Prof. Dr.-Ing. A. Siggelkow

VHDL Design Guide

Hochschule Ravensburg-Weingarten

Inhaltsverzeichnis

The intention of this paper is to give hints about the design environment of the EIS-laboratory of the University of Applied Science Ravensburg-Weingarten. This is:

- setting up a project
- simulate a VHDL code
- write synthesisable VHDL
- synthesize the chip.

Zweck dieses Papiers ist, die Designumgebung in EIS-Labor der Hochschule Ravensburg-Weingarten darzustellen. Dazu gehört das Aufsetzen eines Projekts, den entworfenen VHDL-Code zu simulieren, den VHDL-Code synthetisierbar zu beschreiben und letzendlich den Code zu synthetisieren.

Teil I

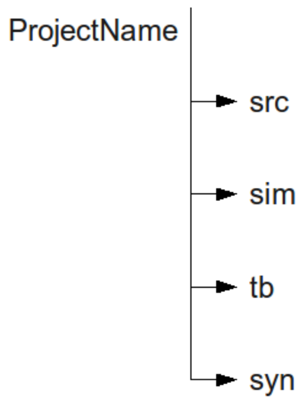
Design Environment

Kapitel 1

Directory Structure

The directory should be strucured to be able to distinguish the needed different design steps. First of all, the project must have a name, this is the name of the first directory level. Then the design steps will be named: VHDL-coding (src), simmulation (sim), testbench (tb) and synthesis (syn).

Die Verzeichnisstruktur sollte so angelegt werden, dass aus dieser die verschiedenen Entwicklungsschritte erkennbar sind. Dazu gehört auch der Projektname als Wurzelverzeichnis. Die darunterliegenden Namen der Verzeichnisse sind: VHDL-Code in “src”, die Simulation in “sim”, die Testumgebung/Testbench in “tb” und letztlich die Synthese in “syn”.



The src-directory can be divided, if the design will be complex, into some more sub-directories: for the entities (separate files), for the architecture files and for the configuration

Diese Verzeichnisse, z.B. das “src”-Verzeichnis, können weiter unterteilt werden, z.B. für die entities, die architectures und die configurations.

files. Befehle dies zu erreichen:

Commands to do this:

1. cd
2. mkdir myLabName
3. cd myLabName
4. mkdir ProjectName
5. cd ProjectName
6. mkdir src
7. mkdir sim
8. mkdir tb
9. mkdir syn

1.1 Project Name

It should be self-explaining name for your project.

1.2 Source

Only the VHDL source files will be allowed to be here.

1.3 Simulation

This directory is reserved for the simulation and all of its binaries.

1.4 Testbench

Only the TestBench VHDL- or MODELSIM-.do files will be allowed here.

1.5 Synthesis

All kinds of synthesis jobs could be located here.

Teil II

Design Simulation

For the simulation with MODELSIM a few commands will be needed.

- qhlib work: Go to the sim-directory of your project and type this command. This will generate the binary container for your simulation. It will be done at the beginning of the project once and never again.
- qvhcom ../src/myfile.vhd: Go to the sim-directory of your project and type this command. this will compile the chosen VHDL file.
- qhsim: This command starts the MODELSIM simulator.

Teil III

Design Guide

Kapitel 2

Naming Conventions

2.1 Naming of Design Parts

2.1.1 Entity

The entity names should be marked by the extension “_e” at the end of the name.

Der Entity-Name sollte mit dem Anhängsel “_e” enden.

- entityName_e

2.1.2 Architecture

The architecture names should be marked by the extension “_a” at the end of the name.

Der Architecture-Name sollte mit dem Anhängsel “_a” enden.

- architectureName_a

2.2 Naming of Connections

2.2.1 Inputs

The naming of an input, defined in the entity, should end with an _i. This makes within the following design clear, that this is an input. If de-

Die Namensgebung von Eingängen, in der Entity definiert, endet mit einem _i. Somit kann man im folgenden Design sehen, das dieses signs will be bigger this could help to keep the overview. For example:

Signal ein Eingang ist. Man kann so in größeren Designs besser den Überblick behalten. Z.B.:

- clk_i
- rst_n_i

In the example above, the reset has an additional _n, this makes clear, that it is a active-low signal.

Im obigen Besipiel hat die Resetleitung im Namen den Teil _n. Dieser stellt klar, dass es ein low-aktives Signal ist

2.2.2 Outputs

The naming of an output, defined in the entity, should end with an _o. This makes within the following design clear, that this is an input. For example:

Die Namensgebung von Ausgängen, in der Entity definiert, endet mit einem _o. Somit kann man im folgenden Design sehen, das dieses Signal ein Eingang ist. Z.B.:

- result_o

2.2.3 Signals

The naming of an internal defined signal, defined in the architecture, should end with an _s. This makes within the following design clear, that this is an internal signal. For example:

Die Namensgebung von internen Signalen, in der Architecture definiert, endet mit einem _s. Somit kann man im folgenden Design sehen, das dieses Signal ein internes Signal ist. Z.B.:

- internalSignal_s

2.2.4 Variables

The naming of an internal defined variables, defined in the pro-

Die Namensgebung von internen variablen, im Process oder Functicess or function/procedure, should end with an _v. This makes within the following design clear, that this is an local variable. For example:

on/Procedure definiert, endet mit einem _v. Somit kann man im folgenden Design sehen, das dieses Signal eine lokale Variable ist. Z.B.:

- internalVariable_v

2.3 Naming of Others

2.3.1 Types

Names of **types** should end with an _t. For example:

Die Namen von **Typen** enden mit einem _t. Z.B.:

- internalType_t

2.3.2 States/Zustände

Names of **states** should end with an _st. For example:

Die Namen von **Zuständen** enden mit einem _st. Z.B.:

- internalState_st

Kapitel 3

Summary

3.1 Un-Categorized

- use _i, _o, _s, _e, _a
- entity and architecture are in separate files
- the project setup has the directories: *src*, *sim*, *syn* and *tb*
- use *std_logic* instead of *bit*
- implement asynchronous resets and synchronous clocks for all FlipFlops
- use *(others => '0')* instead of *00000000*
- the bit-width of vectors must be given as generic or as a constant in the package, not fixed in the code
- In general, no numbers, except 0 will be allowed in the code. The needed numbers will be defined as generic or as a constant in the package.
- use *integer range my_const downto 0* wherever possible; my_const is a generic or a constant in a package
- implement limits only with constants (if (cnt >= max_cnt) then)
- no initial signal assignments in synthesizable code
- no latches
- use structural code wherever it makes sense
- All simulations must be shown with Modelsim

- All synthesis runs must be done with ISE/ Vivado
- The Top-Level has only components, no functional VHDL code.

3.2 Guidelines Category I

Line-Break: Maximal 80 characters per line, including comments.

Large and lower case: All VHDL language elements like ENTITY, ARCHITECTURE, BEGIN, IF, END, DOWNTO, AND, OR, NOT, etc. in capital letters. All self defined names in lower case letters. No mixed style. IEEE definitions like std_logic, signed oder rising_edge in lower case letters.

Indent: Within hierarchy, loops, condition make a two space indent (no tab).

Libraries: For synthesizable code only std_logic_1164 and numeric_std will be allowed.

Names: All designators should have names according its meaning. Do not use reserved VHDL words.

3.3 Guidelines Category II

Package: For the project, a package is mandatory. Every module should be referenced in the package with the COMPONENT-declaration (So, it is not needed anymore in the ARCHITECTURE).

Signal types: For the syntesizable code use std_logic, std_logic_vector, signed, unsigned and integer range top downto low only.

3.4 Guidelines Category III

Documentation: Every module must be documented in its own data sheet. The function, the inputs and outputs, the signals, the processes and the registers must be described (latex document).

Self-Checking Testbench: Every module must have its own testbench which covers the function completely.

Kapitel 4

VHDL for Synthesis

4.1 Register

4.1.1 D-FlipFlop

Entity:

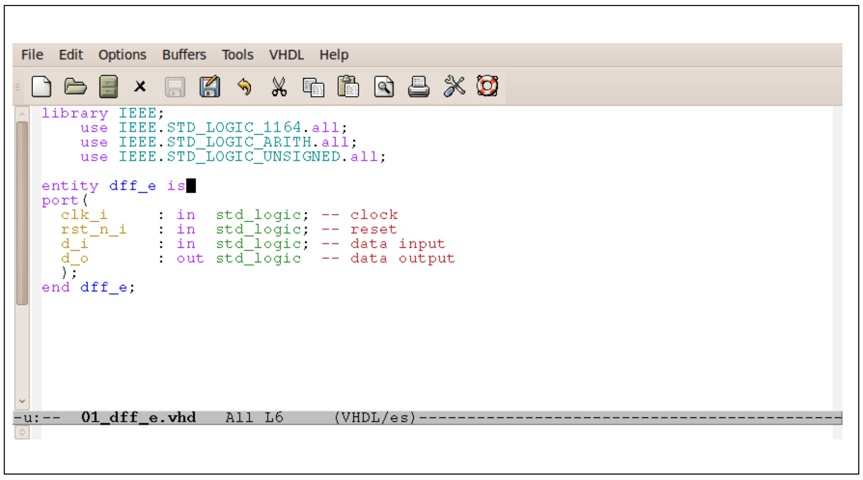
```
ENTITY d_ff_e IS
    PORT(   clk_i, d_i        : IN std_logic;
            q_o               : OUT std_logic);
END d_ff_e;
```

Architecture:

```
ARCHITECTURE d_ff_a OF d_ff_e IS
BEGIN
    PROCESS(clk_i)
    BEGIN
            IF (clk_i'event AND clk_i = '1') THEN
                            q_o <= d_i;
            END IF;
    END PROCESS;
END d_ff_a;
```

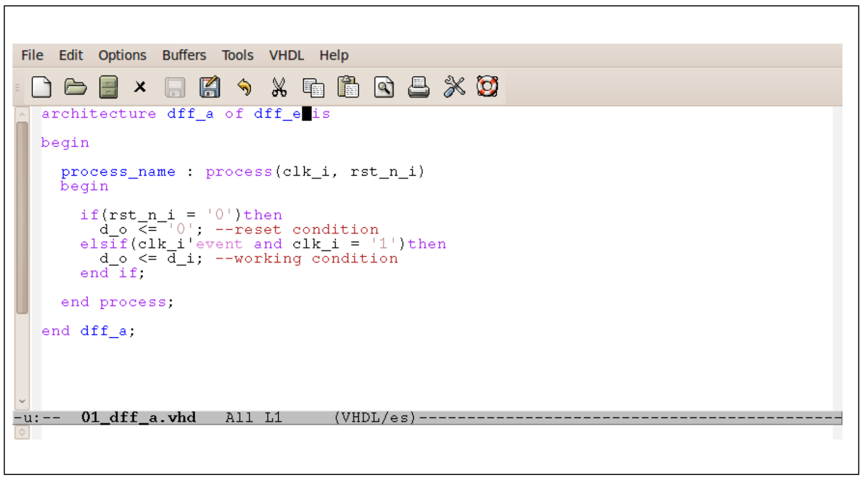
4.1.2 D-FlipFlop with A-Synchronous Reset

Entity:


File Edit Options Buffers Tools VHDL Help
-u:-- 01_dff_e.vhd All L6 (VHDL/es)

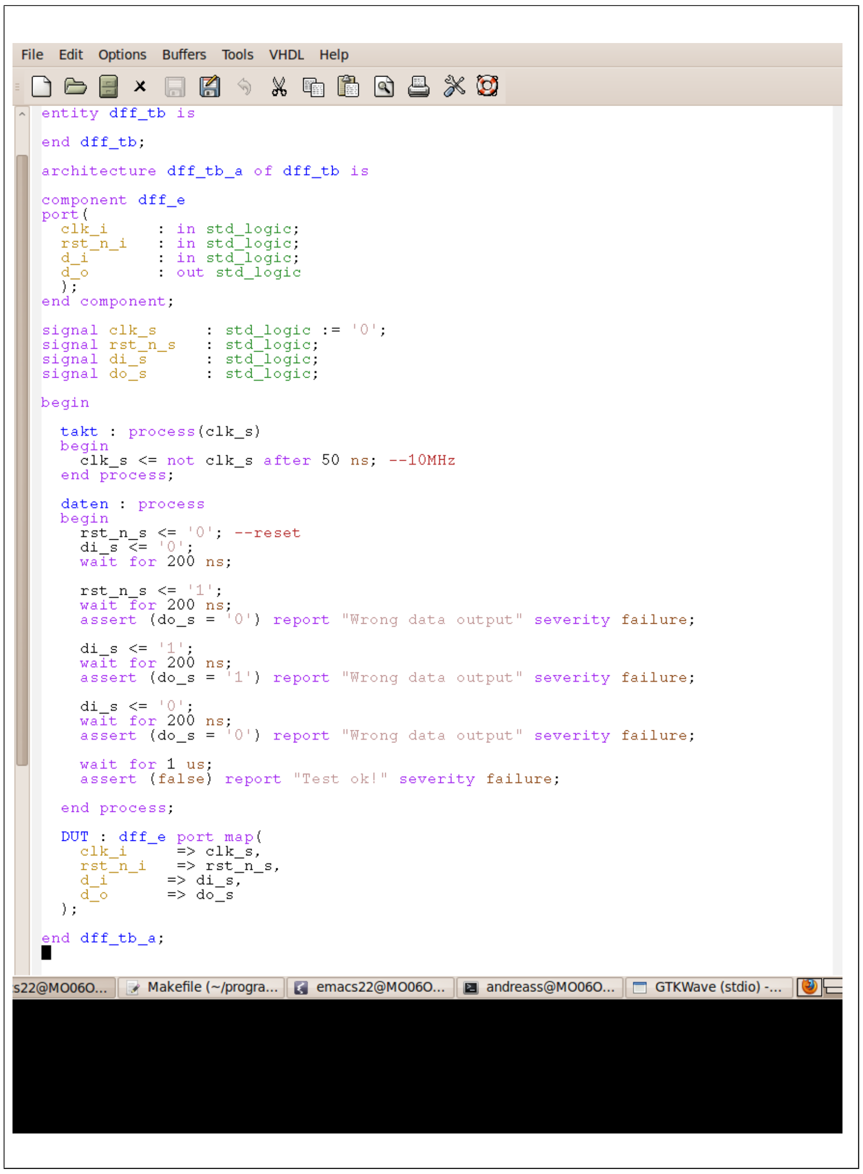
```
library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.STD_LOGIC_ARITH.all;
    use IEEE.STD_LOGIC_UNSIGNED.all;

entity dff_e is
port(
  clk_i      : in  std_logic; -- clock
  rst_n_i    : in  std_logic; -- reset
  d_i        : in  std_logic; -- data input
  d_o        : out std_logic  -- data output
  );
end dff_e;
```

Architecture:


File Edit Options Buffers Tools VHDL Help
-u:-- 01_dff_a.vhd All L1 (VHDL/es)

```
architecture dff_a of dff_e is

begin

  process_name : process(clk_i, rst_n_i)
  begin

    if(rst_n_i = '0')then
      d_o <= '0'; --reset condition
    elsif(clk_i'event and clk_i = '1')then
      d_o <= d_i; --working condition
    end if;

  end process;

end dff_a;
```

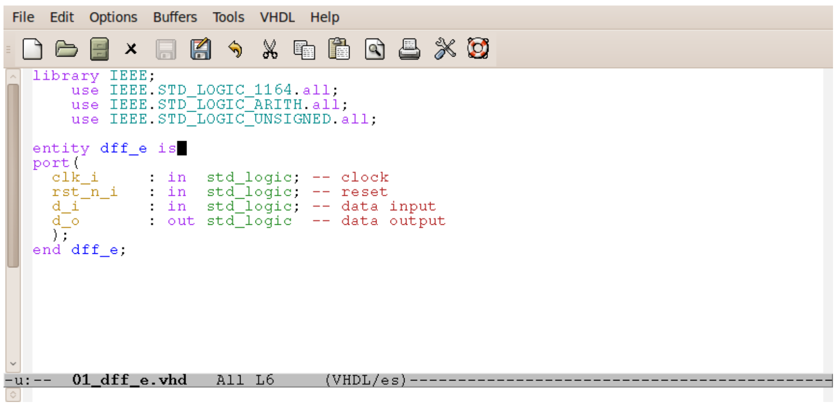
Test-Bench:



```vhdl
entity dff_tb is

end dff_tb;

architecture dff_tb_a of dff_tb is

component dff_e
port(
  clk_i     : in std_logic;
  rst_n_i   : in std_logic;
  d_i       : in std_logic;
  d_o       : out std_logic
  );
end component;

signal clk_s     : std_logic := '0';
signal rst_n_s   : std_logic;
signal di_s      : std_logic;
signal do_s      : std_logic;

begin

  takt : process(clk_s)
  begin
    clk_s <= not clk_s after 50 ns; --10MHz
  end process;

  daten : process
  begin
    rst_n_s <= '0'; --reset
    di_s <= '0';
    wait for 200 ns;

    rst_n_s <= '1';
    wait for 200 ns;
    assert (do_s = '0') report "Wrong data output" severity failure;

    di_s <= '1';
    wait for 200 ns;
    assert (do_s = '1') report "Wrong data output" severity failure;

    di_s <= '0';
    wait for 200 ns;
    assert (do_s = '0') report "Wrong data output" severity failure;

    wait for 1 us;
    assert (false) report "Test ok!" severity failure;

  end process;

  DUT : dff_e port map(
    clk_i     => clk_s,
    rst_n_i   => rst_n_s,
    d_i      => di_s,
    d_o      => do_s
  );

end dff_tb_a;
```

4.1.3 D-FlipFlop with Synchronous Reset

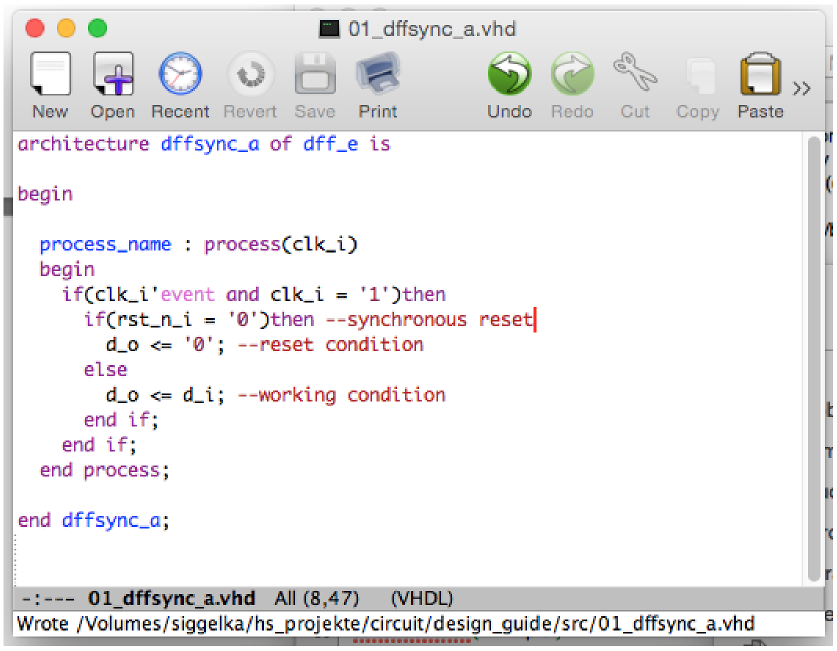
Entity:



```
library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.STD_LOGIC_ARITH.all;
    use IEEE.STD_LOGIC_UNSIGNED.all;

entity dff_e is
port(
  clk_i       : in  std_logic; -- clock
  rst_n_i     : in  std_logic; -- reset
  d_i         : in  std_logic; -- data input
  d_o         : out std_logic  -- data output
  );
end dff_e;
```

```
-u:--  01_dff_e.vhd    All L6     (VHDL/es)------------------------------------------
```

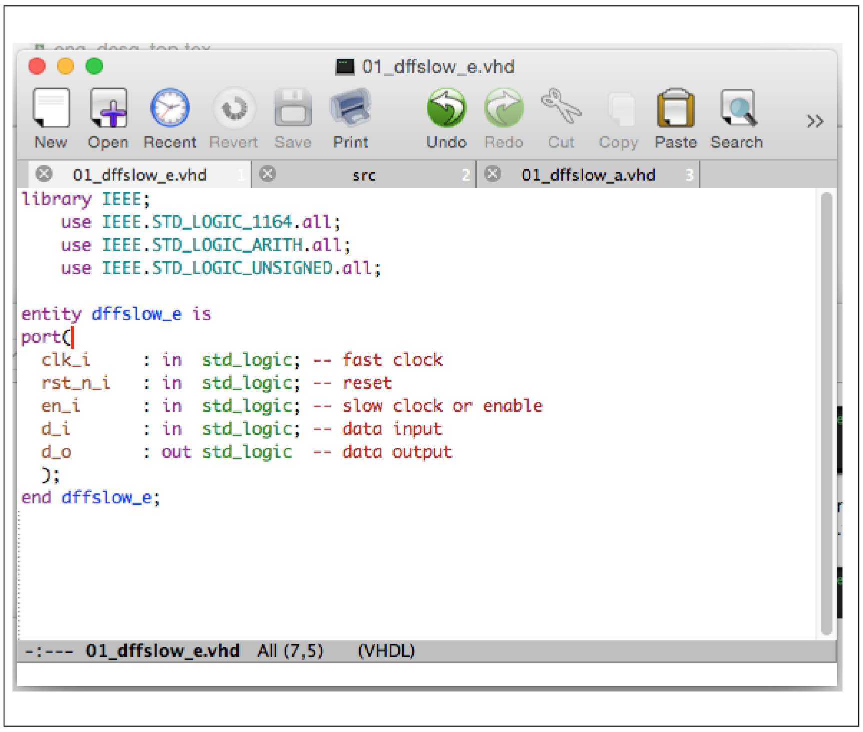
Architecture:



```
architecture dffsync_a of dff_e is

begin

  process_name : process(clk_i)
  begin
    if(clk_i'event and clk_i = '1')then
      if(rst_n_i = '0')then --synchronous reset
        d_o <= '0'; --reset condition
      else
        d_o <= d_i; --working condition
      end if;
    end if;
  end process;

end dffsync_a;
```

```
-:---  01_dffsync_a.vhd   All (8,47)    (VHDL)
Wrote /Volumes/siggelka/hs_projekte/circuit/design_guide/src/01_dffsync_a.vhd
```

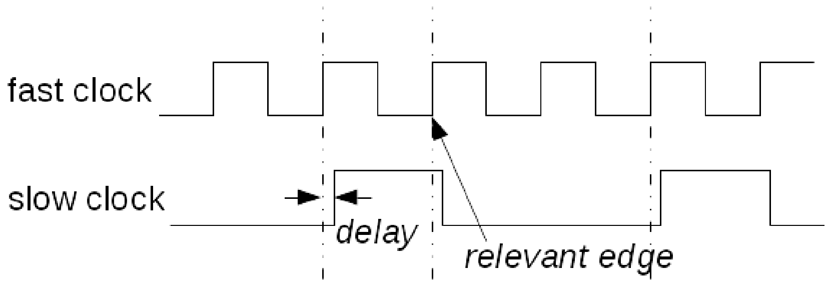
4.1.4 D-FlipFlop with Reset - Running with Slower Clock

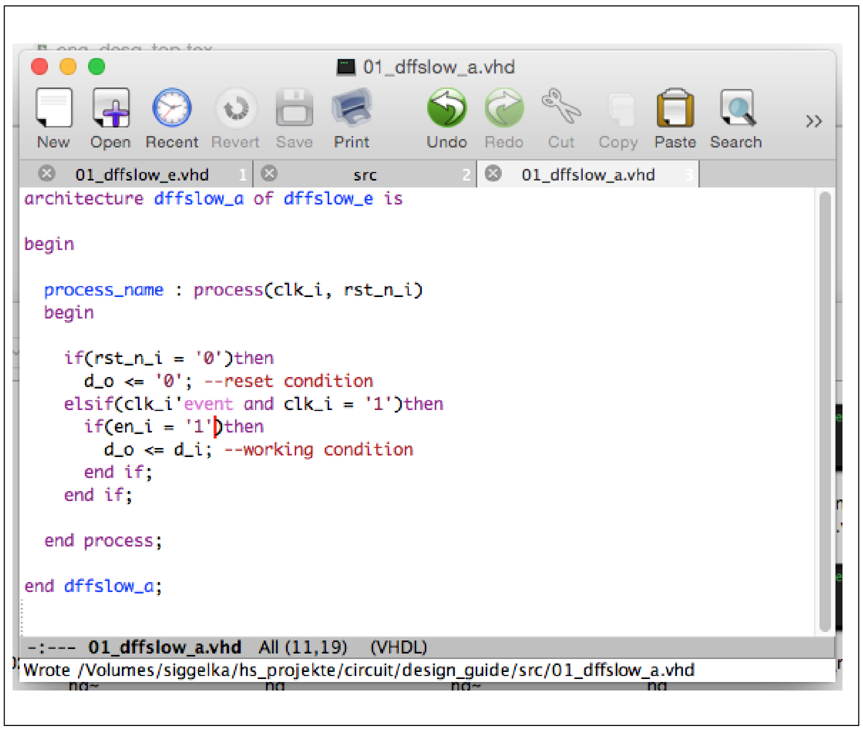


```
library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.STD_LOGIC_ARITH.all;
    use IEEE.STD_LOGIC_UNSIGNED.all;

entity dffslow_e is
port(
  clk_i     : in  std_logic; -- fast clock
  rst_n_i   : in  std_logic; -- reset
  en_i      : in  std_logic; -- slow clock or enable
  d_i       : in  std_logic; -- data input
  d_o       : out std_logic  -- data output
  );
end dffslow_e;
```

In order to achieve a synchronous design, all FlipFlops must be clocked with a single clock, the basic clock. But sometimes it is required to react on a slower clock. Here, it is not allowed to use the slower clock in the line like “elsif(slow_clock’event and slow_clock = ’1’)”. The basic clock must be used here. But how to slow down the system? Just go below the “elsif”line and start with a new “if” statement reacting on the slow clock.

The shown “slow clock” (= en_i) should be generated from the “fast clock” (clk_i) by a divider circuit as shown in the figure. The divider is a counter (clocked with the basic clock), it should count endlessly from 0 up to 2 and at every 0 state, the slow clock must be set to high and for every other state it should be set to low. It is absolutely required, that the generated “slow clocks” (= en_i) high duration is not longer than one original clock period.





```
architecture dffslow_a of dffslow_e is

begin

  process_name : process(clk_i, rst_n_i)
  begin

    if(rst_n_i = '0')then
      d_o <= '0'; --reset condition
    elsif(clk_i'event and clk_i = '1')then
      if(en_i = '1')then
        d_o <= d_i; --working condition
      end if;
    end if;

  end process;

end dffslow_a;
```

```
-:--- 01_dffslow_a.vhd   All (11,19)   (VHDL)
Wrote /Volumes/siggelka/hs_projekte/circuit/design_guide/src/01_dffslow_a.vhd
```

4.1.5 Synchronous Logic

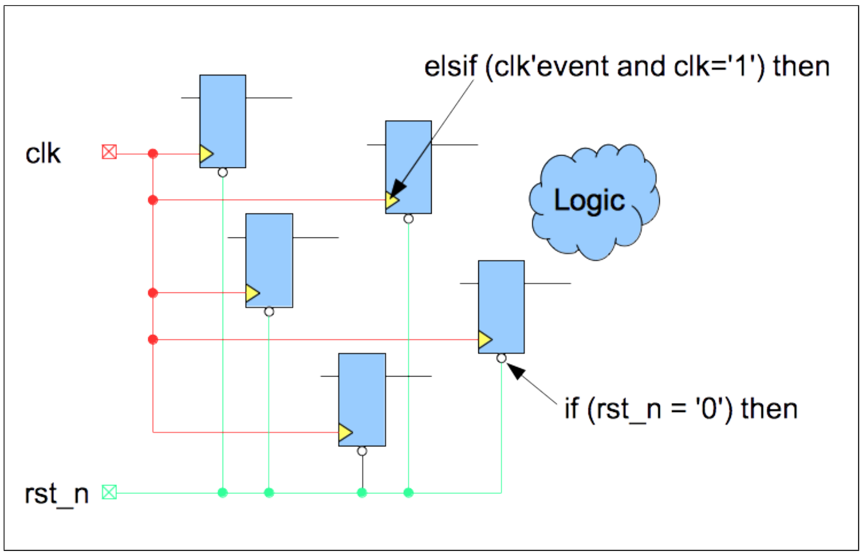
The generation of logic with FlipFlops like counter synchronizer, etc., can be done in the same way as the generation of simple D-FlipFlops.

- A process with the sensitivity-list containing only clk and rst is needed.
- The asynchronous reset requires the opening of an ”if” statement with ”if (rst_n = ’0’) then”.
- The reaction on rising clock edges requires ”elsif (clk’event and clk = ’1’) then”.
- This hierarchy of the ”if” statement needs to be finished without any ”else” statement, than it will be a storage element.
- Between the ”elsif (clk’event and clk = ’1’) then” and the ”end if”, there can be any logical description, also a new ”if” statement level with a ”else” in it.

4.1.6 Not Allowed

It is forbidden:

- to combine the resets ”if” part with any other signal (e.g. ”if (rst_n = ’0’ and enable = ’1’) then”), the reset is a pure signal from outside the chip, which brings the chip initially into a well defined state (Every FlipFlop requires a reset (or a set) stand-alone),
- to combine the clocks ”elsif” part with any other signal (e.g. ”elsif (clk’event and clk = ’1’ and enablex = ’1’) then”), the clock is a pure signal from outside the chip, which drives the chip (Every FlipFlop requires a clock),
- to have more than one ”elsif (clk’event and clk = ’1’) then” statement within one process,
- to clock the chip with mixed falling edges and rising edges, chose one.



4.1.7 Summary:

To describe a FlipFlop or other synchronous elements in VHDL it is required to write a process:

- with a sensitivity list which contains clk and reset,
- with a non-finished if - elsif - end if statement (NO else on the highest hierarchy),
- the “if”must handle the reset - only the reset, nothing else,
- the “elsif”must be the description of the clock (clk’event and clk = ’1’) - NOTHING else is allowed in this line,

- between “elsif” and “end if” is the place to describe any function you need: FlipFlop, counter, arbitrary functions.

If this procedure will be followed, both, simulation and synthesis will be successful.

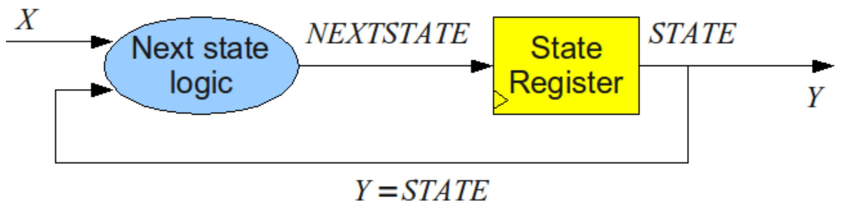
4.2 Finite State Machines - FSM

4.2.1 Medvedev

Principle:

The three different finite state machines (Medvedev, Moore und Mealy) differ in the calculation of the output values. For the Medvedev-FSM is the value of the output always equal to the value of the state. So, the output logic is just wiring.

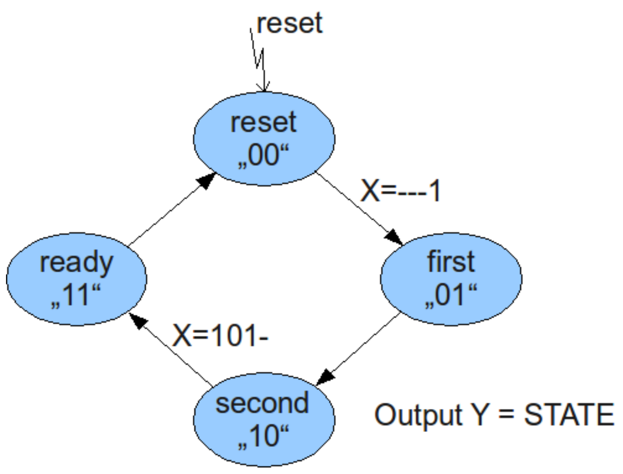
Die grundlegenden drei verschiedenen Automaten aus der Theorie (Medvedev, Moore und Mealy Automaten) unterscheiden sich in der Berechnung der Ausgangswerte. Beim Medvedev Automat ist der Wert des Ausgangs immer gleich dem Wert des Zustandvektors. Das heißt die Logik für den Ausgang besteht aus einer reinen Verdrahtung; nämlich der Verdrahtung des Zustandvektors mit dem Ausgangsvektor.



The FSM can be represented by means of a state diagram (bubble-diagram). The bubbles represents the states, names (e.g., first) and coding (“01”) and the edges represent the state transitions dependent of the input X and the current state. The conditions for a state change will be written to the edge. If nothing is mentioned at the edge, the state tran-

tbd

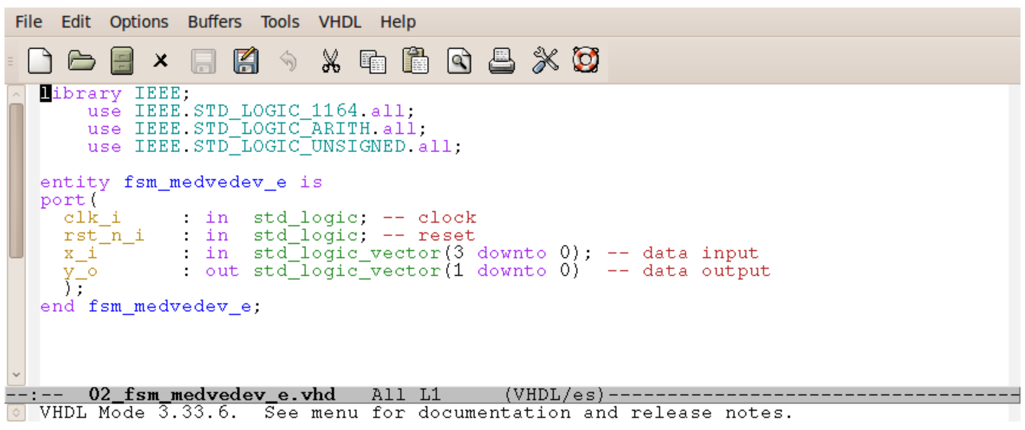
sition will be done without condition at the next delay (e.g., rising clock edge). If a specified condition is not fulfilled, the state will not change (evtl. edge to itself).



Entity:

An example of an simple Medvedev-FSM. The input vector is four bits wide, the output vector is two bits wide.

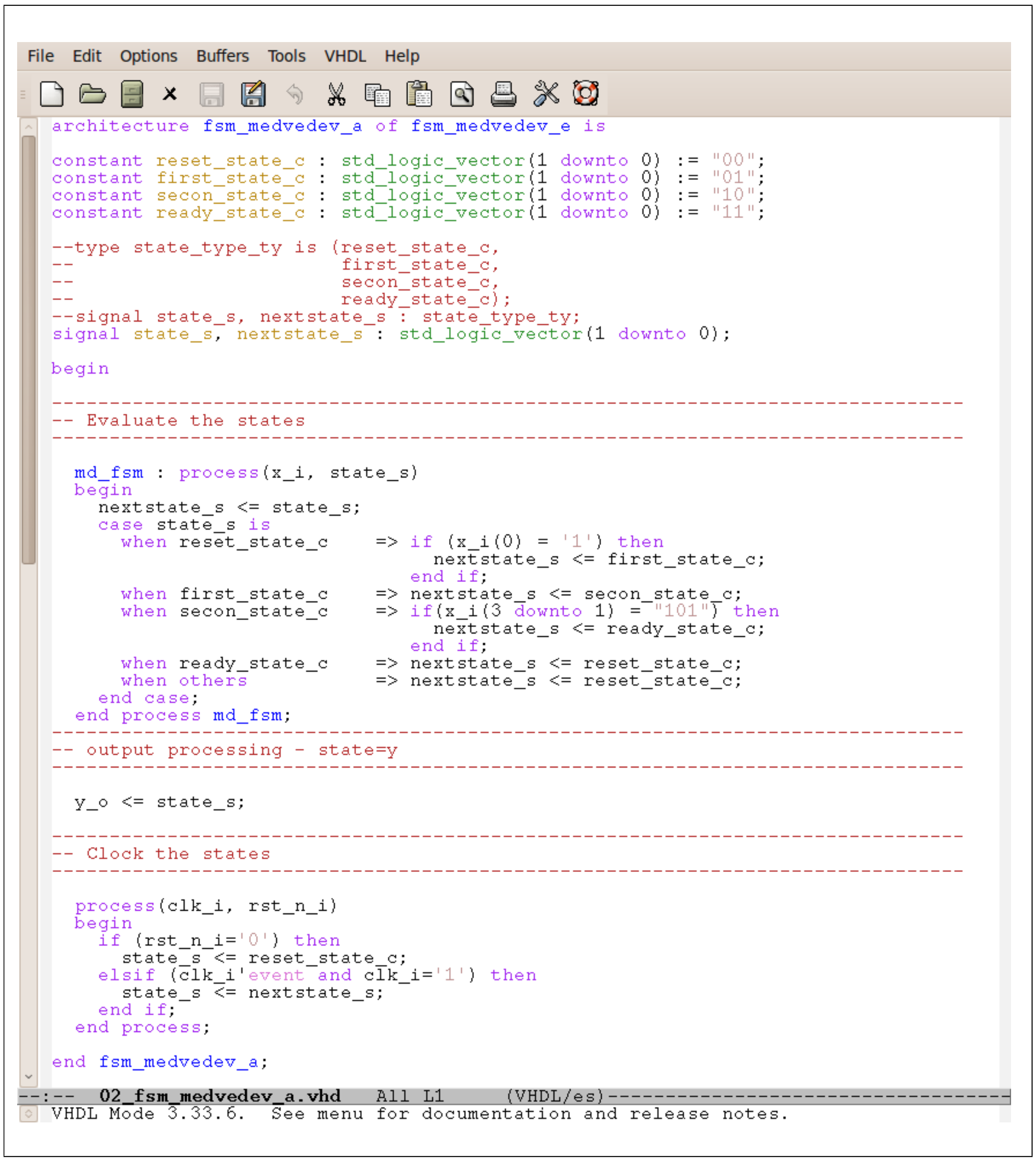
tbd



```
library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.STD_LOGIC_ARITH.all;
    use IEEE.STD_LOGIC_UNSIGNED.all;

entity fsm_medvedev_e is
port(
  clk_i      : in   std_logic; -- clock
  rst_n_i    : in   std_logic; -- reset
  x_i        : in   std_logic_vector(3 downto 0); -- data input
  y_o        : out  std_logic_vector(1 downto 0)  -- data output
  );
end fsm_medvedev_e;
```

```
--:--  02_fsm_medvedev_e.vhd    All L1       (VHDL/es)----------------------------------
VHDL Mode 3.33.6.   See menu for documentation and release notes.
```

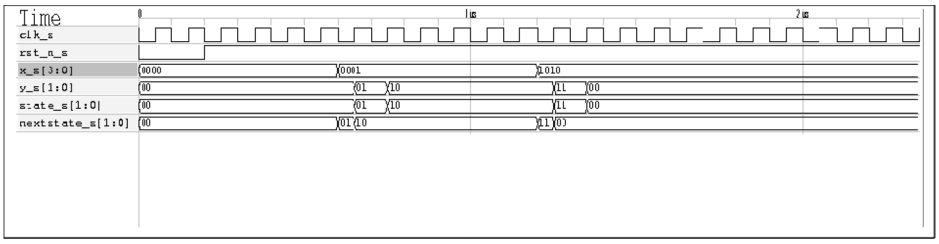
Architecture:



```vhdl
architecture fsm_medvedev_a of fsm_medvedev_e is

constant reset_state_c : std_logic_vector(1 downto 0) := "00";
constant first_state_c : std_logic_vector(1 downto 0) := "01";
constant secon_state_c : std_logic_vector(1 downto 0) := "10";
constant ready_state_c : std_logic_vector(1 downto 0) := "11";

--type state_type_ty is (reset_state_c,
--                       first_state_c,
--                       secon_state_c,
--                       ready_state_c);
--signal state_s, nextstate_s : state_type_ty;
signal state_s, nextstate_s : std_logic_vector(1 downto 0);

begin

-------------------------------------------------------------------------------
-- Evaluate the states
-------------------------------------------------------------------------------

  md_fsm : process(x_i, state_s)
  begin
    nextstate_s <= state_s;
    case state_s is
      when reset_state_c    => if (x_i(0) = '1') then
                                 nextstate_s <= first_state_c;
                               end if;
      when first_state_c    => nextstate_s <= secon_state_c;
      when secon_state_c    => if(x_i(3 downto 1) = "101") then
                                 nextstate_s <= ready_state_c;
                               end if;
      when ready_state_c    => nextstate_s <= reset_state_c;
      when others           => nextstate_s <= reset_state_c;
    end case;
  end process md_fsm;
-------------------------------------------------------------------------------
-- output processing - state=y
-------------------------------------------------------------------------------

  y_o <= state_s;

-------------------------------------------------------------------------------
-- Clock the states
-------------------------------------------------------------------------------

  process(clk_i, rst_n_i)
  begin
    if (rst_n_i='0') then
      state_s <= reset_state_c;
    elsif (clk_i'event and clk_i='1') then
      state_s <= nextstate_s;
    end if;
  end process;

end fsm_medvedev_a;
```

```
--:--  02_fsm_medvedev_a.vhd   All L1     (VHDL/es)----------------------------------
VHDL Mode 3.33.6.  See menu for documentation and release notes.
```

Simulation:

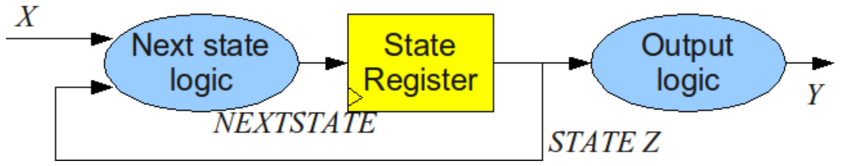


4.2.2 Moore

Principle:

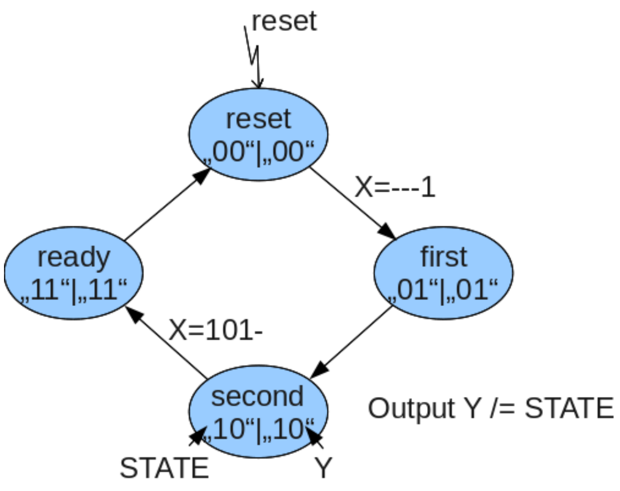
For the Moore-FSM is the value of the output a transformed value of the state. This is done by means of pure combinational logic.

tbd.



The bubbles represents the states, names (e.g., first) the coding of the states is not important here. The edges represent the state transitions dependent of the input X and the current state. The conditions for a state change will be written to the edge. If nothing is mentioned at the edge, the state transition will be done without condition at the next delay (e.g., rising clock edge). If a specified condition is not fulfilled, the state will not change (evtl. edge to itself). The output coding should be written to the states (instead of the state coding).

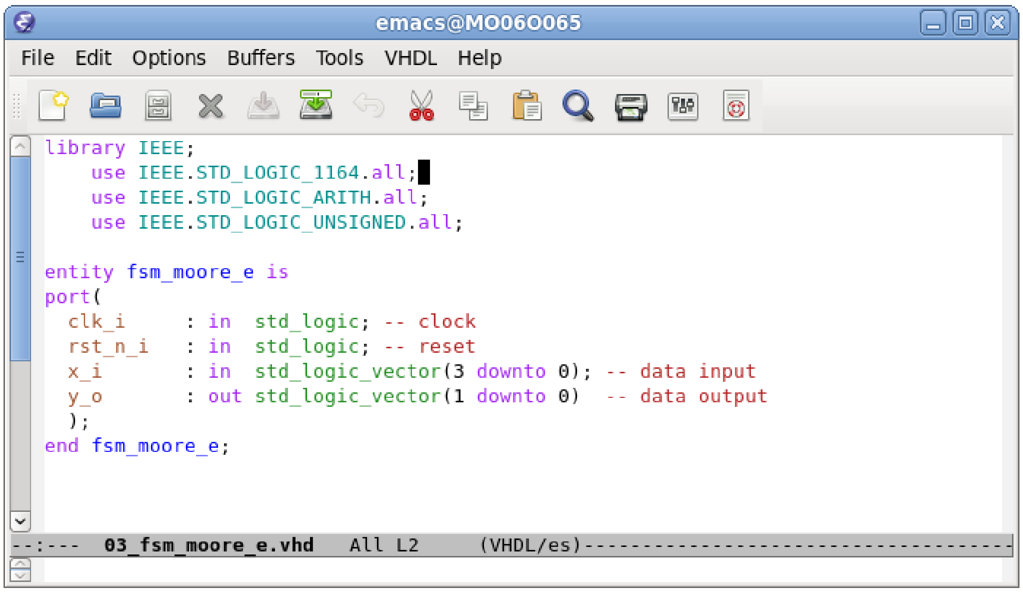
tbd



Entity:

An example of an simple Moore-FSM. The input vector is four bits wide, the output vector is two bits wide.

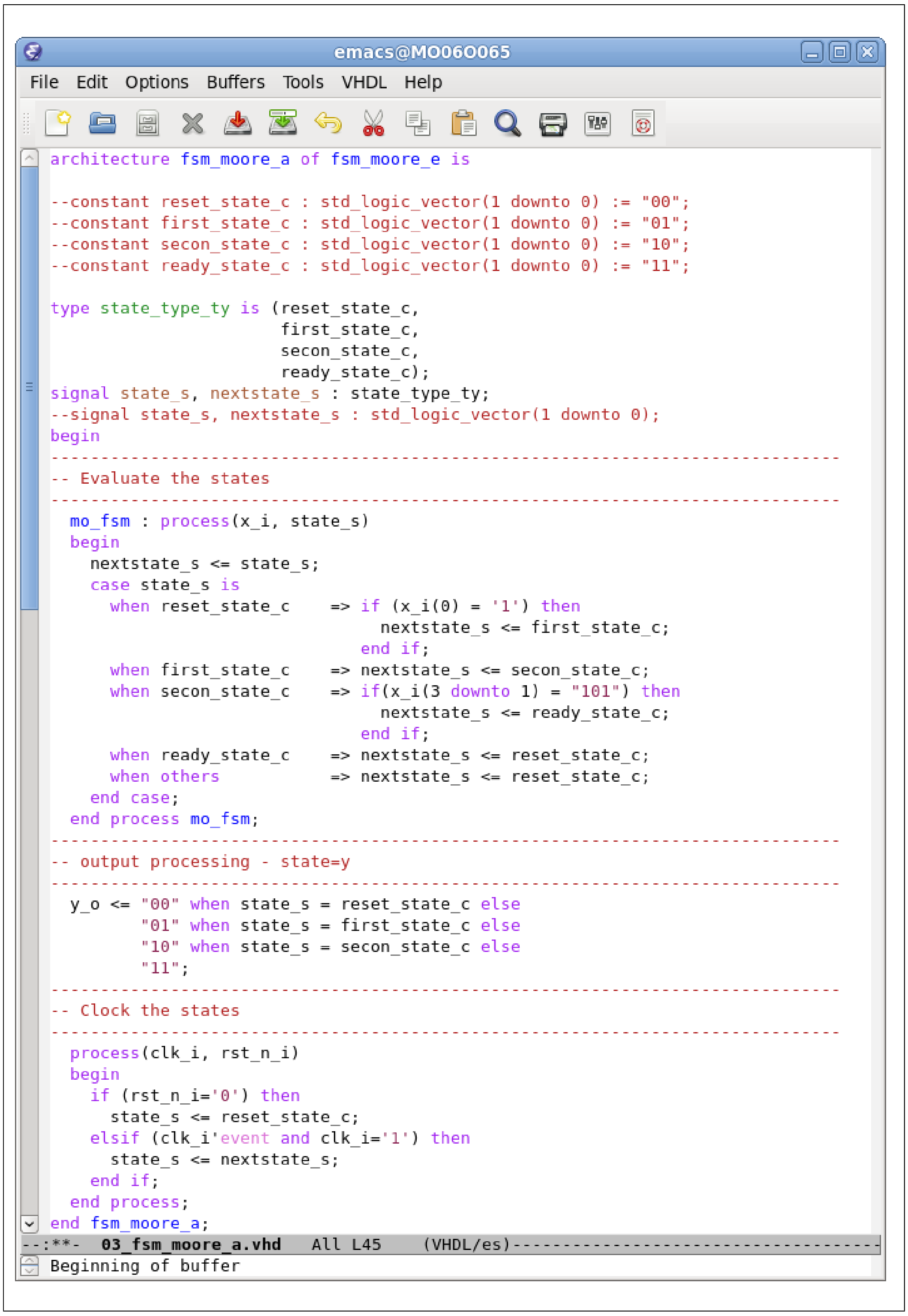
tbd



```
library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.STD_LOGIC_ARITH.all;
    use IEEE.STD_LOGIC_UNSIGNED.all;

entity fsm_moore_e is
port(
  clk_i      : in  std_logic; -- clock
  rst_n_i    : in  std_logic; -- reset
  x_i        : in  std_logic_vector(3 downto 0); -- data input
  y_o        : out std_logic_vector(1 downto 0)  -- data output
  );
end fsm_moore_e;
```

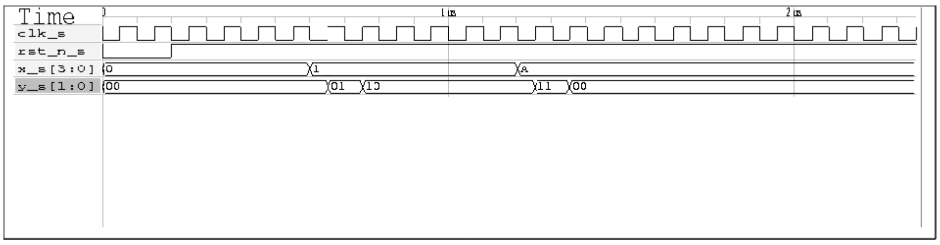
```
--:---  03_fsm_moore_e.vhd   All L2     (VHDL/es)-----------------------------------
```

Architecture:



```
architecture fsm_moore_a of fsm_moore_e is

--constant reset_state_c : std_logic_vector(1 downto 0) := "00";
--constant first_state_c : std_logic_vector(1 downto 0) := "01";
--constant secon_state_c : std_logic_vector(1 downto 0) := "10";
--constant ready_state_c : std_logic_vector(1 downto 0) := "11";

type state_type_ty is (reset_state_c,
                       first_state_c,
                       secon_state_c,
                       ready_state_c);
signal state_s, nextstate_s : state_type_ty;
--signal state_s, nextstate_s : std_logic_vector(1 downto 0);
begin
-------------------------------------------------------------------------------
-- Evaluate the states
-------------------------------------------------------------------------------
  mo_fsm : process(x_i, state_s)
  begin
    nextstate_s <= state_s;
    case state_s is
      when reset_state_c     => if (x_i(0) = '1') then
                                  nextstate_s <= first_state_c;
                                end if;
      when first_state_c     => nextstate_s <= secon_state_c;
      when secon_state_c     => if(x_i(3 downto 1) = "101") then
                                  nextstate_s <= ready_state_c;
                                end if;
      when ready_state_c     => nextstate_s <= reset_state_c;
      when others            => nextstate_s <= reset_state_c;
    end case;
  end process mo_fsm;
-------------------------------------------------------------------------------
-- output processing - state=y
-------------------------------------------------------------------------------
  y_o <= "00" when state_s = reset_state_c else
         "01" when state_s = first_state_c else
         "10" when state_s = secon_state_c else
         "11";
-------------------------------------------------------------------------------
-- Clock the states
-------------------------------------------------------------------------------
  process(clk_i, rst_n_i)
  begin
    if (rst_n_i='0') then
      state_s <= reset_state_c;
    elsif (clk_i'event and clk_i='1') then
      state_s <= nextstate_s;
    end if;
  end process;
end fsm_moore_a;
--:**-  03_fsm_moore_a.vhd    All L45     (VHDL/es)-----------------------------------
Beginning of buffer
```

Simulation:

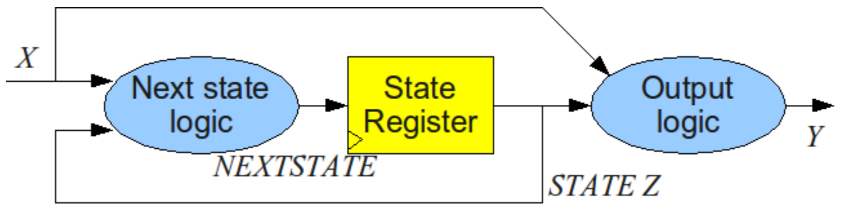


4.2.3 Mealy

Principle:

For the Mealy-FSM is the value of the output a transformed value of the state together with the direct information of the input. This is done by means of pure combinational logic.

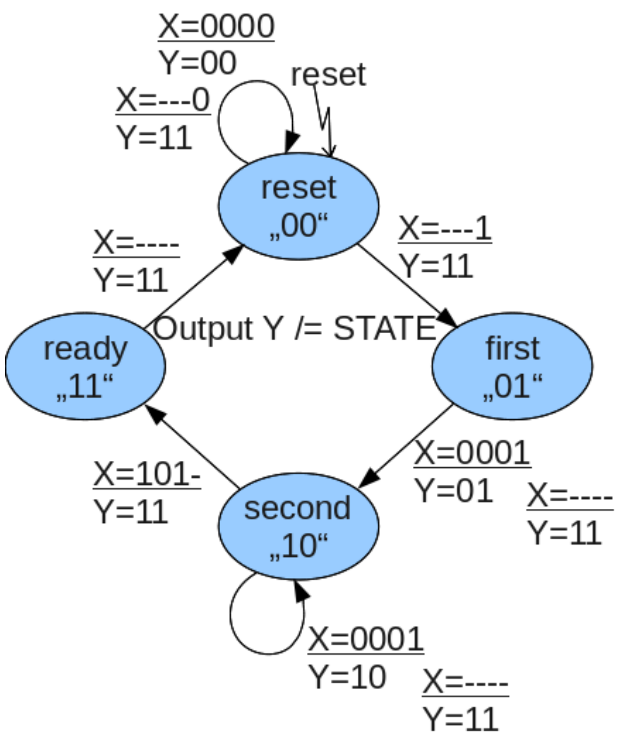
tbd.



The bubbles represents the states. They can be named (e.g., first), the coding of the states is not important here. The edges represent the state transitions dependent of the input X and the current state. The conditions for a state change will be written to the edge. The output coding will be written to the edges, also. Normally, the corresponding input will be located above a fraction bar and the desired output below. In this special case (figure below), the multiple entry at the edges mean: If we are in the reset-state and X = “0000“ is

Die Kreise repräsentieren die Zustände. Sie können formal benannt werden (z.B. first), die Codierung ist bei diesem Automaten nicht wichtig. Die Kanten repräsentieren die Zustandsübergänge, in Abhängigkeit vom Eingang X und vom aktuellen-Zustand. Die Bedingung für einen Zustandsübergang wird an die Kante geschrieben. Auch die Ausgangsbelegung (Y) wird an diese Kante geschrieben. Die Eingangsbelegung steht über einem Bruchstrich, die Ausgangsbelegung darunter. In diesem speziellen Fall (Bild), be-true, the output will be Y = “00“. But, if we are in the reset-state and X = “0000“ is not true (nevertheless, the LSB of X must be ’0’), the output will be Y = “11“.

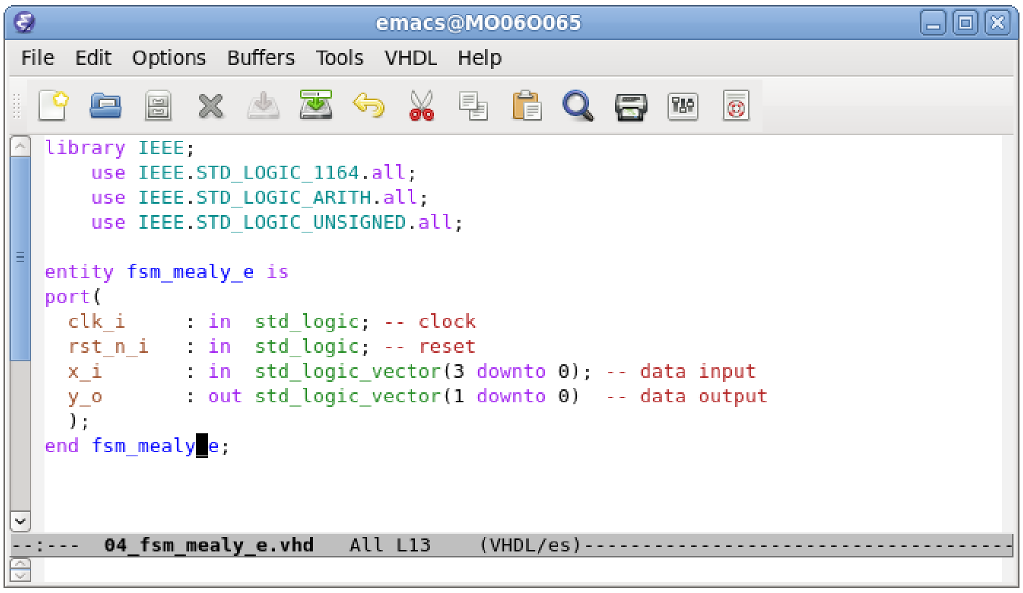
deuten die mehrfachen Einträge an den Kanten: Falls wir uns im Reset-Zustand befinden und die Eingangsbelegung gleich X = “0000“ ist, ist der Ausgang Y = “00“. Falls aber im Reset-Zustand die Eingangsbelegung nicht X = “0000“ ist, sondern irgendwie anders (das LSB von X muss allerdings ’0’ sein), ist der Ausgang Y = “11“ (die Angabe X = “ − − − 0“ ist hier natürlich nicht korrekt).



Entity:

An example of an simple Mealy-FSM. The input vector is four bits wide, the output vector is two bits wide.

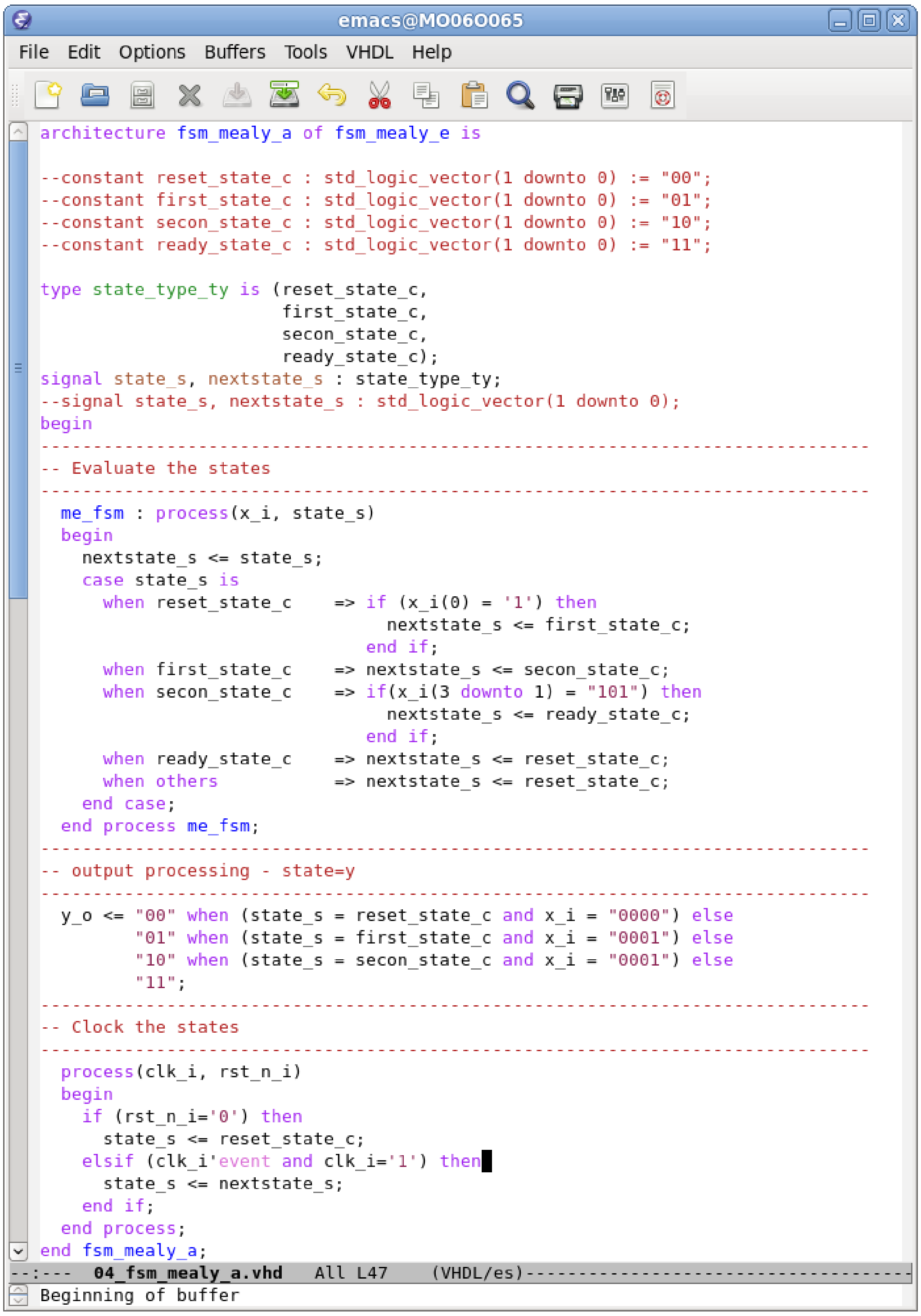
tbd



```
library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.STD_LOGIC_ARITH.all;
    use IEEE.STD_LOGIC_UNSIGNED.all;

entity fsm_mealy_e is
port(
  clk_i      : in  std_logic; -- clock
  rst_n_i    : in  std_logic; -- reset
  x_i        : in  std_logic_vector(3 downto 0); -- data input
  y_o        : out std_logic_vector(1 downto 0)  -- data output
  );
end fsm_mealy_e;
```

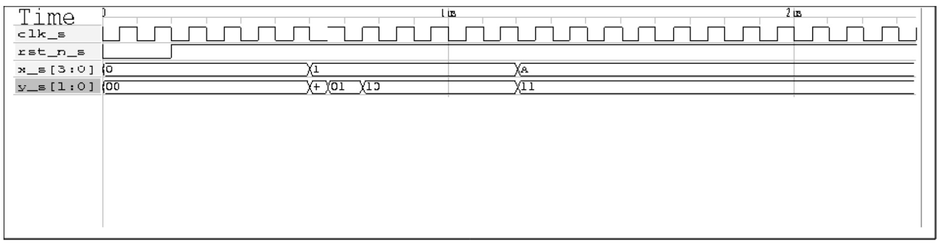
```
--:---  04_fsm_mealy_e.vhd    All L13    (VHDL/es)-----------------------------------
```

Architecture:



```vhdl
architecture fsm_mealy_a of fsm_mealy_e is

--constant reset_state_c : std_logic_vector(1 downto 0) := "00";
--constant first_state_c : std_logic_vector(1 downto 0) := "01";
--constant secon_state_c : std_logic_vector(1 downto 0) := "10";
--constant ready_state_c : std_logic_vector(1 downto 0) := "11";

type state_type_ty is (reset_state_c,
                       first_state_c,
                       secon_state_c,
                       ready_state_c);
signal state_s, nextstate_s : state_type_ty;
--signal state_s, nextstate_s : std_logic_vector(1 downto 0);
begin
-------------------------------------------------------------------------------
-- Evaluate the states
-------------------------------------------------------------------------------
  me_fsm : process(x_i, state_s)
  begin
    nextstate_s <= state_s;
    case state_s is
      when reset_state_c    => if (x_i(0) = '1') then
                                 nextstate_s <= first_state_c;
                               end if;
      when first_state_c    => nextstate_s <= secon_state_c;
      when secon_state_c    => if(x_i(3 downto 1) = "101") then
                                 nextstate_s <= ready_state_c;
                               end if;
      when ready_state_c    => nextstate_s <= reset_state_c;
      when others           => nextstate_s <= reset_state_c;
    end case;
  end process me_fsm;
-------------------------------------------------------------------------------
-- output processing - state=y
-------------------------------------------------------------------------------
  y_o <= "00" when (state_s = reset_state_c and x_i = "0000") else
         "01" when (state_s = first_state_c and x_i = "0001") else
         "10" when (state_s = secon_state_c and x_i = "0001") else
         "11";
-------------------------------------------------------------------------------
-- Clock the states
-------------------------------------------------------------------------------
  process(clk_i, rst_n_i)
  begin
    if (rst_n_i='0') then
      state_s <= reset_state_c;
    elsif (clk_i'event and clk_i='1') then
      state_s <= nextstate_s;
    end if;
  end process;
end fsm_mealy_a;
--:---  04_fsm_mealy_a.vhd   All L47    (VHDL/es)-----------------------------------
Beginning of buffer
```

Simulation:



4.3 Design for Testability - DFT

4.3.1 Reset

We are using the "asynchronous reset" design style throughout our projects. This is:

- No latches are allowed.
- Every flip-flop must have its "set" or "reset" pin connected; no flip-flops without "set" or "reset" are allowed.
- There must be a dedicated reset pin as a primary input to the chip.
- This reset input must be connected to every flip-flops "reset" or inverted to every flip-flops "set".
- This so called reset net is the ONLY input to the flip-flops "set" or "reset". No other influences are allowed.
- The reset input should not be used by other logic.

4.3.2 Clock

We are using the "fully synchronous clock" design style throughout our projects.

Requirements:

- Use a single clock for all flip-flops clock input.
- If you need a slower clock, generate a "pulse stealing" logic and use the slower clock as enable input to the flip-flops.

- If you realy need multiple clocks, build a clean clock divider (attachment) and write a clear clock concept. You need to know which clock is needed by which flip-flop. Everey logic with different clock speeds must be implemented as seperate design units. If you need to route signals between different clock domains, you have to use synchronizer cells (attachment).

Kapitel 5

Attachments

5.1 Attachment 1

ASIC Design Guidelines

Application Specific IC (ASIC)

Application Note

Introduction

The Atmel ASIC Design Guidelines constitute a general set of recommendations intended for use by designers when preparing circuits for fabrication by Atmel. The guidelines are independent of any particular CAD tool or silicon process. They are applicable to Gate Arrays, Cell-Based ASICs (CBICs) and full-custom designs. Although they do not give specific coding recommendations, they apply equally to designs captured in Verilog or VHDL as to designs captured as schematics.

These guidelines do not cover general principles of ASIC design; rather they highlight specific design practices which are regarded as unsafe, and which can lead to devices which are difficult to test, and whose correct operation cannot be guaranteed under all circumstances. For each unsafe, and therefore non-recommended design practice, an alternative safe, and therefore recommended practice is proposed.

The current paradigm shift towards system level integration (SLI), incorporating multiple complex functional blocks and a variety of memories on a single circuit, gives rise to a new set of design requirements at integration level. These design guidelines do not fully address these issues yet. The recommendations are principally aimed at the design of the blocks and memory interfaces which are to be integrated into the system-on-chip. However, the guidelines given here are fully consistent with the requirements of system level integration. Respect for these guidelines will significantly ease the integration effort, and ensure that the individual blocks are easily reusable in other systems.

These design guidelines have been drawn up in the light of experience with large numbers of ASIC designs over more than a decade.

The Atmel ASIC Design Guidelines have a particular significance during the signoff of each design prior to submission for fabrication:

*Atmel customers must sign off a design to confirm that it complies with **all** the recommendations in the Atmel ASIC Design Guidelines. For each case of non-compliance, the case must be discussed with the ASIC Support Center, and if necessary a formal Authorization must be obtained.*

Rev. 1205A–12/99

Synchronous Circuits

Experience has shown that the safest methodology for time-domain control of an ASIC is **synchronous design**.

A synchronous circuit is one in which:

- all data storage elements are clocked, and in normal operation change state **only** in response to the clock signal
- the same active edge of a single clock signal is applied at **precisely the same point in time** at every clocked cell in the device.

Examples of circuit elements which contradict these principles are given below, and methods of achieving synchronous design are given in the four sections which follow.

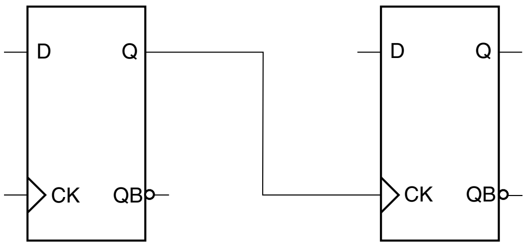
Non-recommended Circuits

Circuits which violate the principles of synchronous design include the following elements:

Flip-flop driving clock input of another flip-flop

The clock input of the second flip-flop is skewed by the clock-to-q delay of the first flip-flop, and is not activated on every clock edge. See Figure 1.

Figure 1. Flip-flop driving clock input of another flip-flop

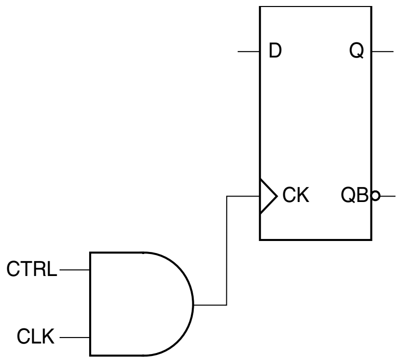


An example of a circuit containing this element is a ripple counter.

Gated clock line

Gating in a clock line (Figure 2) causes clock skew and can introduce spikes which trigger the flip-flop. This is particularly the case when there is a multiplexer in the clock line.

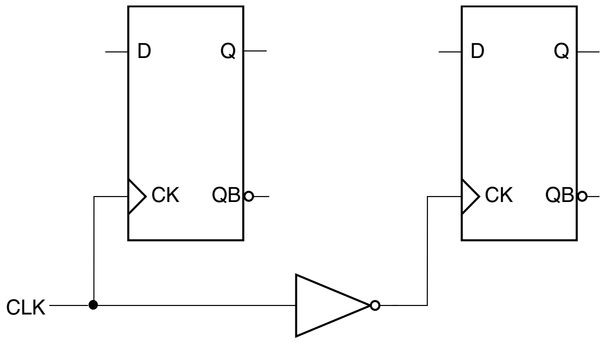
Figure 2. Gated clock line



Double-edged clocking

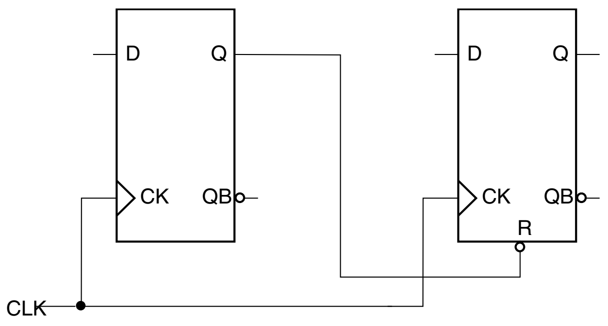
The two flip-flops are clocked on opposite edges of the clock signal (Figure 3). This makes synchronous resetting and test methodologies such as scan-path insertion difficult, and causes difficulties in determining critical signal paths.

Figure 3. Double-edged clocking



Flip-flop driving asynchronous reset of another flip-flop.
In Figure 4, the second flip-flop can change state at a time other than the active clock edge, violating the principle of synchronous design. In addition, this circuit contains a potential race condition between the clock and reset of the second flip-flop.

Figure 4. Flip-flop driving asynchronous reset of another flip-flop



An example of a circuit containing this element is an asynchronously reset counter.

Recommended Circuits

Methods of achieving the requirements of synchronous design, and avoiding the non-recommended situations described above are dealt with in subsequent sections, as follows:

- Synchronous clocking by means of clock buffering: See “Clock Buffering” on page 4.
- Flip-flop driving clock signal of another flip-flop: See “Gated Clocks” on page 10.
- Gated clocks: See “Gated Clocks” on page 10.
- Double-edged clocking: See “Double-edged Clocking” on page 11.
- System clock generation: See “Clock Generation and Overall Circuit Control” on page 12.
- Asynchronous resets: See “Asynchronous Resets” on page 13.

Clock Buffering

To achieve the requirement of a simultaneous application of a single clock signal at all storage elements in a design, and avoid problems due to fanout, a clock buffering scheme needs to be implemented consistently throughout a circuit. This is often done automatically as part of placement and routing; if not, the principles described in this section should be followed.

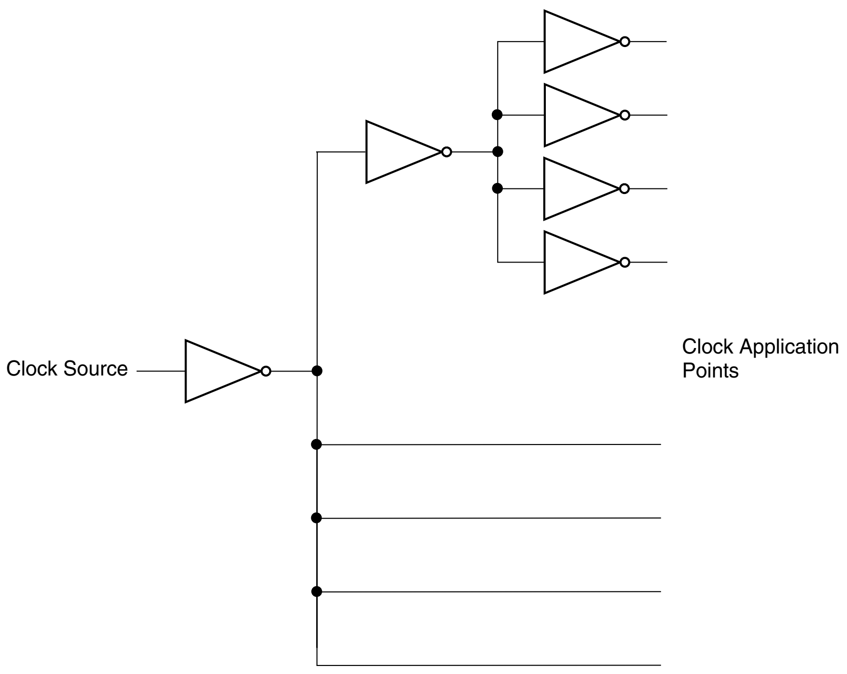
Non-recommended Circuits

Circuits which violate the principles of consistent clock buffering include the following elements:

Unequal depth of clock buffering

The depth of clock buffering differs between different clock application points, causing clock skew. See Figure 5.

Figure 5. Unequal depth of clock buffering



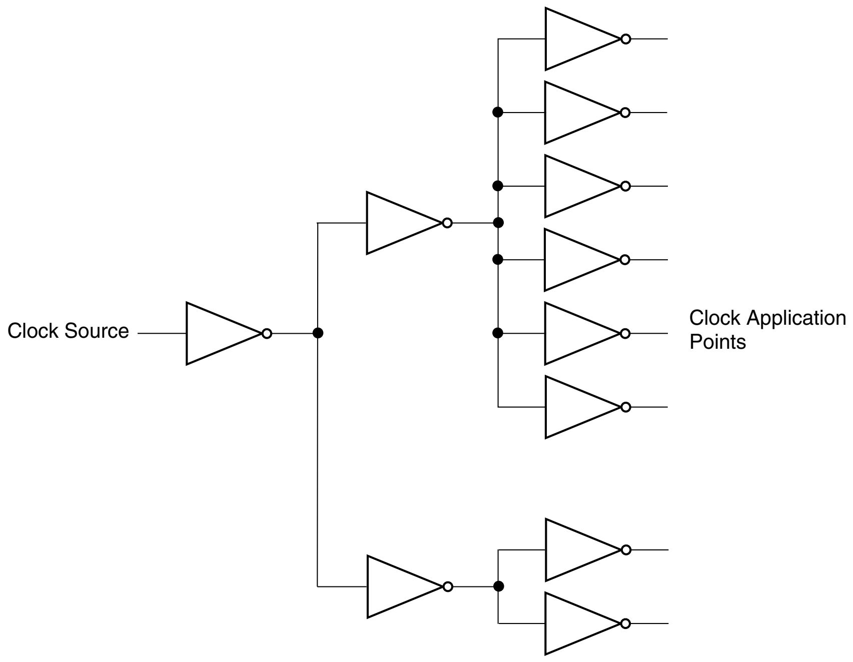
Unbalanced fanout on clock buffers

As shown in Figure 6, the difference between the fanouts at the two intermediate buffers gives rise to different load-dependent delays, causing clock skew.

Excessive clock fanout

Excessive clock fanout leads to slow clock edges, which can cause a number of problems, including an increased risk of metastability in flip-flops which capture external asynchronous signals.

Figure 6. Unbalanced fanout on clock buffers



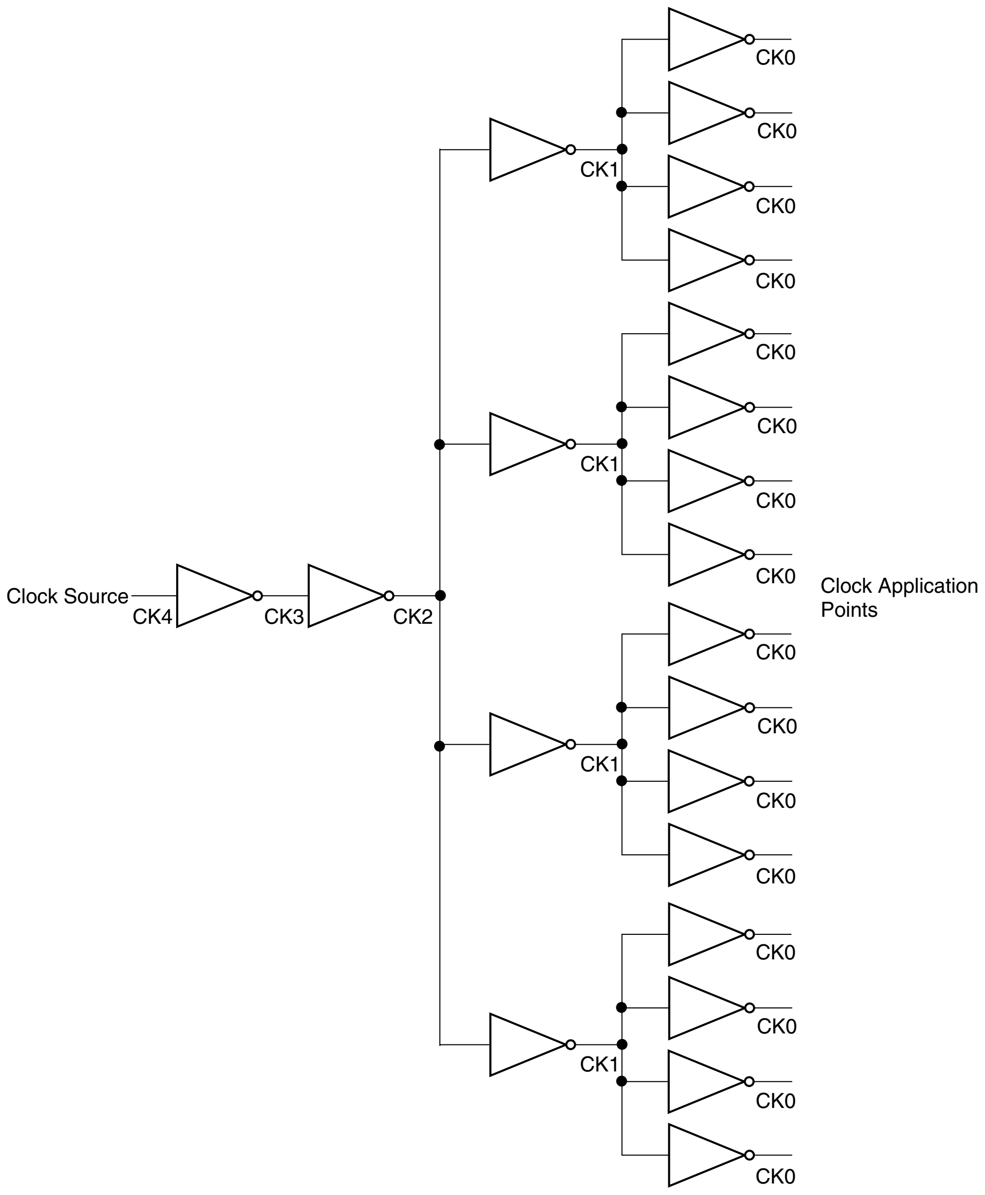
Recommended Circuits

The recommended clock buffering scheme is balanced tree buffering, which must satisfy the following conditions:

1. The same depth of buffering to all clocked cells. (A suggestion is to use the naming convention: ck0 at application point, then ck1, ck2, ... ckn, and join equivalent levels up the circuit hierarchy. Note that n must be even to retain clock polarity.) See Figure 7.
2. The same fanout on all buffers. This must be checked after placement and routing, to ensure that tracking capacitances do not unbalance the fanout.
3. Lightly loaded buffers to keep clock edges sharp (max 50% of max relative fanout). An alternative is to use a combination of geometric and tree buffering, as illustrated in Figure 8.

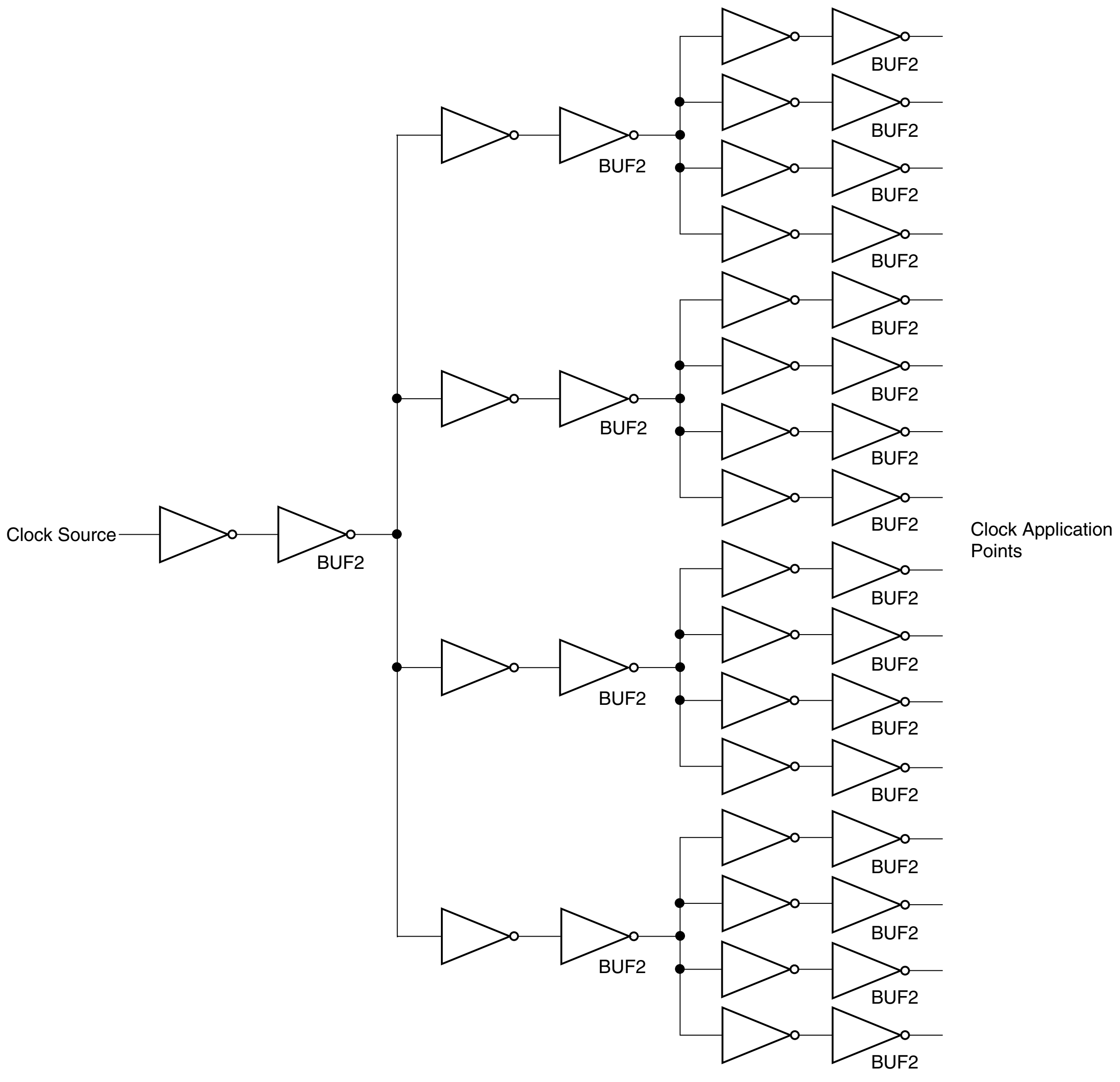
Balanced clock tree buffering

Figure 7. Balanced clock tree buffering



Combined geometric/tree buffering
By using an intermediate buffer of a suitable drive strength at each clock fanout point, the relative fanout at each buffer is reduced, and clock edges remain sharp.

Figure 8. Combined geometric/tree buffering



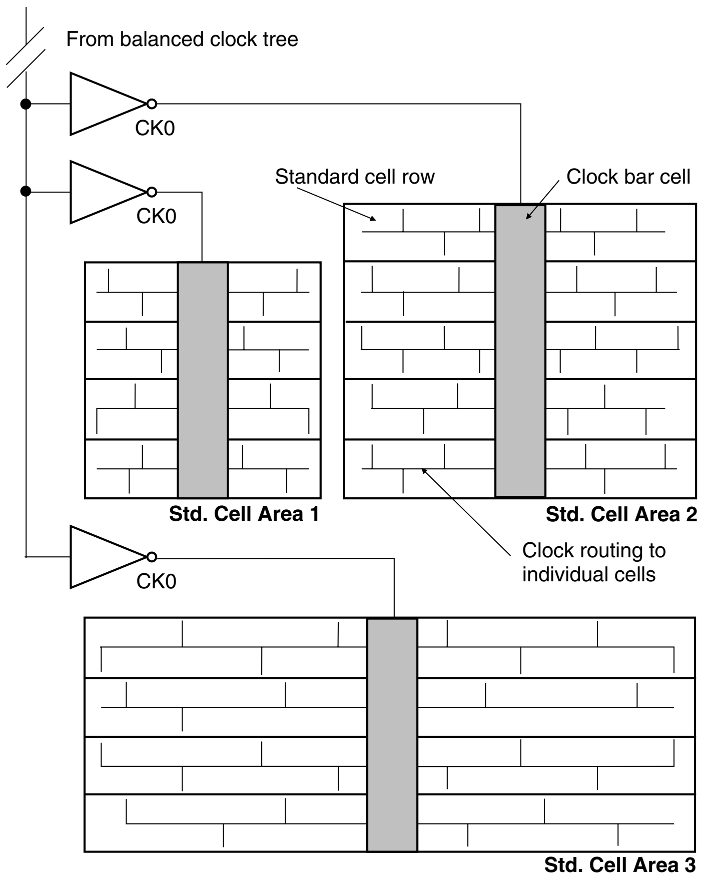
Clock Bar Cells

The use of clock bar cells for clock distribution from within a standard cell area is recommended (during placement and routing, if they are available), as shown in Figure 9. A single Clock bar cell, positioned correctly in the centre of the standard cell area, can provide a balanced clock net distribution. This runs a vertical clock trunk through the middle of the cell area, allowing clock net branches to feed cells on either side of the trunk. This method reduces the risk of clock skew by halving the effective clock path length along a row of cells, compared with a clock supplied from one end of the cell row. It also guides the router to prevent a long clock path being threaded through the standard cells, and prevents clock net looping.

It is recommended to use only one clock bar cell per standard cell area (otherwise clock looping may occur). By using clock bar cells, there will be a balanced clock net distribution within each standard cell area.

Balanced clock routing using clock bar cells

Figure 9. Balanced clock routing using clock bar cells

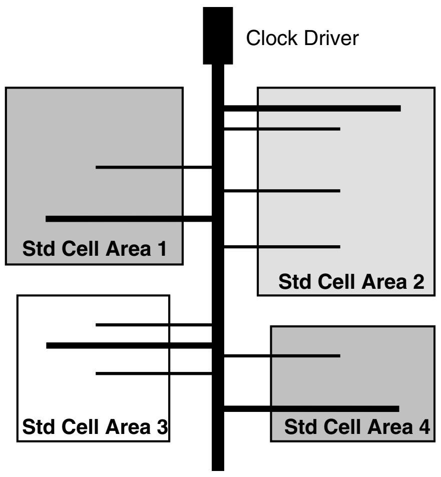


Clock Guidance

The use of clock guidance is recommended if available, before starting place and route. A central clock trunk should be run between the standard cell areas with branches feeding off either side into the standard cell areas themselves, as shown in Figure 10. A bad example of Clock guidance is given in Figure 11, highlighting the risk of clock skew.

Good example of clock guidance

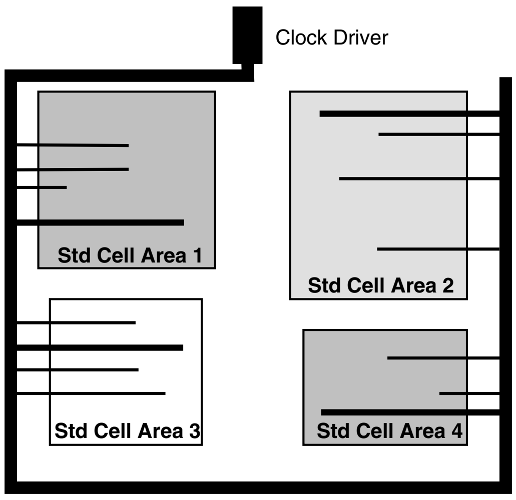
Figure 10. Good clock guidance for routing



It is important to have an even number of rows (in the standard cell areas), because an odd number of rows can force the place and route software to create loops on the clock net.

Bad example of clock guidance

Figure 11. Bad clock guidance for routing



Clock Compilers

If Clock compilers are available, they help to maintain a a balanced clock network, but should be used with care. The clock compiler automatically adjusts the clock buffering to make the equivalent delays for each cell area the same as the longest delay. This means additional buffer cells may be added both outside and inside the standard cell areas, and the cell areas themselves may be split.

Gated Clocks

A seemingly obvious way of controlling the operation of a flip-flop is to gate the clock signal with a control signal, or to multiplex two alternative clocks into its clock input. *This practice is dangerous on two counts:*

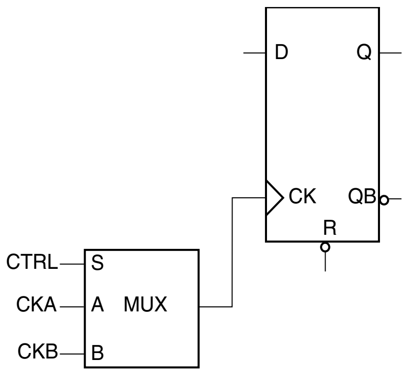
- A glitch on the gate output can cause a clock edge.
- Gating in the clock line introduces clock skew.

Non-recommended Circuits

A **particularly unsafe** circuit element is shown in Figure 12.

Multiplexer on clock line

Figure 12. Multiplexer on clock line



Toggling the multiplexer control signal inevitably causes a glitch on the ck input to the flip-flop, which may cause it to capture invalid data.

Recommended Circuits

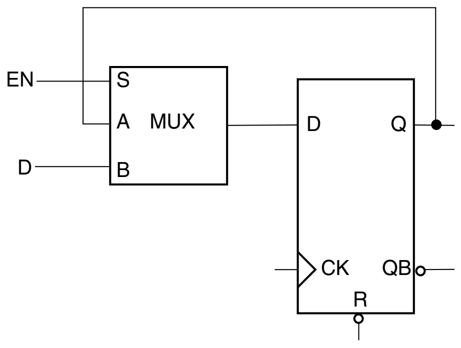
Two circuit elements which are recommended for use in synchronous designs are illustrated here. They are the enabled (E-type) flip-flop and the toggle (T-type) flip-flop. They remove the need for gated clocks, or for using the output from one flip-flop as the input to another.

Enabled (E-type) flip-flop

The enable signal (the multiplexer select line) controls the input of data to the flip-flop. If enable is low, the existing value of q is re-input at the next clock cycle. If enable is high, a new data value is clocked in. See Figure 13.

Note: A version of the E-type flip-flop can be constructed with a synchronous reset. A recommended way of constructing an E-type flip-flop is using AOI logic. See “Design for Speed” on page 31.

Figure 13. E-type flip-flop

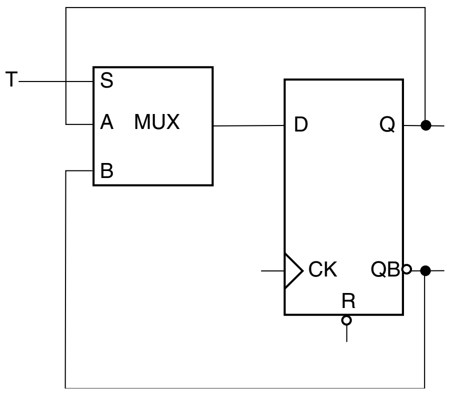


Toggle (T-type) flip-flop

The toggle flip-flop is the basic element in synchronous counters. The toggle signal (the multiplexer select line) controls state of the flip-flop. If toggle is low, the flip-flop retains its existing value at the next clock edge; if toggle is high, it takes the opposite value. See Figure 14.

Note: A version of the T-type flip-flop can be constructed with a synchronous reset. A recommended way of constructing a T-type flip-flop is using AOI logic. See “Design for Speed” on page 31.

Figure 14. T-type flip-flop



Double-edged Clocking

In an attempt to increase data throughput rates, use is sometimes made of both the rising and the falling clock edge for clocked elements. This practice, however, violates the principles of synchronous design given in "Synchronous Circuits" on page 2, and causes a number of problems, in particular:

- An asymmetrical clock duty cycle can cause setup and hold violations.
- It is difficult to determine critical signal paths.
- Test methodologies such as scan-path insertion are difficult, as they rely on all flip-flops being activated on the same clock edge. If scan insertion is required in a circuit with double-edged clocking, multiplexers must be inserted in the clock lines to change to single-edged clocking in test mode. *See, however, the warning in "Multiplexer on clock line" on page 10.*

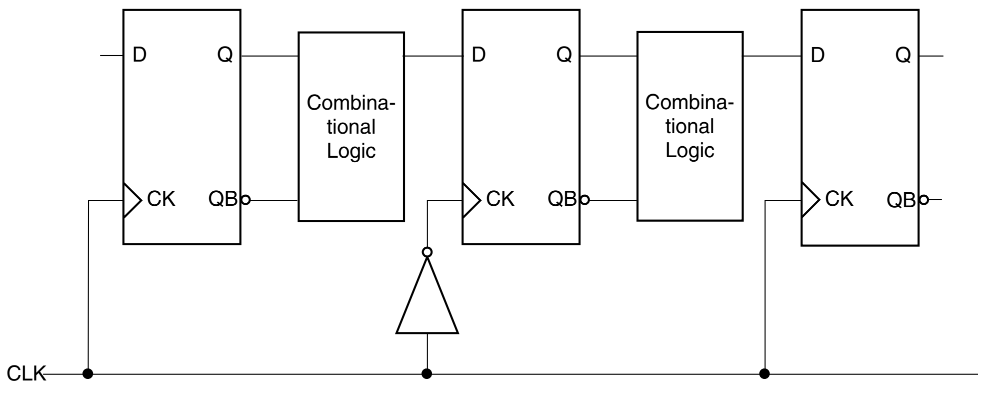
The recommended alternative is to use a single-edged clocking scheme with a higher clock frequency.

A general principle of synchronous circuit design is that the minimum time resolution available within the circuit is the duration of one complete clock cycle.

Non-recommended Circuit

Pipelined logic with double-edged clocking
In a circuit as shown in Figure 15, an asymmetrical clock duty cycle could cause setup and hold time violations, and a scan-path cannot easily be threaded through the flip-flops.

Figure 15. Pipelined logic with double-edged clocking

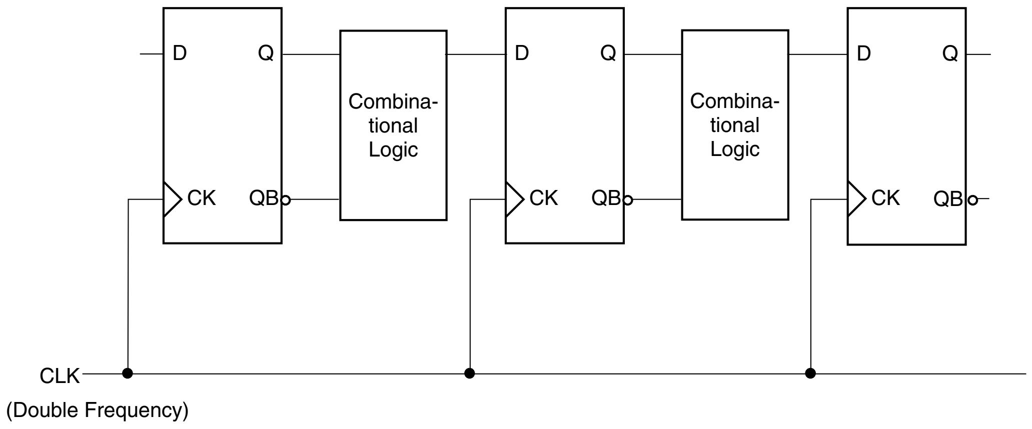


Recommended Circuit

Pipelined logic with single-edged clocking
The equivalent synchronous circuit (Figure 16) requires a clock frequency of double the previous version.

It is also recommended that enabled logic is used where required. See "Gated Clocks" on page 10.

Figure 16. Pipelined logic with single-edged clocking



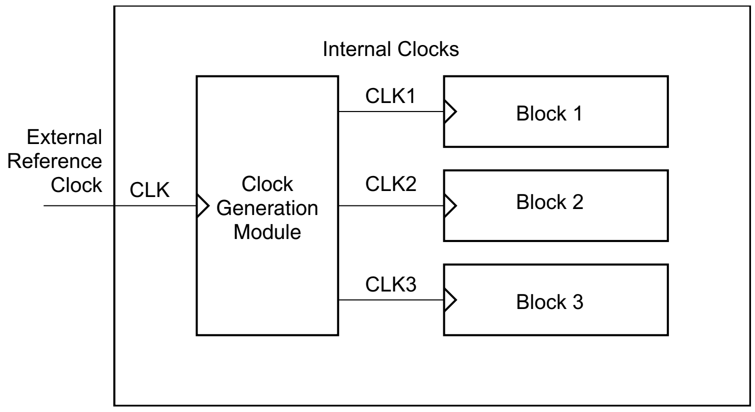
Clock Generation and Overall Circuit Control

If clocks of different speeds are required by different blocks, or the internal clock is required at a speed faster or slower than the externally available clock, it is recommended that a single clock generation block is constructed at the top level of a circuit. This produces the internal clocks required by all the functional blocks in the circuit. See Figure 17.

Communication between the internal blocks is achieved by the same principles as for asynchronous external inputs. See "Asynchronous Inputs" on page 17.

Recommended Circuit

Figure 17. Clock generation module at circuit top level



Generating higher- or lower-speed internal clocks

If the externally available clock signal is of a higher frequency than that required for an internal clock, a synchronous binary counter (made from T-type flip-flops) is recommended to perform the required clock division. Latching of data conditionally, or at a lower frequency than this internal clock is achieved by the use of individual E-type flip-flops for data storage.

Alternatively, a PLL can be used to produce a higher-speed internal clock than the external reference clock.

Asynchronous Resets

The general recommendations for dealing with resets within an ASIC are as follows:

1. The circuit must be brought to a known state, both within test and in operation, within a stated and agreed number of clock cycles. The known state is generally achieved by means of a reset mechanism.
2. If an asynchronous reset is required, use a *single global asynchronous reset* driven by an external input. A tree buffering scheme similar to that for clock distribution may be required to ensure a sharp edge on the reset signal. The benefit of a reset of this nature is that it places the entire circuit in a known state in response to a change on a single input signal, with no clock cycles required for the known state to propagate.
3. If a power-on reset (POR) pad is used, the circuit must contain another global reset for test purposes.
4. If a local reset is required, use a synchronous reset.

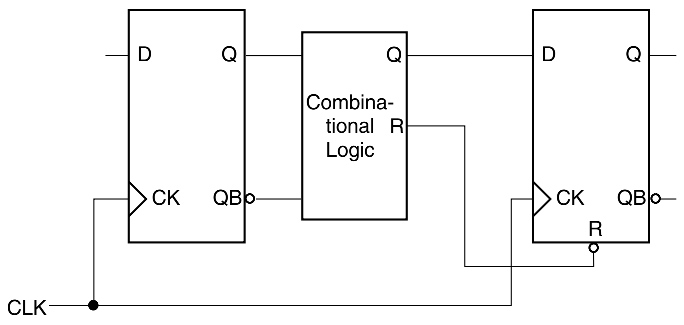
Non-recommended Circuit

A local asynchronous reset such as on a counter causes a change of state in a storage element which is not triggered by the active clock edge, and therefore violates the principles of synchronous design given in "Synchronous Circuits" on page 2.

Local asynchronous reset of a flip-flop

In Figure 18, the local asynchronous reset causes a change of state on the second flip-flop which is not synchronized with the active clock edge.

Figure 18. Flip-flop driving asynchronous reset of another flip-flop

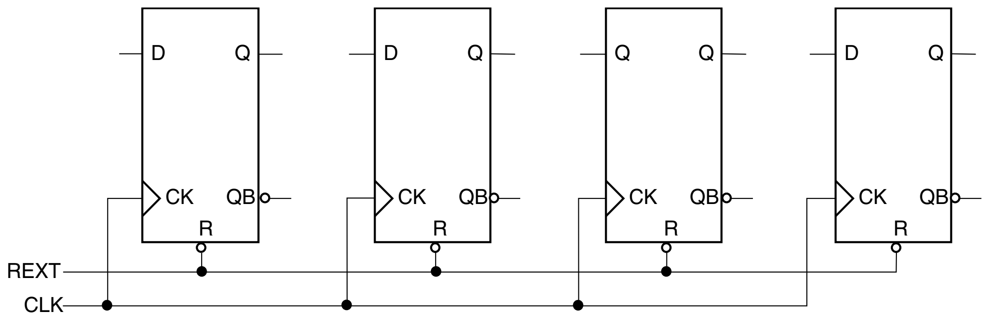


Recommended Circuits

The circuits given below overcome the problems discussed in the previous section.

A general recommendation is, if necessary, to organize resets into a hierarchy, from global (which may be asynchronous) to local (which must be synchronous).

Figure 19. Global asynchronous reset of all flip-flops



Global asynchronous reset of all flip-flops

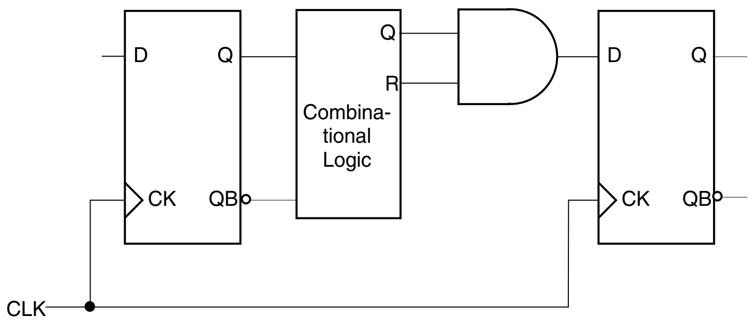
In Figure 19, a single external reset signal (rext) is connected to all flip-flops. The buffering which may be required is not shown.

Local synchronous reset of a flip-flop

In Figure 20, the (active low) reset signal (r) is gated with the d-input of the second flip-flop, making it synchronous.

The second flip-flop changes state only on an active clock edge.

Figure 20. Flip-flop driving a synchronous reset of another flip-flop



Shift Registers

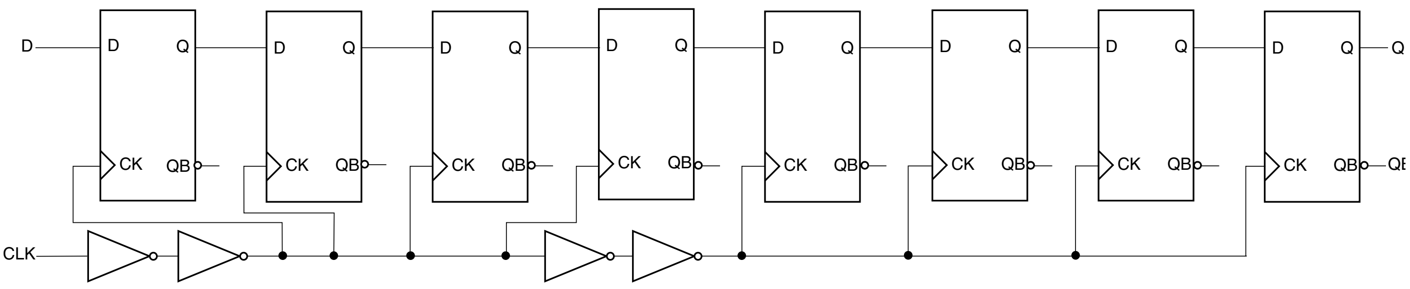
Shift registers are particularly intolerant of clock skew. A problem which occurs in their design is that long shift registers may require internal clock buffering. If not properly designed, this buffering can cause clock skew within the shift register, and interfacing problems between the shift register and the rest of the circuit.

Non-recommended Circuits

Not recommended is a chain of clock buffers within shift register, in either the forward or the reverse direction. These cases are illustrated below.

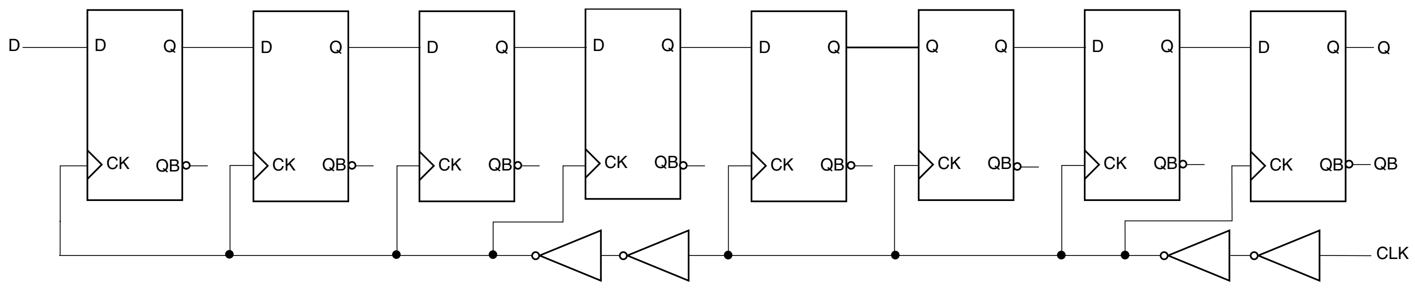
Shift register with forward chain of clock buffers
The problem with a forward chain of clock buffers (Figure 21) is that internal clock skew can cause data fallthrough (where one stage of the shift register is skipped).

Figure 21. Shift register with forward chain of clock buffers.



Shift register with reverse chain of clock buffers
As shown in Figure 22 below, the problem with a reverse chain of clock buffers is the timing interface between the first D-type and the input data received from the rest of the circuit.

Figure 22. Shift register with reverse chain of clock buffers.



Recommended Circuits

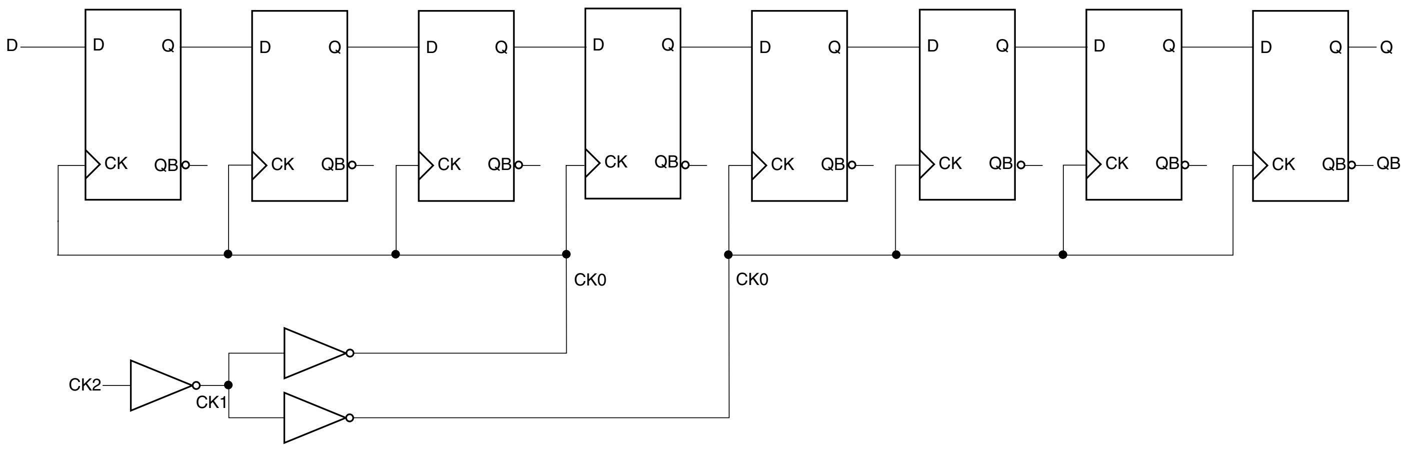
There are two recommended ways of constructing the clock buffering scheme within a shift register:

1. Use balanced clock tree buffering as in the rest of the circuit. See “Clock Buffering” on page 4 and Figure 23 below. As an additional safety feature, buffering can be introduced in the data lines between each flip-flop.
2. Use a FIFO.

Shift register with balanced clock tree buffering

As shown in Figure 23, the clock tree within the shift register must be balanced (in terms of relative fanout) with the same levels of clock tree in other parts of the circuit. Note the naming convention for clock signals which facilitates this.

Figure 23. Shift register with balanced tree of clock buffers.



Asynchronous Inputs

A problem arises at the interface between a synchronous circuit and an external asynchronous input. At the flip-flop which captures the asynchronous input, there is a probability of **metastability** occurring. This section suggests some circuits which capture an external asynchronous input with a minimal risk of metastability.

Note: For large designs, inter-block communication is similar to external asynchronous interfacing.

Non-recommended Circuits

Not recommended is any circuit using a complicated feedback loop to capture an asynchronous input. The function of such circuits is obscure, and they run the risk of creating more problems than they solve. They are also very sensitive to noise, and their function can be altered by placement and routing delays.

Recommended Circuits

There are two recommended approaches to the problem of capturing an asynchronous input signal:

1. Two (or more) D-type registers in series to reduce the probability of metastability (Figure 24).
2. Use an asynchronous handshake circuit (Figure 25).

In all cases, the asynchronous event is a rising edge on the d (external) input to the first flip-flop. The pulse width of this signal is indeterminate, but is at least one clock cycle. The asynchronous event may occur simultaneously with a rising clock edge.

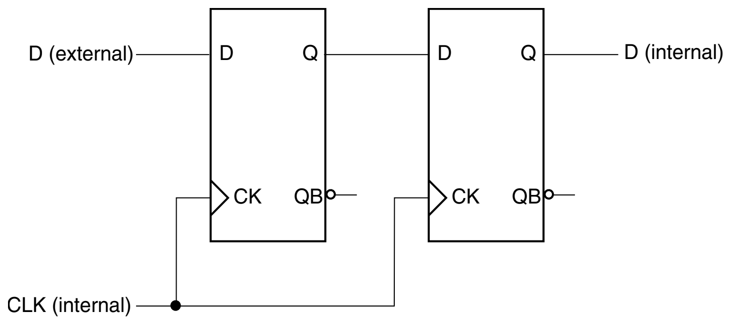
A general point which applies to all situations where metastability is possible is as follows:

- The rise and fall times of both the clock and data signals are significant: *fast edges reduce the probability of metastability.*

Two D-type flip-flops in series to capture an asynchronous input

If the first flip-flop goes into a metastable state, the probability that it will still be in that state at the next rising clock edge is low. Should this, however, occur, the metastable state is propagated to the d (internal) output and into the rest of the circuit. The probability of this situation is reduced by additional flip-flops in series.

Figure 24. Two D-type flip-flops in series to capture an asynchronous input



The common characteristics of circuits of this nature are as follows:

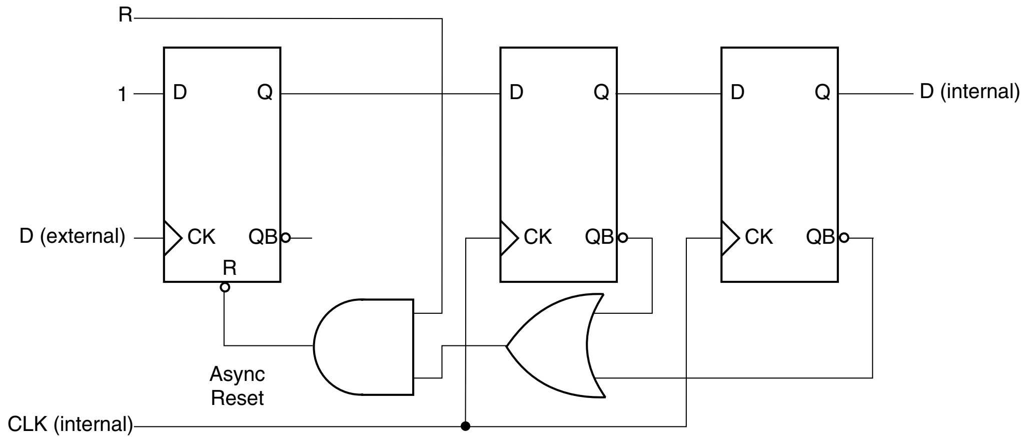
- In order for the d (external) rising edge to cause a rising edge on the d (internal) output, there must be at least one clock cycle between asynchronous inputs during which d (external) is low. This reduces the maximum frequency for the recognition of external events to half that of the internal clock frequency.
- If the flip-flop which receives the asynchronous d (external) rising edge settles (after a period of metastability) into the state with q = 0, the external input is lost unless it persists beyond the next rising clock edge.
- Metastability can be caused by a rising or a falling edge on the d (external) input.

Asynchronous handshake circuit

A circuit of the type shown in Figure 25 can be used to detect an asynchronous event: a rising edge on d (external). *These events must occur at longer time intervals than two clock cycles.*

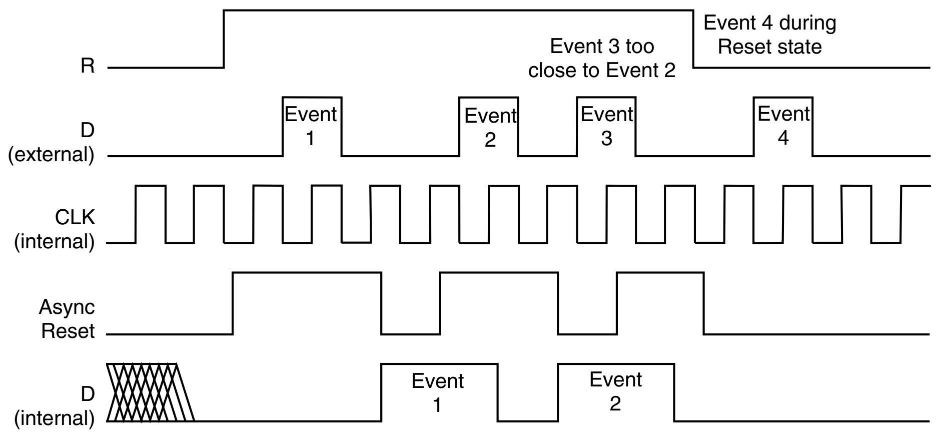
The external event (d) drives the clock input of the first flip-flop. This is the **only** flip-flop in the circuit which has a clock input not driven by the system clock (clk). The d-input to this flip-flop is tied to logic 1. It has an asynchronous input driven from the system reset (r) and from the qb outputs of the second and third flip-flops.

Figure 25. Asynchronous handshake circuit



In reset mode (r = 0), the first flip-flop is reset asynchronously. This state takes two clock cycles to propagate to the d (internal) signal. In active mode (r = 1), a rising edge on d (external) immediately drives the q-output from the first flip-flop high. After one rising clock edge, this propagates to the q-output from the second flip-flop, and after a second clock edge, to the d (internal) output from the third flip-flop. At this time, the qb outputs from the second and third flip-flops are both low. This logic level propagates through the OR and the AND gates in the feedback loop, forcing a reset on the first flip-flop, which is now ready to receive another rising edge on the d (external) input. The circuit function is illustrated in Figure 26.

Figure 26. Operation of asynchronous handshake circuit



The d (internal) signal can be used as an acknowledge signal to the external system which is supplying the d (external) inputs.

The risk of metastability is at the second flip-flop: caused by simultaneous rising edges on the (asynchronous) q-output from the first flip-flop and the system clock. If this occurs, there are three possibilities:

- The second flip-flop settles into a q = 1 state before the next rising clock edge. This is then clocked by the third flip-flop, and the circuit functions normally.

- The second flip-flop settles into a q = 0 state before the next rising clock edge. This causes no change to the third flip-flop, and the feedback loop to the first flip-flop is unaffected. Therefore the first flip-flop retains its q = 1 value to be clocked by the second flip-flop on the next rising clock edge. The effect of this is to delay the recognition of the asynchronous event by one clock cycle.
- The metastable state persists until the next rising clock edge. In this case there is a possibility of the third flip-flop entering a metastable state as well. However, the probability of a metastable state persisting for an entire clock cycle, and forcing the third flip-flop into a similar state, is extremely low. This risk can be further reduced by inserting additional flip-flops, at the expense of an additional clock cycle as the minimum delay between recognized inputs.

Note: Metastability can only be caused by a rising edge of the d (external) input, whereas in the previous two circuits it can be caused by either edge. The only restriction on pulse width for the asynchronous handshake circuit is the minimum pulse width of the first flip-flop.

This circuit will enter an unknown state if it receives simultaneous rising edges on the d (external) and reset (r) signals.

Delay Lines and Monostables

There is often an apparent requirement to create a short pulse within a circuit, of duration less than a clock cycle. This generally requires the use of a **delay line** within a monostable element, as shown in Figure 29 below. A multi-vibrator circuit (Figure 31) is based on a similar principle. More generally, asynchronous circuits often rely on delay lines for their correct operation, for example in an attempt to overcome race conditions.

The practice of delay-line dependent circuits is not recommended, as the actual timing of the delay line is difficult to predict, and is highly sensitive to temperature and process spread.

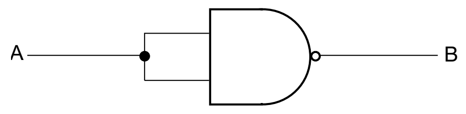
In particular, due to simulation model constraints it is not permitted to short two inputs of a logic gate to the same source signal (Figure 27). The problem is that the gate delays are characterized with one signal changing. For a NAND3 driven to a one (Figure 28), if two signals change simultaneously there are two transistors pulling the output high, instead of one. This will reduce the delay time by about 50% compared to the simulation model.

Non-recommended Circuits

In general, any circuit which relies on delays for its operation is not recommended. All gates in series which are not used for buffering must be considered as delay lines. Five specific examples are given below:

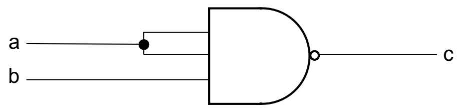
NAND2 gate used as delay element

Figure 27. NAND2 gate used as a delay element



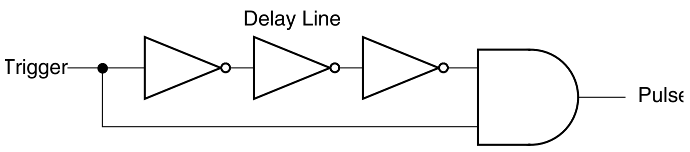
NAND3 gate with two inputs connected together

Figure 28. NAND3 gate with two inputs connected together



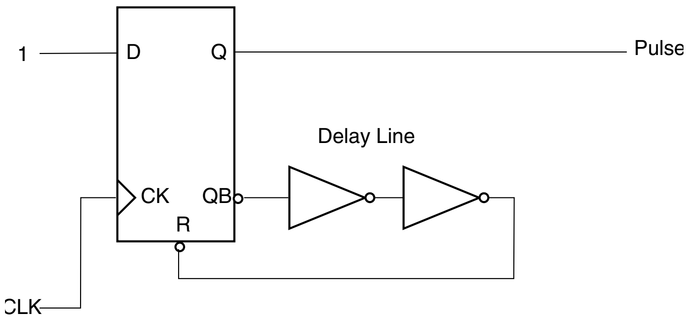
Monostable pulse generator

Figure 29. Monostable pulse generator



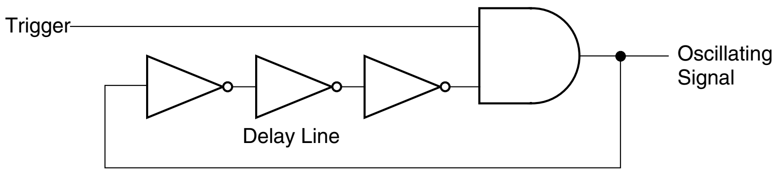
Pulse generator using a flip-flop

Figure 30. Pulse generator using a flip-flop



Multivibrator

Figure 31. Multivibrator



Care must be taken not to create inadvertently an equivalent circuit to this one, for example, in the (synchronous) reset loop of a counter.

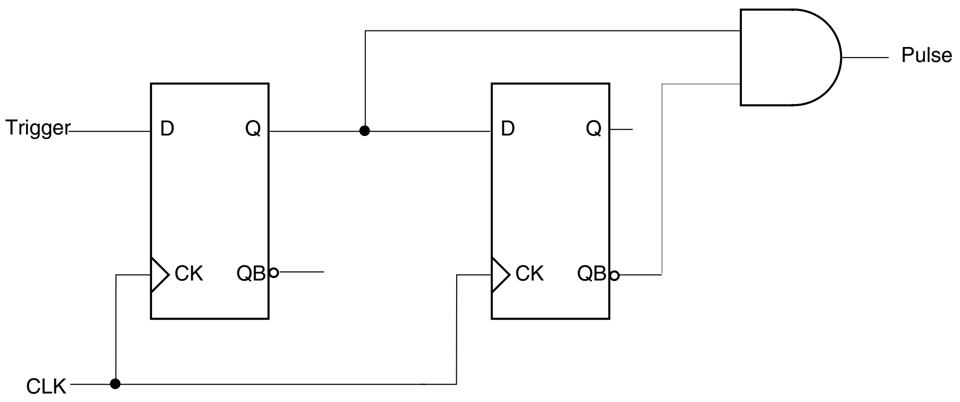
Recommended Circuit

If at all possible, delay-line dependent circuits should be avoided completely. The safe solution to the problem is as follows:

1. Use a higher clock speed. *The best time resolution available in a circuit is the width of one clock cycle.*
2. Use a synchronous pulse generator, as illustrated in Figure 32 below.

Synchronous pulse generator

Figure 32. Synchronous pulse generator



Authorization

Delay-dependent circuitry is only accepted by Atmel when it is accompanied by post-layout (H)Spice simulation results of the relevant circuit elements.

Bistable Elements

Data storage elements should not be created by cross-coupling NAND or NOR gates to form bistable elements. There are a number of problems associated with bistable elements of this nature, including asynchronous operation, unknown output states for certain input combinations, sensitivity to input spikes, and the lack of timing constraint checking in simulation.

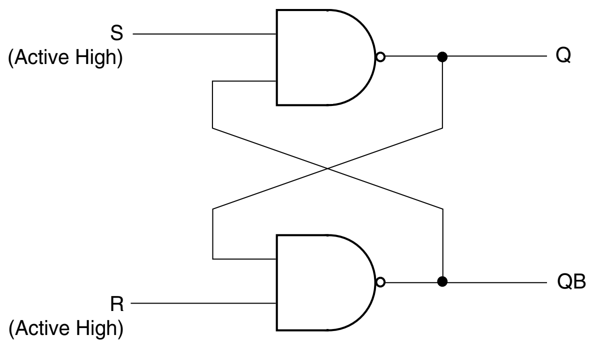
Non-recommended Circuits

Non-recommended circuits include cross-coupled NAND or NOR gates and RS flip-flops. These are illustrated in Figure 33, Figure 34 and Figure 35 below.

It is important to avoid the inadvertent creation of cross-coupled NAND/NOR gates by means of feedback loops within combination logic.

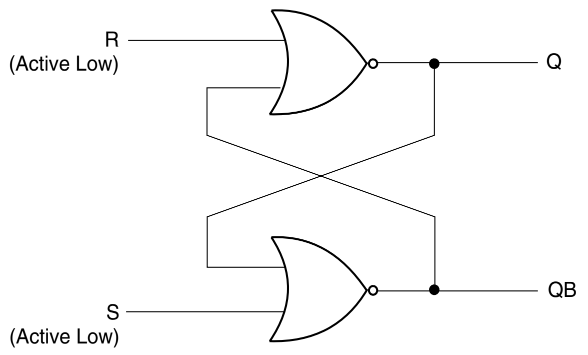
Cross-coupled NAND gates

Figure 33. Cross-coupled NAND gates forming bistable storage element



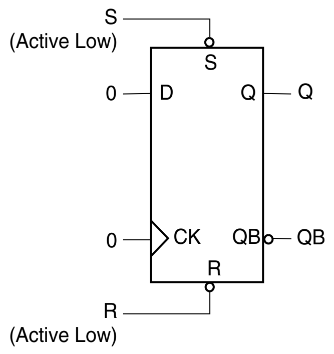
Cross-coupled NOR gates

Figure 34. Cross-coupled NOR gates forming bistable storage element



RS flip-flop

Figure 35. Asynchronous RS flip-flop



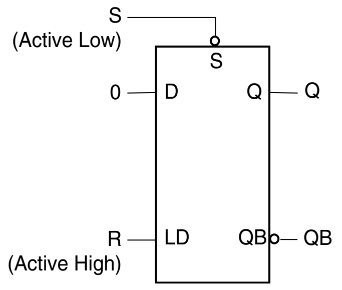
Recommended Circuits

The recommended methods of overcoming the problems listed in the previous section are as follows:

1. Use D-types with gated set/reset as required.
2. Use a latch configured as RS flip-flop. See the example circuit in Figure 36 below.
3. Avoid R-S races in the control of RS flip-flops.

Latch configured as RS flip-flop

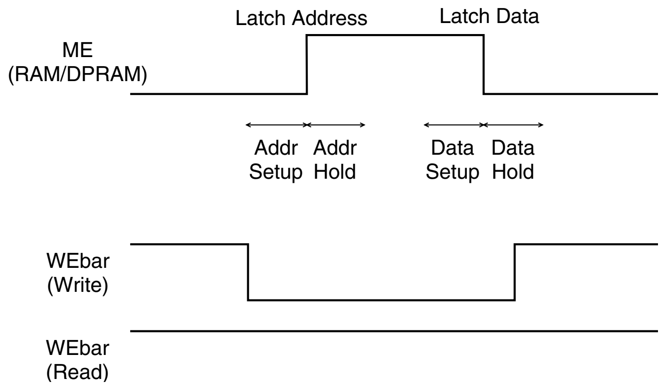
Figure 36. Latch configured as RS flip-flop



RAMs/ROMs in Synchronous Circuits

The problem of interfacing RAMs and dual-port RAMs into synchronous circuits is that they are double-edge triggered: the address is latched on the opposite clock edge to the data. This scheme is shown in relation to the ME and WEbar signals used by RAM and dual-port RAM in Figure 37 below. The ROM ME signal also latches the address on the rising edge.

Figure 37. ME and WEbar (RAM/DPRAM) timing scheme



Recommended Circuits

ME and WEBar Generation

To achieve synchronicity with the rest of the circuit, connect the RAM or dual-port RAM ME signal to an inverted system clock. One method of generating the WEbar signal is to use a D-type flip flop, with the inverted ME signal driving the clock, and an active-high external write request (wext) driving the d-input. The Webar signal is taken from the qb output. This produces the required delay of WEbar with respect to ME. This configuration is shown in Figure 38, and the resulting waveforms for a write cycle in Figure 39.

Figure 38. Interfacing RAM/DPRAM into a synchronous circuit

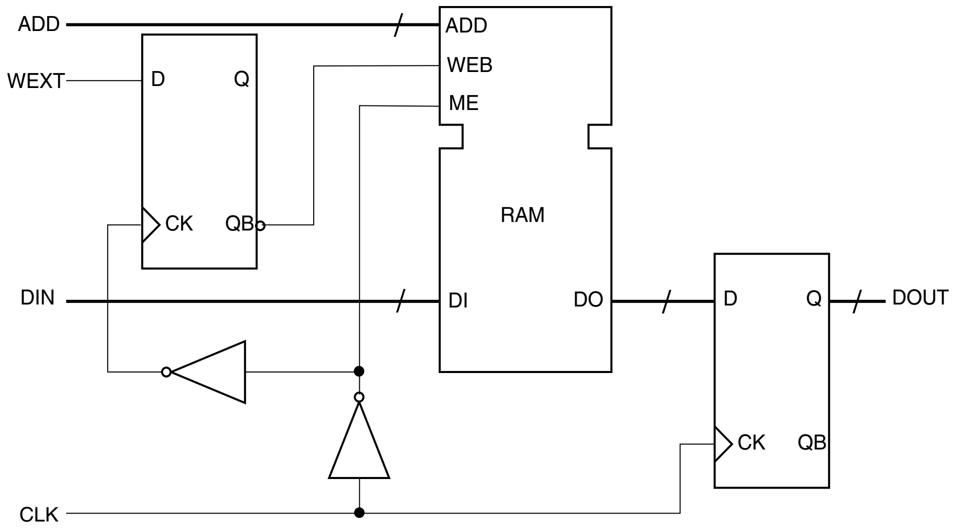
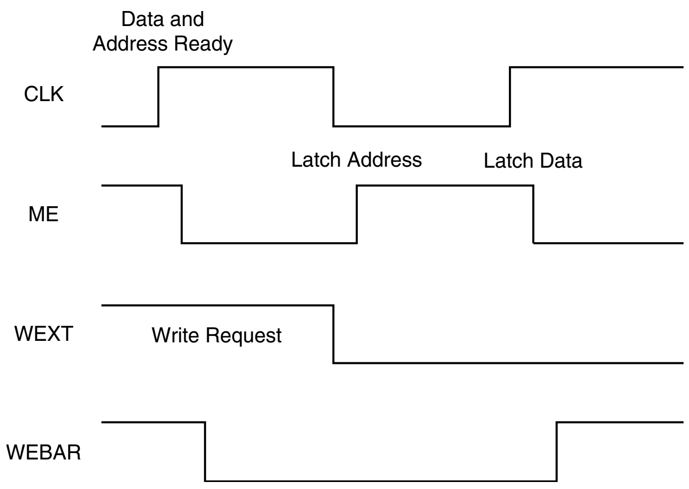


Figure 39. ME and WEbar timing scheme using flip-flop for WEbar generation

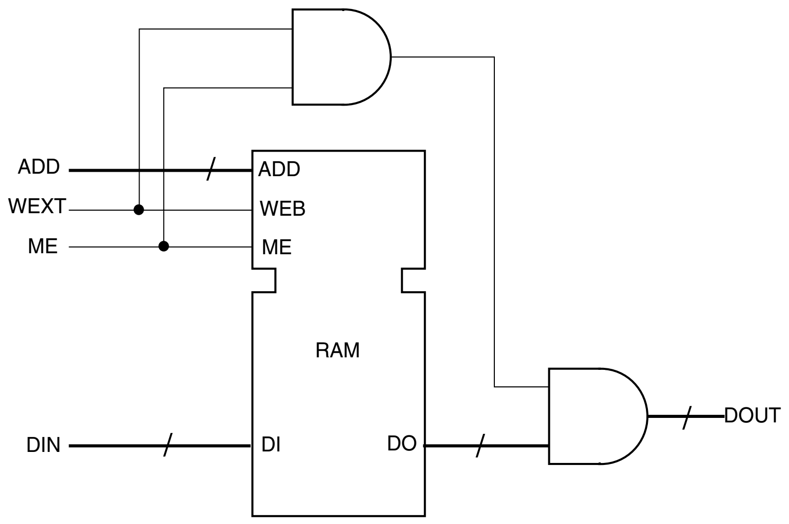


A consequence is that the clock duty cycle needs to be checked: the shorter phase needs to be longer than the setup and hold times and maximum propagation delay in the RAM, ROM, dual-port RAM and interfacing circuitry.

Avoiding Floating Outputs during Write Phase

During a write cycle, the output of a RAM/DPRAM (with tristate outputs) is floating. The propagation of this state can be avoided by means of the circuitry shown in Figure 40.

Figure 40. Avoiding floating RAM/DPRAM output propagation



Internal Tristates

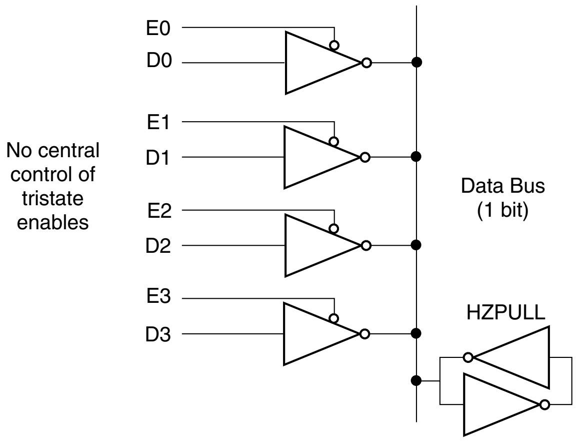
Internal tristates for data bus access within a circuit must be used with care, and should be avoided if possible. Potential problems are an undriven bus (particularly at initialization time) and conflicting bus drivers. An undriven bus floats to an intermediate state, causing high static currents.

Non-recommended Circuit

The general configuration of a circuit which is susceptible to problems of tristate control is shown in Figure 41 below.

Local control of tristate enables

Figure 41. Tristate bus with no central control of tristate enables. Do **not** use the Hzpull cell as a memory device.



The tristate enables are controlled locally, with no means of ensuring that there is no conflict (two driving simultaneously) or no undriven state, with no driver switched on. The Hzpull part retains the existing state of the bus, but it cannot initialize a tristated bus and creates asynchronous storage.

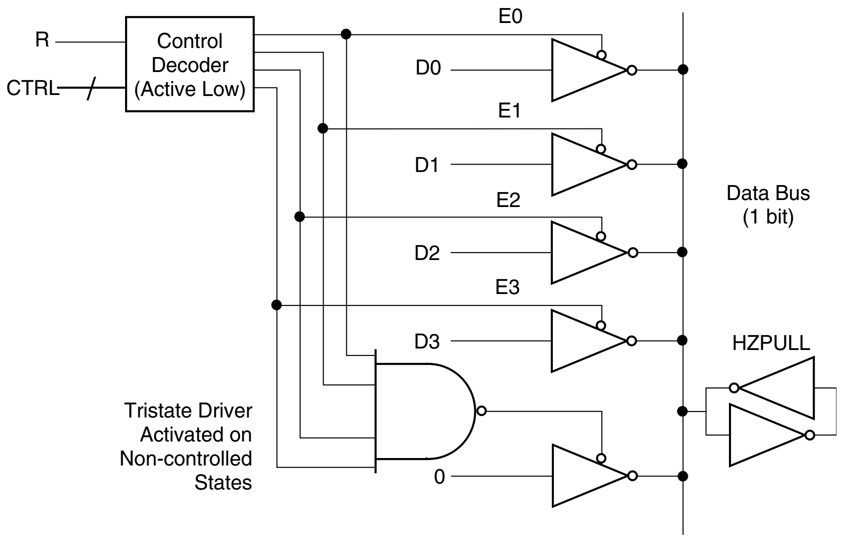
Recommended Circuits

1. Decode tristate control through a central control decoder. *It is recommended that the operation of this decoder is documented by means of a truth table or Karnaugh map.*
2. Provide one driver which is activated on non-controlled states. In particular, ensure that this driver is active during the reset state of the circuit.
3. Do **not** rely on Hzpull as a memory device. Its function is to prevent static dissipation, and it has a poor timing check.
4. Eliminate the tristates altogether by using multiplexed data bus lines. See "Multiplexers vs tristates" on page 26.

These three points are illustrated in Figure 42 below.

Central control of tristate enables

Figure 42. Tristate bus with central control of tristate enables and additional driver activated on non-controlled states


R
CTRL
Control Decoder (Active Low)
E0
D0
E1
D1
E2
D2
E3
D3
Data Bus (1 bit)
HZPULL
Tristate Driver Activated on Non-controlled States
0

Note: The Hzpull part is not strictly necessary in the above schematic. It is included for additional security during control transitions.

Multiplexers vs tristates

5. Preferably, **multiplex** data lines instead of using tristate-driven buses. The factors to be taken into account are as follows:

Tristates (disadvantages):

- large area
- limited buffering
- large routing load, consequently slow

Multiplexers (advantages):

- small area
- efficient routing

Note: The control decoding is the same for a tristate-driven bus as for a multiplexed set of data lines.

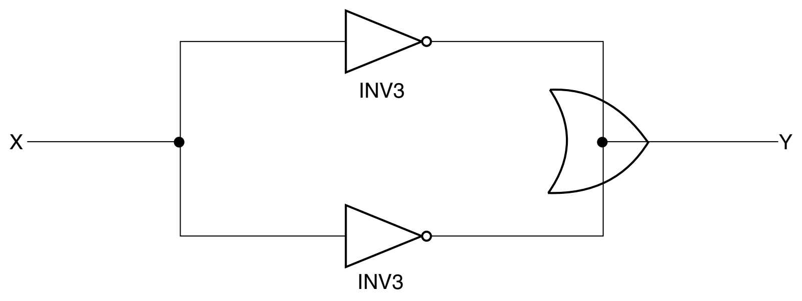
Paralleling Signals

For various reasons it sometimes appears necessary to include a wired OR or equivalent construction in a circuit, in order to provide parallel data signals. This practice is **not** recommended. *The use of wired OR parts should be avoided wherever possible.*

Non-recommended Circuit

Any circuit element which makes implicit or explicit use of the wired OR part is not recommended. An example is shown in Figure 43 below.

Figure 43. Wired OR part used to create higher fanout

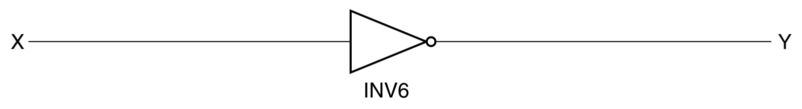


The function of this circuit may not be modeled properly, and there are placement and routing hazards.

Recommended Circuit

Use buffers of the appropriate strength and logic combinations which avoid the use of wired OR gates. The previous circuit can be replaced by the following equivalent:

Figure 44. Higher-fanout buffer replacing wired OR part



Fanout

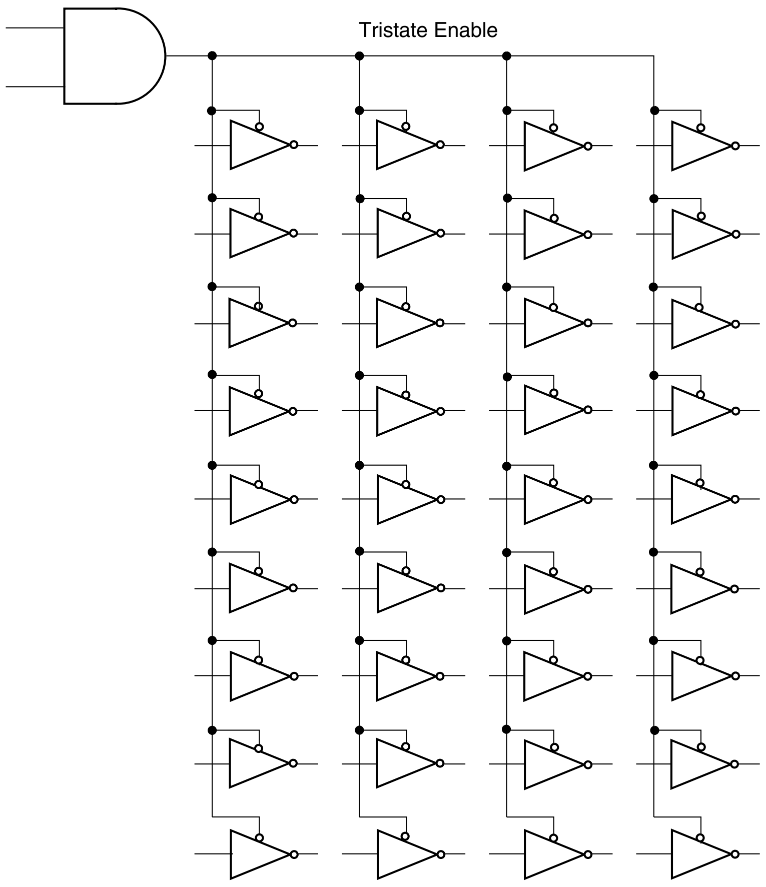
The relative fanout on any net in a circuit is the ratio of the total load (due to driven inputs and tracking capacitance) to the drive strength of the output driving the net. *In general the relative fanout should not exceed 12* (a process-independent figure derived from Atmel cell characterization data), otherwise the signals on the net are unacceptably delayed, and edges are unacceptably slow.

The special case of fanout in clock signals is dealt with in "Clock Buffering" on page 4.

Non-recommended Circuits

Any circuit which has excessive fanout on a data or control signal is not recommended. An example is shown in Figure 45.

Figure 45. Excessive fanout on control signal



Recommended Circuits

Use geometric or tree buffering in order to reduce fanout. Examples of each type are shown in Figure 46 and Figure 47.

Figure 46. Geometric buffering on control signal

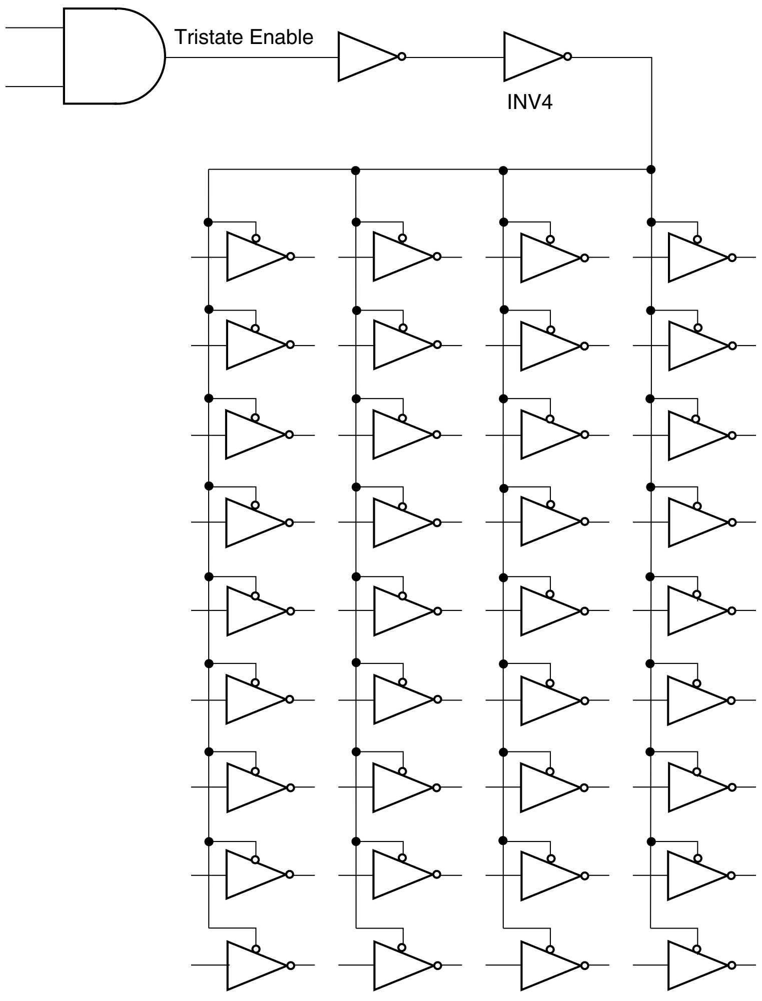
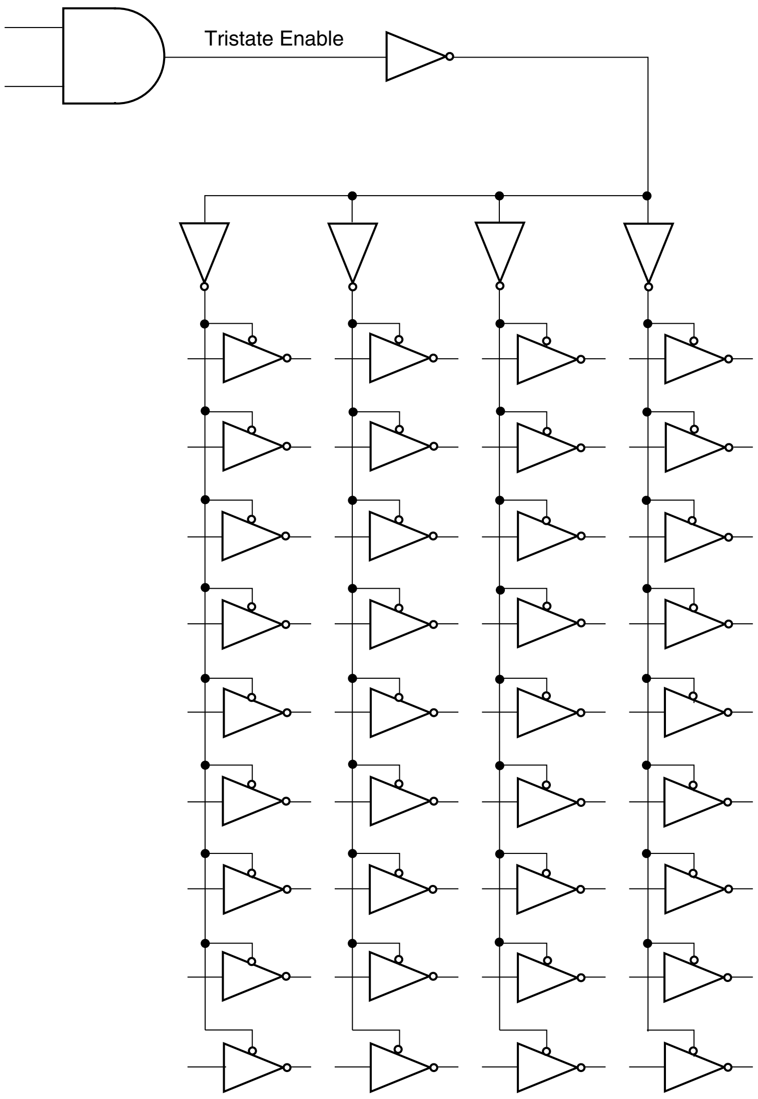


Figure 47. Tree buffering on control signal



Authorization

Relative fanout affects the speed of operation of a circuit. Given sufficient time, highly loaded nets will eventually settle to their correct logical value.

Accordingly, maximum relative fanout may be exceeded if no clock signals are involved, and data signals have sufficient time margin on input to clocked elements.

Design for Speed

A number of techniques can be used to increase the operational speed of a circuit. To an increasing extent, these are implemented automatically during design synthesis. If this is not available, some of the most popular methodologies are discussed in this section. *These generally involve a tradeoff between speed and silicon area.* These techniques are in addition to the fanout reduction methods described in "Fanout" on page 28.

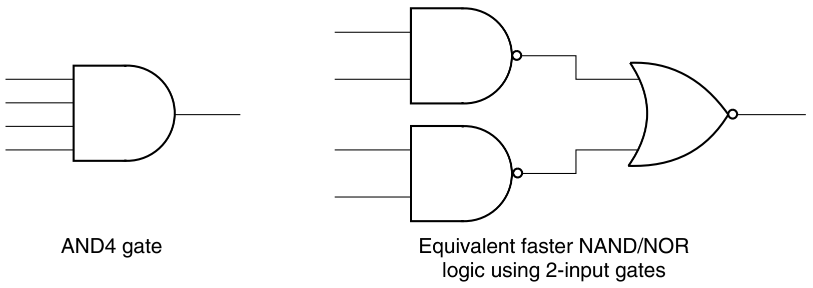
Recommended Circuits

Recommended techniques for increasing circuit speed, all of which involve safe design practices, are given below.

1. Use a maximum of 2 inputs on all combinational logic gates.

For example, an AND4 gate may be replaced by NAND/NOR logic as shown in Figure 48.

Figure 48. 4-input AND gate and equivalent NAND/NOR logic



2. 2 Use AOI or OAI logic where possible.

AND/OR/Inverter (AOI) and OR/AND/Inverter (OAI) gates are particularly economical for both speed and area. *Their use is recommended wherever possible.*

Figure 49, Figure 50 and Figure 51 show three common examples of the use of AOI logic: a multiplexer, an enabled (E-type) flip-flop and a toggle (T-type) flip-flop.

Figure 49. Multiplexer using AOI logic

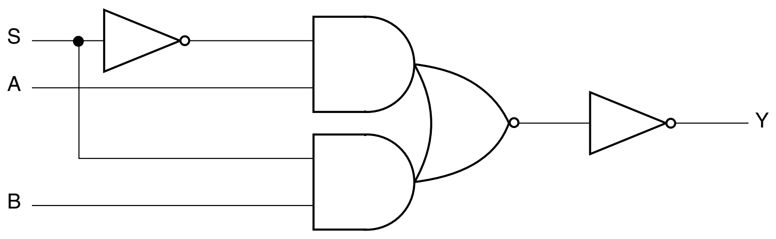


Figure 50. E-type flip-flop with reset constructed from AOI logic

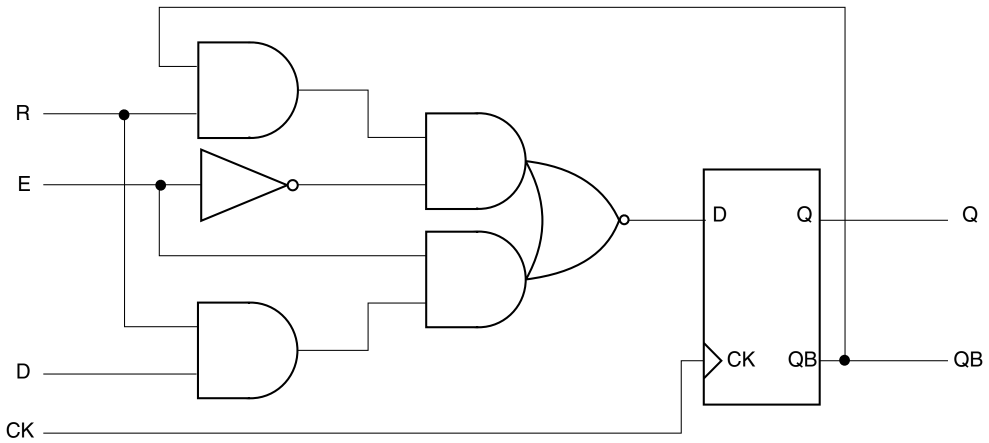
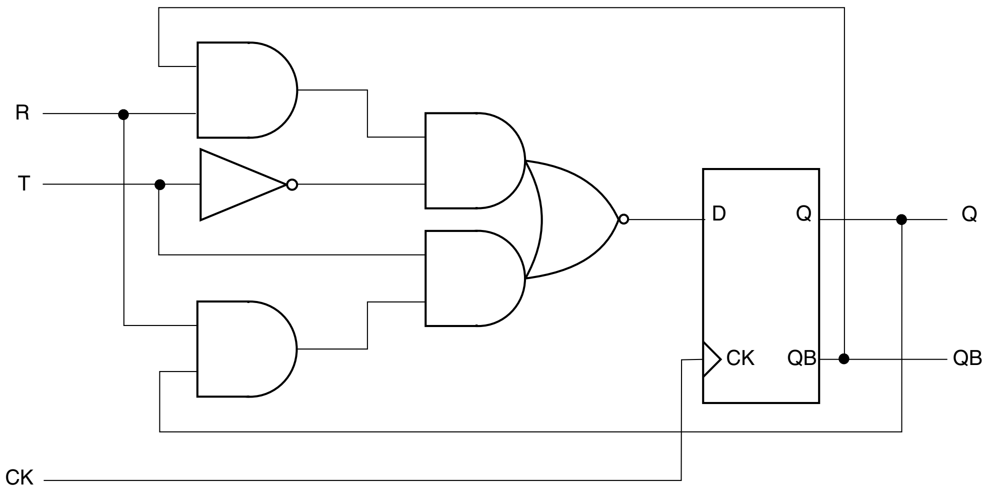


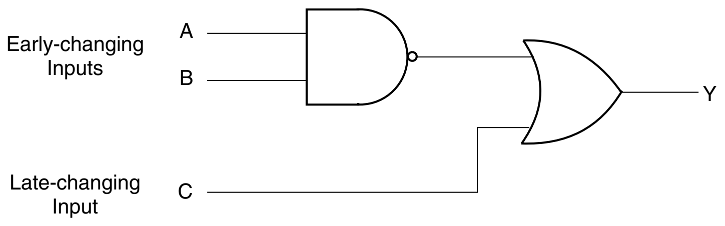
Figure 51. T-type flip-flop with reset constructed from AOI logic



3. Feed late changing inputs late into combinational logic.

An example of this technique is shown in Figure 52. *The aim is, as far as possible, to balance the total gate delay along each path of a combinational circuit.*

Figure 52. Late-changing input fed late into combinational logic



4. Use shift (Johnson) counters instead of binary counters.

A Johnson counter (Figure 53) is a shift register with the inverted output fed back into the primary input. An n-stage Johnson counter produces a set of distinct outputs of length 2n (see the truth table below), which can be decoded to give a count sequence. Its advantage is that, having no combinational logic between flip-flops, it can be run at the maximum speed permitted by setup and hold time constraints. The disadvantage of a Johnson counter is that, for a required count of m, it requires m/2 flip-flops, rather than log2(m) as required by a synchronous binary counter.

A Johnson counter can be provided with a synchronous reset, at the expense of an AND gate feeding into each flip-flop.

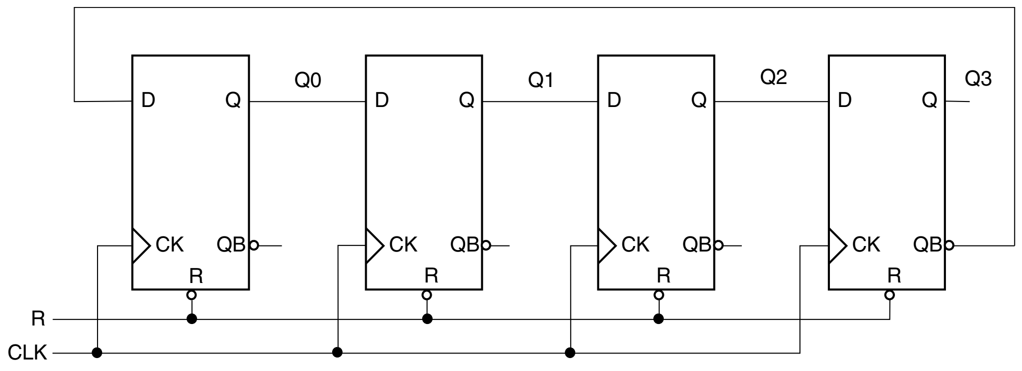
Note that the same buffering considerations apply to the clocking of long Johnson (and other) counters as they do to shift registers. See “Shift Registers” on page 15.

The truth table for a Johnson Counter is shown in Table 1 below.

Table 1. 4-stage Johnson Counter Truth Table

q0	q1	q2	q3
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

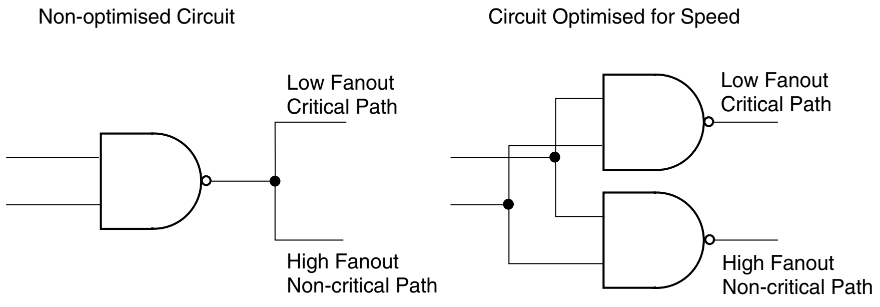
Figure 53. 4-stage Johnson counter



5 Use duplicate logic to reduce fanout.

This technique is analogous to the use of tree buffering to reduce fanout on clock signals. An example is shown in Figure 54.

Figure 54. Using duplicate logic to reduce fanout



Non-optimised Circuit
Low Fanout Critical Path
High Fanout Non-critical Path
Circuit Optimised for Speed
Low Fanout Critical Path
High Fanout Non-critical Path

6 Use fast library cells where available.

Consult the timing figures in the Atmel Databook for the relevant process, in order to identify the fastest available library cell for the required function.

7 Reduce the length of critical signal paths.

If a net priority scheme is available for automatic routing, raise the priority of nets in critical signal paths in order to give them preference in automatic routing.

In addition, if required, use the manual routing facilities of the design tool in order to identify the physical positioning of critical signal paths, and to re-route these manually in order to obtain the shortest path length.

8 Use Schmitt trigger inputs in noisy environments.

The use of Schmitt trigger inputs in noisy environments is strongly recommended.

Design for Testability

Testability within ASICs is based on the single 'stuck-at' fault model: *a faulty device is represented as having a single internal net held permanently at logic 0 or logic 1, regardless of how the net is driven*. Although this model is a simplification of the defects which can occur in an ASIC, experience has shown that it is adequate in practice for most purposes, and can form the basis of successful design-for-testability methodologies.

In terms of the single 'stuck-at' model, testability is based on two factors:

- **Controllability**: the ability to drive (from primary inputs) every internal net to both logic 0 and logic 1. In particular, the circuit must be reset to a known state within a specified number of clock cycles after initialization.
- **Observability**: the ability to detect (at primary outputs) that a single internal net is stuck at a state different from its driven state.

Controllability and observability are achieved by a combination of circuit design techniques and the selection of appropriate test vectors.

The fault coverage of a particular set of test vectors is:

$$\frac{\text{number of stuck-at faults identified by the vectors}}{2 \times (\text{number of nets in the circuit})}$$

Note that this formula is for net (output) `stuck-at' faults as defined in the first paragraph above, and not for input `stuck-at' faults.

Although the theoretical goal of 100% fault coverage is seldom achievable in practice, this section gives some techniques which enable an acceptably close figure to be obtained.

Non-recommended circuits

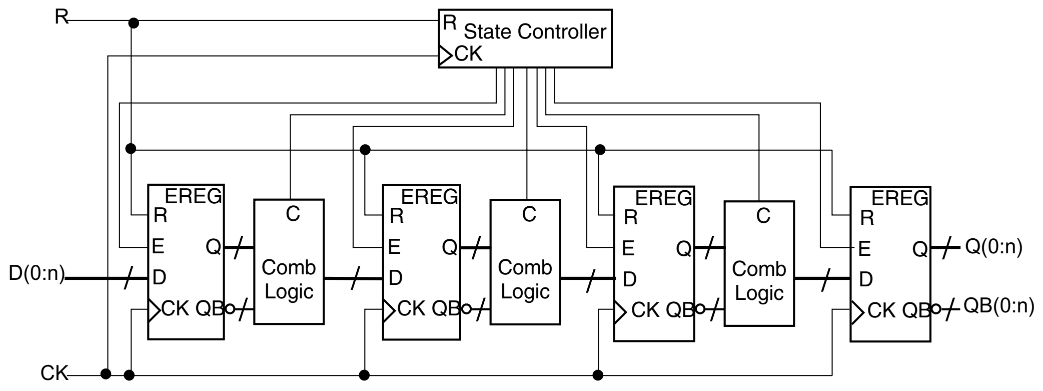
The following are examples of circuits with low observability/controllability. For each type of circuit, a recommendation is given later in this section for improving its testability.

Circuit with inaccessible internal logic

Figure 55 shows a typical circuit with a flow of bus-wide data through a sequence of enabled (E-type) registers linked by combinational logic. Control is by a central state controller, which has connections to the register enable lines and a control input to each combinational block.

From a testability point of view, the problem with this circuit is that only the first combinational block is directly controllable from external inputs, and only the last combination block is directly observable at external outputs. The central combinational block is neither directly controllable nor directly observable.

Figure 55. Circuit with inaccessible internal logic



Badly-designed state machines

For a circuit controlled by a state machine (such as that in Figure 55), problems can occur if the following two conditions are not met:

- all states must be decoded
- there must be no trap or lock states.

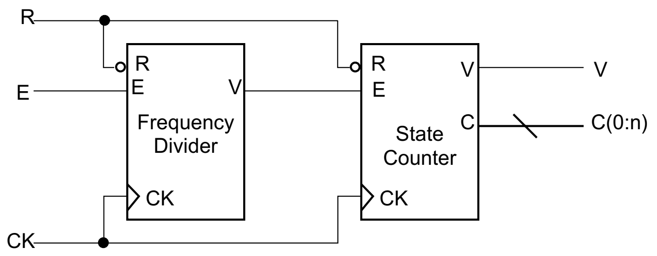
Chain of counters

Figure 56 shows the output of one counter feeding the enable of another. This configuration requires a large number of clock cycles to take the second counter through its entire sequence. The output from the first counter is not directly observable.

This design element violates both testability requirements: the first counter is not directly observable, and the second is not directly controllable.

A similar problem occurs with large single-element counters: they require a large number of test vectors (2^n for an n-bit counter) in order to take them through their entire count cycle.

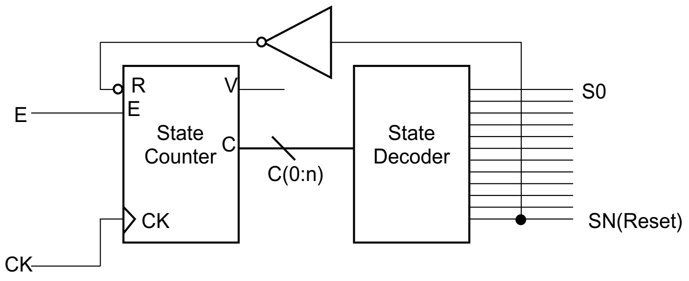
Figure 56. Chain of counters



Counter with feedback loop

The state counter in Figure 57 has its reset activated by a closed feedback loop triggered when it reaches an arbitrary internal state. This makes it impossible to reset to a known state at the start of a simulation.

Figure 57. Counter with closed feedback loop



Recommended circuits

The following techniques are recommended for improving the testability of circuits which include elements of the types given in the previous sections. In any particular case, one or more techniques may be applied to a circuit, depending on its particular configuration. The preferred technique for complex, system-on-chip designs is scan path testing.

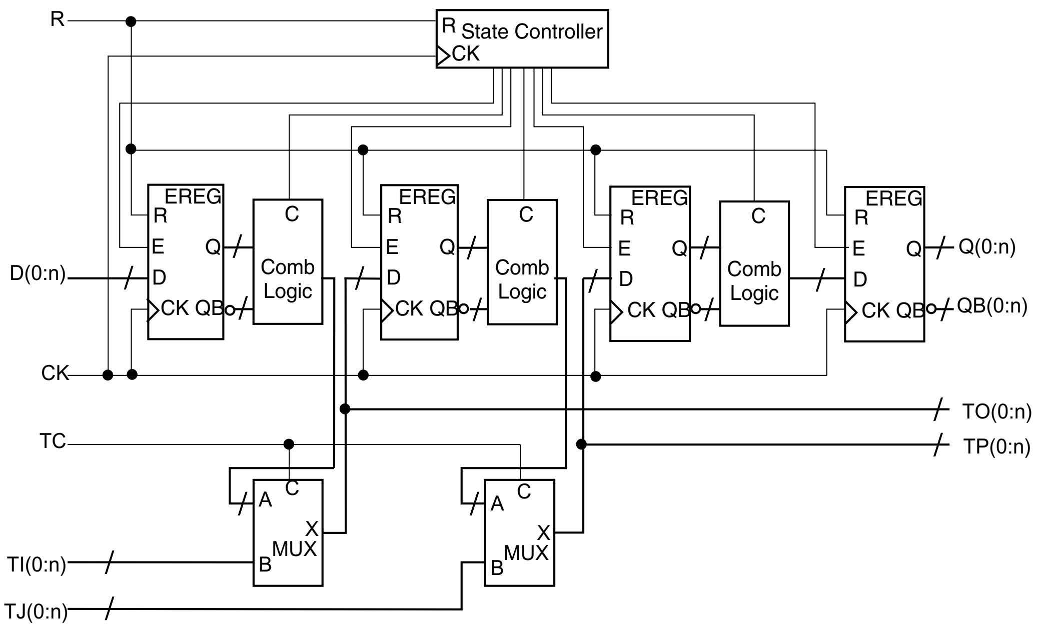
The techniques are described below in brief outline only. They do not form an exhaustive list, but are the methods found to be most successful in practice for synchronous designs based on a single clock signal. For more details, consult a standard reference on testability techniques.

1. Insert test inputs and outputs.

Additional inputs and outputs are inserted in order to make the internal logic of a circuit directly controllable and/or observable. Test inputs (ti(0:n) and tj(0:n)) are connected into the circuit via multiplexers. There is at least one test control signal (tc) in order to control the test multiplexers. Test out outputs to(0:n) and tp(0:n) are taken as outputs of the circuit element. The general concept is illustrated in Figure 58.

In test mode (tc high), test input data is fed directly into the internal registers, and test output is observed directly from the combinational logic blocks.

Figure 58. Circuit with test inputs and outputs



At the level of primary input/outputs, multiplexers may be inserted in order to combine test and operational data signals, thus reducing the pin overhead of test circuitry. Another possibility is to replace uni-directional input/output cells by bi-directional cells, using the other direction for test input/output. However, clock and reset signals must not be treated in this way. This level of multiplexing is not shown in Figure 58.

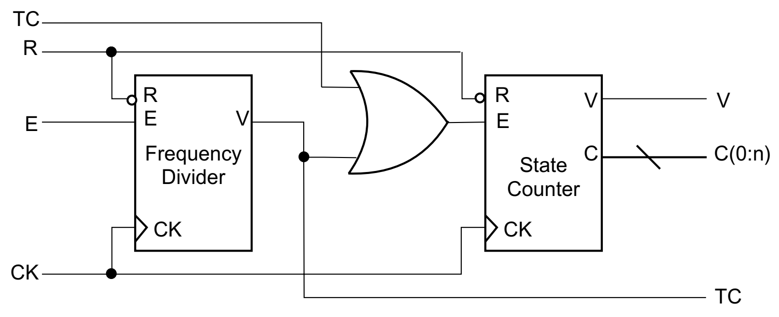
The minimum requirement is for a separate test control signal, unless an otherwise unused combination of control inputs is used for test mode. If test control is achieved by an otherwise unused combination of control inputs, care must be taken to ensure that under no circumstances can the circuit be inadvertently placed in test mode.

2. Break long counter/shift register chains.

Figure 59 shows how a chain of counters can be broken by a test control signal (tc) brought in using an OR gate. When the test control signal is high, the second counter increments at every clock cycle. The output from the first counter is taken to a primary output as the test output (to) signal.

A similar technique can be used to break up long single-element counters and shift registers.

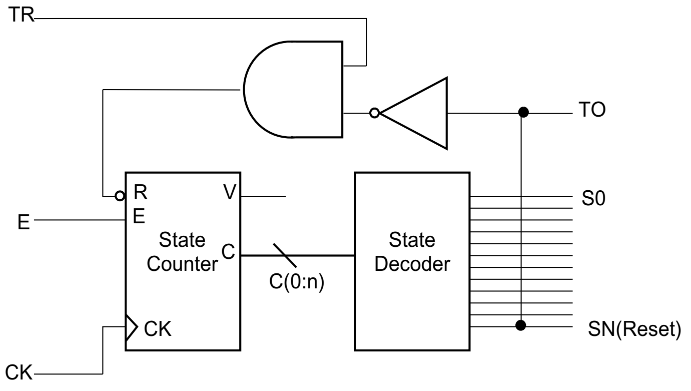
Figure 59. Chain of counters broken by test input and output signals



3. Open feedback loops.

Figure 60 shows how the feedback loop between a state decoder and a state counter is broken by the insertion of a test reset (tr) signal, and the connection of the reset state to a test output (to) signal. When tr is high, the circuit functions normally; the state counter is reset by forcing tr low. The value of the internal reset is monitored by the signal to.

Figure 60. Counter with feedback loop opened by test control and output signals



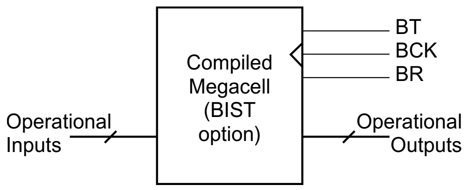
4. Use BIST with compiled megacells.

Compiled megacells (RAM, ROM, PLA, Multiplier, Dual-port RAM and FIFO) are inherently difficult to test, owing to their internal complexity, combinational depth and the small number of input/output signals relative to the number of internal cells.

To provide a solution to this problem giving 100% fault coverage for single ‘stuck-at’ faults, Atmel has implemented a *built-in self-test (BIST) option* for each compiled megacell. Selecting the BIST option produces a compiled megacell with customized test circuitry and three additional pins: BIST select (bt), BIST clock (bck) and BIST result (br). In most cases the BIST clock signal can be connected directly to the main system clock. See Figure 61.

In operational mode, the BIST select signal is low, and the normal working of the compiled megacell is unaffected. In BIST mode, with BIST select high, a sequence of clock edges on the BIST clock signal takes the megacell through a fixed test sequence which completely exercises all internal cells, and results in a unique signature on the BIST result signal after a specific number of clock cycles. On a fabricated device, any fabrication error within the megacell is revealed as a difference between the signature obtained and that resulting from a fault-free simulation of the device.

Figure 61. Compiled megacell with BIST input/outputs



5. Scan path testing.

An established technique for providing a high level of fault coverage in a circuit which has inaccessible internal logic is scan path testing. This requires the insertion of multiplexers in front of all storage elements in the circuit (such as the E-type flip-flop in Figure 62), and linking the additional inputs to form a single shift register which threads the entire device (Figure 63). This forms the scan path, from scan input si to scan output so. Note that buffering may be necessary between the q and so outputs of a scan flip-flop if they are both connected to external part-level or device-level outputs. Scan insertion can be performed automatically by sysntesis tools.

If the circuit contains gated clocks. dual-edged clocks or asynchronous control signals, multiplexers must be inserted where appropriate so that, in test mode, all clocked elements are activated on the same clock edge, and all control signals are synchronous. *Extreme care must be taken in inserting these multiplexers, in order not to introduce spikes or skew on clock or control lines.* See "Gated Clocks" on page 10.

Figure 62. E-type scan path flip-flop

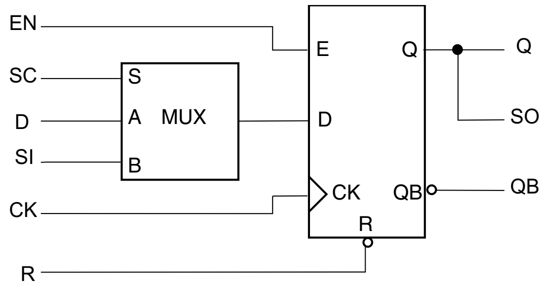
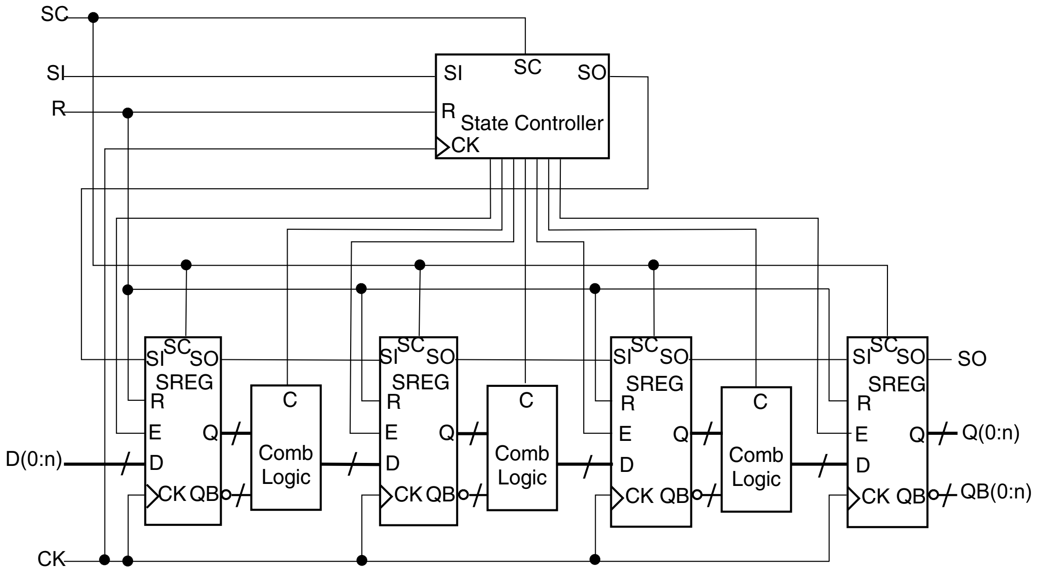


Figure 63. Circuit with scan path



In scan test mode, with the scan control signal sc high, a test pattern is shifted in serially through the scan path. The circuit is then put into operational mode for a single clock cycle, which propagates the test data through the combinational logic and back into the registers in the scan path. Again in scan test mode, the result is then shifted out through the scan chain. The test vectors are selected in order to exercise all the combinational logic in the circuit.

Test vectors for use with a scan path can be produced by an analysis of the logic functions implemented in the combinational logic blocks in the circuit. Alternatively, a pseudo-random binary sequence (PRBS) generator can be inserted at the start of the scan path, to produce a random test sequence. This can be combined with a signature analyser at the end of the scan path, which compresses the bit stream into a short, unique signature. Both the PRBS generator and the signature analyser are based on the technique of linear feedback shift registers. This method eliminates the need to develop a long set of test vectors, for a small overhead in silicon area. It produces an acceptably high level of fault coverage, and also permits in-service testing of devices.

As a further alternative, the software tools which perform scan path insertion also generate automatically a corresponding set of scan test vectors.

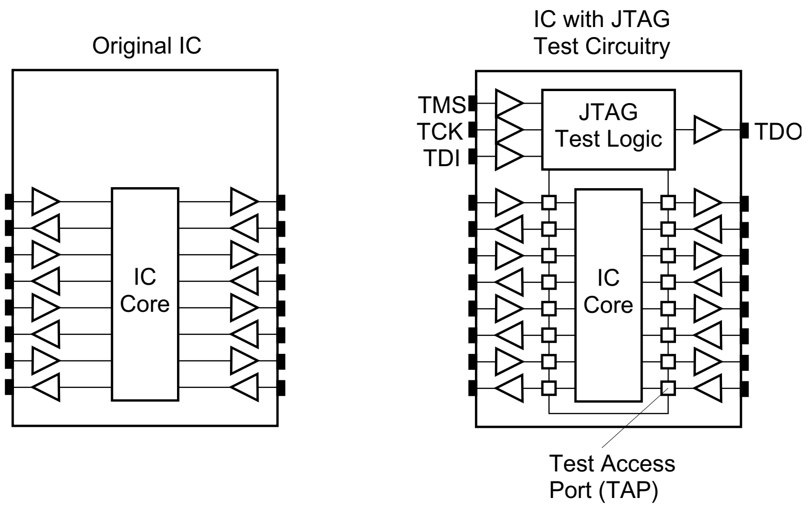
6. JTAG boundary scan path.

In addition to a scan path threading the internal logic of a circuit, a scan path may be constructed around the primary input/output cells. This technique follows the JTAG/IEEE 1149.1 standard, and provides a capability for board-level connectivity tests without the need to propagate test vectors through the core of a device. This is achieved by adding test circuitry and test pins to a device which enable all input and output pins to be connected together in a boundary scan path. The boundary scan path is a shift register with a parallel load facility which can be used to control and read the signal states on all input/output cells. See Figure 64.

In addition, the JTAG methodology allows the tester to interrogate an IC buried in the middle of the PCB, to run diagnostic checks, to identify the IC, or to sample the signal states of its pins during normal operation.

At PCB level, all such scan chains are connected together in series (parallel branches are permitted).

Figure 64. JTAG test circuitry



Test Vector Generation

Functional and Post-fabrication Tests

In the complete ASIC design and fabrication cycle, there are two main test phases:

- **functional testing**, to check whether the circuit design conforms to its functional specification
- **post-fabrication testing**, to ensure that each device has been fabricated without any faults.

It is essential to distinguish between the purposes of these tests: functional tests are concerned with the operation of the ASIC relative to its specification, post-fabrication tests are to check for differences between the operation of each individual die and an ideal, fault-free device (in practice a fault-free simulation of the circuit). This distinction is not always clearly drawn in practice, and in many cases the same simulation stimuli are applied to both types of test.

The fault-free simulation against which the post-fabrication tests are measured should (ideally) exercise all the nets in the circuit, in such a way that a fault in any net will show as a difference in an output signal. This simulation assumes (but does not verify) that the circuit is functioning according to specification.

This section gives some background information required for the generation of vectors intended for post-fabrication tests, their essential properties, and the specific requirements which they must satisfy. To an increasing extent, these test vectors are generated automatically as part of scan insertion. If this is not possible, then the guidelines in this section must be followed. These test vectors (both inputs and expected outputs) are supplied to Atmel, in a recognized format, with the design database for fabrication.

Static Test Vectors

An important concept which underlies the techniques for testing ASICs is that of static and dynamic circuits. These are defined as follows:

A **static** digital circuit is one which reaches a stable state a certain time after a set of inputs is applied, and then remains in that state for as long as it is powered up. For example, if the clock were to stop, the circuit would remain in a stable state as long as power were supplied.

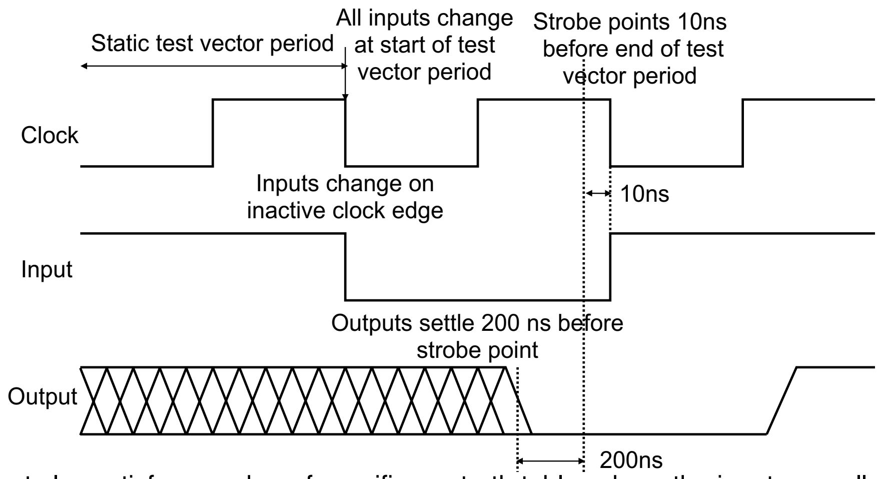
A **dynamic** digital circuit does not reach a stable state after a certain time, or the stable state represents a loss of data, such as leakage from a dynamic RAM. In dynamic circuits, a clock failure would lead to a loss of data.

Test vectors are designed on assumption that the device under test is a static circuit. They must also be compatible with the operation of the automatic test equipment used for post-fabrication tests. Test vectors which satisfy these conditions are called static test vectors. Their essential properties are as follows:

- For each test period, all inputs are applied simultaneously at the start of the period (1000ns, 10000ns or a multiple thereof).
- All outputs are strobed 10ns before the end of the test period
- All outputs and internal nodes must have reached a stable state 200ns before the strobe point.
- Input data must not change on the active edge of the clock which latches it.

The above three requirements for static test vectors are illustrated in Figure 65.

Figure 65. Static test vectors



Static test vector period
All inputs change at start of test vector period
Strobe points 10ns before end of test vector period
Clock
Inputs change on inactive clock edge
10ns
Input
Outputs settle 200 ns before strobe point
Output
200ns

- Static test vectors must also satisfy a number of specific requirements which are discussed in a later section.

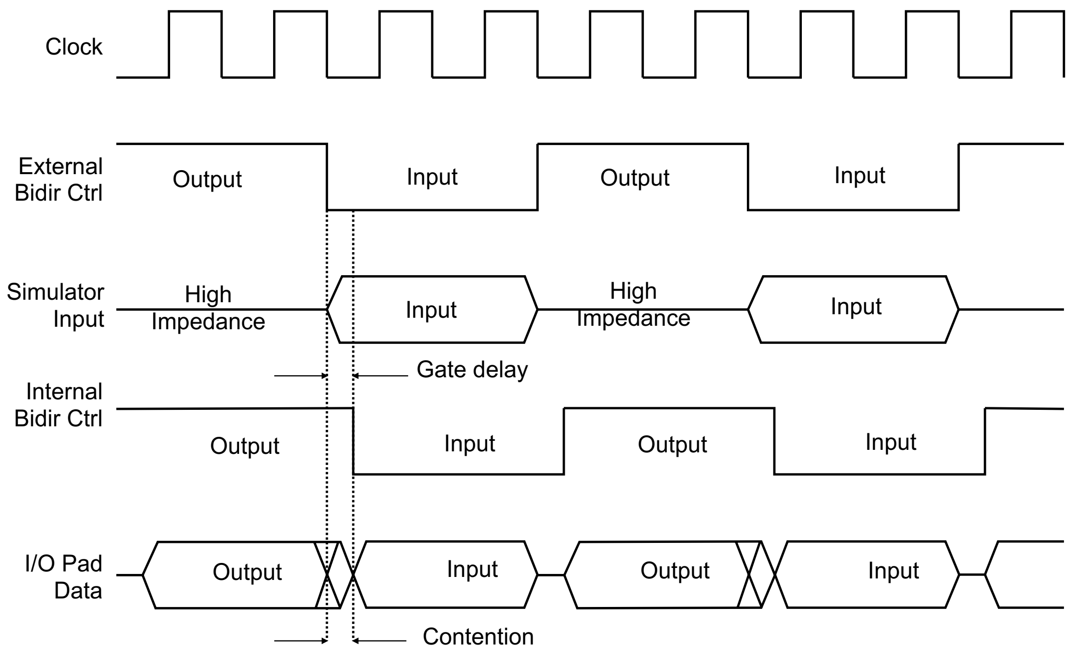
A consequence of the requirement for static circuits is that the behavior of a circuit under test can be described by a truth table, where the inputs are all applied together at the start of the static test vector period, and the outputs are at a stable state after the setting period.

Static Vectors for Bi-directional Input/Outputs

One circuit element which requires special attention in the design of static test vectors is the bi-directional input/output. *The problem is that there is almost always a short period of contention on the pad as it changes from output to input.* This is because the bi-directional control line is usually changed through internal logic, which produces a gate delay, whereas the simulated direction of the pad is determined externally, with no gate delay. Accordingly, the simulator has started to drive input data to the pad while it is still in its output state. The situation is illustrated in Figure 66.

Non-recommended protocol

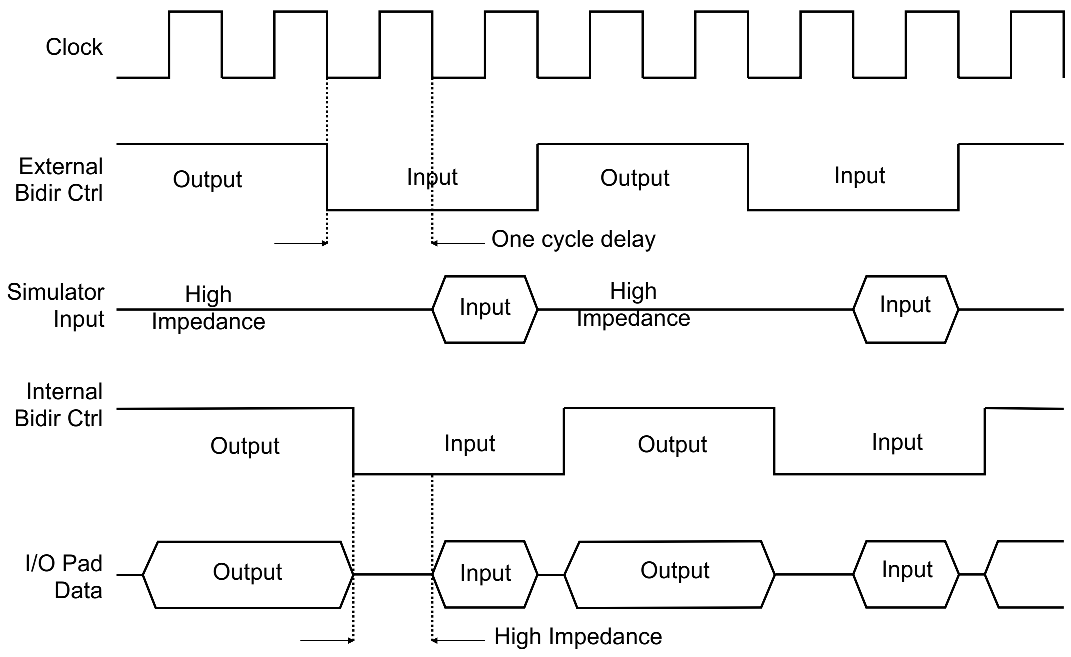
Figure 66. Contention on bi-directional input/output pad



The solution to this problem, to meet tester requirements, is to delay the application of input data by the simulator for a clock period after the transition from output to input. This is illustrated in Figure 67.

Recommended protocol

Figure 67. Delayed input on bi-directional input/output pad



ASIC Test Procedures

Post-fabrication tests are applied to ASICs using industry-standard automatic test equipment (ATE). The test vectors and pinout data submitted with a design are used to configure the ATE, and apply the test sequence. The overall procedure is as follows:

A truth table is extracted from the simulation vectors, and used to set up the ATE. The test equipment applies the test vectors at regular intervals (1000ns for digital designs and 10000ns or a multiple thereof for mixed analog/digital designs) and strobes the outputs from the ASIC 10ns before the end of this interval.

If any output differs from the one produced by fault-free simulation, then the particular device is rejected. A table is produced, showing the vector number at which each failure occurred, for the skew on each input.

In order to test the tolerance of the device to clock skew, the operational test cycles are repeated a number of times, each with one input advanced or retarded in steps of 10ns to a maximum skew of 80ns. (Note that the clock signal is not treated as a special case in the test process.)

If required, a variety of additional tests may be applied, including parametric tests and speed tests using functional vectors at full operational speed. These test such aspects as carry propagation.

Rules for Test Vectors

Design Rules

The following rules for test vectors are to ensure that they (and the associated design) are in general compliance with the design guidelines set out in this and previous sections of this document:

- An external master reset must be used (even if there is a POR cell) to ensure a complete device initialization.
- The package used must be in the Atmel ASIC Package Selector Guide.
- All I/O cell names must be from an Atmel Library or other known source, and not modified. If not, a Buffer Information Base (BIB) file must be created containing the name and details of the new I/O cell.
- Long counters and shift registers must have test access to intermediate stages.
- All internal feedback loops must be broken with test I/Os.
- Redundant logic must be avoided.
- Compiled megacells must either use the BIST option or have direct test access to input/outputs.
- Analog cells are peripherals. At most two analog cells may be connected in series.

- Different types of analog cells may not be placed in parallel, except if one is for input and the other for output.

Simulation Rules

Table 2 shows the nine possible simulation events or signal settings that can occur. The abbreviations shown are used in the list of checks that follows. I denotes either an input pad or a bi-directional pad configured as an input. O denotes an output pad, a bi-directional pad configured as an output, a tristate pad or an (internal) enable signal.

Table 2. Possible Simulation Events

Event value	Signal state		
Event type	0	1	X
I (input)	I:0	I:1	I:X
O (output)	O:0	O:1	O:X
Z (tristate)	Z:0	Z:1	Z:X

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL (408) 441-0311
FAX (408) 487-2600

Europe
Atmel U.K., Ltd.
Coliseum Business Centre
Riverside Way
Camberley, Surrey GU15 3YL
England
TEL (44) 1276-686-677
FAX (44) 1276-686-697

Asia
Atmel Asia, Ltd.
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan
Atmel Japan K.K.
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Atmel Colorado Springs
1150 E. Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL (719) 576-3300
FAX (719) 540-1759

Atmel Rousset
Zone Industrielle
13106 Rousset Cedex
France
TEL (33) 4-4253-6000
FAX (33) 4-4253-6001

Fax-on-Demand
North America:
1-(800) 292-8635
International:
1-(408) 441-0732

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

BBS
1-(408) 436-4309

1205A–12/99/xM

5.2 Attachment 2

Chapter 2
Clocks and Resets

2.1 Introduction

The cost of designing ASICs is increasing every year. In addition to the non-recurring engineering (NRE) and mask costs, development costs are increasing due to ASIC design complexity. To overcome the risk of re-spins, high NRE costs, and to reduce time-to-market delays, it has become very important to design the first time working silicon.

This chapter constitutes a general set of recommendations intended for use by designers while designing a block or an IP (Intellectual Property). The guidelines are independent of any CAD tool or silicon process and are applicable to any ASIC designs and can help designers to plan and to execute a successful System on Chip (SoC) with a well-structured and synthesizable RTL code.

The current paradigm shift towards system level integration (SLI), incorporating multiple complex functional blocks and a variety of memories on a single circuit, gives rise to a new set of design requirements at integration level. The recommendations are principally aimed at the design of the blocks and memory interfaces which are to be integrated into the system-on-chip. However, the guidelines given here are fully consistent with the requirements of system level integration and will significantly ease the integration effort, and ensure that the individual blocks are easily reusable in other systems.

These guidelines can form as a basis of checklist that can be used as a signoff for each design prior to submission for fabrication.

2.2 Synchronous Designs

Synchronous designs are characterized by a single master clock and a single master set/reset driving all sequential elements in the design.

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5_2,
© Springer Science+Business Media, LLC 2012

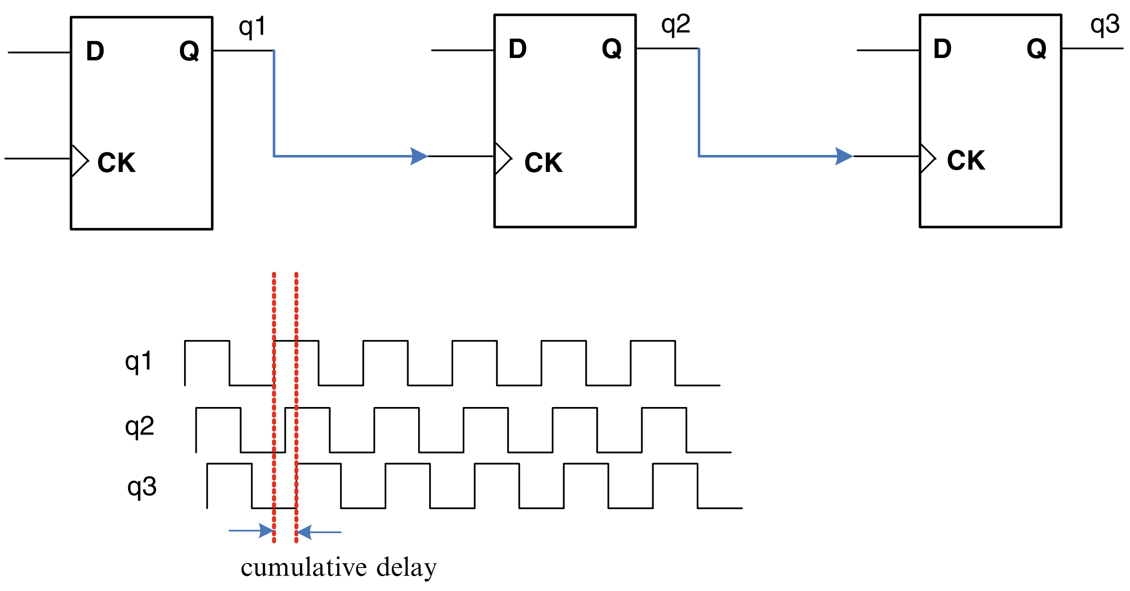


Fig. 2.1 Flip flop driving the clock input of another flip flop (ripple counter)

Experience has shown that the safest methodology for time domain control of an ASIC is synchronous design. Some of the problems with the circuits not being synchronous have been shown in this section.

2.2.1 *Avoid Using Ripple Counters*

Flip Flops driving the clock input of other flip flops is somewhat problematic. The clock input of the second flip-flop is skewed by the *clock-to-q* delay of the first flip-flop, and is not activated on every clock edge. This cumulative effect with more than two Flip Flops connected in a similar manner forms a Ripple counter as shown in Fig. 2.1. Note the cumulative delay gets added on with more number of flip flops and hence the same is not recommended. More details on the ripple counter are given in Sect. 5.6.7.

2.2.2 *Gated Clocks*

Gating in a clock line causes clock skew and can introduce spikes which trigger the flip-flop. This is particularly the case when there is a multiplexer in the clock line as shown in Fig. 2.2.

Simulating a gated clock design might work perfectly fine but the problem arises when such a design is synthesized.

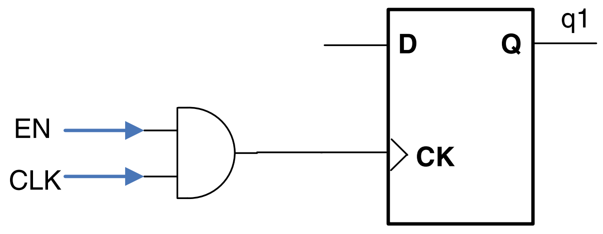


Fig. 2.2 Gated clock line

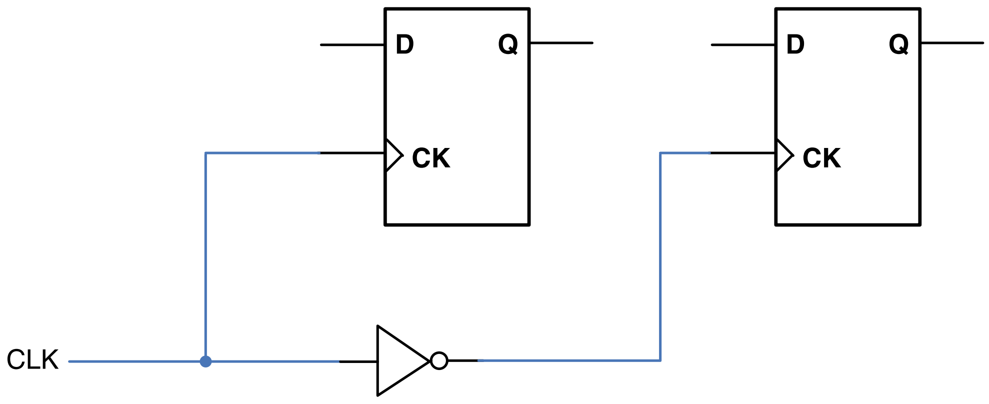


Fig. 2.3 Double-edged clocking

2.2.3 Double-Edged or Mixed Edge Clocking

As shown in Fig. 2.3, the two flip-flops are clocked on opposite edges of the clock signal. This makes synchronous resetting and test methodologies such as scan-path insertion difficult, and causes difficulties in determining critical signal paths.

2.2.4 Flip Flops Driving Asynchronous Reset of Another Flop

In Fig. 2.4, the second flip-flop can change state at a time other than the active clock edge, violating the principle of synchronous design. In addition, this circuit contains a potential race condition between the clock and reset of the second flip-flop.

The subsequent sections show the methods to avoid the above non-recommended circuits.

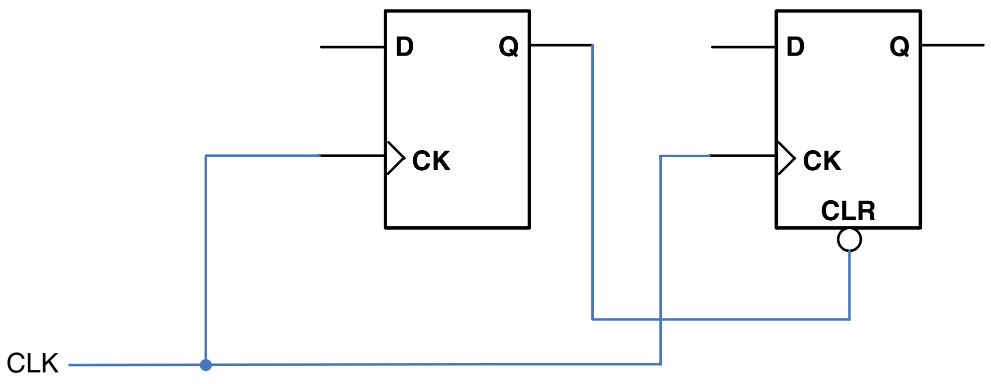


Fig. 2.4 Flip flop driving asynchronous reset of another flop

2.3 Recommended Design Techniques

When designing with HDL code, it is important to understand how a synthesis tool interprets different HDL coding styles and the results to expect. It is very important to think in terms of hardware as a particular design style (or rather coding style) can affect gate count and timing performance. This section discusses some of the basic techniques to ensure optimal synthesis results while avoiding several causes of unreliability and instability.

2.3.1 Avoid Combinational Loops in Design

Combinational loops are among the most common causes of instability and unreliability in digital designs. In a synchronous design, all feedback loops should include registers. Combinational loops violate synchronous design principles by establishing a direct feedback with no registers.

In terms of HDL language, combinational loops occur when the generation of a signal depends on itself through several combinational *always*[1] blocks or when the left-hand side of an arithmetic expression also appears on the right-hand side. Combo loops are a hazard to a design and synthesis tools will always give errors when combo loops are encountered, as these are not synthesize-able.

The generation of combo loops can be understood from the following bubble diagram in Fig. 2.5. Each bubble represents a combo always block and the arrow going into it represents the signal being used in that always block while an arrow going out from the bubble represents the output signal generated by that output block. It is evident that the generation of signal ‘a’ depends on itself through signal ‘d’, thereby generation a combinational loop.

[1]For simplicity, any HDL languages that this book refers to takes Verilog as an example.

```
always@ (a)
begin
b = a;
End

always@ (b)
begin
c = b;
End

always@ (c)
begin
d = c;
End

always@ (c)
Begin
a = c;
end
```

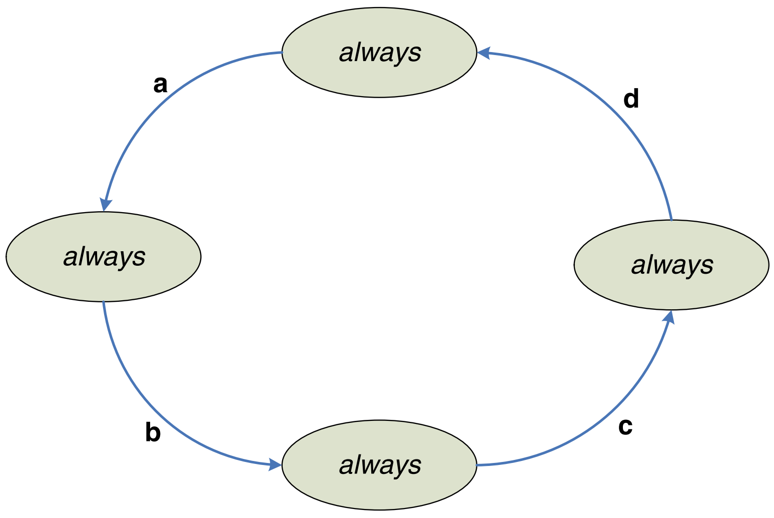


Fig. 2.5 Combinational loop example and bubble diagram

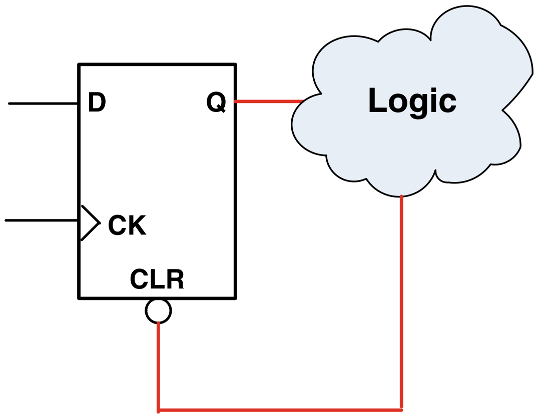


Fig. 2.6 Combinational loop through asynchronous control pins

The code and the bubble diagram are shown below [28]:

In order to remove combo loops, one must change the generation of one of the signals so the dependency of signals on each other is removed. Simple resolution to this problem is to introduce a Flip Flop or register in the combo loop to break this direct path.

Figure 2.6 shows another example where output of a register directly controls the asynchronous pin of the same register through combinational logic.

Combinational loops are inherently high-risk design structures. Combinational loop behavior generally depends on the relative propagation delays through the logic involved in the loop. Propagation delays can change based on various factors and the behavior of the loop may change. Combinational loops can cause endless computation loops in many design tools. Most synthesis tools break open or disable combinatorial loop paths in order to proceed. The various tools used in the design flow may open a given loop a different manner, processing it in a way that may not be consistent with the original design intent.

2.3.2 Avoid Delay Chains in Digital Logic

Delay chains occur when two or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Often inverters are chained together to add delay. Delay chains generally result from asynchronous design practices, and are sometimes used to resolve race conditions created by other combinational logic. In both FPGA and ASIC, delays can change with each place-and-route. Delay chains can cause various design problems, including an increase in a design's sensitivity to operating conditions, a decrease in a design's reliability, and difficulties when migrating to different device architecture. Avoid using delay chains in a design, rely on synchronous practices instead.

2.3.3 Avoid Using Asynchronous Based Pulse Generator

Often design requires generating a pulse based on some events. Designers sometimes use delay chains to generate either one pulse (pulse generators) or a series of pulses (multi-vibrators). There are two common methods for pulse generation; these techniques are purely asynchronous and should be avoided:

- A trigger signal feeds both inputs of a two-input AND or OR gate, but the design inverts or adds a delay chain to one of the inputs. The width of the pulse depends on the relative delays of the path that feeds the gate directly and the one that goes through the delay. This is the same mechanism responsible for the generation of glitches in combinational logic following a change of inputs. This technique artificially increases the width of the spike by using a delay chain.
- A register's output drives the same register's asynchronous reset signal through a delay chain. The register essentially resets itself asynchronously after a certain delay.

Asynchronously generated pulse widths often pose problem to the synthesis and place-and-route software. The actual pulse width can only be determined when routing and propagation delays are known, after placement and routing. So it is difficult to reliably determine the width of the pulse when creating HDL. The pulse may not be wide enough for the application in all PVT conditions, and the pulse width will change when migrating to a different technology node. In addition, static timing analysis cannot be used to verify the pulse width so verification is very difficult.

Multi-vibrators use the principle of the "glitch generator" to create pulses, in addition to a combinational loop that turns the circuit into an oscillator [25].

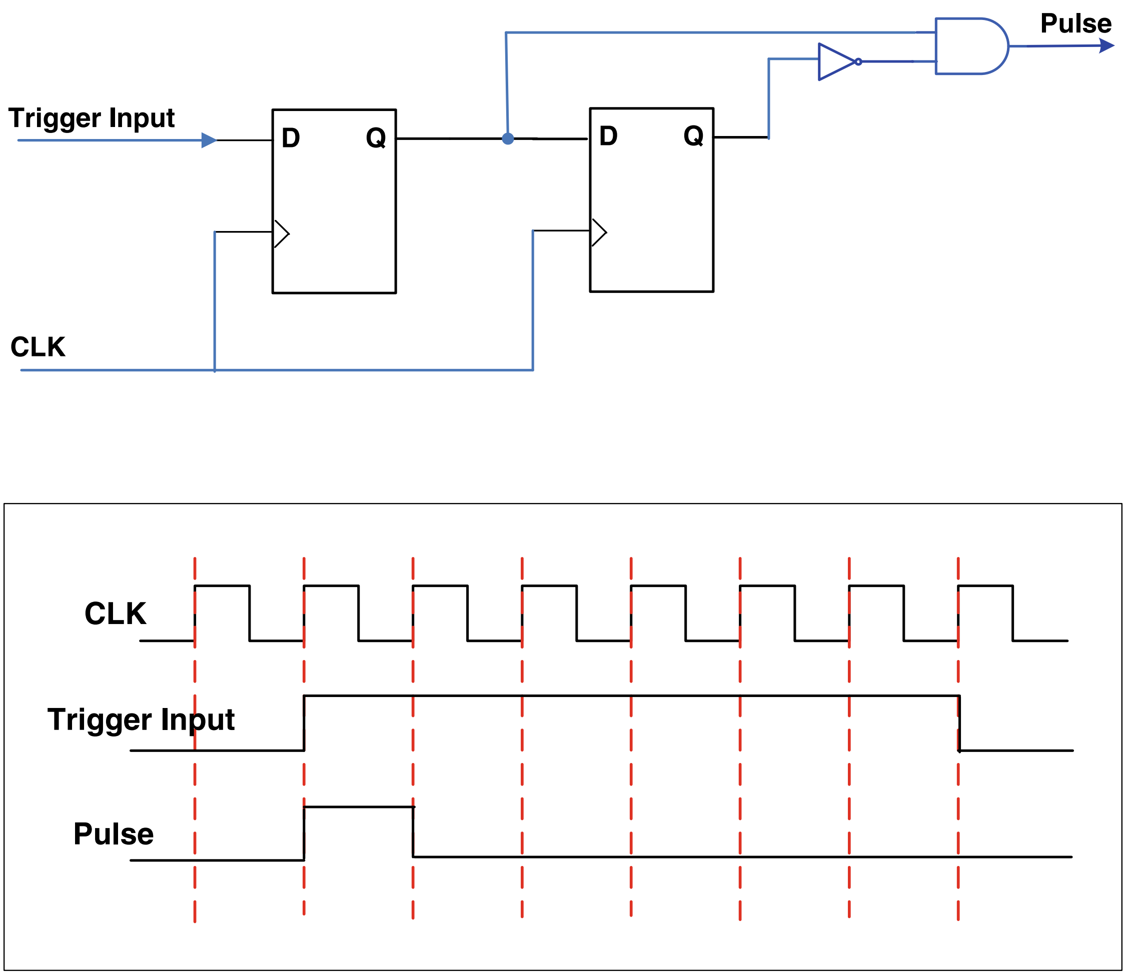


Fig. 2.7 Synchronous pulse generator circuit on start of trigger input

Structures that generate multiple pulses cause even more problems than pulse generators because of the number of pulses involved. In addition, when the structures generate multiples pulses, they also increase the frequency of the design.

A recommended Synchronous Pulse generator is shown in Fig. 2.7.

In the above synchronous pulse generator design, the pulse width is always equal to the clock period. This pulse generator is predictable, can be verified with timing analysis, and is easily migrated to other architectures and is technology independent.

Similar to Fig. 2.7, Fig. 2.8 shows the pulse generator at the end of trigger input.

2.3.4 *Avoid Using Latches*

In digital logic, latches hold the value of a signal until a new value is assigned. Latches should be avoided whereas possible in the design and flip-flops should be used instead.

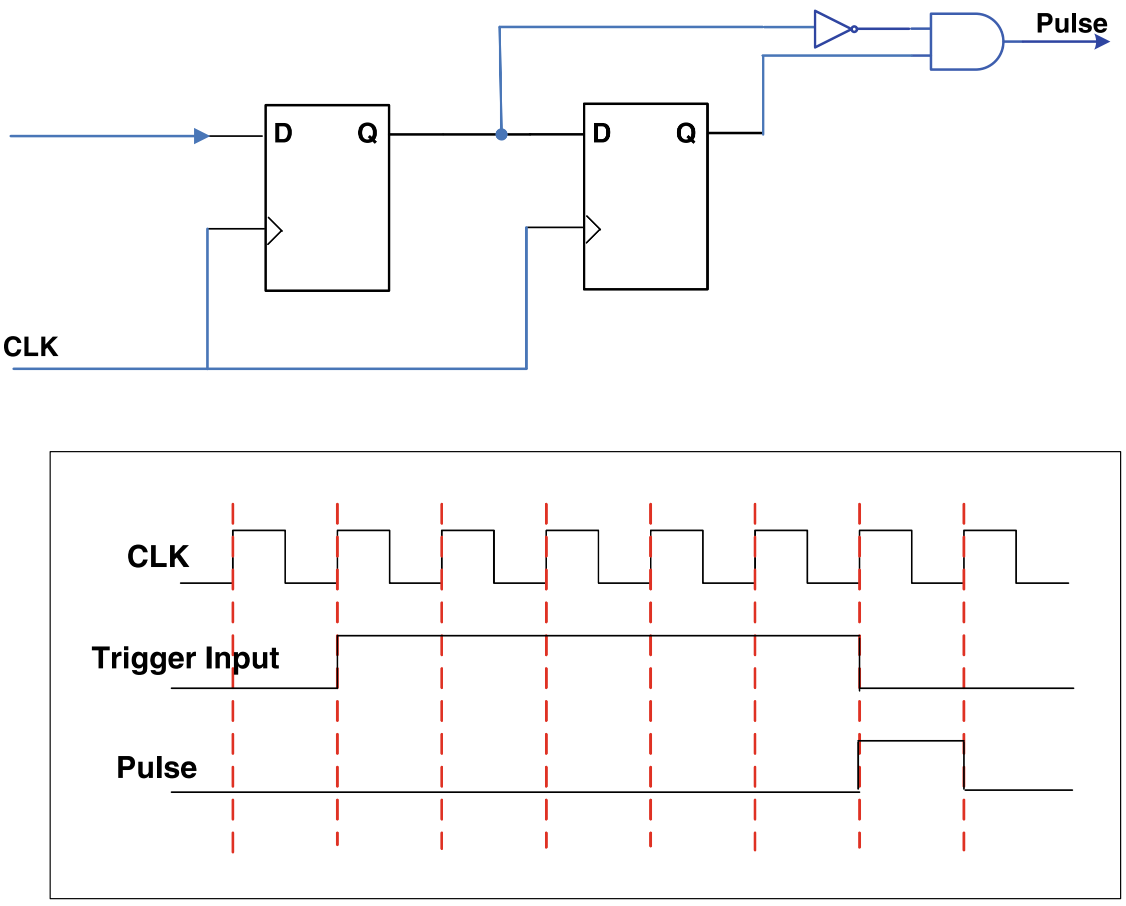


Fig. 2.8 Synchronous pulse generator circuit on end of trigger input

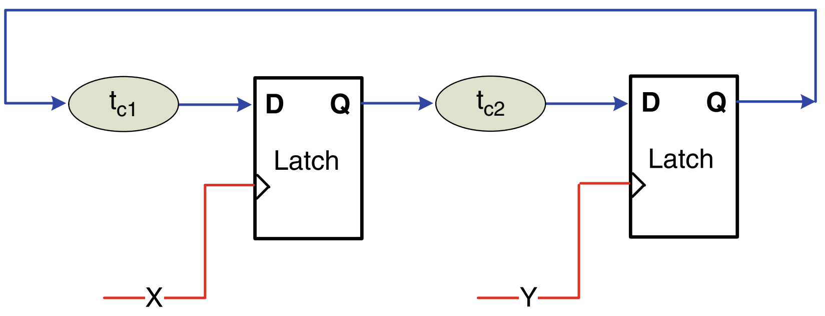


Fig. 2.9 Race conditions in latches

As shown in Fig. 2.9, if both the X and Y were to go high, and since these are level triggered, both the Latches would be enabled resulting in the circuit to oscillate.

Latches can cause various difficulties in the design. Although latches are memory elements like registers, they are fundamentally different. When a latch is in a feed-through mode, there is a direct path between the data input and the output. Glitches on the data input can pass to the output.

```
always @ (clk)
  begin
    if (clk)
      begin
        a = 1'b1;
        b = 1'b0;
      end
    else
        b = 1'b1;
  end
```

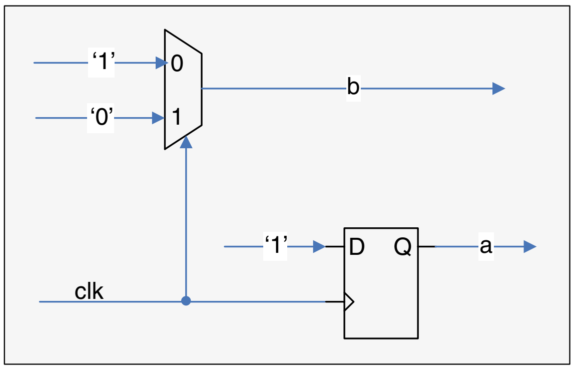


Fig. 2.10 Inferred latch due to incomplete ‘if else’ statement

Static timing analyzers typically make incorrect assumptions about latch transparency, and either find a false timing path through the input data pin or miss a critical path altogether. The timing for latches is also inherently ambiguous. When analyzing a design with a D latch, for example, the tool cannot determine whether you intended to transfer data to the output on the leading edge of the clock or on the trailing edge. In many cases, only the original designer knows the full intent of the design, which implies that another designer cannot easily migrate the same design or reuse the code.

Latches tend to make circuits less testable. Most design for test (DFT) and automatic test program generator (ATPG) tools do not handle latches very well.

Latches pose different challenge in FPGA designs as FPGA’s are register-intensive; therefore, designing with latches uses more logic and leads to lower performance than designing with registers.

Synthesis tools occasionally infer a latch in a design when one is not intended. Inferred latches typically result from incomplete “if” or “case” statements. Omitting the final “else” clause in an “if” or “case” statement can also generate a latch. Figure 2.10 shows a similar example.

As shown in Fig. 2.10, ‘b’ will be synthesized as straight combinational logic while a latch will be inferred on signal ‘a’.

A general rule for latch inferring is that if a variable is not assigned in all possible executions of an always statement (for example, when a variable is not assigned in all branches of an *‘if’* statement), then a latch is inferred.

Some FPGA architectures do not support latches. When such a design is synthesized, the synthesis tool creates a combinational feedback loop instead of a latch (as shown in Fig. 2.11).

Combinational feedback loops as shown above are capable of latching data but pose more problem then latches since they may violate setup, hold requirements which are difficult be determined, whereas latches does not have any setup time, hold time violations since they are level triggered.

Note: *The design should not contain any combinational feedback loops. They should be replaced by flip-flops or latches or be eliminated by fully enumerating RTL conditionals.*

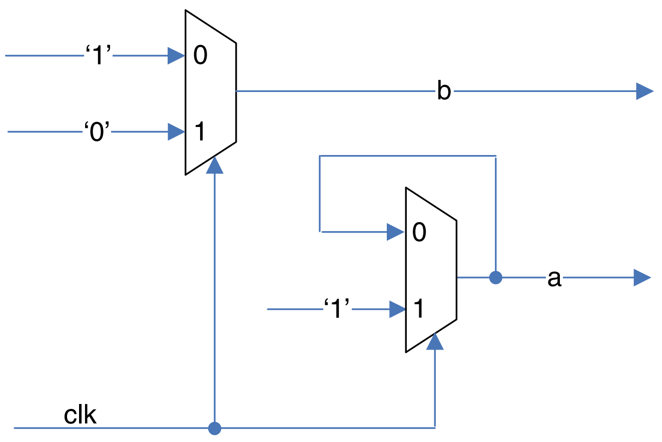


Fig. 2.11 Combinational loop implemented due to incomplete 'if else' statement

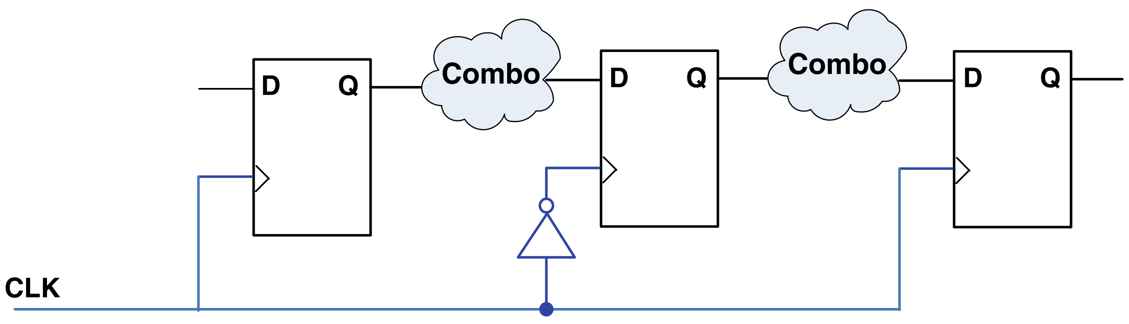


Fig. 2.12 Logic with double edged clocking

To conclude, this does not mean latches should never exist, we will see later how latches could be wonderful when it comes to cycle stealing or time borrowing to meet a critical path in a design.

2.3.5 *Avoid Using Double-Edged Clocking*

Double or Dual edged clocking is the method of data transfer on both the rising and falling edges of the clock, instead of just one or the other. The change allows for double the data throughput for a given clock speed.

Double edge output stage clocking is a useful way of increasing the maximum possible output speed from a design; however this violates the principle of Synchronous circuits and causes a number of problems.

Figure 2.12 shows a circuit triggered by both edges of clock.

Some of the problems encountered with Double Edged clocking are mentioned below:

- An asymmetrical clock duty cycle can cause setup and hold violations.
- It is difficult to determine critical signal paths.

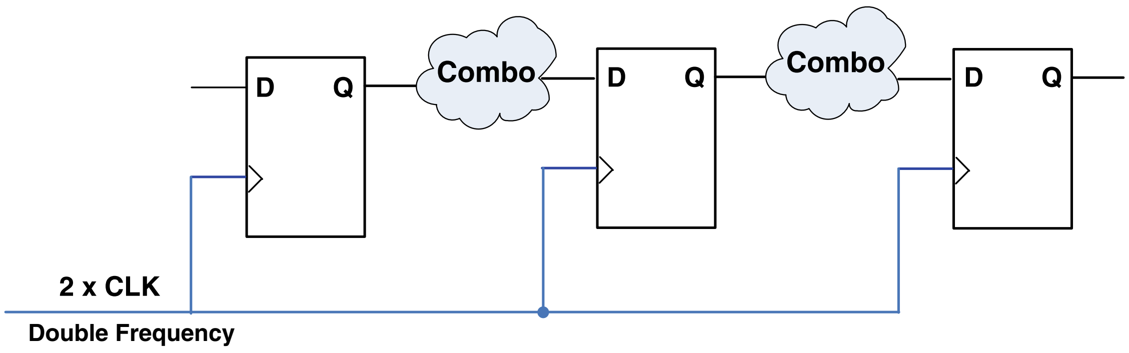


Fig. 2.13 Logic with single edged clocking

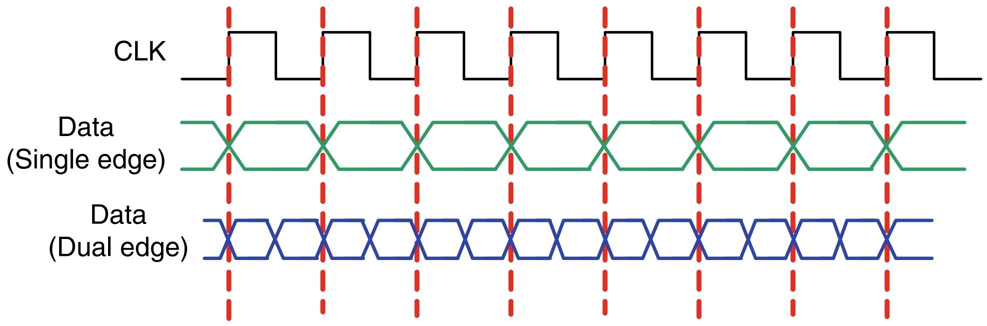


Fig. 2.14 Single/double edged data transfer

- Test methodologies such as scan-path insertion are difficult, as they rely on all flip-flops being activated on the same clock edge. If scan insertion is required in a circuit with double-edged clocking, multiplexers must be inserted in the clock lines to change to single-edged clocking in test mode.

Figure 2.13 shows the normal equivalent pipelined logic with single edge clocking. Note that this synchronous circuit requires a clock frequency that is double the one shown in Fig. 2.12.

Figure 2.14 shows the single transition and double transition clocked data transfer.

The green and blue signals represent data; the “hexagon” shapes are the traditional way of representing a signal that at any given time can be either a one or a zero.

In the circuit shown in Fig. 2.12, an asymmetrical clock duty cycle could cause setup and hold time violations, and a scan-path cannot easily be threaded through the flip-flops.

The above does not means that circuits with dual edge clocking should never be used unless there is an intense desire for higher performance/speed that cannot be met with the equivalent synchronous circuits as the latter comes with an additional overhead of complexity in DFT and verification.

2.3.5.1 Advantages of Dual Edge Clocking

The one constant in the PC world is the desire for increased performance. This in turn means that most interfaces are, over time, modified to allow for faster clocking, which leads to improved throughput. Many newer technologies in the PC world have gone a step beyond just running the clock faster. They have also changed the overall signaling method of the interface or bus, so that data transfer occurs not once per clock cycle, but twice or more.

There are other advantages of circuit operating on dual edge rather than the same synchronous circuit being fed with double the clock frequency. Whatever extent possible, interface designers do regularly increase the speed of the system clock. However, as clock speeds get very high, problems are introduced on many interfaces. Most of these issues are related to the electrical characteristics of the signals themselves. Interference between signals increases with frequency and timing becomes more "tight", increasing cost as the interface circuits must be made more precise to deal with the higher speeds.

The other advantage using double edged clocking is lower power consumptions as clock speeds are decreased by half and hence the system consumes less power than the equivalent synchronous circuits.

So to conclude system integrator should only use dual or double edged clocking unless the same desired performance cannot be met with the equivalent synchronous circuits.

2.4 Clocking Schemes

2.4.1 Internally Generated Clocks

A designer should avoid internally generated clocks, wherever possible, as they can cause functional and timing problems in the design, if not handled properly.

Clocks generated with combinational logic can introduce glitches that create functional problems and the delay due to the combinational logic can lead to timing problems. In a synchronous design, a glitch on the data inputs does not cause any issues and is automatically avoided as data is always captured on the edge of the clock and thus blocks the glitch. However, a glitch or a spike on the clock input (or an asynchronous input of a register) can have significant consequences.

Narrow glitches can violate the register's minimum pulse width requirements. Setup and hold times may also be violated if the data input of the register is changing when a glitch reaches the clock input. Even if the design does not violate timing requirements, the register output can change value unexpectedly and cause functional hazards elsewhere in the design.

Figure 2.15 shows the effect of using a combinational logic to generate a clock on a synchronous counter. As shown in the timing diagram, due to the glitch on the clock edge, the counters increments twice in the clock cycle shown.

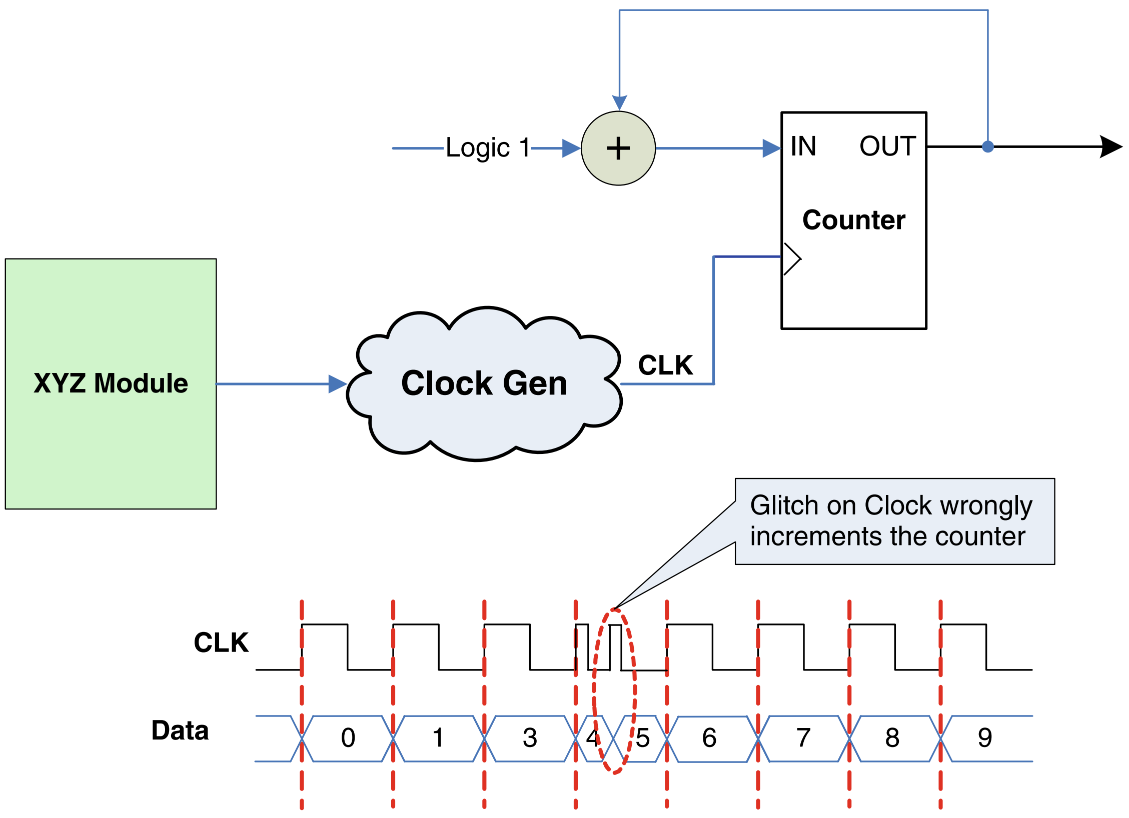


Fig. 2.15 Counter example for using combinational logic as a clock

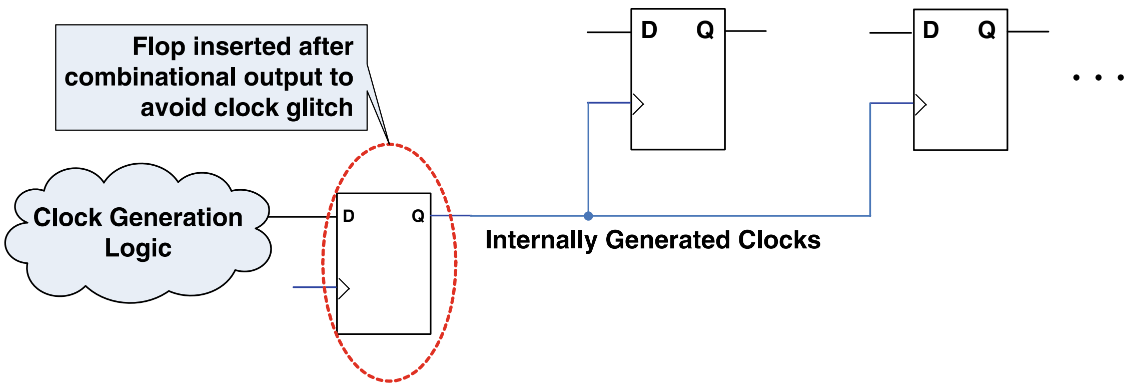

Flop inserted after combinational output to avoid clock glitch
Clock Generation Logic
D
Q
Internally Generated Clocks
D
Q
D
Q
. . .

Fig. 2.16 Recommended clock generation technique

This extra counting may create functional issues in the design where instead of counting the desired count, counter counts an additional count due to the glitch on the clock.

Note: *That for the sake of simplicity, it is assumed that the Counters Flops did not violate the setup/hold requirements on the data due to the glitch.*

A simple guideline to the above problem is to always use a registered output of the combinational logic before using it as a clock signal. This registering ensures that the glitches generated by the combinational logic are blocked on the data input of the register (Fig. 2.16).

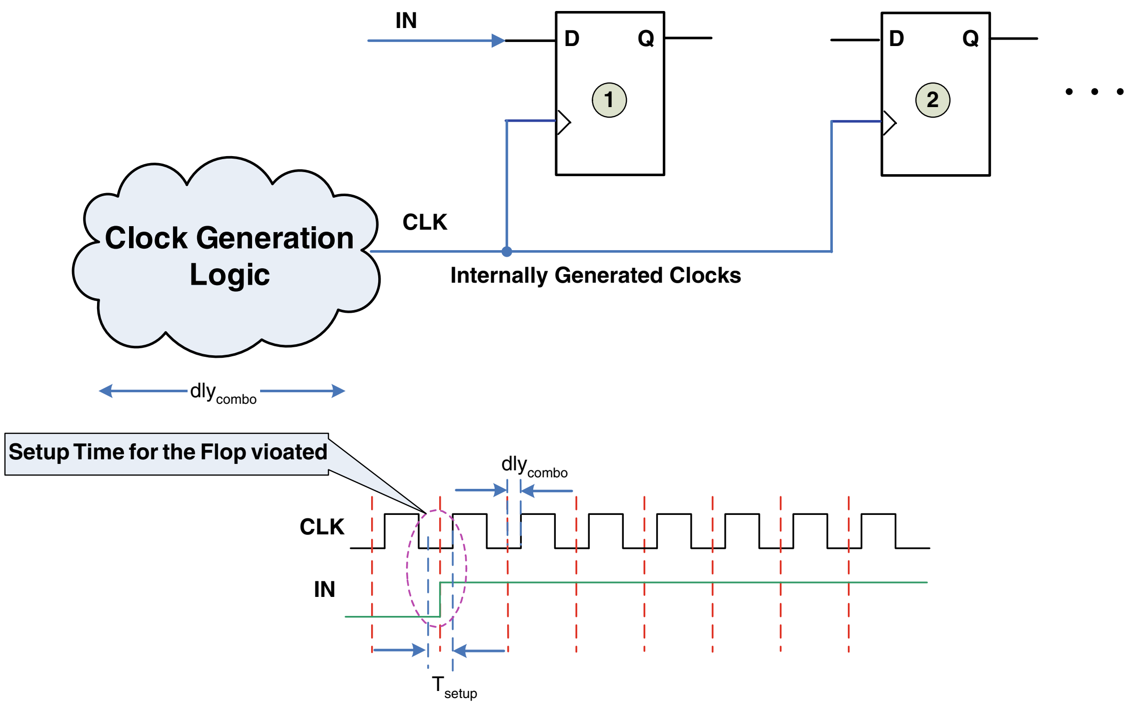


Fig. 2.17 Setup time violated due to skew of clock path

The combinational logic used to generate an internal clock also adds delays on the clock line. In some cases, logic delay on a clock line can result in a clock skew greater than the data path length between two registers. If the clock skew is greater than the data delay, the timing parameters of the register will be violated and the design will not function correctly.

Figure 2.17 shows a similar example where setup time on input "IN" is violated due to skew on the clock path.

Note: *Data path delay is assumed to be zero for simplicity.*

One solution to reduce the clock skew within the clock domain is by assigning the generated clock signal to one of the high-fanout and low-skew clock trees in the SoC. Using a low-skew clock tree can help reduce the overall clock skew for the signal.

2.4.2 Divided Clocks

Many designs require clocks created by dividing a master clock. Design should ensure that most of the clocks should come from the PLL. Using PLL circuitry will avoid many of the problems that can be introduced by asynchronous clock division logic. When using logic to divide a master clock, always use synchronous counters or state machines.

In addition, the design should ensure that registers always directly generate divided clock signals. Design should never decode the outputs of a counter or a state machine to generate clock signals; this type of implementation often causes glitches and spikes.

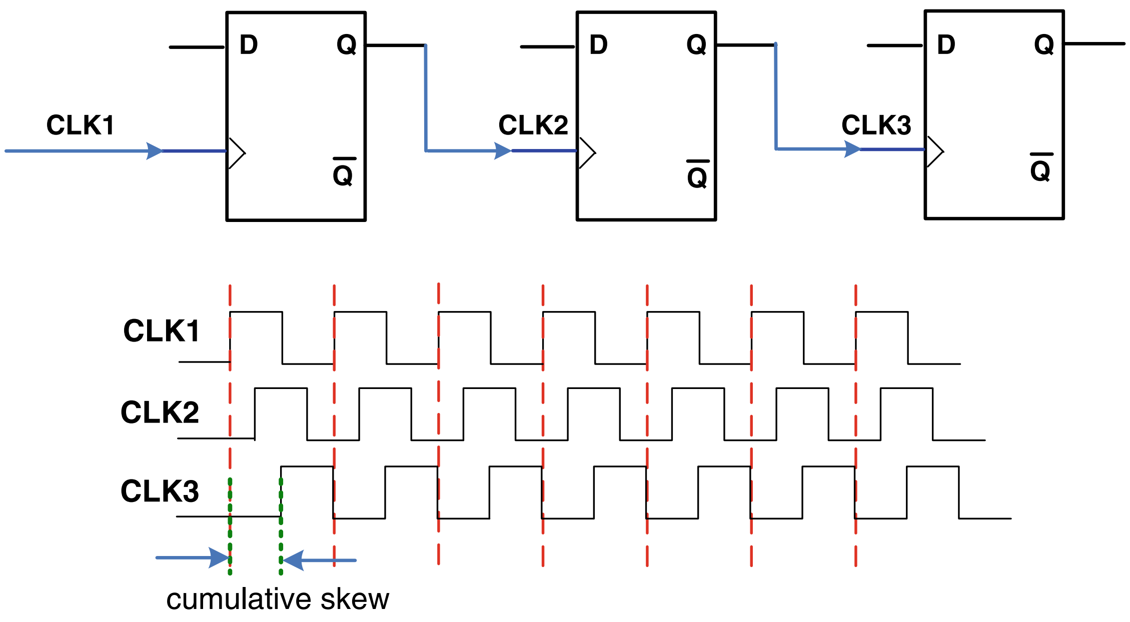


Fig. 2.18 Cascading effort in ripple counters

2.4.3 Ripple Counters

ASIC designers have often implemented ripple counters to divide clocks by a power of 2 because the counters use fewer gates than their synchronous counterparts. Ripple counters use cascaded registers, in which the output pin of each register feeds the clock pin of the register in the next stage (Fig. 2.18).

This cascading can cause problems because the counter creates a ripple clock at each stage. These ripple clocks pose another set of challenges for STA and synthesis tools. One should try to avoid these types of structures to ease verification effort.

Despite of all the challenges and problems with respect to using Ripple counters, these are quite handy in systems that eat power and can be good to reduce the peak power consumed by a logic or SoC.

Note: *Digital designers should consider using this technique in limited cases and under tight control.*

Refer Chap. 5 “Low power design” on more details analysis and techniques of using Ripple counters to save power consumption.

2.4.4 Multiplexed Clocks

Clock multiplexing can be used to operate the same logic function with different clock sources. Multiplexing logic of some kind selects a clock source as shown in Fig. 2.19.

For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

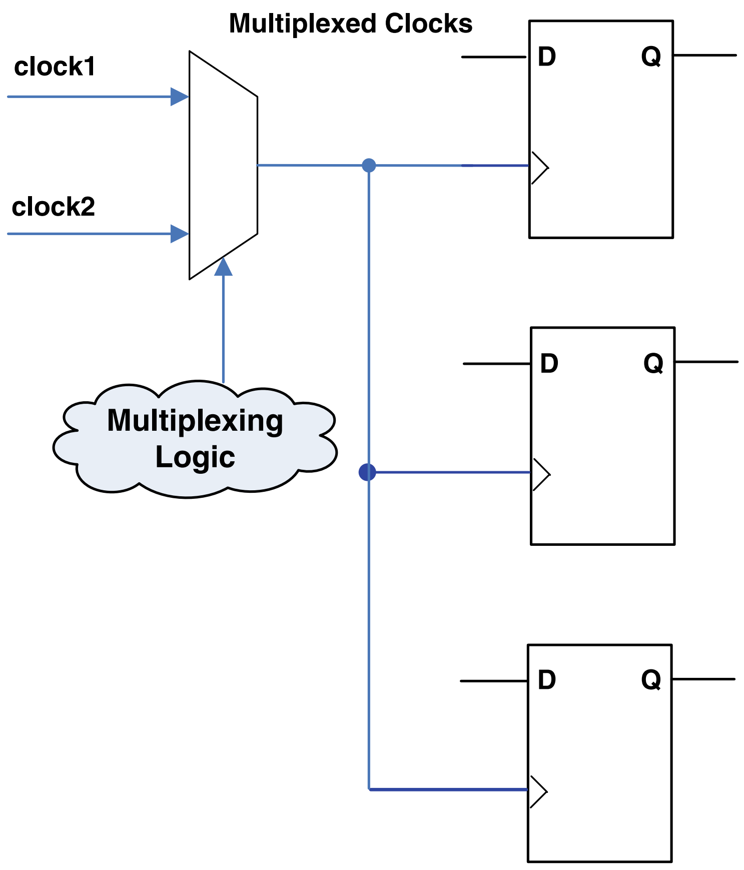


Fig. 2.19 Multiplexing logic and clock sources

Adding multiplexing logic to the clock signal can lead to some of the problems discussed in the previous sections, but requirements for multiplexed clocks vary widely depending on the application.

Clock multiplexing is acceptable if the following criteria are met:

- The clock multiplexing logic does not change after initial configuration
- The design bypasses functional clock multiplexing logic to select a common clock for testing purposes
- Registers are always in reset when the clock switches
- A temporarily incorrect response following clock switching has no negative consequences

If the design switches clocks on the fly with no reset and the design cannot tolerate a temporarily incorrect response of the chip, then one must use a synchronous design so that there are no timing violations on the registers, no glitches on clock signals, and no race conditions or other logical problems.

2.4.5 *Synchronous Clock Enables and Gated Clocks*

Gated clocks turn a clock signal on and off using an enable signal that controls some sort of gating circuitry. As shown in Fig. 2.20, when a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

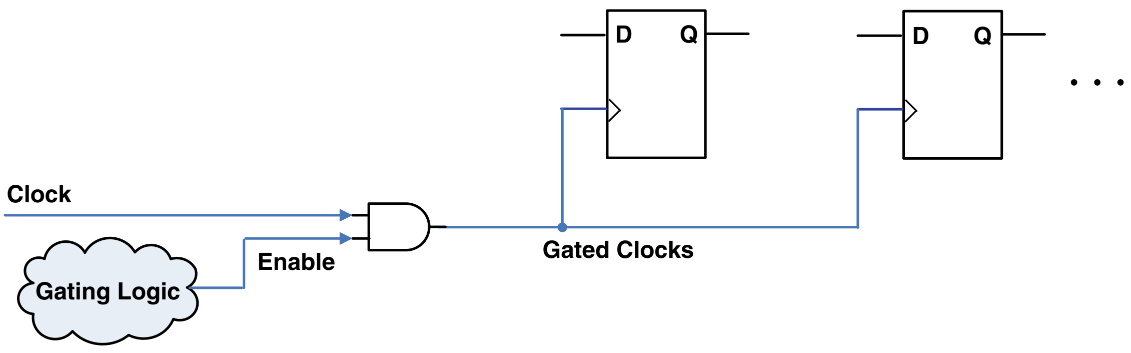


Fig. 2.20 Gated clock

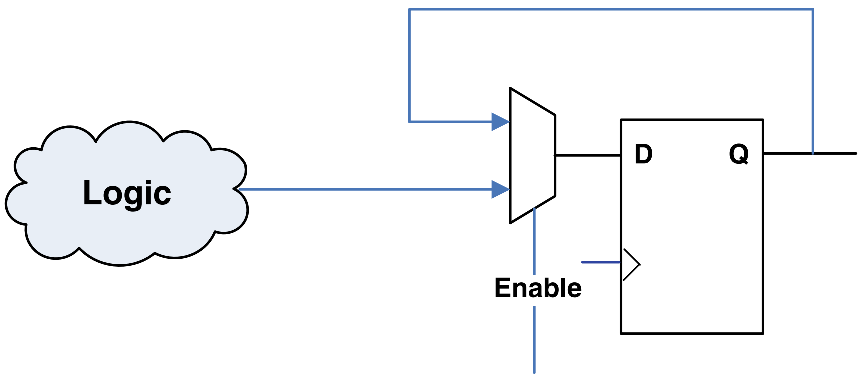


Fig. 2.21 Synchronous clock enable

Gated clocks can be a powerful technique to reduce power consumption. When a clock is gated both the clock network and the registers driven by it stop toggling, thereby eliminating their contributions to power consumption. However, gated clocks are not part of a synchronous scheme and therefore can significantly increase the effort required for design implementation and verification. Gated clocks contribute to clock skew and are also sensitive to glitches, which can cause design failure.

A clock domain can be turned off in a in a purely synchronous manner using a synchronous clock enable. However, when using a synchronous clock enable scheme, the clock tree keeps toggling and the internal circuitry of each Flip Flop remains active (although outputs do not change values), which does not reduce power consumption. A synchronous clock enable technique is shown in Fig. 2.21.

This Synchronous Clock Enable Clocking scheme does not reduce power consumption as much as gating the clock at the source because the clock network keeps toggling, but it will perform the same function as a gated clock by disabling a set of Flip Flops. As shown in Fig. 2.21, multiplexer in front of the data input of every Flip Flop either load new data or copy the output of the Flip Flop based on the Enable signal.

The next section is dedicated to efficient clock gating methodology that should be used where ever clocking gating is desired due to tight power specifications.

2.5 Clock Gating Methodology

In the traditional synchronous design style, the system clock is connected to the clock pin on every flip-flop in the design. This results in three major components of power consumption:

1. Power consumed by combinatorial logic whose values are changing on each clock edge (due to flops driving those combo cells).
2. Power consumed by flip-flops (this has non-zero value even if the inputs to the flip-flops, and therefore, the internal state of the flip-flops, is not changing).
3. Power consumed by the clock tree buffers in the design.

Gating the clock path substantially reduces the power consumed by a Flip Flop. Clock Gating can be done at the root of the clock tree, at the leaves, or somewhere in between.

Since the clock tree constitutes almost 50% of the whole chip power, it is always a good idea to generate and gate the clock at the root so that entire clock tree can be shut down instead of implementing the gating along the clock tree at the leaves.

Figure 2.22 shows an example of a clock gating for a three bit Counter.

The circuit is similar to the traditional implementation except that a clock gating element has been inserted into the clock network, which causes the flip-flops to be clocked only when the *INC* input is high. When the *INC* input is low, the flip-flops are not clocked and therefore retain the old data. This saves three multiplexers in front of the flip-flops which would had been there in case the gating was implemented by Synchronous Clock Enable as described in Fig. 2.21. This can result in significant area saving when wide banks of registers are being implemented.

2.5.1 Latch Free Clock Gating Circuit

The latch-free clock gating style uses a simple AND or OR gate (depending on the edge on which flip-flops are triggered) as shown in Fig. 2.23.

For the correct operation the circuit imposes a requirement that all enable signals be held constant from the active (rising) edge of the clock until the inactive (falling) edge of the clock to avoid truncating the generated clock pulse prematurely or generating multiple clock pulses (or glitches in clock) where one is required.

Figure 2.24 shows the case where generated clock is truncated prematurely when the above requirement is not satisfied.

This restriction makes the latch-free clock gating style inappropriate for our single-clock flip-flop based design.

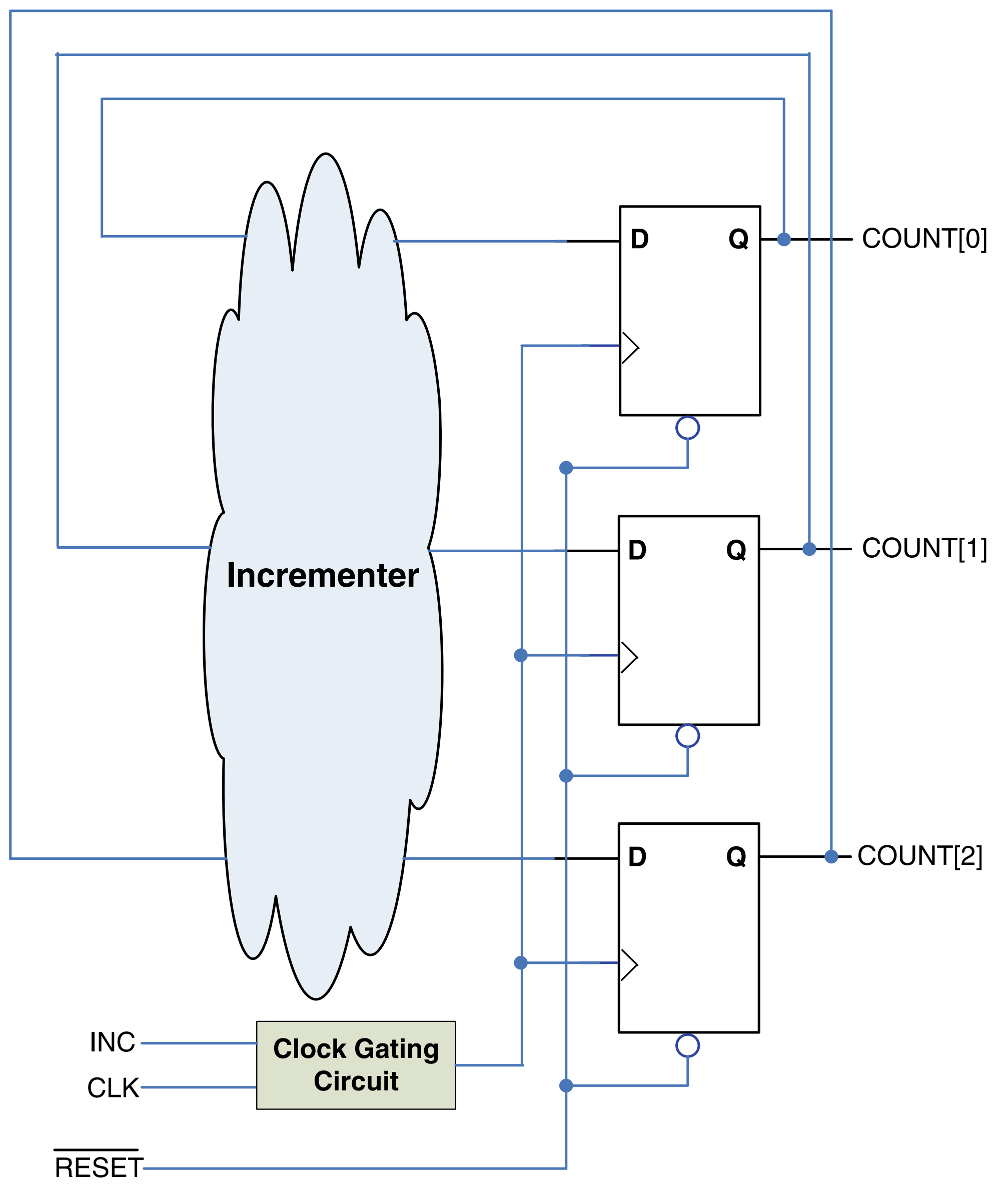


Fig. 2.22 Three bit counter with clock gating

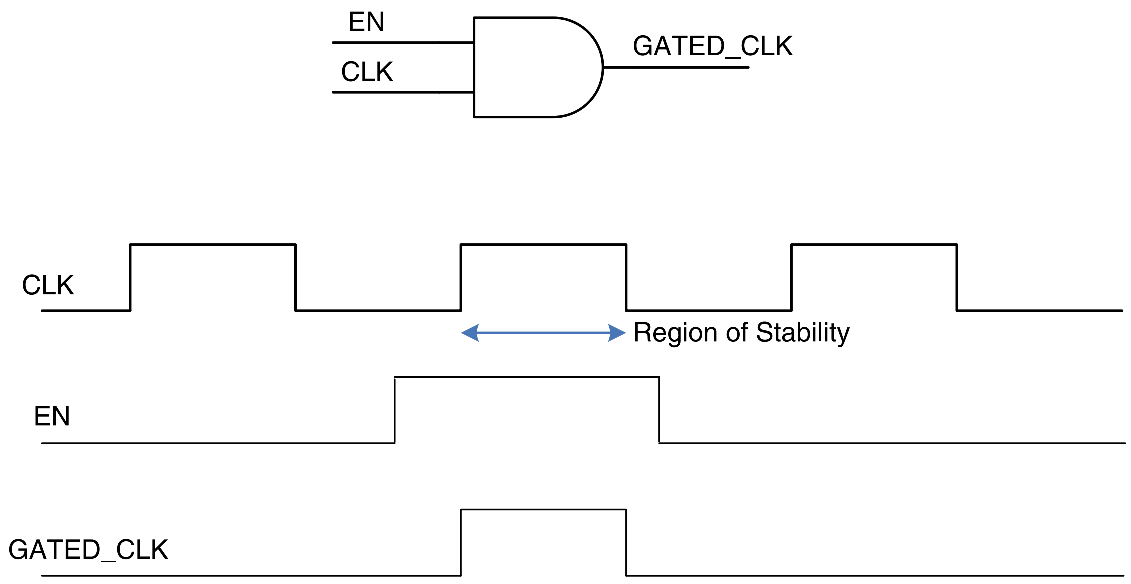


Fig. 2.23 Latch free clock gating circuit

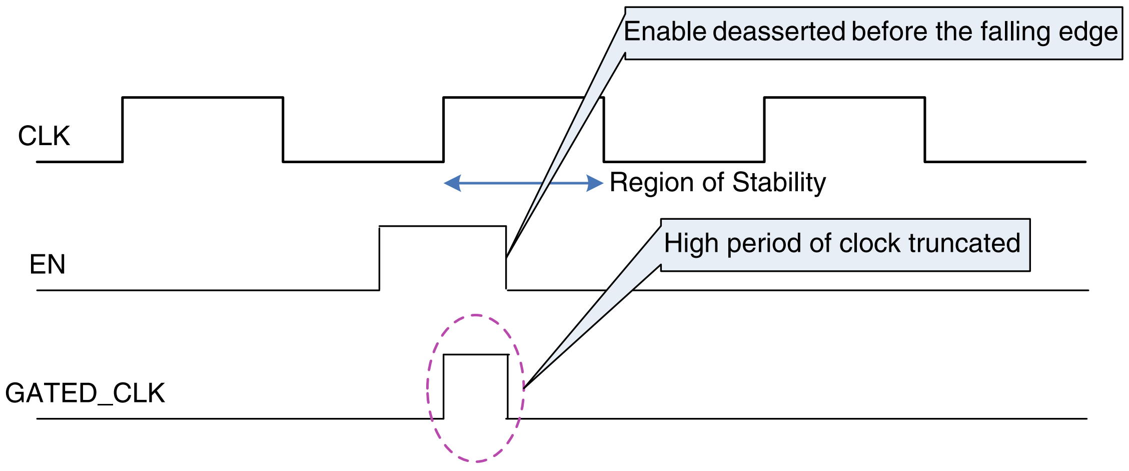

Enable deasserted before the falling edge
CLK
Region of Stability
EN
High period of clock truncated
GATED_CLK

Fig. 2.24 Generated clock terminated prematurely

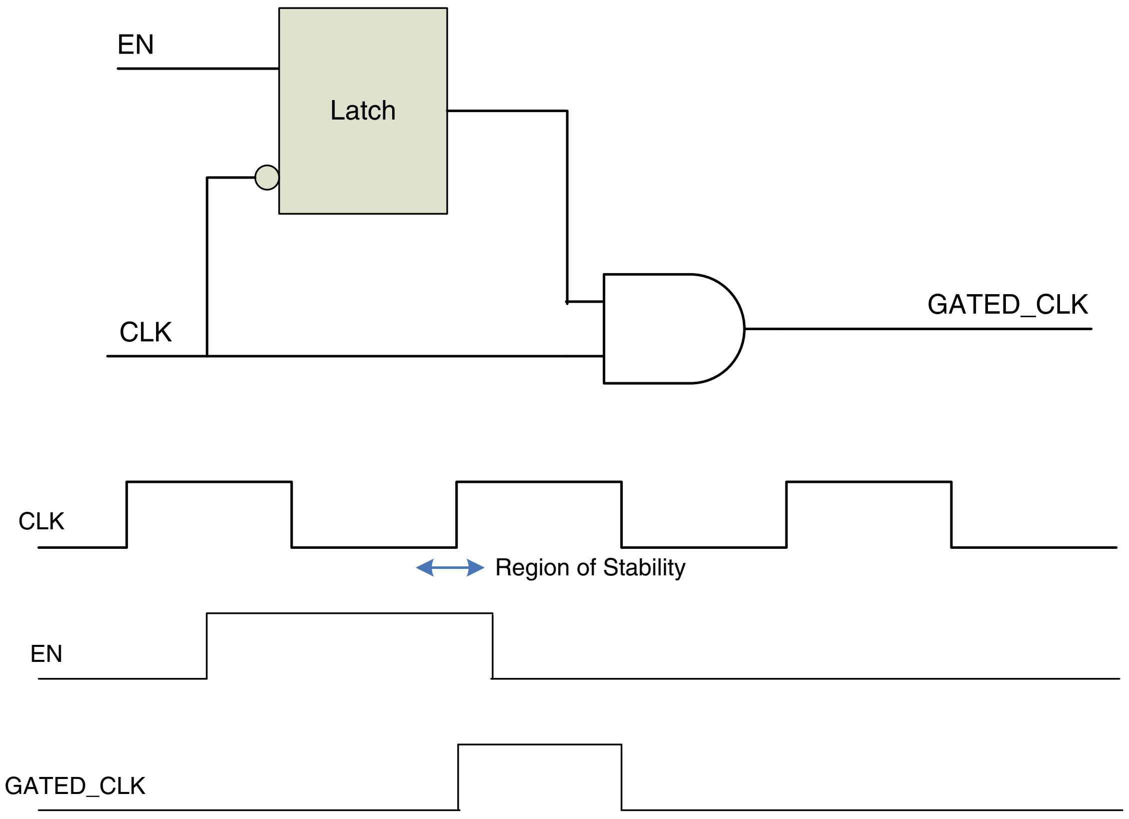


Fig. 2.25 Latch based clock gating circuit

2.5.2 *Latch Based Clock Gating Circuit*

The latch-based clock gating style adds a level-sensitive latch to the design to hold the enable signal from the active edge of the clock until the inactive edge of the clock, making it unnecessary for the circuit to itself enforce that requirement as shown in Fig. 2.25.

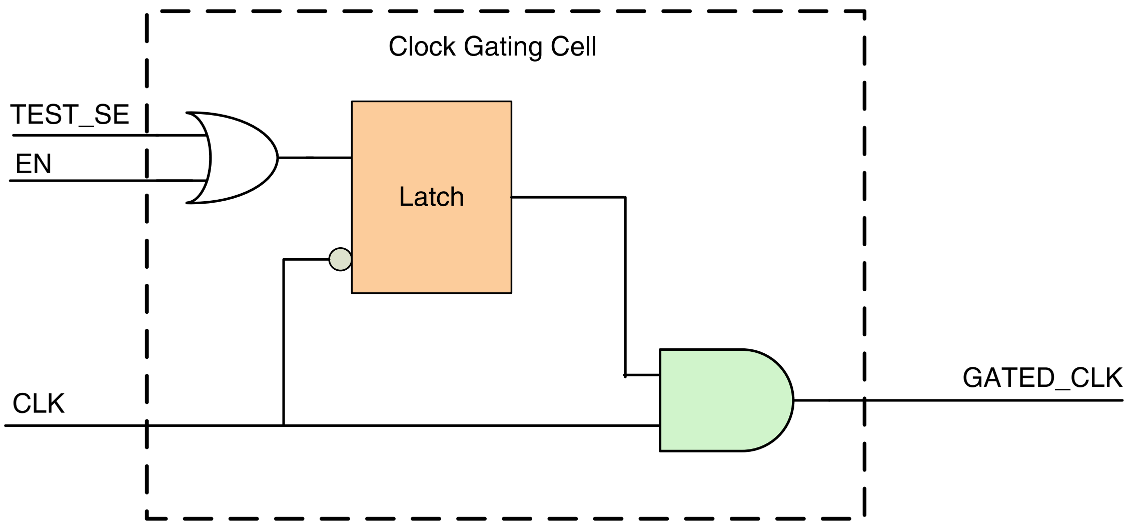


Fig. 2.26 Standard clock gating cell

Since the latch captures the state of the enable signal and holds it until the complete clock pulse has been generated, the enable signal need only be stable around the rising edge of the clock.

Using this technique, only one input of the gate that turns the clock on and off changes at a time, ensuring that the circuit is free from any glitches or spikes on the output.

Note: *Use an AND gate to gate a clock that is active on the rising edge. For a clock that is active on the falling edge, use an OR gate to gate the clock and register the enable with a positive edge-triggered Latch.*

When using this technique, special attention should be paid to the duty cycle of the clock and the delay through the logic that generates the enable signal, because the enable signal must be generated in half the clock cycle. This situation might cause problems if the logic that generates the enable command is particularly complex, or if the duty cycle of the clock is severely unbalanced. However, being careful with the duty cycle and logic delay may be acceptable compared with the problems created by other methods of gating clocks.

To ensure high manufacturing fault coverage, it is necessary to make sure the clock gating circuit is full controllable and observable to use within a scan methodology. A controllability signal which causes all flip-flops in the design to be clocked, regardless of the enable term value, can be added to allow the scan chain to shift information normally.

This signal can be ORed in with the enable signal before the latch and can be connected to either a test mode enable signal which is asserted throughout scan testing or to a scan enable signal which is asserted only during scan shifting.

The modified circuit is shown in Fig. 2.26. Most of the ASIC vendors do provide this “Clock Gating Cell” as a part of their standard library cell.

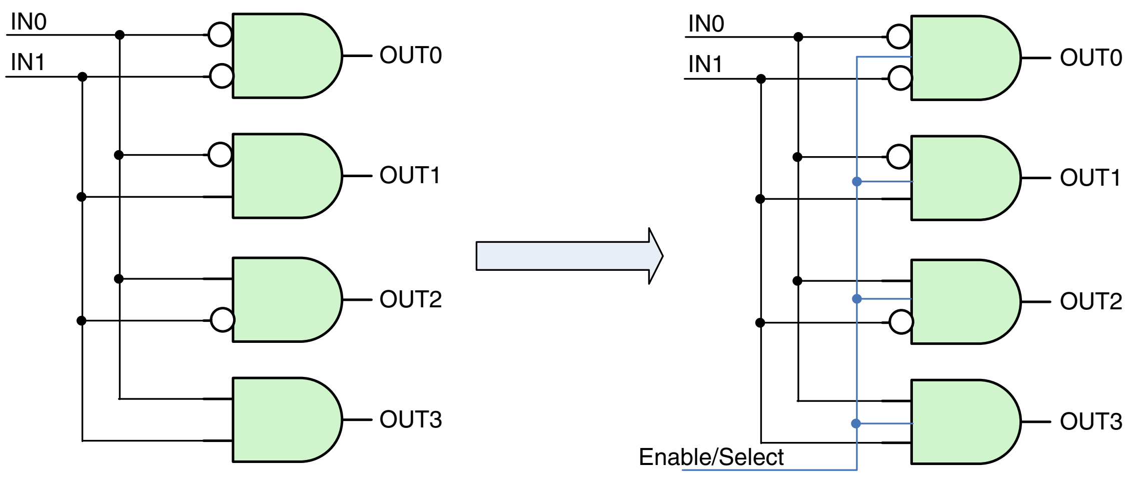


Fig. 2.27 Decoder with enable

2.5.3 *Gating Signals*

Effective power implementation can be achieved using gating signals for particular parts of the design. Similar to the concept of gating clock, signal gating reduces the transitions in clock free signals. The most common example is the decoder enable.

As part of an address decoding mechanism, signals used by other parts of the design may toggle as a reflection of activity in these parts. Switching activity on one input of the decoder will induce a large number of toggling gates. Controlling this with an enable or select signal prevents the propagation of their switching activity, even if the logic is slightly more complex (Fig. 2.27).

2.5.4 *Data Path Re-ordering to Reduce Switching Propagation*

Several data path elements, such as decoders or comparison operators, as well as "glitchy" logic may significantly contribute to power dissipation. The glitches, caused by late arrival signals or skews, propagate through other data path elements and logic until they reach a register. This propagation burns more power as the transitions traverse the logic levels. To reduce this wasted dissipation, designers need to rewrite the HDL code and shorten the propagation paths as much as possible. Figure 2.28 illustrates two implementations of the priority mux where the "glitchy" and "stable" conditions are ordered differently.

2.6 Reset Design Strategy

Many design issues must be considered before choosing a reset strategy for an ASIC design, such as whether to use synchronous or asynchronous resets, will every flip-flop receive a reset etc.

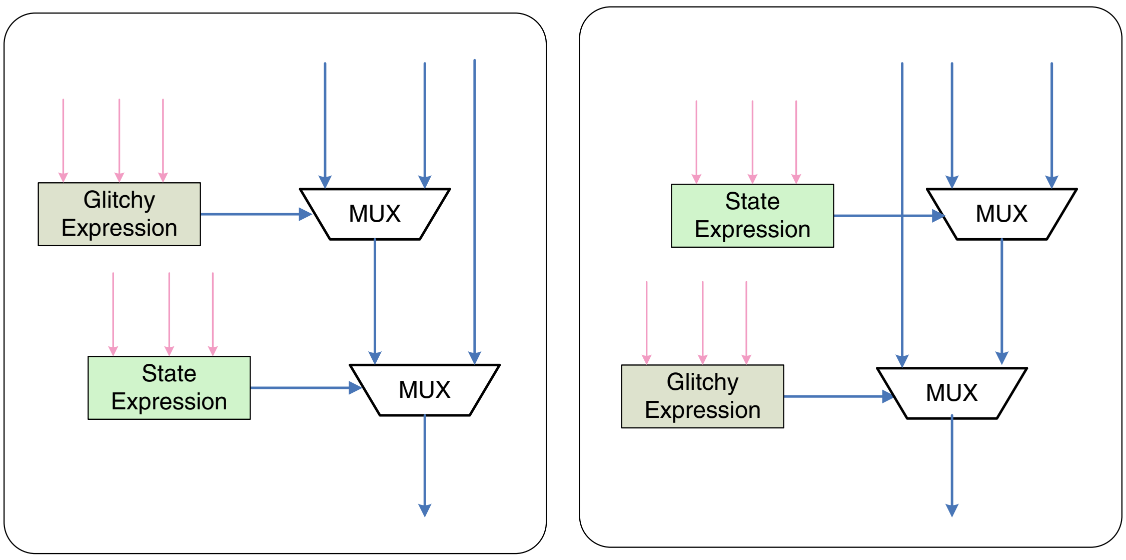


Fig. 2.28 Data path re-ordering to reduce switching propagation

The primary purpose of a reset is to force the SoC into a known state for stable operations. This would avoid the SoC to power on to a random state and get hanged. Once the SoC is built, the need for the SoC to have reset applied is determined by the system, the application of the SoC, and the design of the SoC. A good design guideline is to provide reset to every flip-flop in a SoC whether or not it is required by the system. In some cases, when pipelined flip-flops (shift register flip-flops) are used in high speed applications, reset might be eliminated from some flip-flops to achieve higher performance designs.

A design may choose to use either an Asynchronous or Synchronous reset or a mix of two. There are distinct advantages and disadvantages to use either synchronous or asynchronous resets and either method can be effectively used in actual designs. The designer must use an approach that is most appropriate for the design.

2.6.1 Design with Synchronous Reset

Synchronous resets are based on the premise that the reset signal will only affect or reset the state of the flip-flop on the active edge of a clock. In some simulators, based on the logic equations, the logic can block the reset from reaching the flip-flop. This is only a simulation issue, not a real hardware issue.

The reset could be a "late arriving signal" relative to the clock period, due to the high fanout of the reset tree. Even though the reset will be buffered from a reset buffer tree, it is wise to limit the amount of logic the reset must traverse once it reaches the local logic.

Figure 2.29 shows one of the RTL code for a loadable Flop with Synchronous Reset. Figure 2.30 shows the corresponding hardware implementation.

```
module load_syn_ff ( clk, in, out, load, rst_n);
  input clk, in, load, rst_n;
  output out;

  always @(posedge clk)
    if (!rst_n)
      out <= 1'b0; // sync reset
    else if (load)
      out <= in; // sync

endmodule
```

Fig. 2.29 Verilog RTL code for loadable flop with synchronous reset

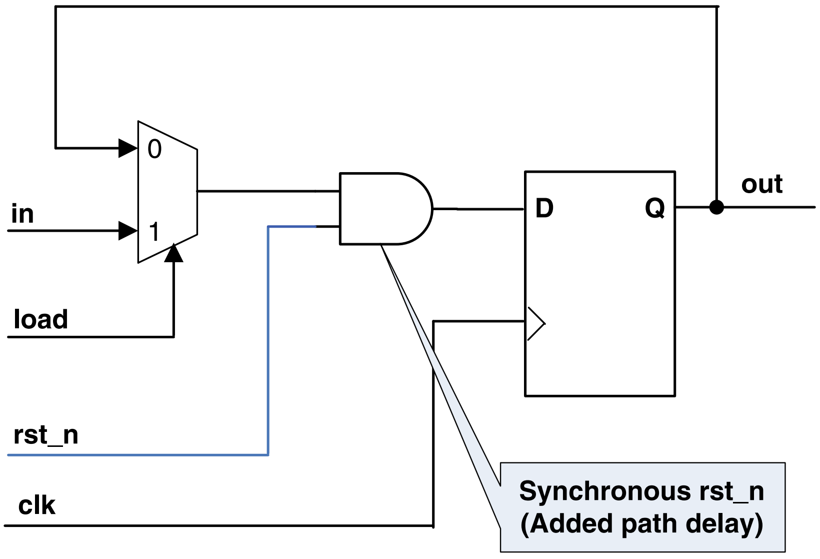


Fig. 2.30 Loadable flop with synchronous reset (hardware implementation)

One problem with synchronous resets is that the synthesis tool cannot easily distinguish the reset signal from any other data signal. The synthesis tool could alternatively have produced the circuit of Fig. 2.31.

Circuit shown in Fig. 2.31 is functionally identical to implementation shown in Fig. 2.30 with the only difference that reset AND gates are outside the MUX. Now, consider what happens at the start of a gate-level simulation. The inputs to both legs of the MUX can be forced to 0 by holding "*rst_n*" asserted low, however if "*load*" is unknown (X) and the MUX model is pessimistic, then the flops will stay unknown (X) rather than being reset. Note this is only a problem during simulation. The actual circuit would work correctly and reset the flop to 0.

Synthesis tools often provide compiler directives which tell the synthesis tool that a given signal is a synchronous reset (or set). The synthesis tool will "pull" this signal as close to the flop as possible to prevent this initialization problem from occurring.

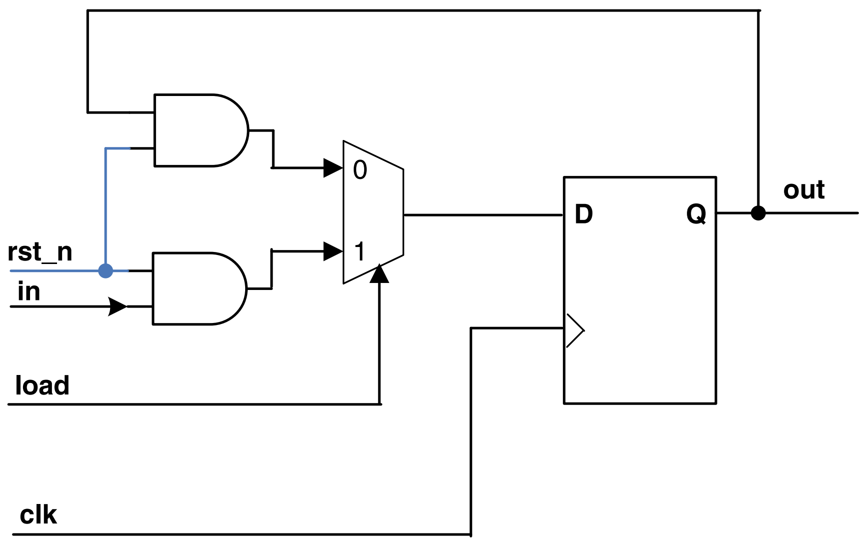


Fig. 2.31 Alternate implementation for loadable flop with synchronous reset

It would be recommended to add these directives to the RTL code from the start of project to avoid re-synthesizing the design late in the project schedule.

2.6.1.1 Advantages of Using Synchronous Resets

1. Synchronous resets generally insure that the circuit is 100% synchronous.
2. Synchronous reset logic will synthesize to smaller flip-flops, particularly if the reset is gated with the logic generating the Flop input.
3. Synchronous resets ensure that reset can only occur at an active clock edge. The clock works as a filter for small reset glitches.
4. In some designs, the reset must be generated by a set of internal conditions. A synchronous reset is recommended for these types of designs because it will filter the logic equation glitches between clocks.

2.6.1.2 Disadvantages of Using Synchronous Resets

Not all ASIC libraries have flip-flops with built-in synchronous resets. Since synchronous reset is just another data input, the reset logic can be easily synthesized outside the flop itself (as shown in Figs. 2.30 and 2.31).

1. Synchronous resets may need a pulse stretcher to guarantee a reset pulse width wide enough to ensure reset is present during an active edge of the clock. This is an issue that is important to consider when doing multi-clock design. A small counter can be used that will guarantee a reset pulse width of a certain number of cycles.
2. A potential problem exists if the reset is generated by combinational logic in the SoC or if the reset must traverse many levels of local combinational logic. During simulation, depending on how the reset is generated or how the reset is applied

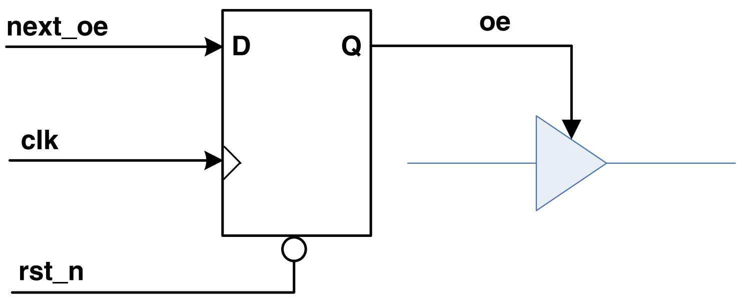


Fig. 2.32 Asynchronous reset for output enable

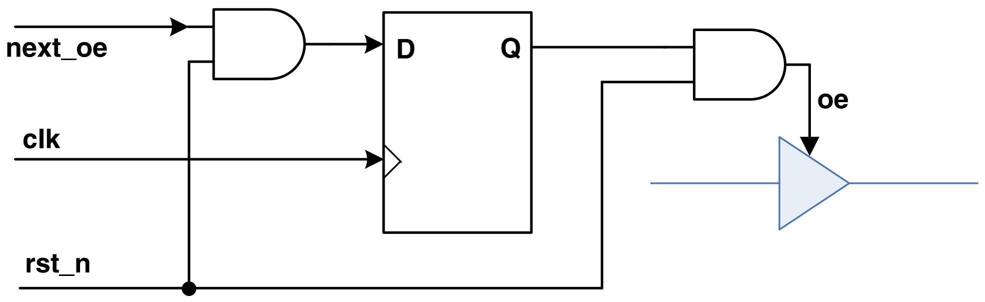


Fig. 2.33 Synchronous reset for output enable

to a functional block, the reset can be masked by X's. The problem is not so much what type of reset you have, but whether the reset signal is easily controlled by an external pin.

3. By its very nature, a synchronous reset will require a clock in order to reset the circuit. This may be a problem in some case where a gated clock is used to save power. Clock will be disabled at the same time during reset is asserted. Only an asynchronous reset will work in this situation, as the reset might be removed prior to the resumption of the clock.

The requirement of a clock to cause the reset condition is significant if the ASIC/FPGA has an internal tristate bus. In order to prevent bus contention on an internal tristate bus when a chip is powered up, the chip should have a power-on asynchronous reset as shown in Fig. 2.32.

A synchronous reset could be used; however you must also directly de-assert the tristate enable using the reset signal (Fig. 2.33). This synchronous technique has the advantage of a simpler timing analysis for the reset-to-HiZ path.

2.6.2 Design with Asynchronous Reset

Asynchronous reset flip-flops incorporate a reset pin into the flip-flop design. With an active low reset (normally used in designs), the flip-flop goes into the reset state when the signal attached to the flip-flop reset pin goes to a logic low level.

```
module load_asyn_ff ( clk, in, out, load, rst_n);
   input clk, in, load, rst_n;
   output out;

   always @(posedge clk or nededge rst_n)
     if (!rst_n)
       out <= 1'b0; // sync reset
     else if (load)
       out <= in; // sync

endmodule
```

Fig. 2.34 Verilog RTL code for loadable flop with asynchronous reset

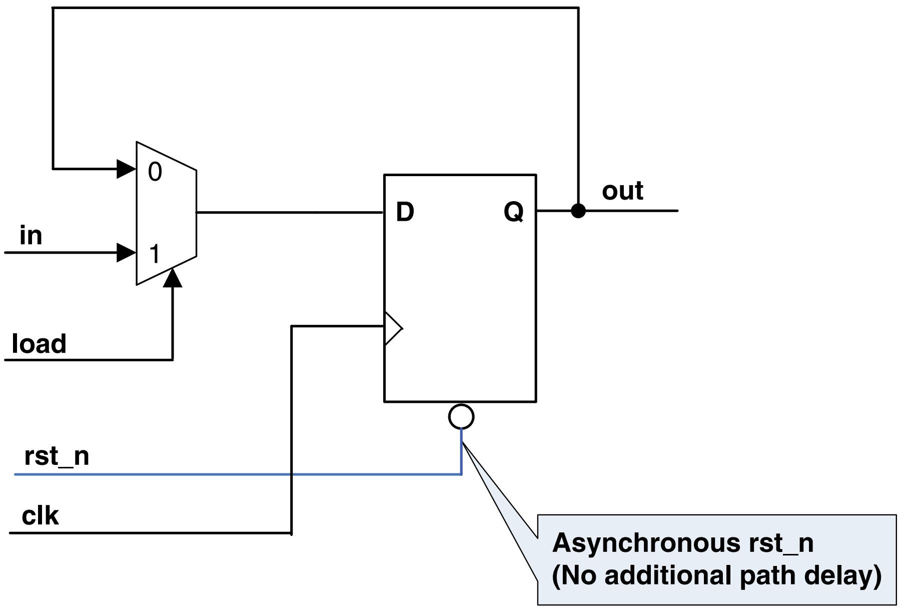


Fig. 2.35 Loadable flop with asynchronous reset (hardware implementation)

Figure 2.34 shows one of the RTL code for a loadable Flop with Asynchronous Reset. Figure 2.35 shows corresponding hardware implementation.

2.6.2.1 Advantages of Using Asynchronous Resets

1. The biggest advantage to using asynchronous resets is that, as long as the vendor library has asynchronously reset-able flip-flops, the data path is guaranteed to be clean. Designs that are pushing the limit for data path timing, cannot afford to have added gates and additional net delays in the data path due to logic inserted to handle synchronous resets. Using an asynchronous reset, the designer is guaranteed not to have the reset added to the data path (Fig. 2.35).

```
module dff_set_reset ( clk, in, out, rst_n, set_n);
  input clk, in, rst_n, set_n;
  output out;

always @(posedge clk or nededge rst_n or negedge set_n)
  if (!rst_n)
    out <= 1'b0; // async reset
  else if (!set_n)
    out <= 1'b1; // async set
  else
    out <= in;
endmodule
```

Fig. 2.36 Verilog RTL for the flop with async reset and async set

2. The most obvious advantage favoring asynchronous resets is that the circuit can be reset with or without a clock present. Synthesis tool tend to infer the asynchronous reset automatically without the need to add any synthesis attributes.

2.6.2.2 Disadvantages of Using Asynchronous Resets

1. For DFT, if the asynchronous reset is not directly driven from an I/O pin, then the reset net from the reset driver must be disabled for DFT scanning and testing [30].
2. The biggest problem with asynchronous resets is that they are asynchronous, both at the assertion and at the de-assertion of the reset. The assertion is a non issue, the de-assertion is the issue. If the asynchronous reset is released at or near the active clock edge of a flip-flop, the output of the flip-flop could go metastable and thus the reset state of the SoC could be lost.
3. Another problem that an asynchronous reset can have, depending on its source, is spurious resets due to noise or glitches on the board or system reset. Often glitch filters needs to be designed to eliminate the effect of glitches on the reset circuit. If this is a real problem in a system, then one might think that using synchronous resets is the solution.
4. The reset tree must be timed for both synchronous and asynchronous resets to ensure that the release of the reset can occur within one clock period. The timing analysis for a reset tree must be performed after layout to ensure this timing requirement is met. One approach to eliminate this is to use distributed reset synchronizer flip-flop.

2.6.3 Flip Flops with Asynchronous Reset and Asynchronous Set

Most synchronous designs do not have flop-flops that contain both an asynchronous set and asynchronous reset, but at times such a flip-flop is required.

Figure 2.36 shows the Verilog RTL for the Asynchronous Set/Reset Flip Flop.

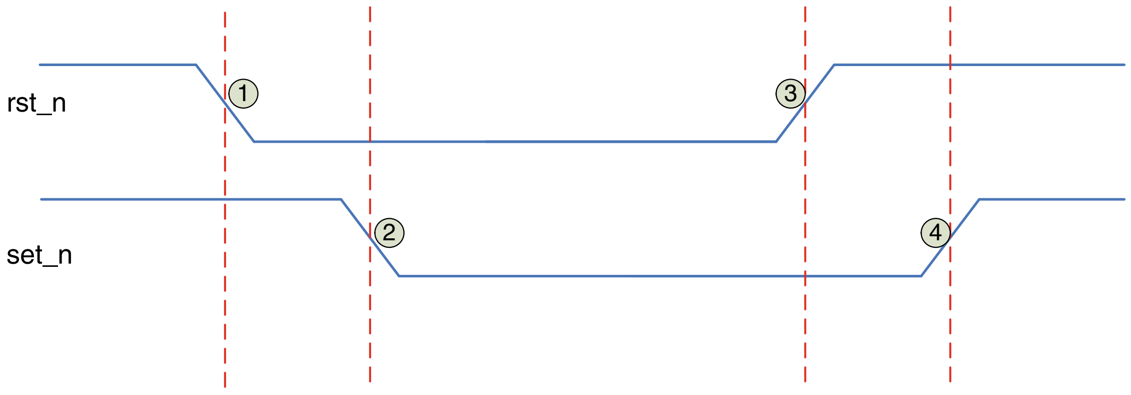


Fig. 2.37 Timing waveform for any asynchronous set/reset condition

```
// Add Compiler specific directive to
// ignore the following block during synthesis
always @(rst_n or set_n)
if (rst_n && !set_n) force q = 1;
else release q;
// End the compiler directive here
endmodule
```

Fig. 2.38 Simulation model for flop with asynchronous set/reset

Synthesis tool should be able to infer the correct flip flop with the asynchronous set/reset but this is not going to work in simulation. The simulation problem is due to the always block that is only entered on the active edge of the set, reset or clock signals.

If the reset becomes active, followed then by the set going active, then if the reset goes inactive, the flip-flop should first go to a reset state, followed by going to a set state (Timing waveform shown in Fig. 2.37).

With both these inputs being asynchronous, the set should be active as soon as the reset is removed, but that will not be the case in Verilog since there is no way to trigger the always block until the next rising clock edge. Always block will be only triggered for 1 and 2 events shown in Fig. 2.37 and would skip the events 3 and 4.

For those rare designs where reset and set are both permitted to be asserted simultaneously and then reset is removed first, the fix to this simulation problem is to model the flip-flop using self-correcting code enclosed within the correct compiler directives and force the output to the correct value for this one condition. The best recommendation here is to avoid, as much as possible, the condition that requires a flip-flop that uses both asynchronous set and asynchronous reset.

The code shown in Fig. 2.38 shows the fix that will simulate correctly and guarantee a match between pre- and post-synthesis simulations.

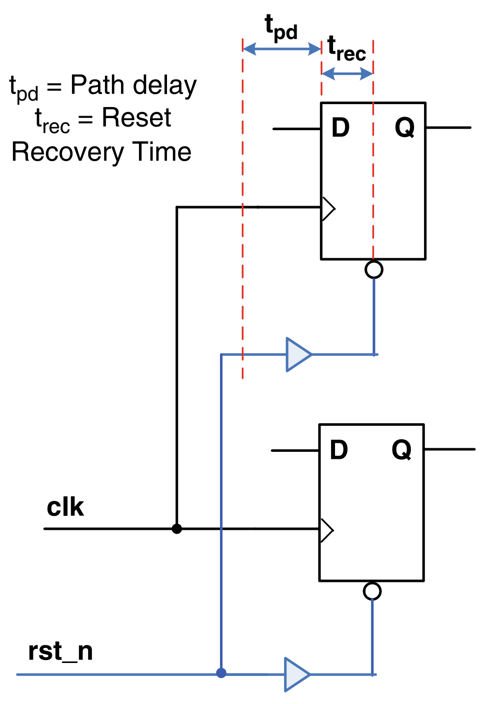


Fig. 2.39 Asynchronous reset removal recovery time problem

2.6.4 *Asynchronous Reset Removal Problem*

Releasing the Asynchronous reset in the system could cause the chip to go into a metastable unknown state, thus avoiding the reset all together. Attention must be paid to the release of the reset so as to prevent the chip from going into a metastable unknown state when reset is released. When a synchronous reset is being used, then both the leading and trailing edges of the reset must be away from the active edge of the clock.

As shown in Fig. 2.39, there are two potential problems when an asynchronous reset signal is de-asserted asynchronous to the clock signal.

1. Violation of reset recovery time. Reset recovery time refers to the time between when reset is de-asserted and the time that the clock signal goes high again. Missing a recovery time can cause signal integrity or metastability problems with the registered data outputs.
2. Reset removal happening in different clock cycles for different sequential elements. When reset removal is asynchronous to the rising clock edge, slight differences in propagation delays in either or both the reset signal and the clock signal can cause some registers or flip-flops to exit the reset state before others.

2.6.5 *Reset Synchronizer*

Solution to asynchronous reset removal problem described in Sect. 2.6.4 is to use a Reset Synchronizer. This is the most commonly used technique to guarantee correct reset removal in the circuits using Asynchronous Resets. Without a reset synchronizer, the usefulness of the asynchronous reset in the final system is void even if the reset works during simulation.

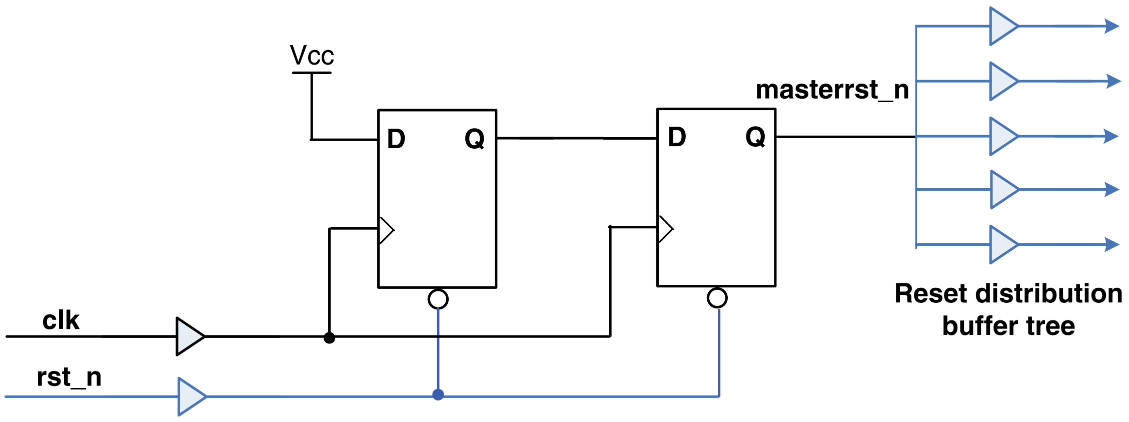


Fig. 2.40 Reset synchronizer block diagram

The reset synchronizer logic of Fig. 2.40 is designed to take advantage of the best of both asynchronous and synchronous reset styles.

An external reset signal asynchronously resets a pair of flip-flops, which in turn drive the master reset signal asynchronously through the reset buffer tree to the rest of the flip-flops in the design. The entire design will be asynchronously reset.

Reset removal is accomplished by de-asserting the reset signal, which then permits the d-input of the first master reset flip-flop (which is tied high) to be clocked through a reset synchronizer. It typically takes two rising clock edges after reset removal to synchronize removal of the master reset.

Two flip-flops are required to synchronize the reset signal to the clock pulse where the second flip-flop is used to remove any metastability that might be caused by the reset signal being removed asynchronously and too close to the rising clock edge.

Also note that there are no metastability problems on the second flip-flop when reset is removed. The first flip-flop of the reset synchronizer does have potential metastability problems because the input is tied high, the output has been asynchronously reset to a 0 and the reset could be removed within the specified reset recovery time of the flip-flop (the reset may go high too close to the rising edge of the clock input to the same flip-flop). This is why the second flip-flop is required.

The second flip-flop of the reset synchronizer is not subjected to recovery time metastability because the input and output of the flip-flop are both low when reset is removed. There is no logic differential between the input and output of the flip-flop so there is no chance that the output would oscillate between two different logic values.

The following equation calculates the total reset distribution time

$$T_{rst\_dis} = t_{clk-q} + t_{pd} + t_{rec}$$

where

t_{clk-q} = Clock to Q propagation delay of the second flip flop in the reset synchronizer
t_{pd} = Total delay through the reset distribution tree
t_{rec} = Recovery time of the destination flip flop

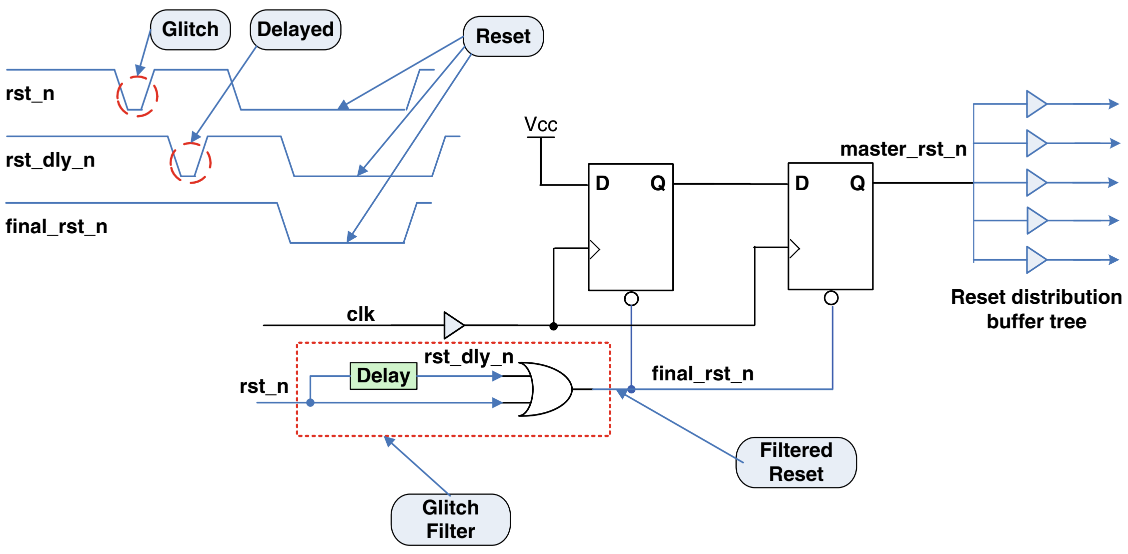


Fig. 2.41 Reset glitch filtering

2.6.6 Reset Glitch Filtering

Asynchronous Reset are susceptible to glitches, that means any input wide enough to meet the minimum reset pulse width for a flip-flop will cause the flip-flop to reset. If the reset line is subject to glitches, this can be a real problem. A design may not have a very high frequency sampling clock to detect small glitch on the reset; this section presents an approach that will work to filter out glitches [30]. This solution requires a digital delay to filter out small glitches. The reset input pad should also be a Schmidt triggered pad to help with glitch filtering. Figure 2.41 shows the reset glitch filter circuit and the timing diagram.

In order to add the delay, some vendors provide a delay hard macro that can be hand instantiated. If such a delay macro is not available, the designer could manually instantiate the delay into the synthesized design after optimization. A second approach is to instantiated a slow buffer in a module and then instantiated that module multiple times to get the desired delay. Many variations could expand on this concept.

Since this approach uses delay lines, one of the disadvantages is that this delay would vary with temperature, voltage and process. Care must be taken to make sure that the delay meets the design requirements across all PVT corners.

2.7 Controlling Clock Skew

Difference in clock signal arrival times across the chip is called clock skew. It is a fundamental design principle that timing must satisfy register setup and hold time requirements. Both data propagation delay and clock skew are parts of these

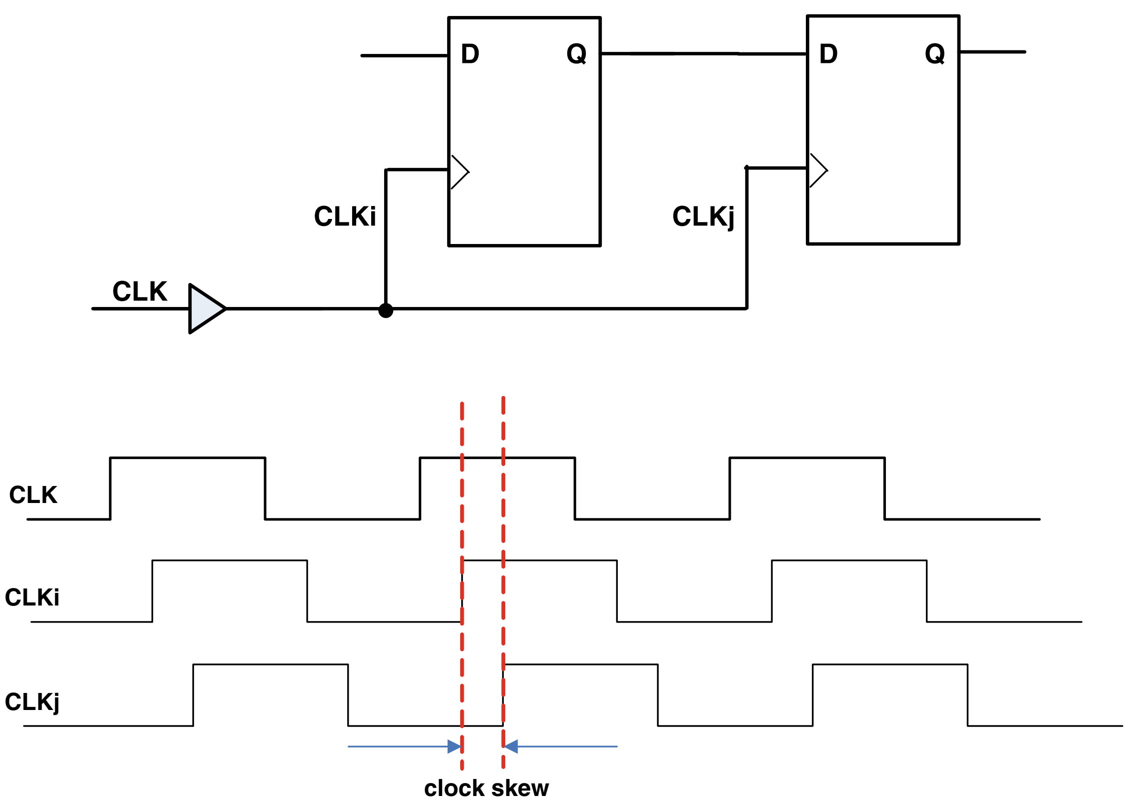


Fig. 2.42 Clock skew in two sequentially adjacent flip-flops

calculations. Clocking sequentially-adjacent registers on the same edge of a high-skew clock can potentially cause timing violations or even functional failures. Probably this is one of the largest sources of design failure in an ASIC.

Figure 2.42 shows an example of clock skew for two sequentially adjacent flip-flops.

Given two sequentially-adjacent flops, F_i and F_j, and an equi-potential clock distribution network, the clock skew between these two flops is defined as

$$\text{Tskew}_{i,j} = T_{c_i} - T_{c_j}$$

where Tc_i and Tc_j are the clock delays from the clock source to the Flops F_i and F_j, respectively.

2.7.1 Short Path Problem

The problem of short data paths in the presence of clock skew is very similar to hold-time violations in flip-flops. The problem arises when the data propagation delay between two adjacent flip-flops is less than the clock skew.

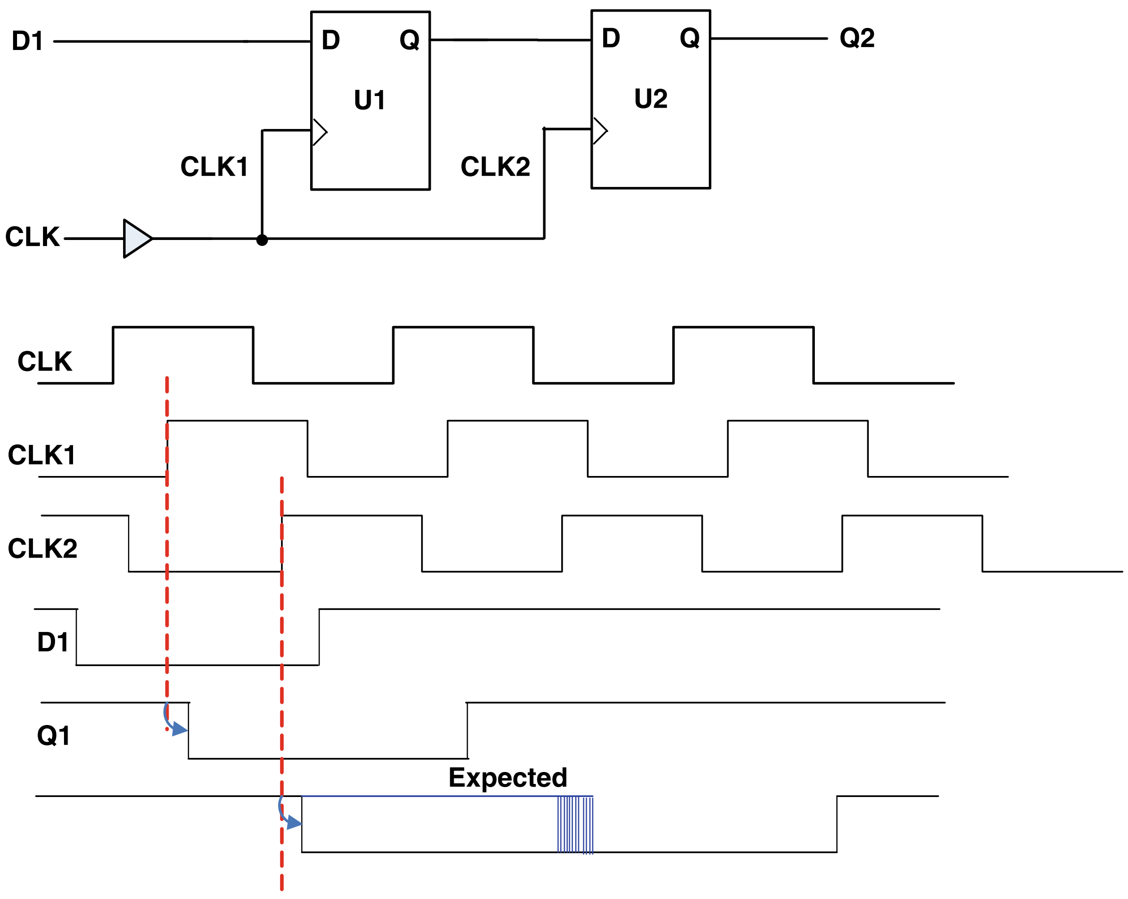


Fig. 2.43 Circuit with a short path problem

Figure 2.43 shows a circuit with timings to illustrate a short-path problem.

Since the same clock edge arrives at the second flip-flop later than the new data, the second flip-flop output switches at the same edge as the first flip-flop and with the same data as the first flip-flop. This will cause U2 to shift the same data on the same edge as U1, resulting in a functional error.

2.7.2 Clock Skew and Short Path Analysis

As mentioned earlier, clock skew and short-path problems emerge when the data propagation path delay between two sequentially adjacent flip-flops is less than the clock skew between the two. Figure 2.44 shows the general diagram of the delay blocks in a sample circuit [33].

The delays in Fig. 2.44 are as follows:

- T_{cq1}: The clock to out delay of the first flip-flop
- T_{rdq1}: The propagation delay from the output of the first flip-flop to the input of the second one
- T_{ck2}: The clock arrival time at the second flip-flop minus the clock arrival time at the first flip-flop

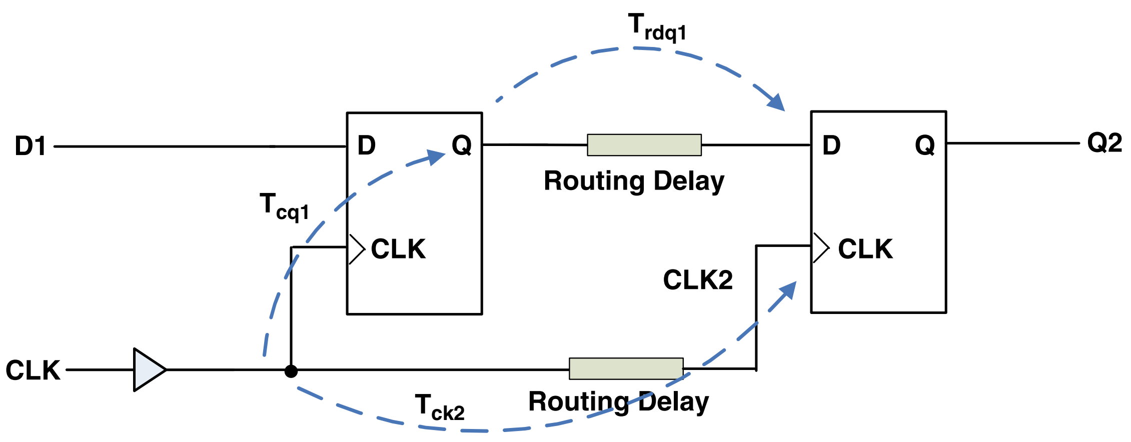
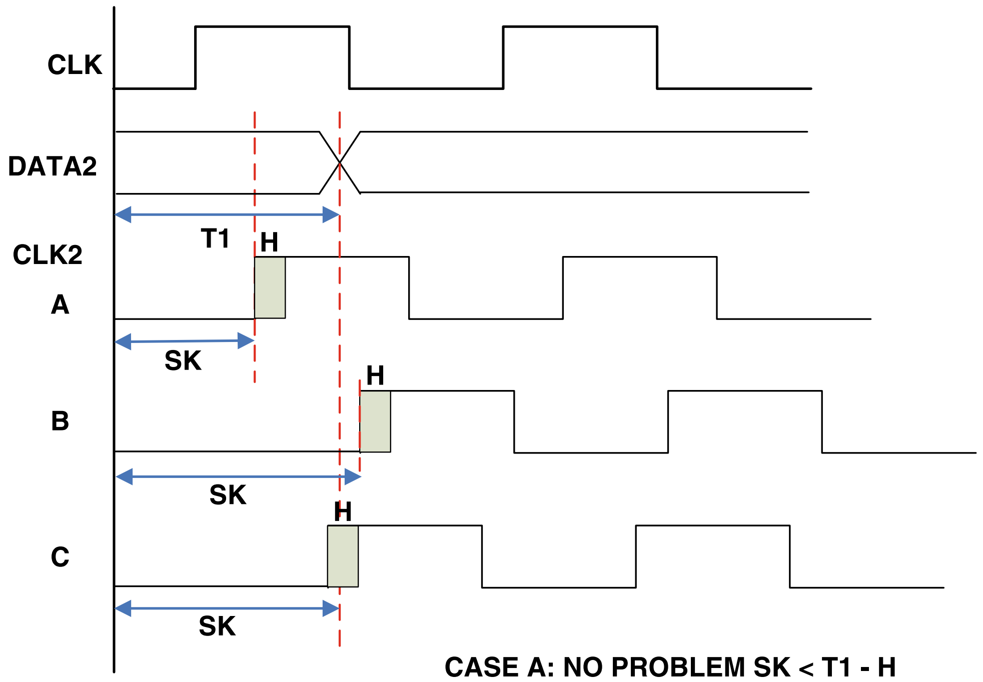


Fig. 2.44 General delay blocks in a simple circuit



CASE A: NO PROBLEM SK < T1 - H

CASE B: PROBLEM SK > T1 - H

CASE C: PROBLEM SK > T1 - H

Fig. 2.45 Illustration of short path problem

The short-path problem will definitely emerge in this circuit if

$$T_{ck2} > T_{cq1} + T_{rdq1} - T_{HOLD2}$$

where T_{HOLD2} is the hold-time requirement of the sink flip-flop.
The regions are illustrated in Fig. 2.45.

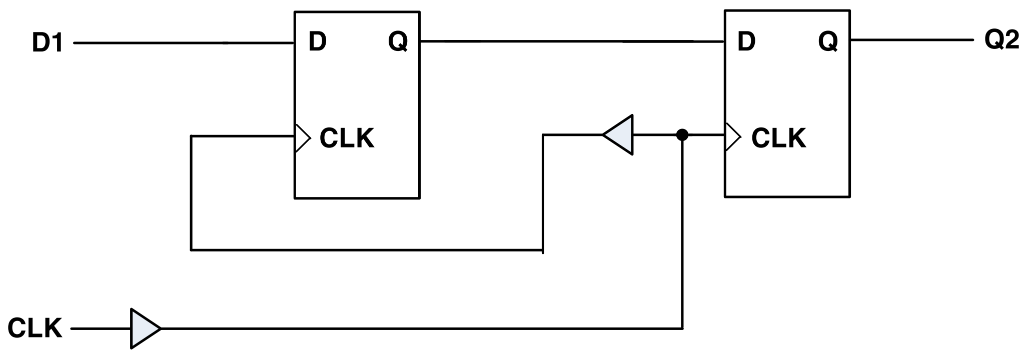


Fig. 2.46 Clock reversing methodology

Therefore, in order to identify the paths with the problem, the user needs to extract the clock skew (e.g. T_{ck2}) and the short-path delays (e.g. $T_{cq1} + T_{rdq1} - T_{HOLD2}$).

2.7.3 Minimizing Clock Skew

Reducing the Clock skew to the minimum is the best approach to reduce the risk of short-path problems. Maintaining the clock skew at a value less than the smallest Flop-to-Flop delay in the design will improve the robustness of the design against any short-path problems.

The following sections are a few well-known design techniques to make designs more robust against clock skew.

2.7.3.1 Adding Delay in Data Path

As Shown in Fig. 2.44, by increasing the Routing Delay in the data path (T_{rdq1}) that eventually increases the total delay of the data path to a value greater than the clock skew, will eliminate the short path problem.

The amount of the inserted delay in the data path should be large enough so that the data path delay becomes sufficiently greater than the clock skew.

2.7.3.2 Clock Reversing

Clock reversing is another approach to get around the problem of short data paths and clock skew. In this technique Clock is applied in the reverse direction with respect to data so that clock skew is automatically eliminated.

The receiving Flop will clock in the transmitting (source) value before the transmitting register receives its clock edge. Figure 2.46 shows a simple example of implementing the clock reversing approach.

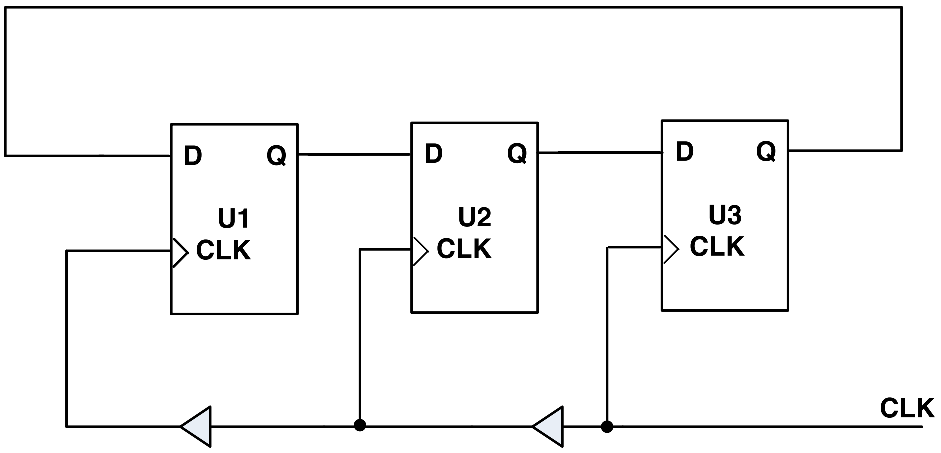


Fig. 2.47 Clock reversing in a circular structure

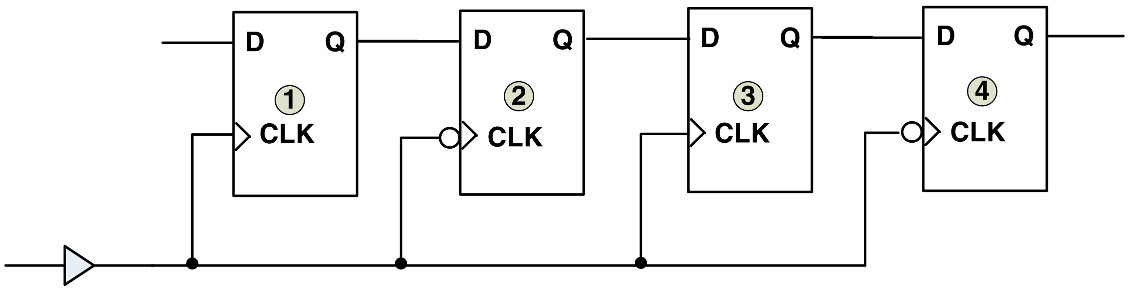


Fig. 2.48 Alternate edge clocking

As shown when sufficient delay is inserted, the receiving Flop will receive the active-clock edge before the source Flop. This improves the Hold time at the expense of Setup Time.

The clock reversing method will not be effective in circular structures such as Johnson counters and Linear Feedback Shift Registers (LFSRs), because it is not possible to define the Sink Flop explicitly. Figure 2.47 shows an example of a circular structure with clock reversing interconnection. As shown, short-path problem exists between flip-flops U1 and U3.

2.7.3.3 Alternate Phase Clocking

One of the known methodologies to avoid clock skew issues is alternate-phase clocking. The following sections mentions few design techniques of alternate phase clocking.

Clocking on Alternate Edges

In this method, sequentially adjacent Flops are clocked on the opposite edges of the clock as shown in Fig. 2.48.

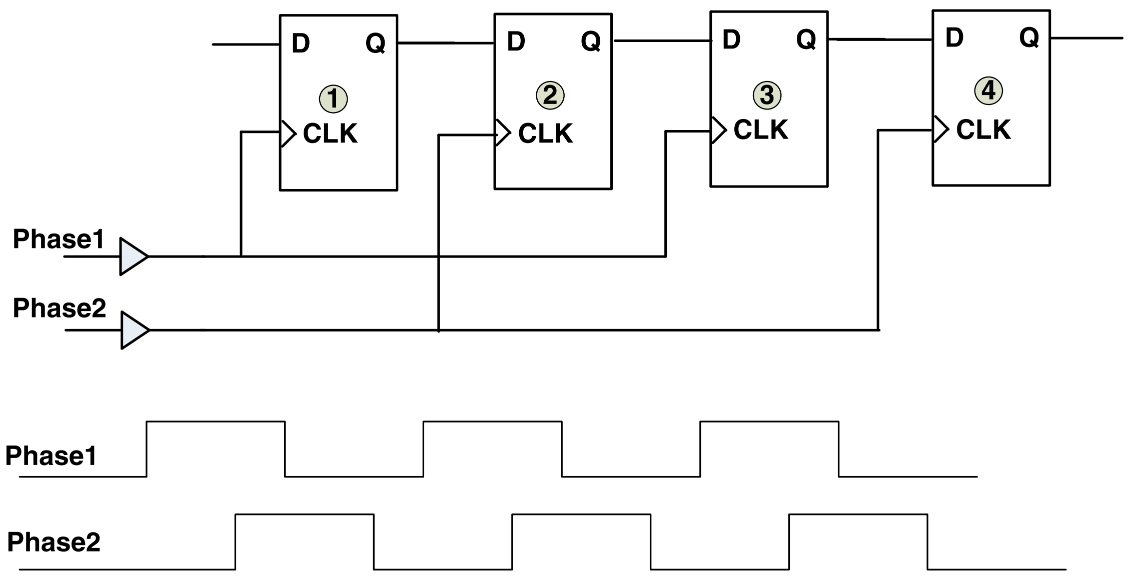


Fig. 2.49 Alternate phase clocking

As shown this method provides a short path-clock skew margin of about one half clock cycle for clock skew.

Clocking on Alternate Phases

Figure 2.49 shows a set of adjacent Flops, which are alternately clocked on two different phases of the same clock. In this case, between each two adjacent Flops, there is a safety margin approximately equal to the phase difference of the two phases.

The user should note that the usage of alternate-phase clocking may require completely different clock constraints on the original clock signal. For example, in the case of clocking on alternate edges, the new constraint on the clock frequency will be half the original frequency since the adjacent Flops are clocked on opposite edges of the same clock cycle.

Ripple Clocking Structure

In a ripple structure, each Flop output drives the next Flop clock port just like the way a Ripple counter is implemented. Here the sink Flop will not clock unless the source Flop toggled as shown in Fig. 2.50.

As shown in Fig. 2.50, the output of each counter flop drives the clock port of the next Flop instead of its data input port. This will eliminate the clock skew since the Flops do not toggle on the same clock. The first Flop is clocked on the positive edge of the CLK signal and the second- and third-stage Flops are clocked on the positive edge of the output of the previous Flop.

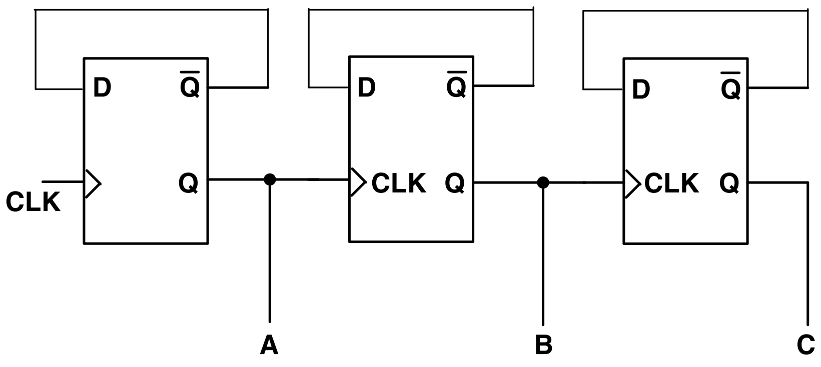


Fig. 2.50 Three bit ripple down-counter

Different techniques as mentioned above may be used to minimize clock skew and avoid short path problems depending on the design complexity and methodology being used.

2.7.3.4 Balancing Trace Length

Techniques described in the previous section are more on design techniques that may be planned much before the final project phase. Of course alternative to the above, Designers may choose to balance the trace length for low skew clock drivers. Apart from merely providing equal traces on all clock nets, the same termination strategy should be used on each trace by placing the same load at the end of the line. This would make sure trace lengths are properly balanced.

Below are some of the guidelines that should be followed:

1. Pay close attention to the specifications for input-to-output delay on the drivers.
2. Use the same drivers at every level of the clock hierarchy.
3. Balance the nominal trace delays at each level.
4. Use the same termination strategy on each line.
5. Balance the loading on each line, even if that means adding dummy capacitors to one branch to balance out loads on the other branches.

References

1. Mohit Arora, Prashant Bhargava, Amit Srivastava, *Optimization and Design Tips for FPGA/ ASIC(How to make the best designs)*, DCM Technologies, SNUG India, 2002
2. Application Note, ASIC design guidelines, Atmel Corporation, 1999
3. Cummings CE, Sunburst Design, Inc.; Mills D, LCDM Engineering (2002) Synchronous resets? Asynchronous resets? I am so confused! How will I ever know which to use? SNUG, San Jose
4. Application Note, Clock skew and short paths timing, Actel Corporation, 2004

5.3 Attachment 3

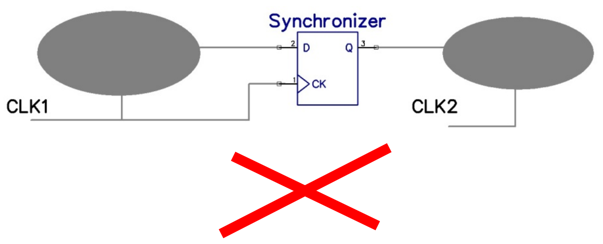
ECE 428   Programmable ASIC Design

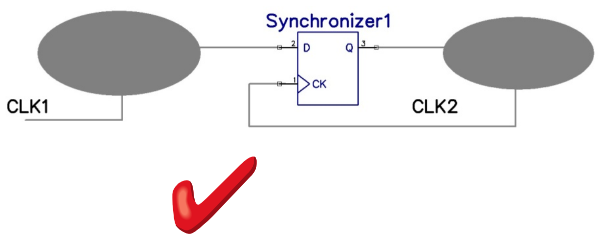
Issues in ASIC Design with Multi-Clock Domains

Haibo Wang
ECE Department
Southern Illinois University
Carbondale, IL 62901

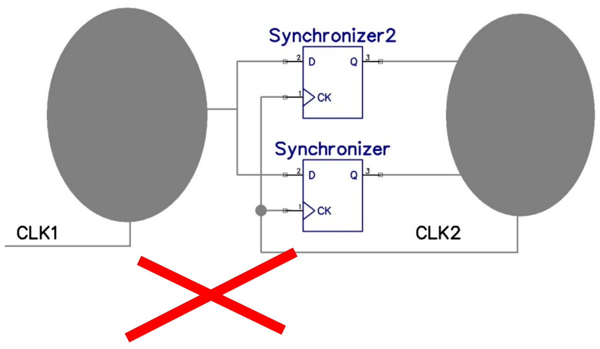
Mistakes in the Use of Synchronizer

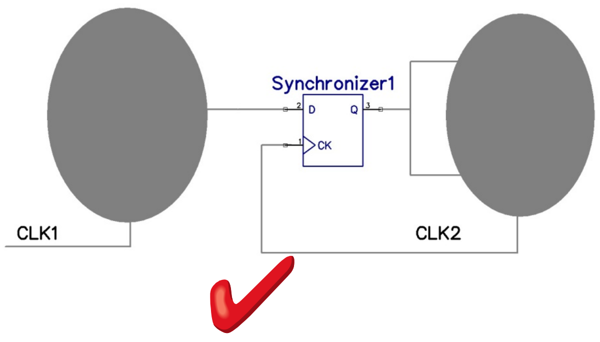
- ❑ Does not use the correct clock





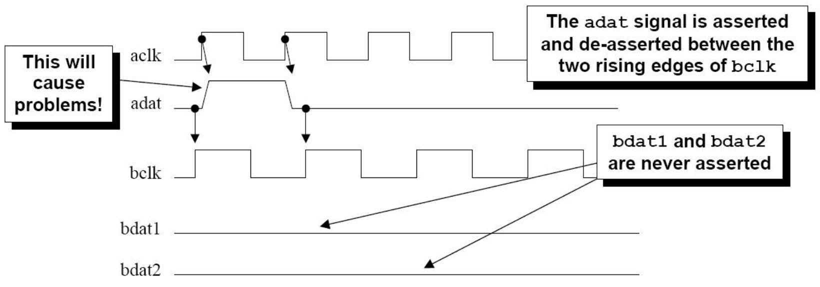
- ❑ Synchronize the same signal more than once



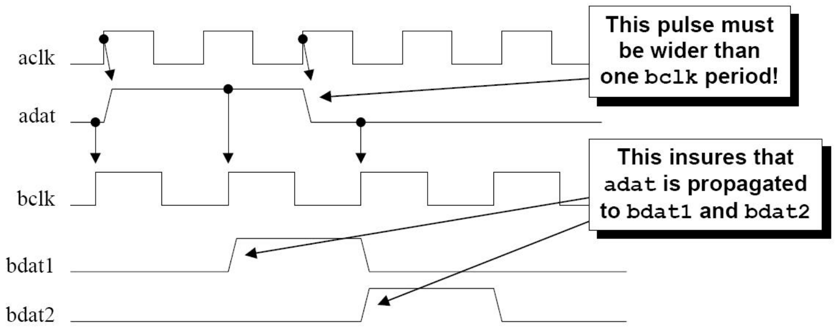


Passing a control signal from slow clock domain to fast clock domain

- ❑ The control signal may not be captured by the slow clock domain


This will cause problems!
aclk
adat
bclk
bdat1
bdat2
The adat signal is asserted and de-asserted between the two rising edges of bclk
bdat1 and bdat2 are never asserted

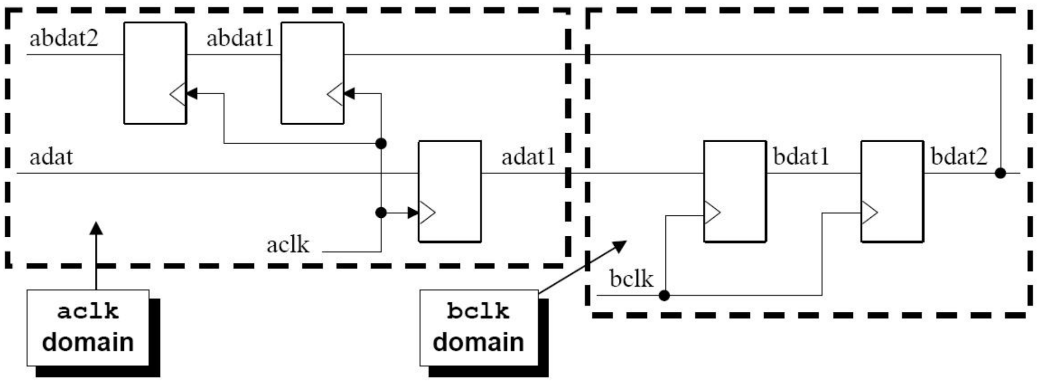
- ❑ Assert the control signal for a time period longer than the slow clock period


aclk
adat
bclk
bdat1
bdat2
This pulse must be wider than one bclk period!
This insures that adat is propagated to bdat1 and bdat2

Source: Clifford Cummings, "Synthesis and Scripting Techniques for Designing Multi Asynchronous Clock Designs"

Passing a control signal from slow clock domain to fast clock domain

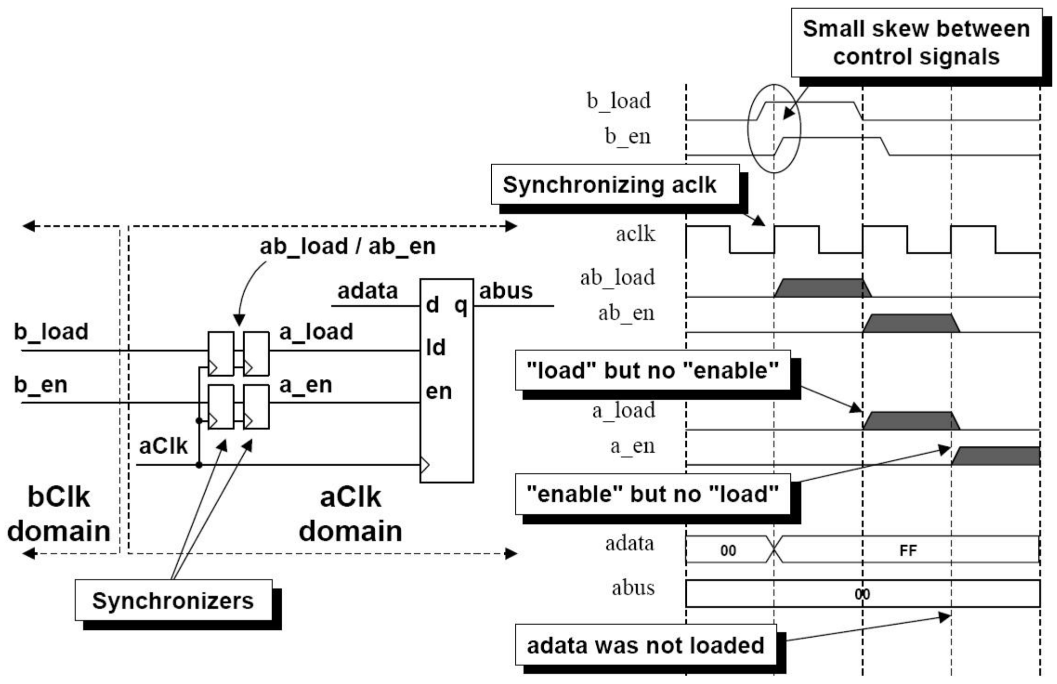
- Feedback the control signal as the acknowledge signal



- ❖ Don’t have to consider the relation between clock periods
- ❖ Introduce significant delay due to the latency of the synchronizer

Passing Multiple Control Signals

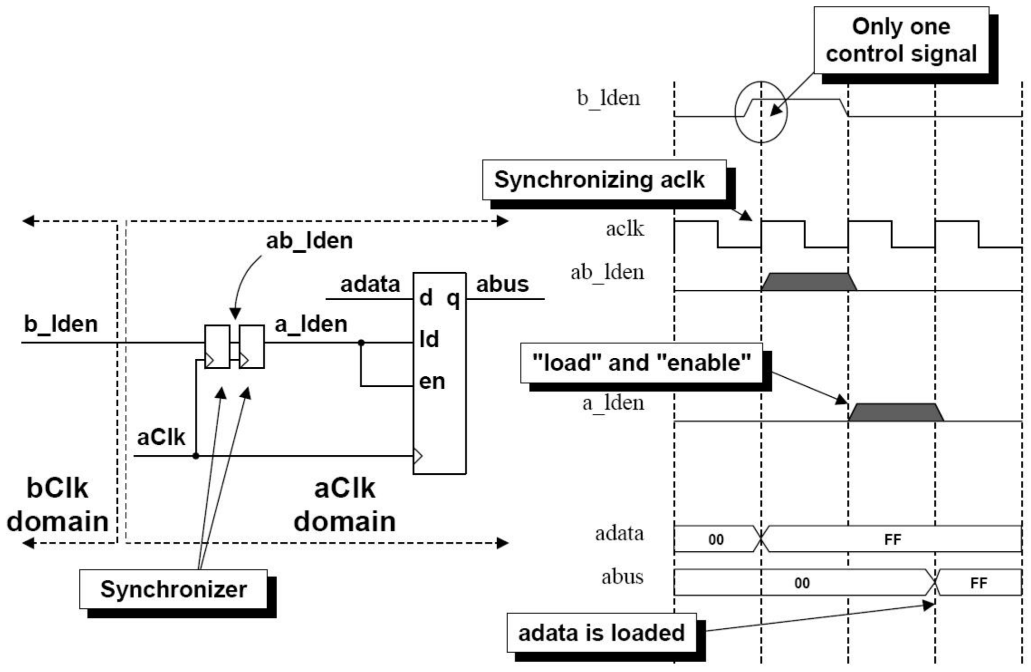
- ❑ CASE 1: two simultaneously required control signal



Source: Clifford Cummings, "Synthesis and Scripting Techniques for Designing Multi Asynchronous Clock Designs"

Passing Multiple Control Signals

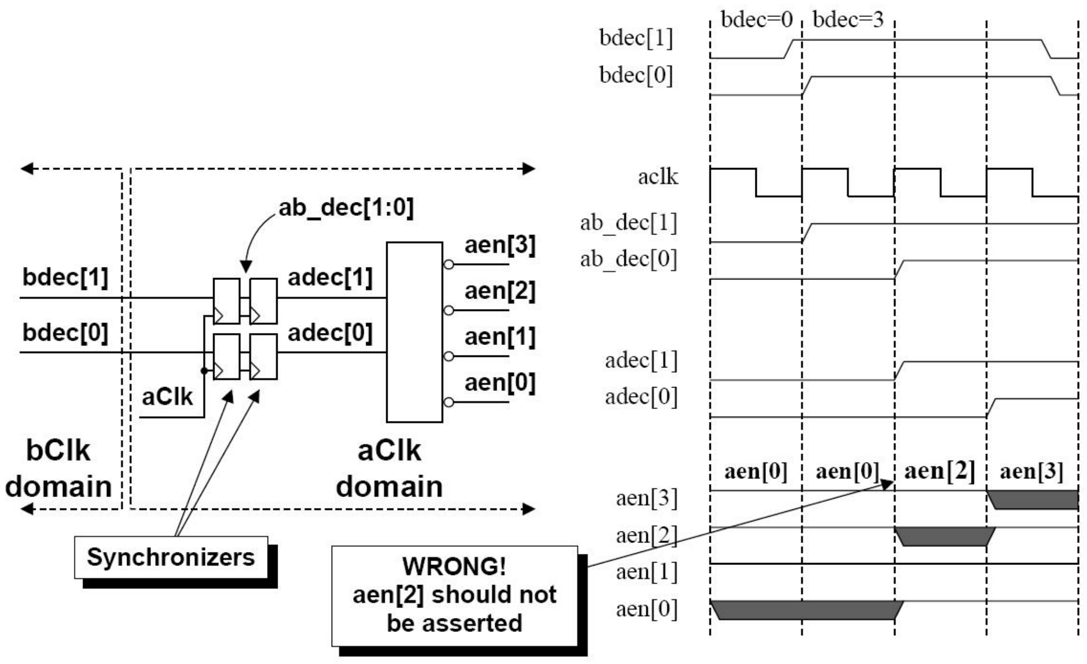
- Derived the two control signal from a single signal



Source: Clifford Cummings, "Synthesis and Scripting Techniques for Designing Multi Asynchronous Clock Designs"

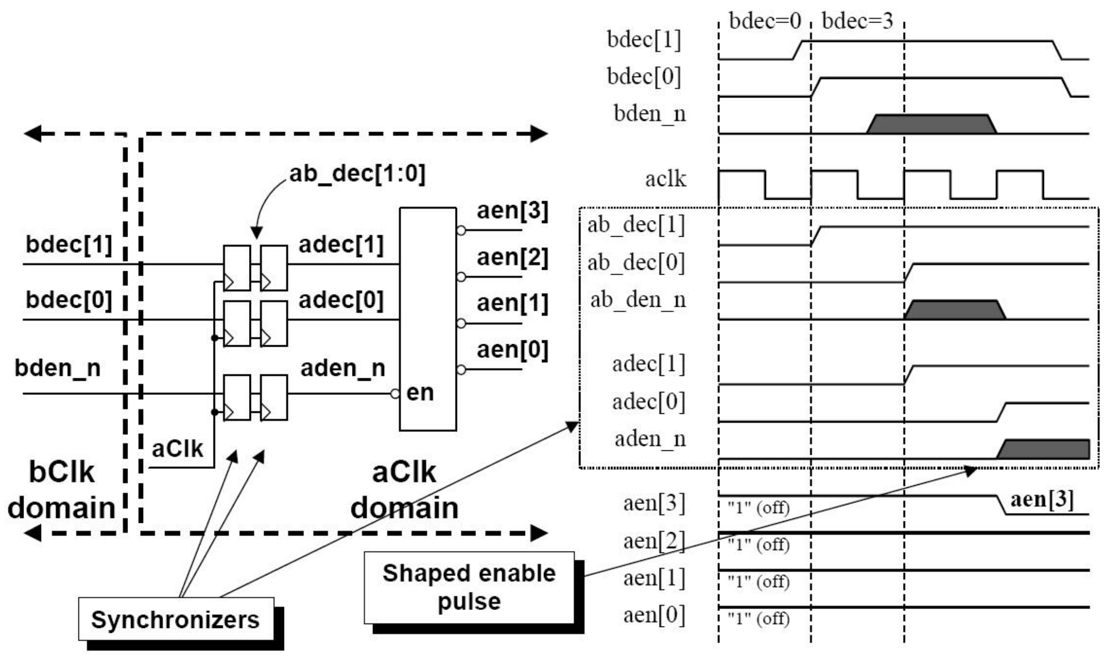
Passing Multiple Control Signals

- CASE 2: two encoded signals



Passing Multiple Control Signals

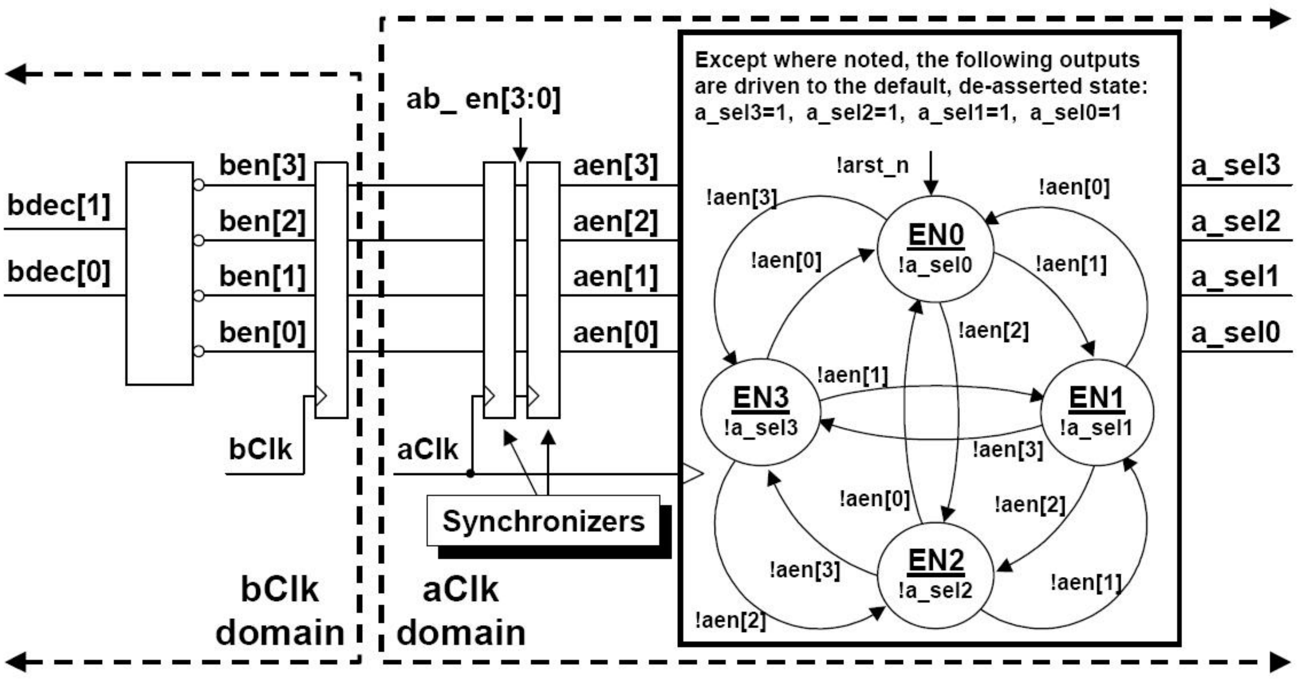
- Solution 1: added synchronized enable signal



Source: Clifford Cummings, "Synthesis and Scripting Techniques for Designing Multi Asynchronous Clock Designs"

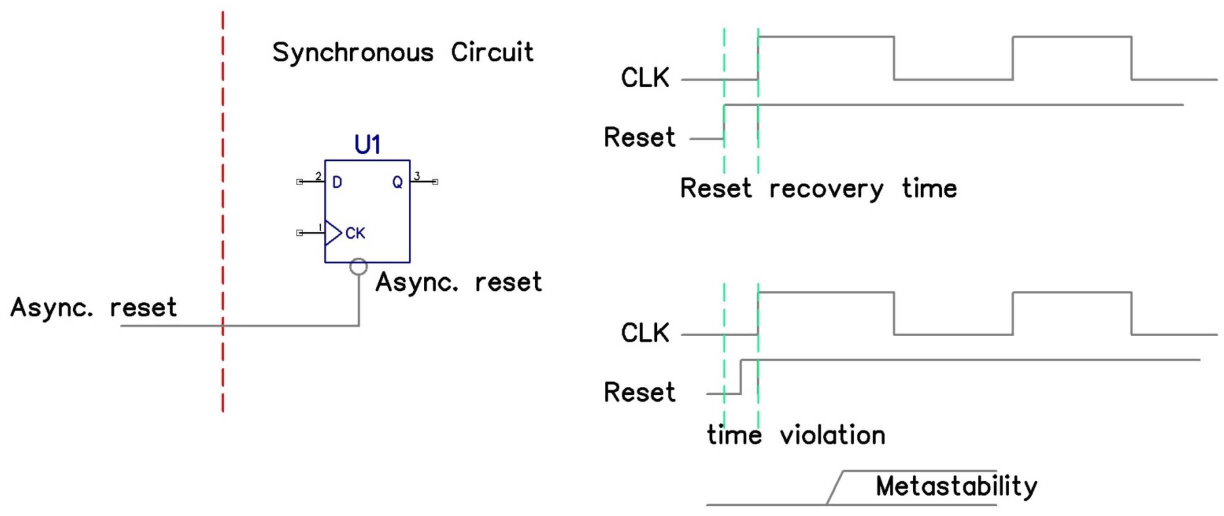
Passing Multiple Control Signals

- Solution 2:



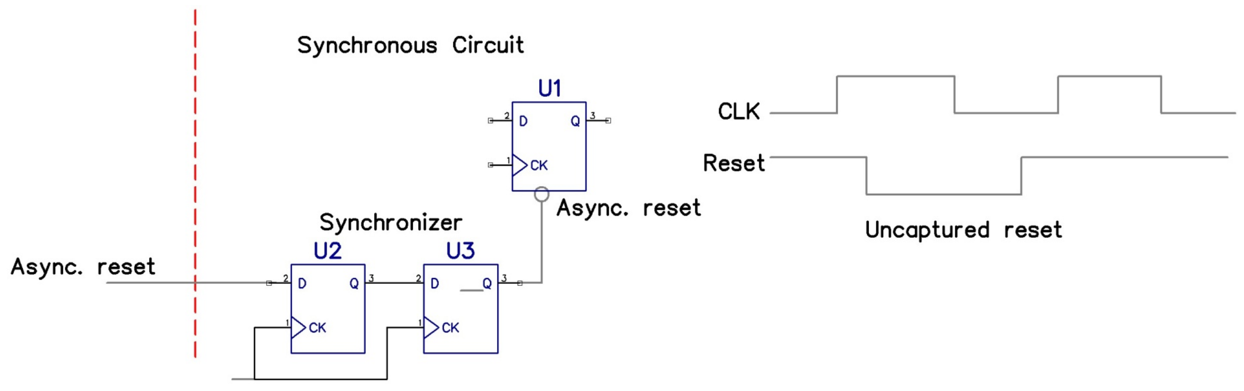
Source: Clifford Cummings, "Synthesis and Scripting Techniques for Designing Multi Asynchronous Clock Designs"

Asynchronous Reset/Set



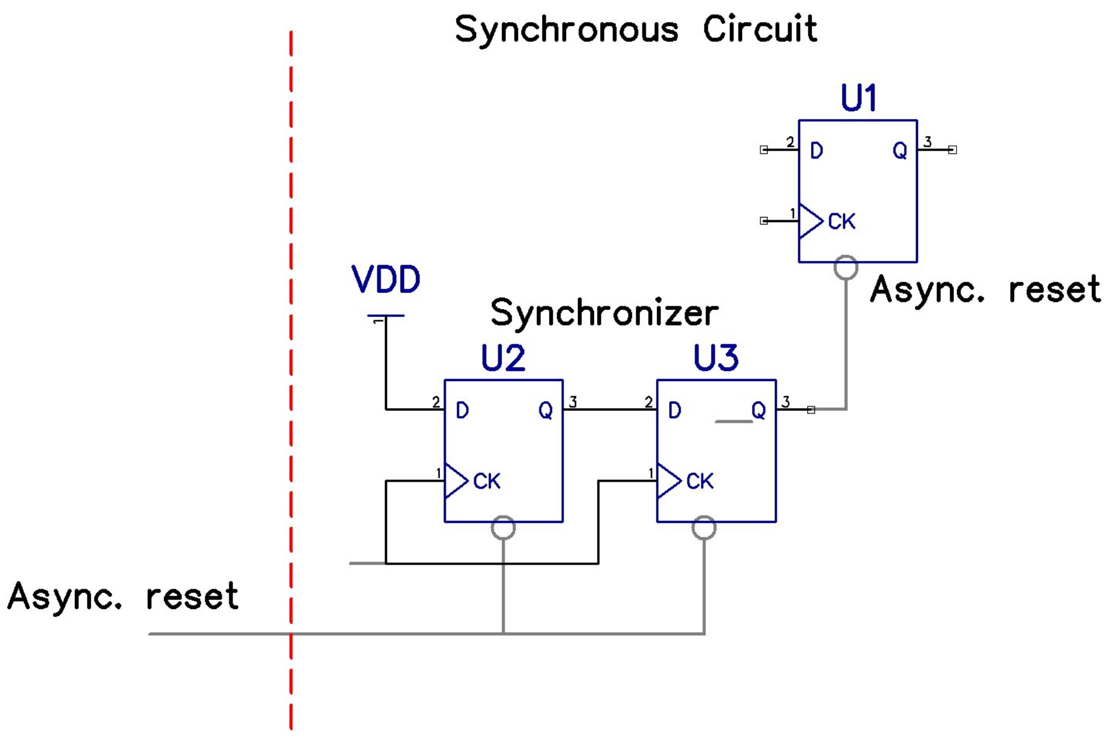
Source: Steve Kilts, “Advanced FPGA Design”

Synchronous Reset/Set



Source: Steve Kilts, “Advanced FPGA Design”

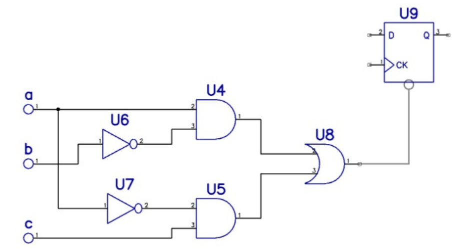
Asynchronous Assertion and Synchronous Desertion Reset/Set

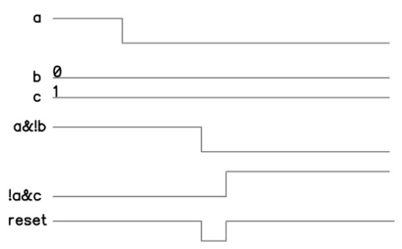


Source: Steve Kilts, "Advanced FPGA Design"

Internally Generated Reset/Set

- ❑ It is preferred to use synchronous reset/set if the reset/set signal is internally generated
- ❑ If asynchronous reset/set has to be used, need make sure the circuit that generates the reset/set signal is glitch free.





- ❑ Truth table of the reset logic

	00	01	10	11
0	0	0	1	0
1	1	1	1	0

- ❑ Adding a prime implicant

	00	01	10	11
0	0	0	1	0
1	1	1	1	0

Source: Steve Kilts, "Advanced FPGA Design"