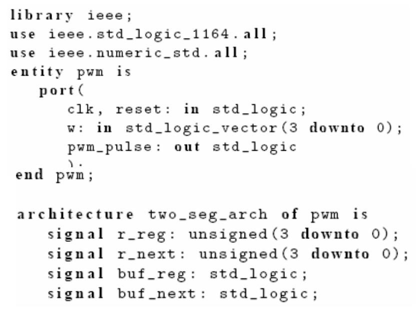
```
begin
   -- register
   process (clk,reset)
   begin
      if (reset='1') then
         r_reg <= SEED;
      elsif (clk'event and clk='1') then
   end process;
   -- next-state logic
   fb <= r_reg(1) xor r_reg(0);
   r_next <= fb & r_reg(3 downto 1) ;
   -- output logic
   q <= r_reg;
end no_zero_arch;
```

- Read remaining of Section 9.2.3 (design to including 00..00 state)
- Read Section 9.2.4 (BCD counter, design similar to the second/minute counter in Section 9.1.3

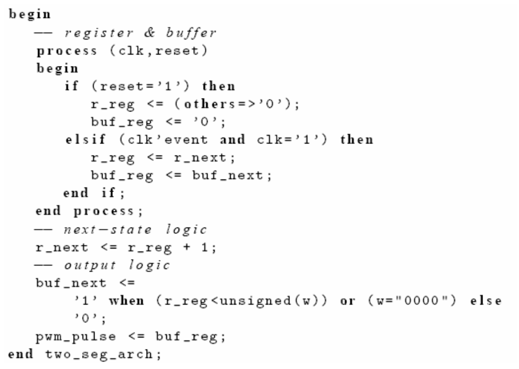
PWM (pulse width modulation)

- Duty cycle: percentage of time that the signal is asserted
- PWM: use 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”
- 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;

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 & 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;
   -- next-state logic
   r_next <= r_reg + 1;
   -- output logic
   buf_next <=
        '1' when (r_reg<unsigned(w)) or (w="0000") else
        '0';
   pwm_pulse <= buf_reg;
end two_seg_arch;
```

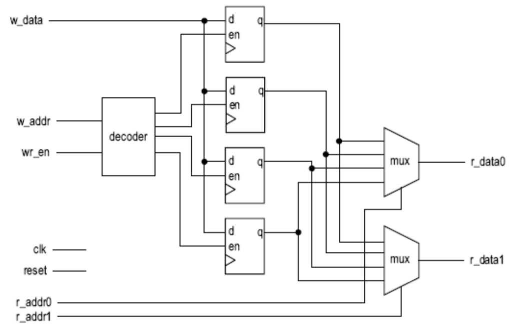
3. Register as fast temporary storage

- RAM
  – RAM cell designed at transistor level
  – Cell use minimal area
  – Behave like a latch
  – For mass storage
  – Need a special interface logic
- Register
  – D FF requires much larger area
  – Synchronous
  – For small, fast storage
  – E.g., register file, fast FIFO, Fast CAM (content addressable memory)

Register file

- Registers arranged as an 1-d array
- Each register is identified with an address
- Normally has 1 write port (with write enable signal)
- Can has multiple read ports

- E.g., 4-word register file w/ 1 write port and two read ports



- Register array:
  - 4 registers
  - Each register has an enable signal
- Write decoding circuit:
  - 0000 if wr_en is 0
  - 1 bit asserted according to w_addr if wr_en is 1
- Read circuit:
  - A mux for each read por

- 2-d data type needed

```
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;

architecture no_loop_arch of reg_file is
   constant W: natural:=2; -- number of bits in address
   constant B: natural:=16; -- number 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);
```

```
-- 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');
   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
```

```
-- 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;
   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;
```

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

- “Elastic” storage between two subsystems

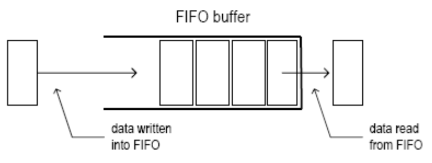
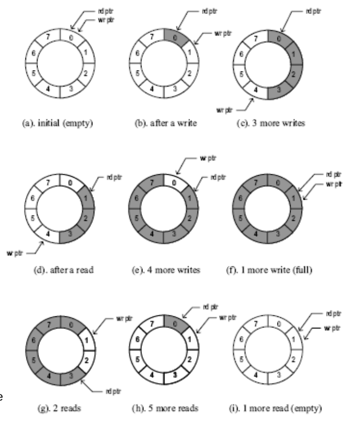
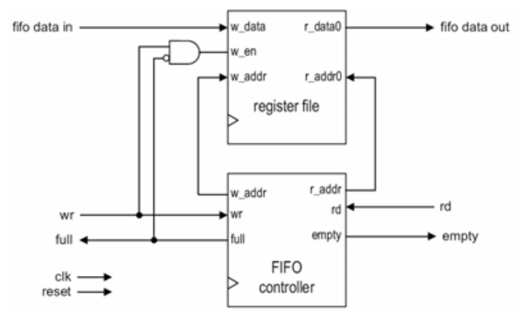


Figure 9.11 Conceptual diagram of a FIFO buffer.

- Circular queue implementation
- Use two pointers and a “generic storage”
  - Write pointer: point to the empty slot before the head of the queue
  - Read pointer: point to the tail of the queue



- FIFO controller
  - Read and write pointers: 2 counters
  - Status circuit:
    - Difficult
    - Design 1: Augmented binary counter
    - Design 2: with status FFs
  - LSFR as counter



- Augmented binary counter:
  - increase the counter by 1 bits
  - 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;

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;

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
```

```
-- 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
   w_ptr_next <=
      w_ptr_reg + 1 when wr='1' and full_flag='0' else
      w_ptr_reg;
   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;
   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;
```

- 2 extra status FFs
  - Full_erg/empty_reg memorize the current staus
  - Initialized as 0 and 1
  - Modified according to wr and rd signals:
    - 00: no change
    - 11: advance read pointer/write pointer; full/empty no change
    - 10: advance write pointer; de-assert empty; assert full if needed (when write pointer=read pointer)
    - 01: advance read pointer; de-assert full; asserted empty if needed (when write pointer=read pointer)

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

```
-- successive value for 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 op
         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;
         when others => -- write/read;
            w_ptr_next <= w_ptr_succ;
            r_ptr_next <= r_ptr_succ;
       end case;
   end process;
```

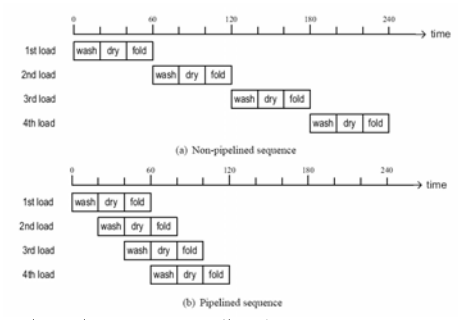
- Non-binary counter for the pointer
  - Exact location does not matter as long as the write pointer and read pointer follow the same pattern
  - Other counters can be used for the second scheme
  - E.g, use LFSR

```
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);
```

4. Pipelined circuit

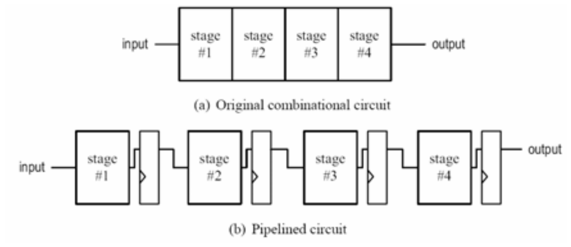
- Two performance criteria:
  - Delay: required time to complete one task
  - Throughput: number of tasks completed per unit time.
- E.g., ATM machine
  - Original: 3 minutes to process a transaction
    delay: 3 min; throughput: 20 trans per hour
  - Option 1: faster machine 1.5 min to process
    delay: 1.5 min; throughput: 40 trans per hour
  - Option 2: two machines
    delay: 3 min; throughput: 40 trans per hour
- Pipelined circuit: increase throughput

- Pipeline: overlap certain operation
- E.g., pipelined laundry:



- Non-pipelined:
  - Delay: 60 min
  - Throughput 1/60 load per min
- Pipelined:
  - Delay: 60 min
  - Throughput k/(40+k*20) load per min
    about 1/20 when k is large
  - Throughput 3 times better than non-pipelined

Pipelined combinational circuit



(a) Original combinational circuit

(b) Pipelined circuit

$$T_{comb} = T_0 + T_1 + T_2 + T_3 \qquad \frac{1}{T_{comb}}.$$

$$T_{max} = \max(T_0, T_1, T_2, T_3) \qquad \frac{k}{3*T_c+k*T_c}$$

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

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

$$T_{pipe} = 4 * T_c \approx 4 * T_{max} = T_{comb}$$

$$\frac{1}{T_c} \approx \frac{1}{T_{max}} = \frac{4}{T_{comb}}$$

Adding pipeline to a comb circuit

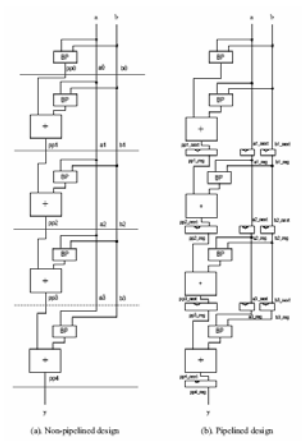
- Candidate circuit for pipeline:
  - enough input data to feed the pipelined circuit
  - throughput is a main performance criterion
  - comb circuit can be divided into stages with similar propagation delays
  - propagation delay of a stage is much larger than the setup time and the clock-to-q delay of the register.

- Procedure
  - 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
  - Insert registers for these signals in the boundary.

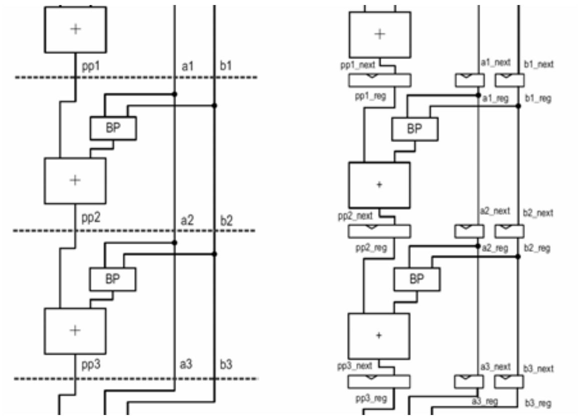
Pipelined comb multiplier

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

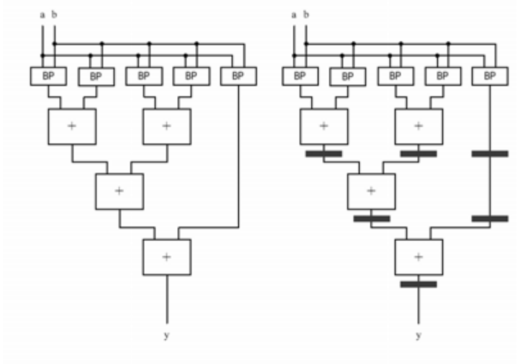
```
begin
   au <= unsigned(a);
   bv0 <= (others=>b(0));
   bv1 <= (others=>b(1));
   bv2 <= (others=>b(2));
   bv3 <= (others=>b(3));
   bv4 <= (others=>b(4));
   bv5 <= (others=>b(5));
   bv6 <= (others=>b(6));
   bv7 <= (others=>b(7));
   p0 <="00000000" & (bv0 and au);
   p1 <="0000000" & (bv1 and au) & "0";
   p2 <="000000" & (bv2 and au) & "00";
   p3 <="00000" & (bv3 and au) & "000";
   p4 <="0000" & (bv4 and au) & "0000";
   p5 <="000" & (bv5 and au) & "00000";
   p6 <="00" & (bv6 and au) & "000000";
   p7 <="0" & (bv7 and au) & "0000000";
   prod <= ((p0+p1)+(p2+p3))+((p4+p5)+(p6+p7));
   y <= std_logic_vector(prod);
end comb1_arch;
```



(a). Non-pipelined design (b). Pipelined design



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


--  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;
```



(a). Non-pipelined design (b). Pipelined design

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