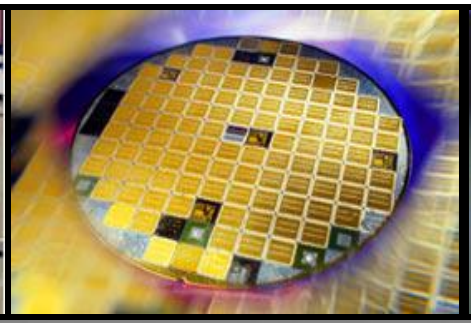
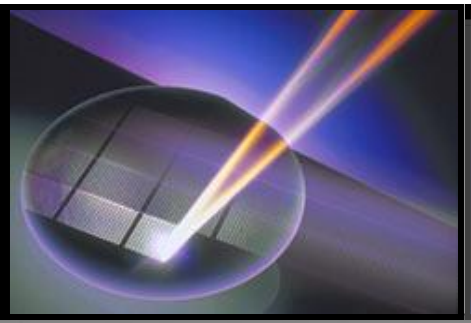
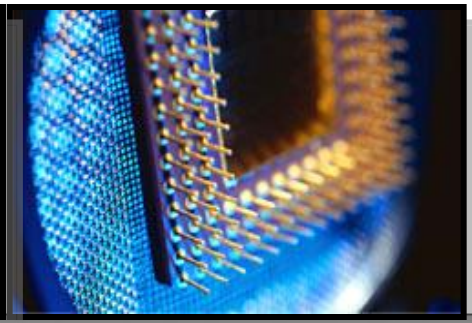
FPGA异步电路处理方法

课程安排

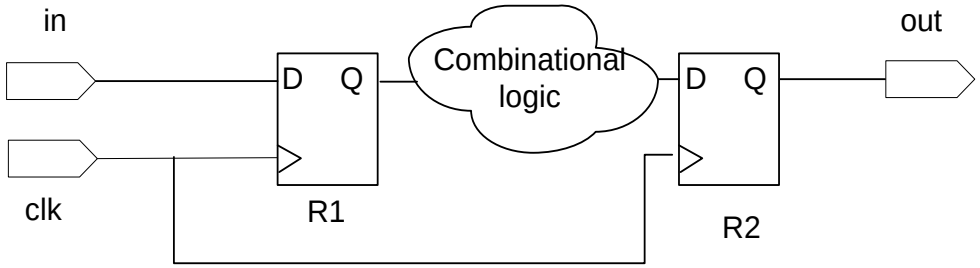
- 亚稳态的基本概念
- 慢时钟信号进入快时钟域的处理方法
- 快时钟信号进入慢时钟域的处理方法
- 异步复位（Reset）路径处理方法
- 设计实例

课程安排

- 亚稳态的基本概念
- 慢时钟信号进入快时钟域的处理方法
- 快时钟信号进入慢时钟域的处理方法
- 异步复位（Reset）路径处理方法
- 设计实例

同步电路

- What is synchronous design circuit?
  - All clocked element, such as flip flops (FFs) or registers, s hare a common clock signal (a globally distributed clock)
  - Data changes based on clk edges
  - Example:



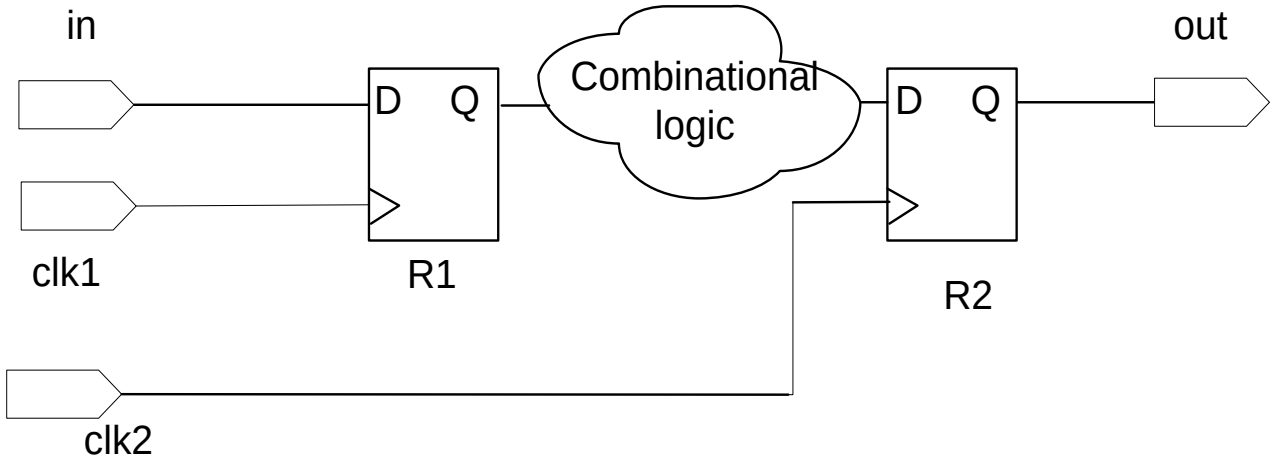
clk reaches to R1 and R2 at the same time

同步电路- cont

- Fully supported by EDA tools
- Static timing analysis tools are designed to report t iming problem on one-clock synchronous designs
- Easy to implement

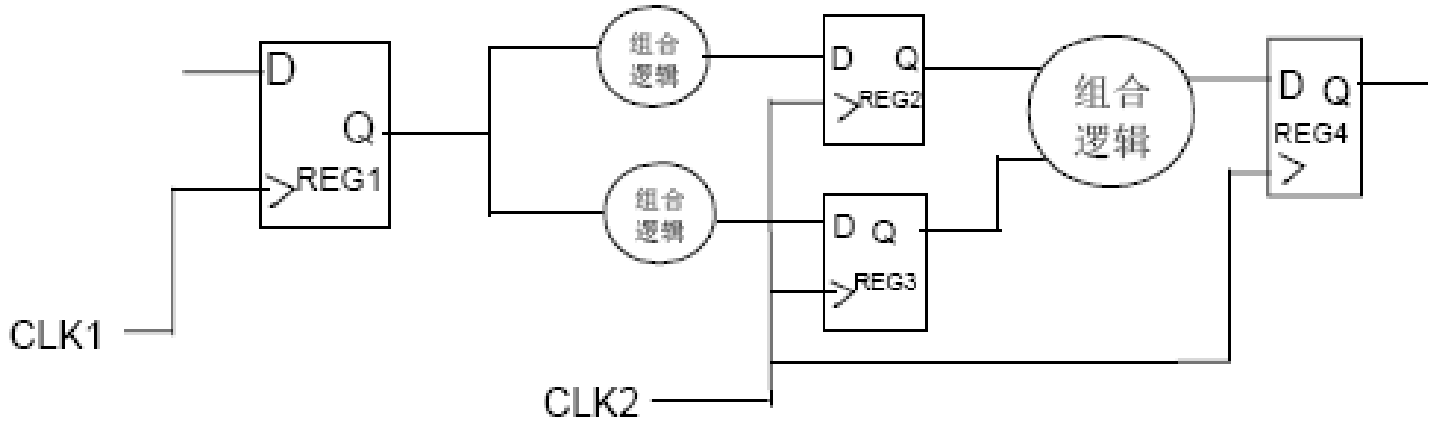
异步电路

- Transition can be done at any time - Not controlled by any gl obal or local clock



多时钟系统

- 许多系统要求在同一设计内采用多时钟，最常见的例子是两个异步微处理器之间的接口，或微处理器和异步通信通道的接口。 由于两个时钟信号之间要求一定的建立和保持时间， 所以上述应用引进了附加的定时约束条件， 它们会要求将某些异步信号同步化。



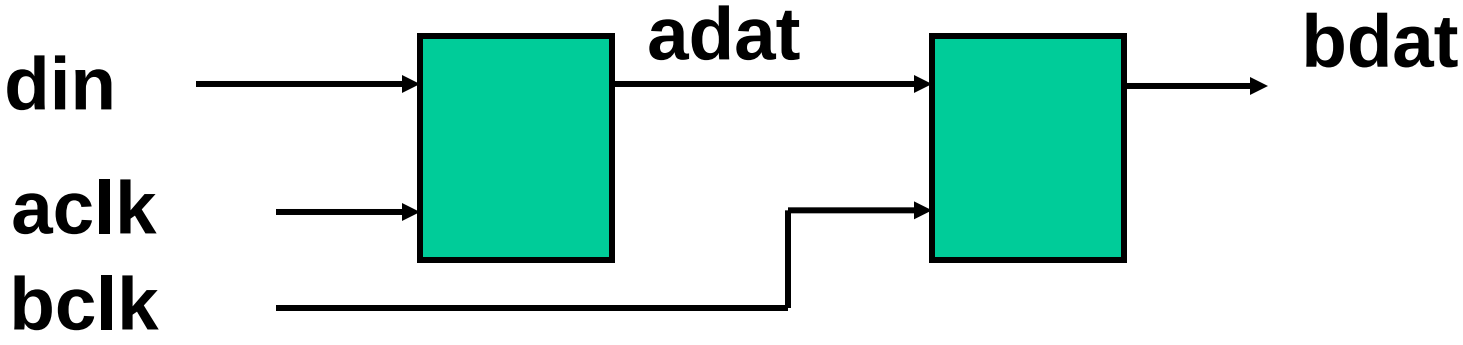
亚稳态 Metastability

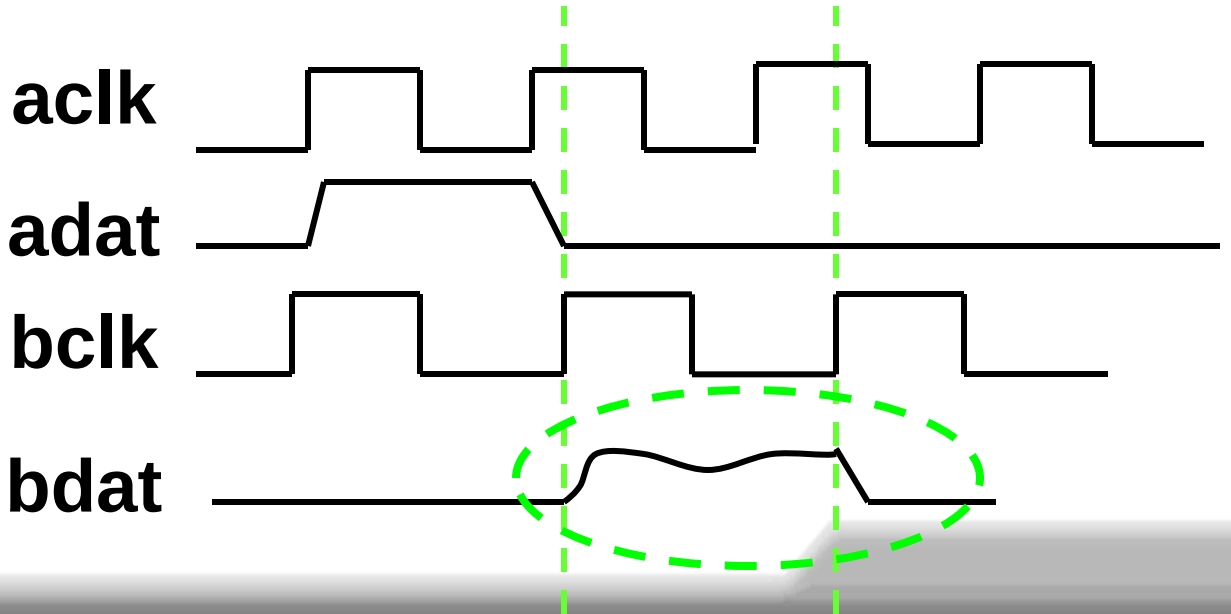
- **Observed:**
  - asynchronous inputs in synchronous systems lead to system failure (also called synchronization failure)

- **Reason:**
  - an asynchronous input which can change at any tim e with respect to the clock edges of the synchronous system. When a FF input signal is changing state at or near the instant of active clk edge occurring.

多时钟系统设计

- 如果一个系统中存在多个独立(异步)时钟，并且存在多时钟域(clock domain)之间的信号传输，那么电路会出现亚稳态。



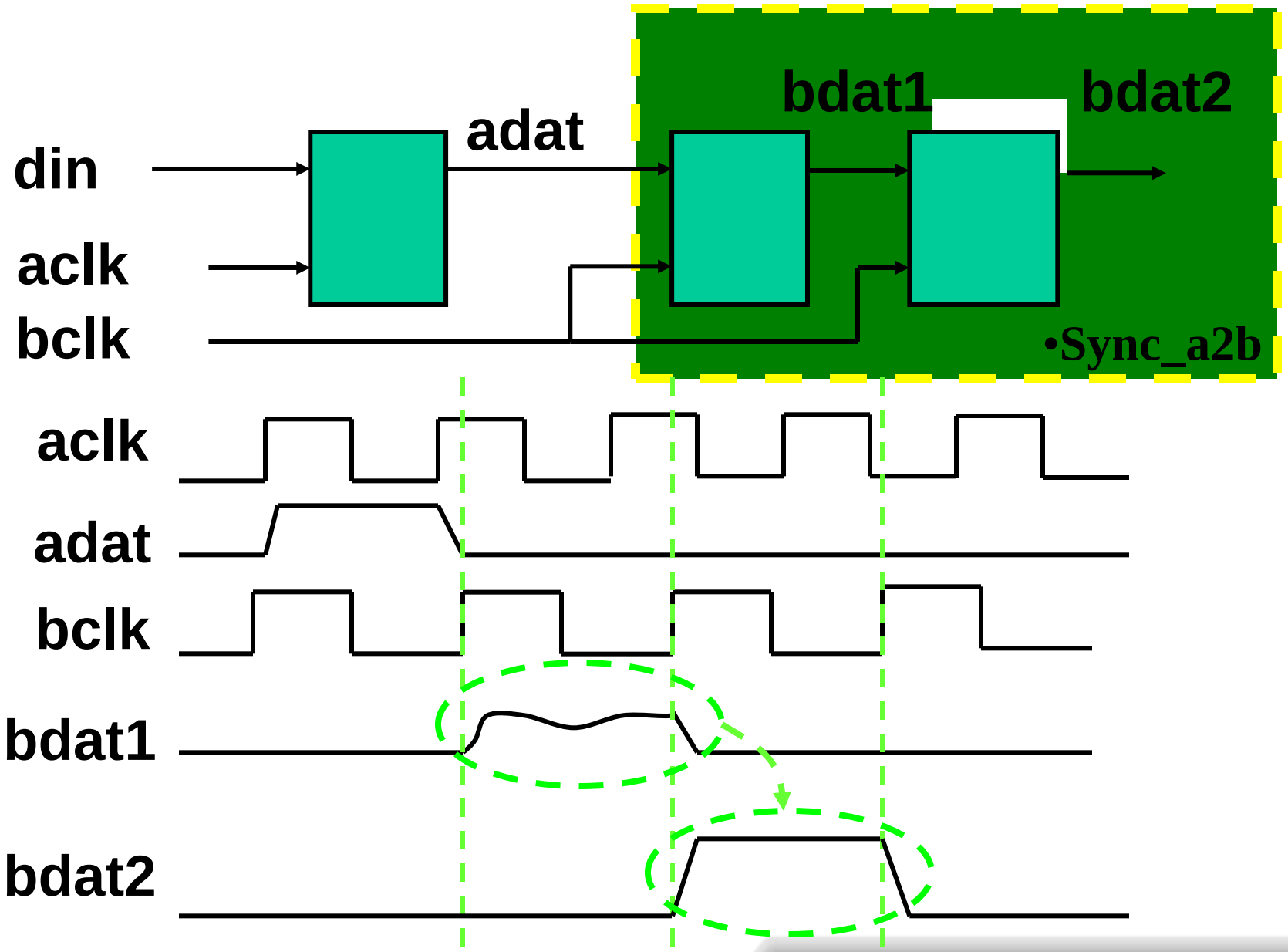


- **Metastable state**
  - The output of the device does not reach either of the valid logic levels but between the two for a time that is long compared with the normal timing delays of t he device or may even oscillate.
  - If the signal bdat is propagated to the rest of the desi gn before it comes to a stable state, synchronous fail ure will occur.
- **More cases in real ASIC design world**
  - Input data from UART, SSI, …devices to another de vice/chip
  - Asynchronous external reset

课程安排

- 亚稳态的基本概念
- **慢时钟信号进入快时钟域的处理方法**
- 快时钟信号进入慢时钟域的处理方法
- 异步复位（Reset）路径处理方法
- 设计实例

避免亚稳态----两级FF同步化



亚稳态分析

- 出现亚稳态的平均时间间隔常用“平均无故障时间”（MTBF，Mean Time Between Failure） 来表示。单个触发器的MTBF为

$$\mathrm{MTBF} = \frac{\exp\left(t_r / \tau\right)}{T_0 \cdot f_{in} \cdot f_{clock}}$$

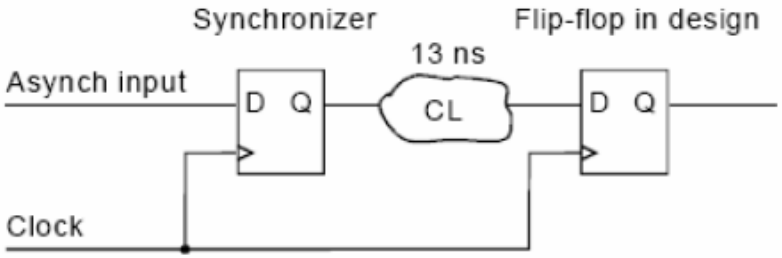
  - t_r：不引起synchronizer failure的前提下，亚稳态可持续的最长时间（MetaStability Resolution Time）；
  - τ和 T_0：与触发器电气特性有关的常数；
  - f_{in}：异步输入信号的频率；
  - f_{clock}：起同步作用的触发器时钟频率。

温度、电压、辐射等因素都对MTBF有影响

两个寄存器方案分析

- 触发器时钟频率为 **10 MHz**，异步输入信号频率为 **3 kHz**：

τ	1 ns
T_o	$5 \cdot 10^5$ s
t_{su}	2 ns



Resolution time:

$$t_r = \frac{1}{f_{clock}} - t_{CL} - t_{su} = (100 - 13 - 2)\,\text{ns} = 85\,\text{ns}$$

$$\text{MTBF} = \frac{\exp(85)}{5 \cdot 10^5 \cdot 10 \cdot 10^6 \cdot 3 \cdot 10^3} = 5.5 \cdot 10^{20}\,\text{s} = 1.74^{13}\,\text{years}$$

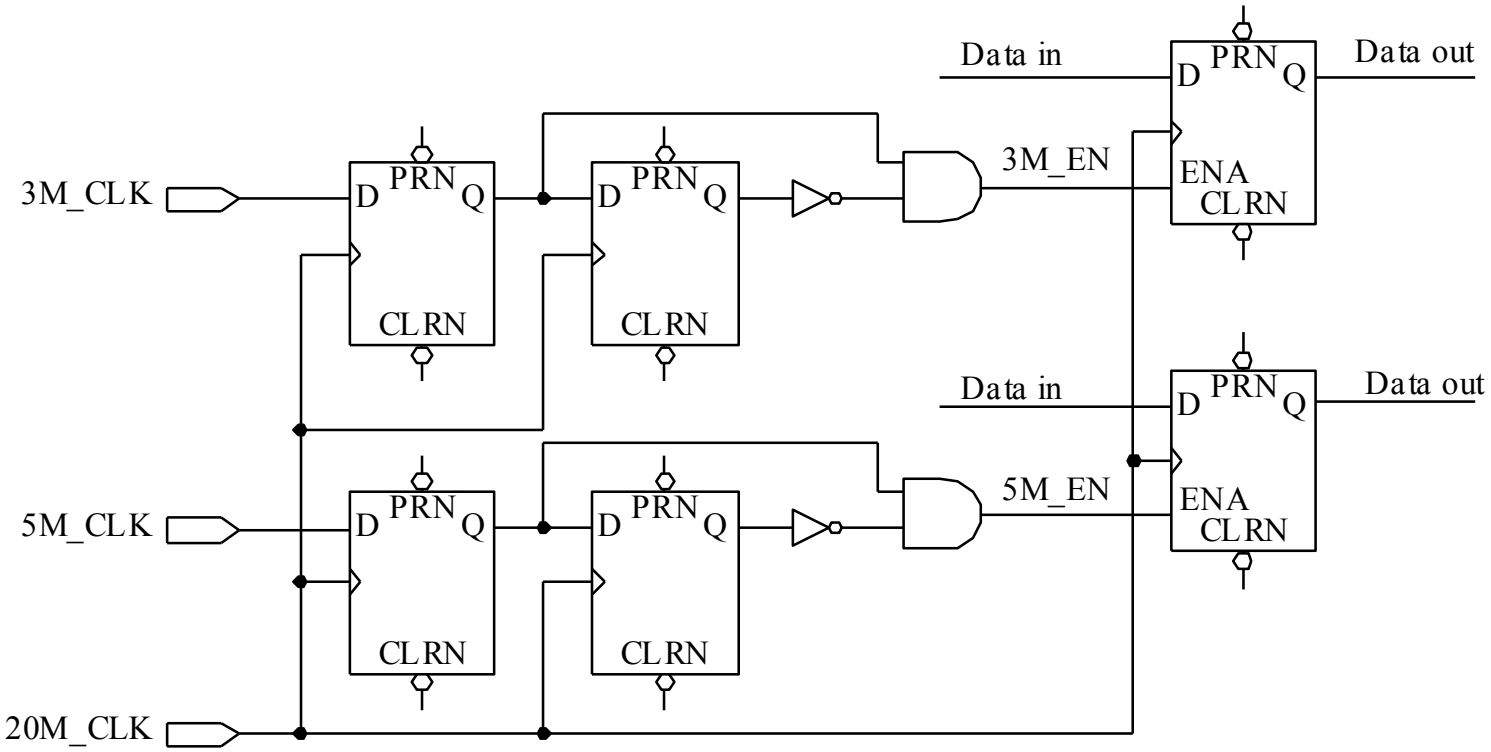
- 优点：
  - 实现简单.
  - 成本低.

- 缺点：
  - 不能完全消除亚稳态
  - 导致延时增加.

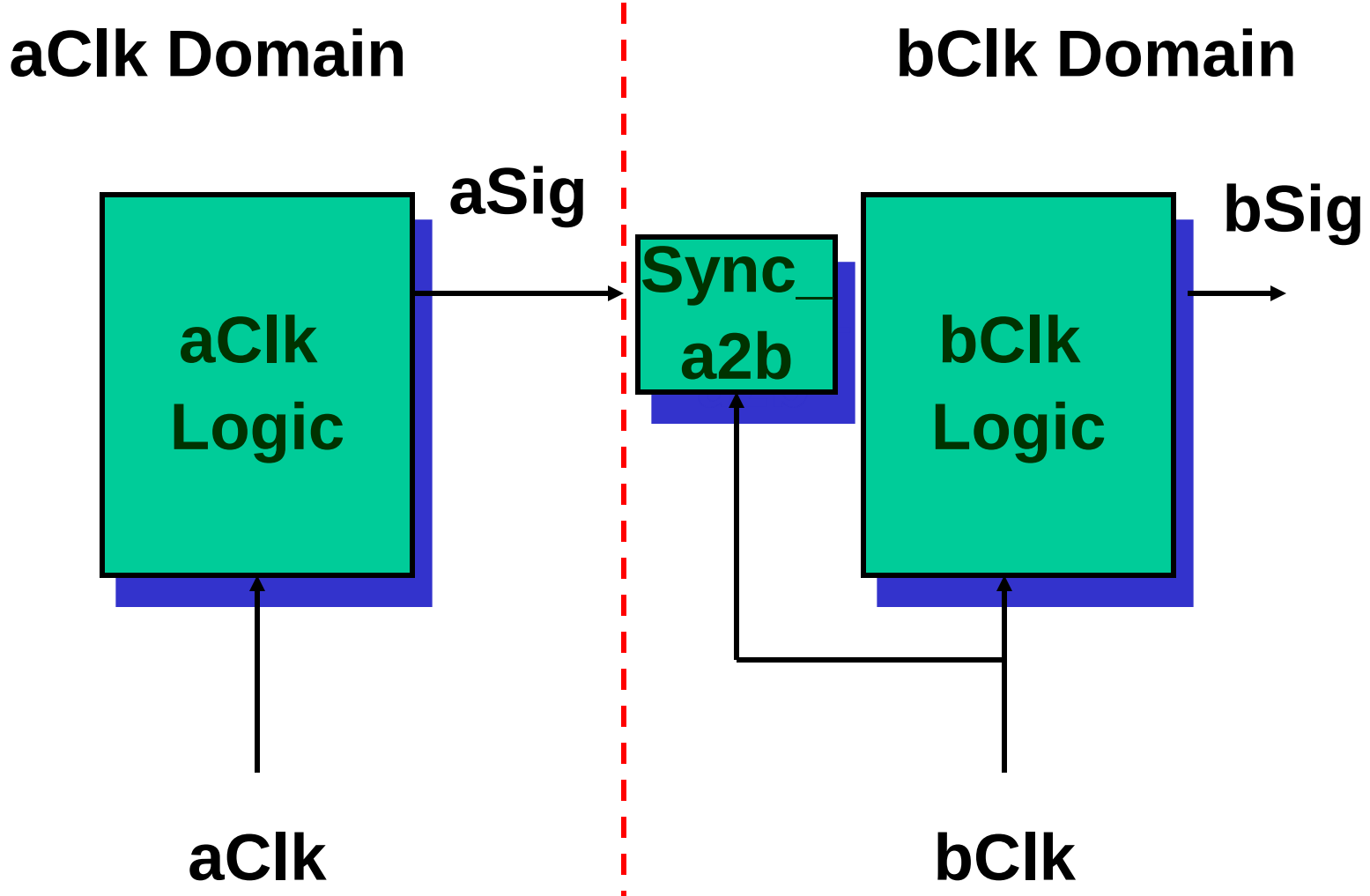
多时钟系统

- 在许多应用中只将异步信号同步化还是不够的，当系统中有两个或两个以上非同源时钟的时候，数据的建立和保持时间很难得到保证，设计人员将面临复杂的时间分析问题。
- 可以将所有非同源时钟同步化。
- 这时就需要引入一个高频时钟来实现信号的同步化。

同步化任意非同源时钟



异步多时钟系统模型

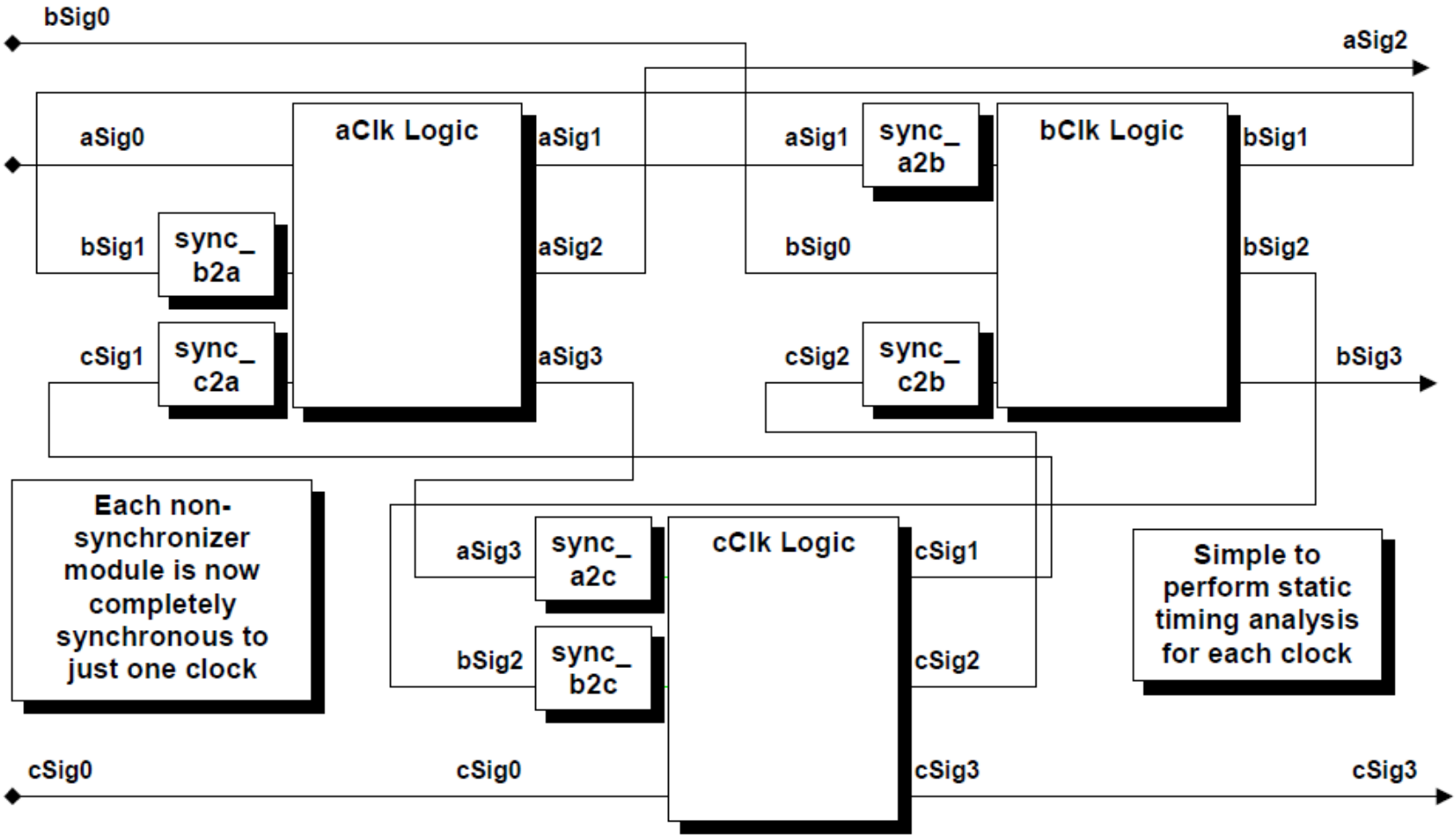

aClk Domain
bClk Domain
aSig
bSig
aClk Logic
Sync_a2b
bClk Logic
aClk
bClk

注意其信号命名和模块划分方法

- 这种信号命名和模块划分的方法有如下优点：
  - 有利于检查信号所通过的时钟域；
  - 有利于各模块进行单独的静态时序分析；
  - 有利于在静态时序分析中快速地设定false path；

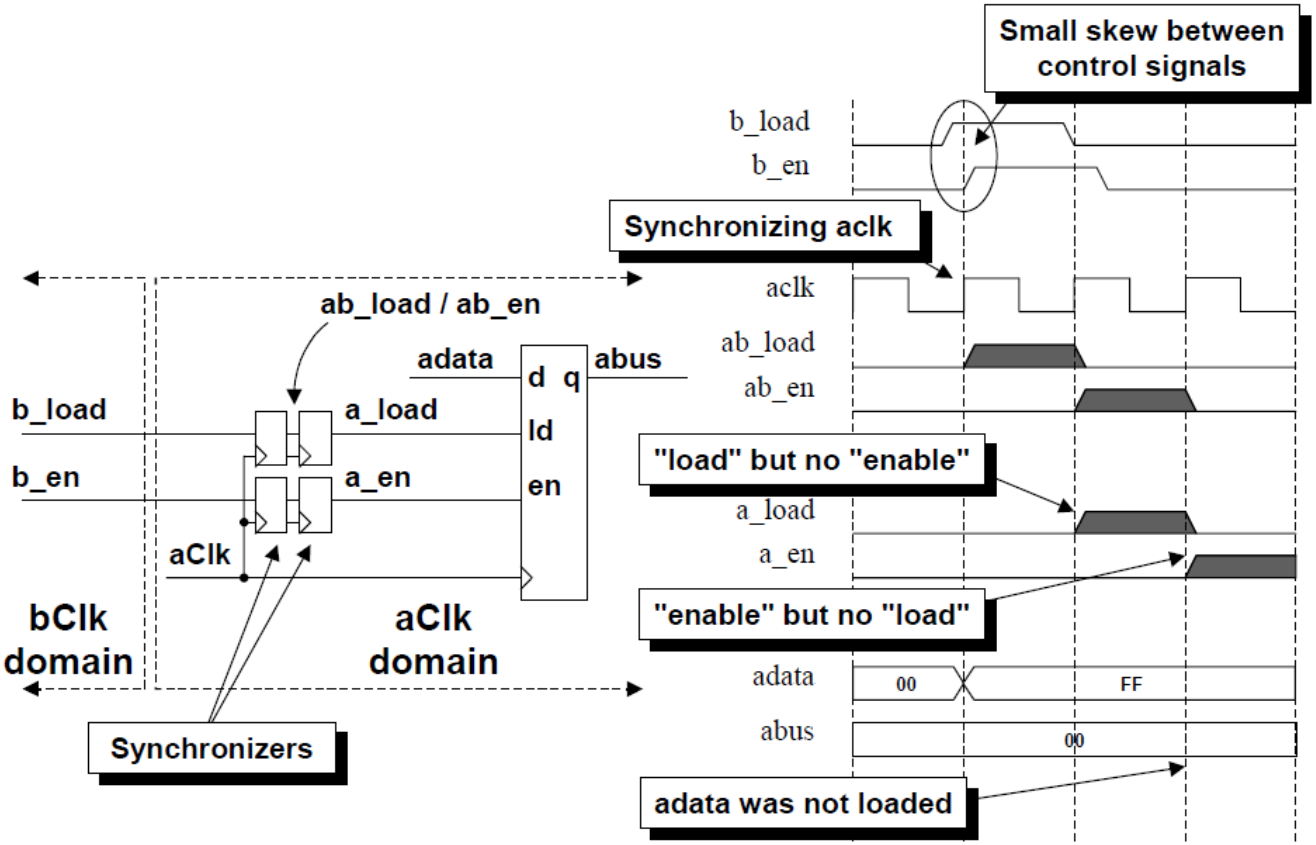
•异步信号穿越时钟域时，这些信号与异步时钟之间的相位关系数是无穷的，所以在整个系统静态时序分析时必须忽略这些信号路径。

按时钟域进行划分

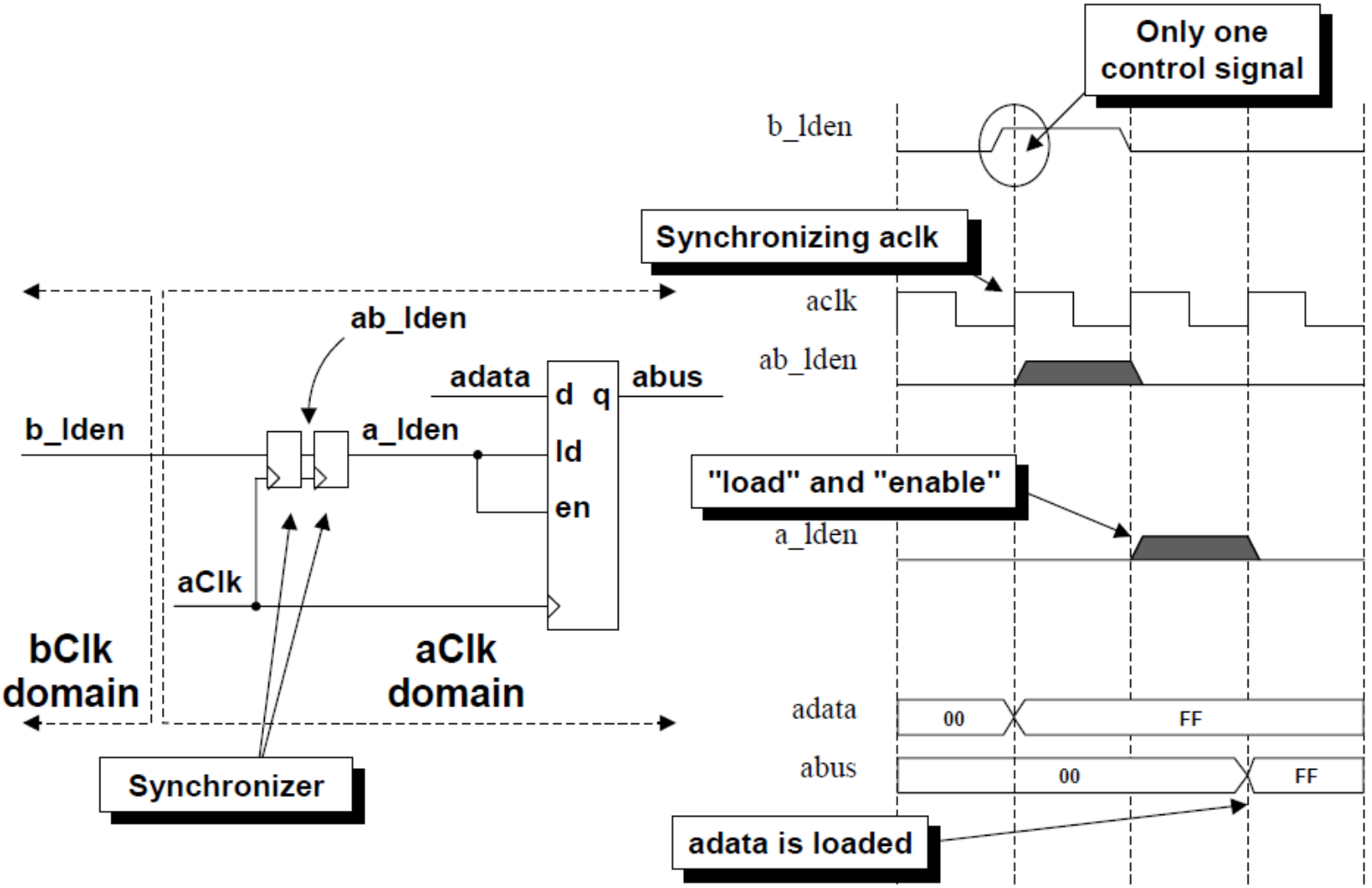

bSig0
aSig2
aSig0
aClk Logic
aSig1
aSig1
sync_
a2b
bClk Logic
bSig1
bSig1
sync_
b2a
aSig2
bSig0
bSig2
cSig1
sync_
c2a
aSig3
cSig2
sync_
c2b
bSig3
Each non-synchronizer module is now completely synchronous to just one clock
aSig3
sync_
a2c
cClk Logic
cSig1
bSig2
sync_
b2c
cSig2
Simple to perform static timing analysis for each clock
cSig0
cSig0
cSig3
cSig3

多bit控制信号

- 同时需要两个控制信号



解决方法：合为一个控制信号



数据接口的同步方法

- 输入、输出的延时(芯片间、PCB 布线、一些驱动接口元件的延时等)不可测，或者有可能变动的条件下，如何完成数据同步？

最常用的缓存单元是DPRAM和FIFO，在输入端口使用上级时钟写数据，在输出端口使用本级时钟读数据。

其他异步问题

A lesson we learnt:

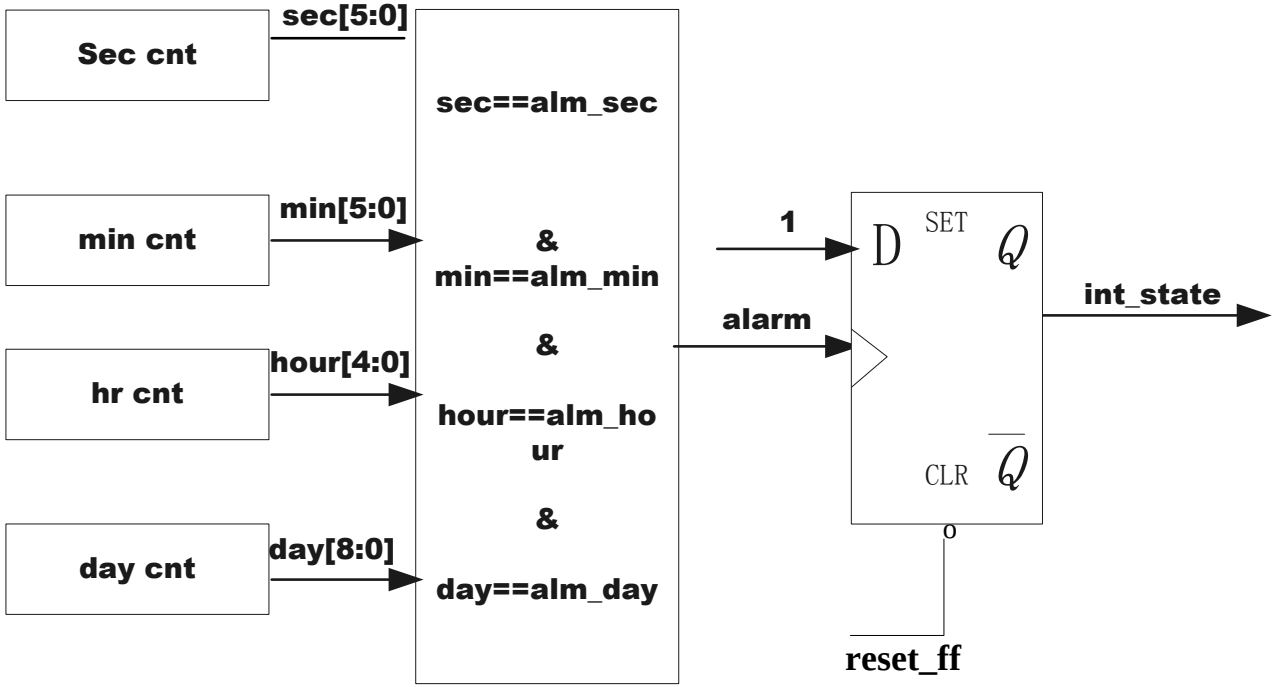
Signal from combinational logic as clock/rest of FF

- Any glitch may let the FF toggle

Glitch on the clk of FF

Example: Real Time Clock module

Problem: alarm signal from combinational logic used as clk signal of FF



Example of RTL code

```
wire       alarm =
   (sec==alm_sec) & (min==alm_min) & (hour==alm_hour)& (day==alm_day);

// use alarm to generate interrupt

always @(posedge alarm or negedge reset_ff)
    if( !reset_ff )
          int_state <= 1'b0;
    else
          int_state <= 1'b1;
```

Note: signal alarm may have glitches due to the change of **alm_sec, alm_min, alm_hour and alm_day**

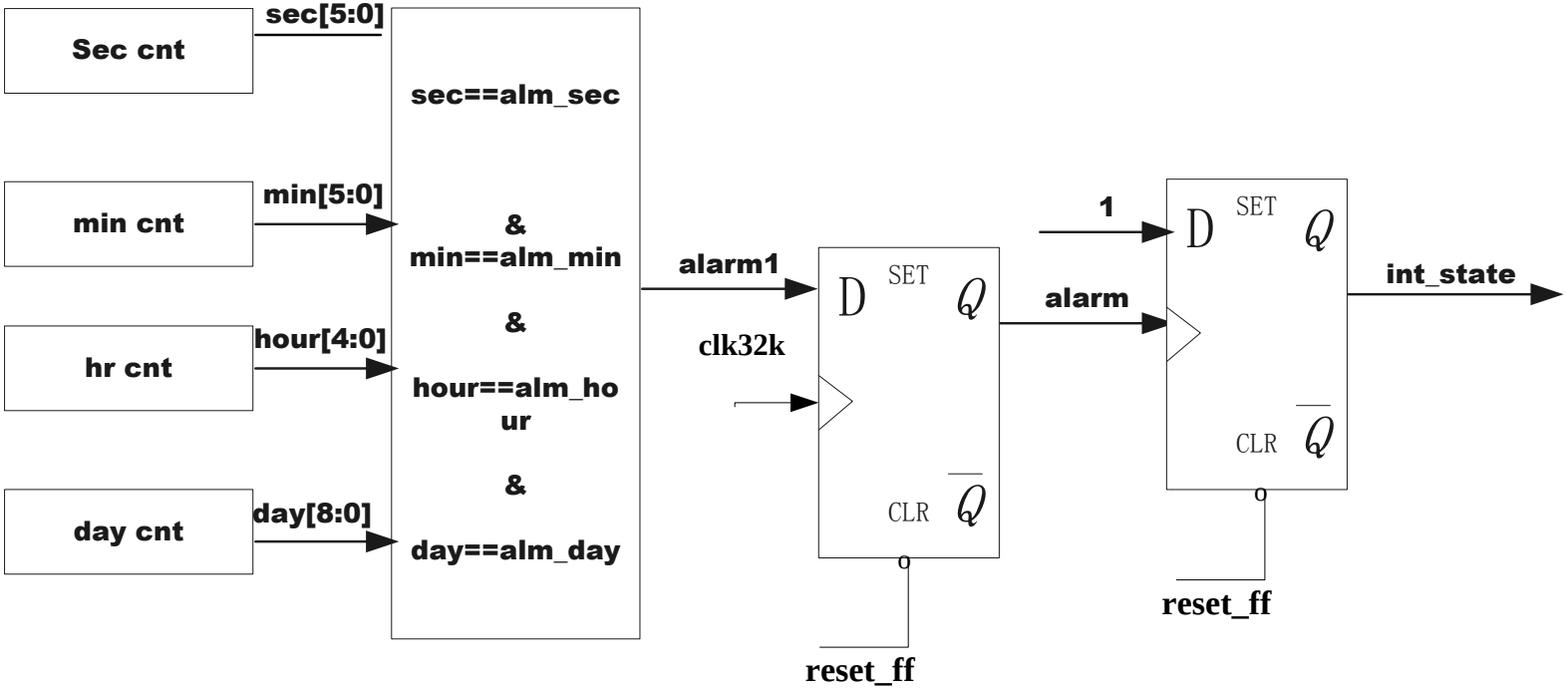
Solution

- Sync the signal by global synchronized clk before using it as a clk for FF
  - Clock clk32k is used as clk signal in the synchronous counters before using it as a clk signal for FF to generate int_state

Modified code:

```
wire alarm1=(sec==alm_sec)&(min==alm_min)&(hour==alm_h
    our)&(day==alm_day);
always @(posedge clk32k) alarm<=alarm1;
always @(posedge alarm or negedge reset_ff)
    if( !reset_ff )
          int_state <= 1'b0;
    else              int_state <= 1'b1;
```

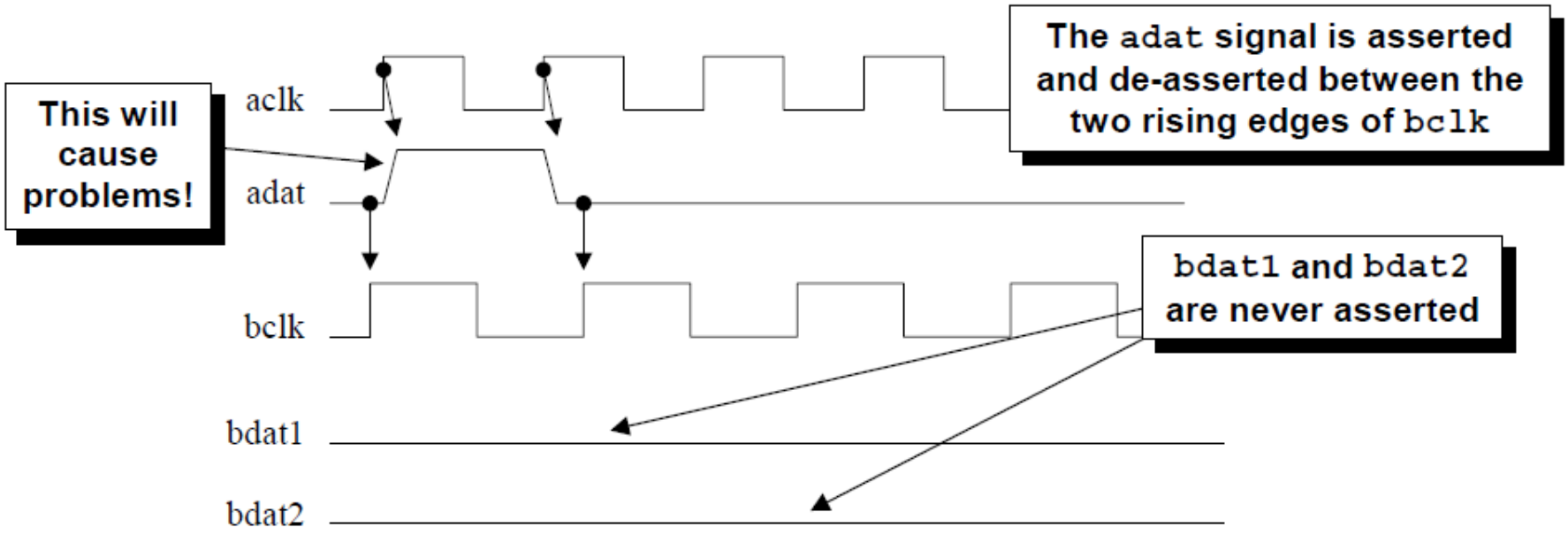
Modified Logic



课程安排

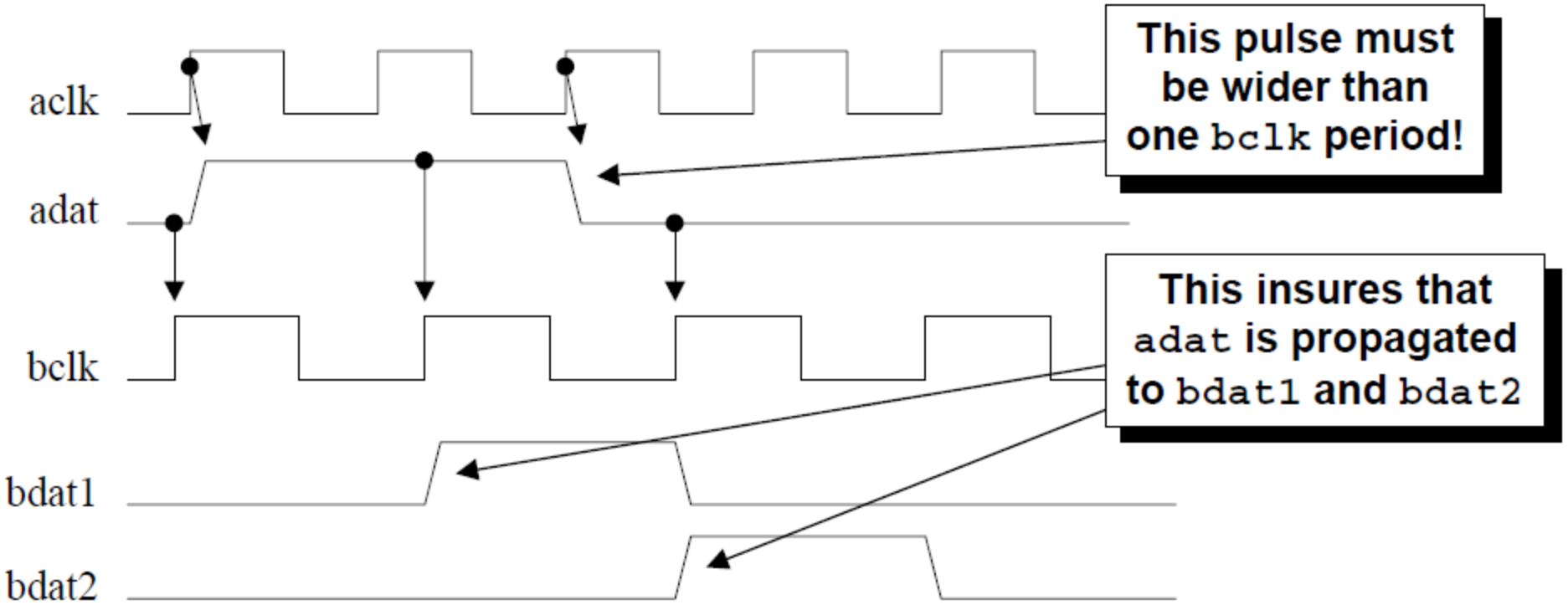
- 亚稳态的基本概念
- 慢时钟信号进入快时钟域的处理方法
- **快时钟信号进入慢时钟域的处理方法**
- 异步复位（Reset）路径处理方法
- 设计实例

慢控制信号进入快时钟域



解决方法

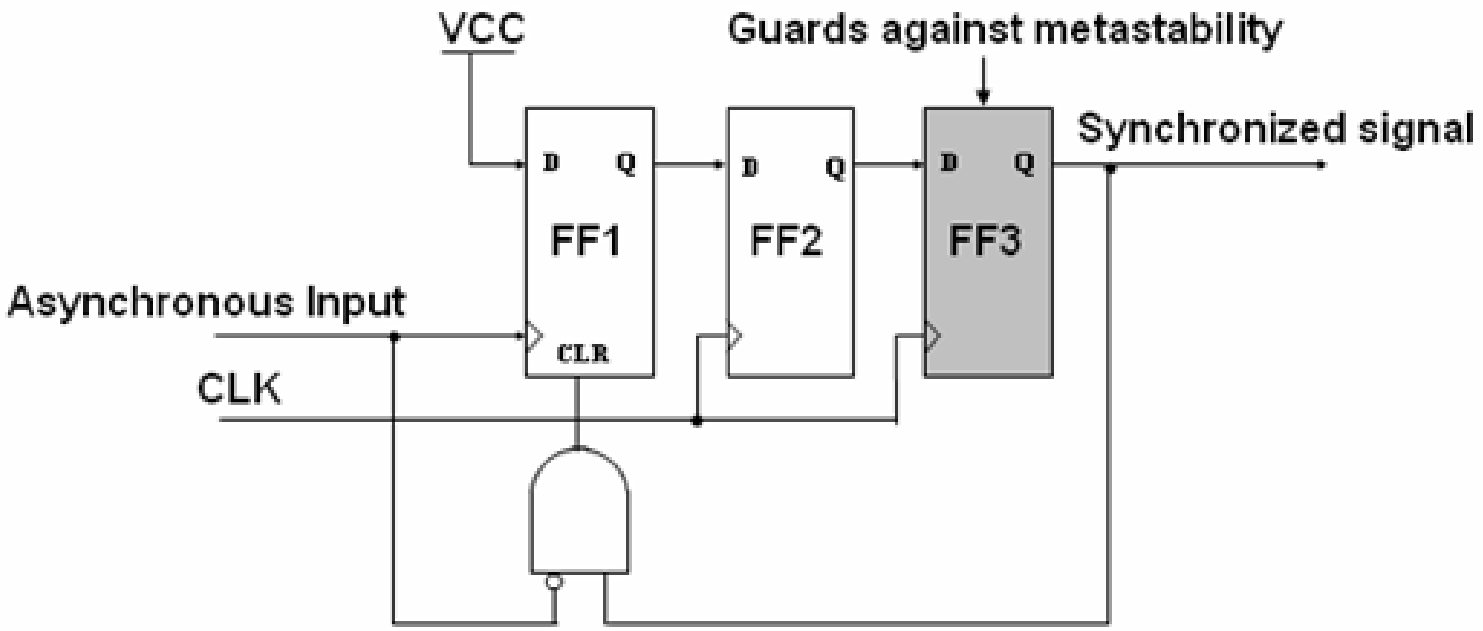
- 插入一个超过一个采样时钟周期的控制信号



解决方法2

- 当输入信号小于一个时钟周期，比如快时钟域的信号进入慢时钟域信号

  - FF1 captures short pulses



Data-Path Synchronization

- 从1个时钟进入另外一个时钟域
- 常见错误:
  - 简单的同步flip-flop来同步总线信号
- 难题:
  - 并不能保证每个bit数据同时变化
- 常见的解决方法:
  - To have a data_valid signals accompany the bus (handshaking)
  - Use asynchronous FIFO (First In First Out memories)

Data-Path Synchronization – cont

- Method 1: to have a data_valid signal accompany the bus
  - Data_valid signal is not asserted until a comfortable timing margin after all the bus bits have transitioned.
  - **To synchronize data_valid signal only and use it to indica te Bus bits are stable**
  - Condition: the rate of sampling clk must be much higher tha n the rate of updates on the Bus

Data-Path Synchronization – cont

- Method 2: use FIFOs or DP-RAM
  - Dual port memory: dual port SRAM or FF array
  - The data first written to it shall also be the first one read from it
  - Store (write) data using one clk domain an d retrieve (read) data using another clk do main

课程安排

- 亚稳态的基本概念
- 慢时钟信号进入快时钟域的处理方法
- 快时钟信号进入慢时钟域的处理方法
- **异步复位路径处理方法**
- 设计实例

复位

复位是可综合编码风格的重要环节。状态机中一般都有复

同步复位

```
module sync( q, clk, r, d);
    input lck, d, rst;
    output q;
    reg q;
    always @( posedge clk)
        if (r)
            q <= 0;
        else
            q <= d;
endmodule
```

同步块的异步复位

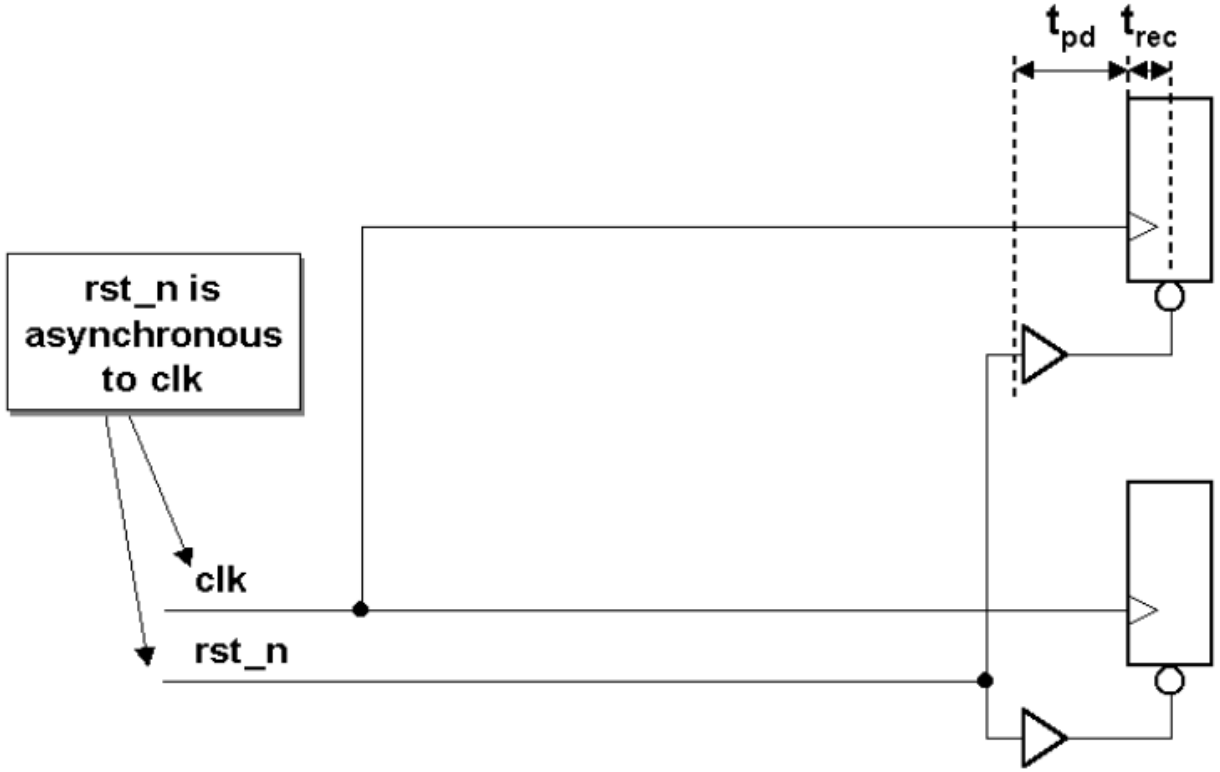
```
module async( q, clk, r, d);
    input clk, d, r;
    output q;
    reg q;
    always @( posedge clk or negedge r)
        if (!r)
            q <= 0;
        else
            q <= d;
endmodule
```

复位信号的设计原则

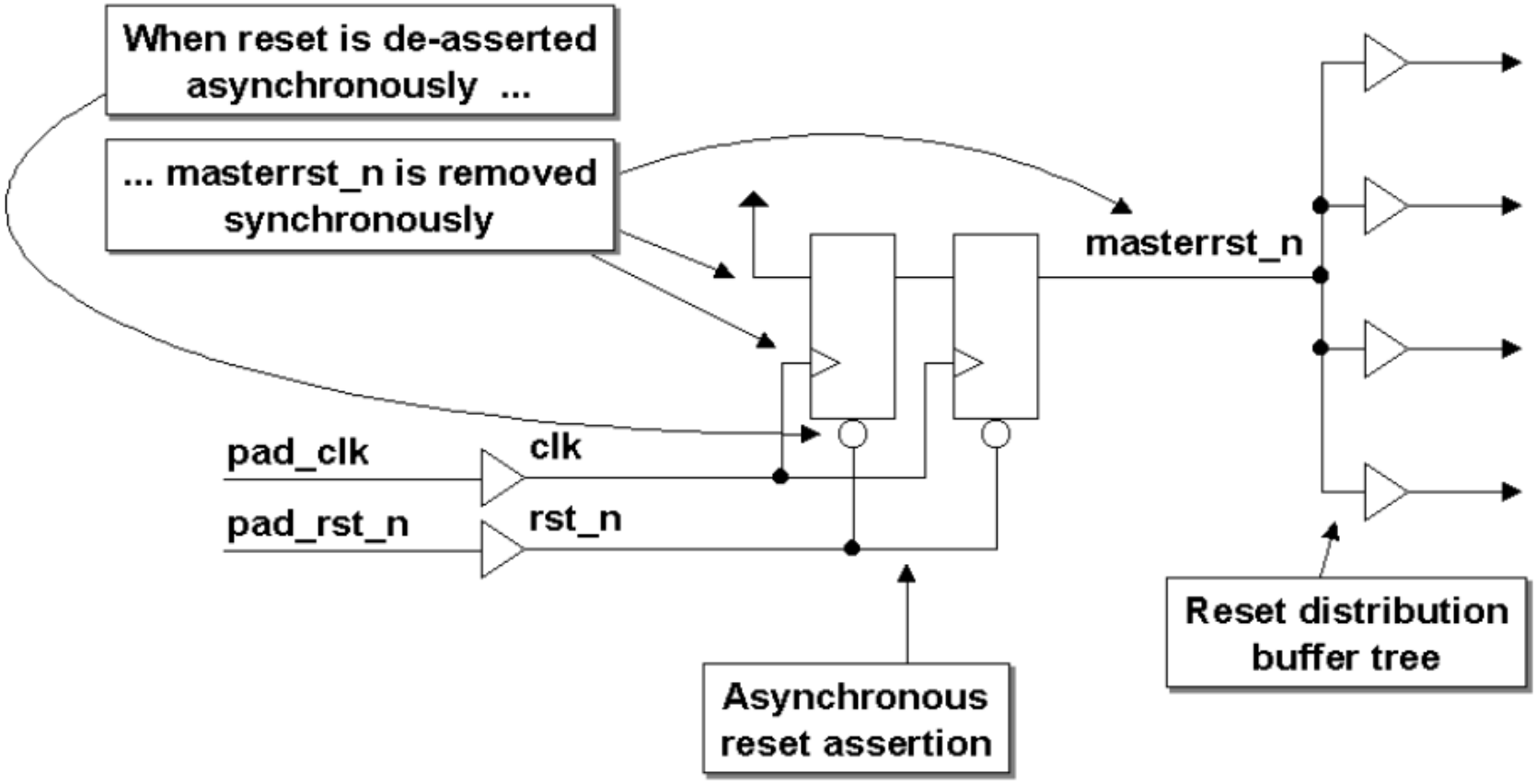
- 复位信号的设计，要采用单一的全局复位信号
- 避免使用模块内部产生的条件复位信号，模块内部产生的条件复位信号可以转换为同步输入的使能信号处理
- 所有的时钟信号和复位信号在芯片的最顶层都必须是可控制和可观测的

Asynchronous reset problem

- **an asynchronous reset signal will be de-asserted asynchrono us to the clock signal.**
  - violation of reset recovery time
  - reset removal happening in different clock cycles for different sequ ential elements



Reset同步器

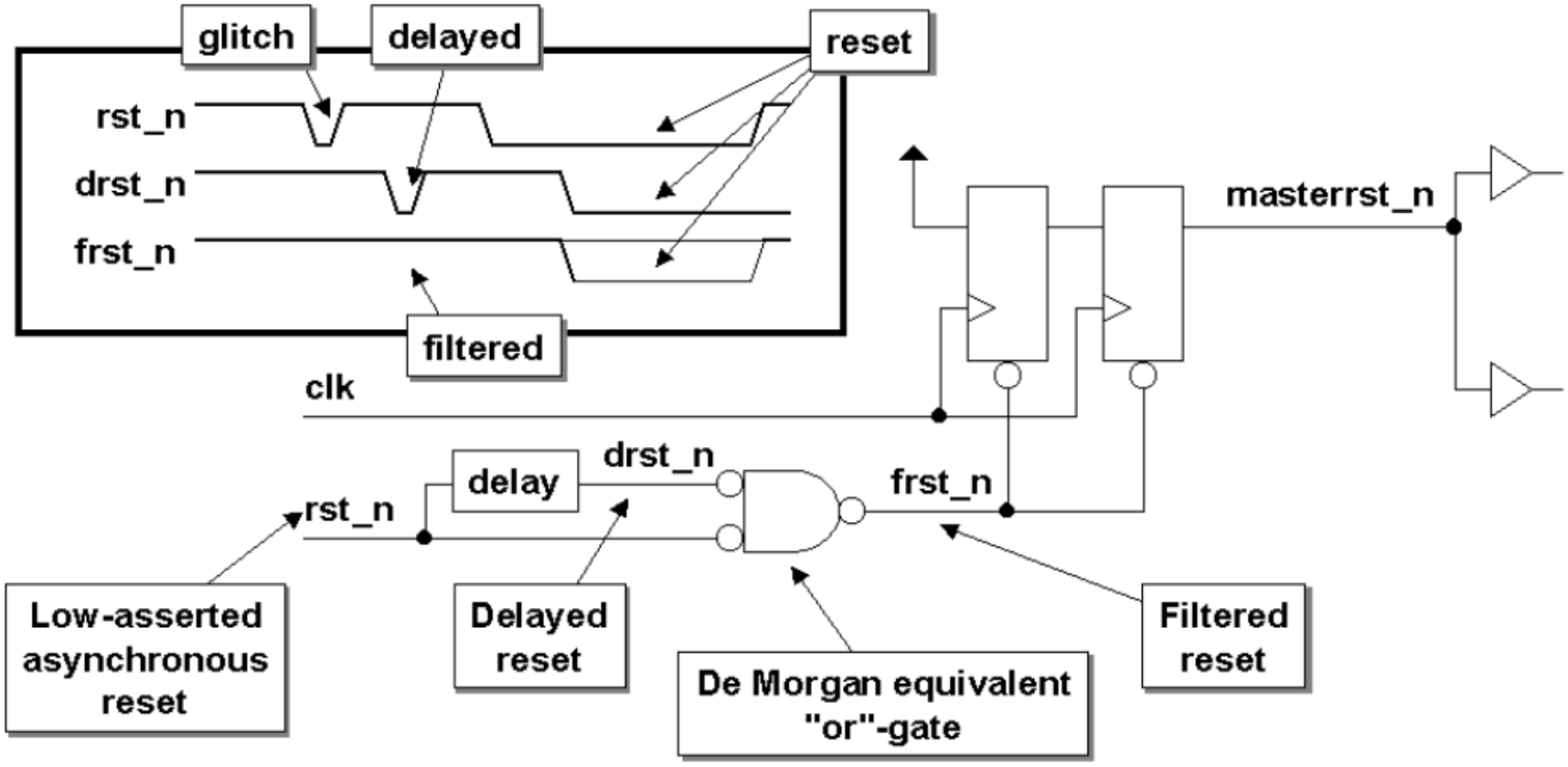

When reset is de-asserted asynchronously ...
... masterrst_n is removed synchronously
masterrst_n
pad_clk
clk
pad_rst_n
rst_n
Asynchronous reset assertion
Reset distribution buffer tree

The code for the reset synchronizer

```
module async_resetFFstyle2 (rst_n, clk, asyncrst_n);
  output rst_n;
  input clk, asyncrst_n;
  reg rst_n, rff1;

  always @(posedge clk or negedge asyncrst_n)
    if (!asyncrst_n) {rst_n,rff1} <= 2'b0;
    else {rst_n,rff1} <= {rff1,1'b1};
endmodule
```

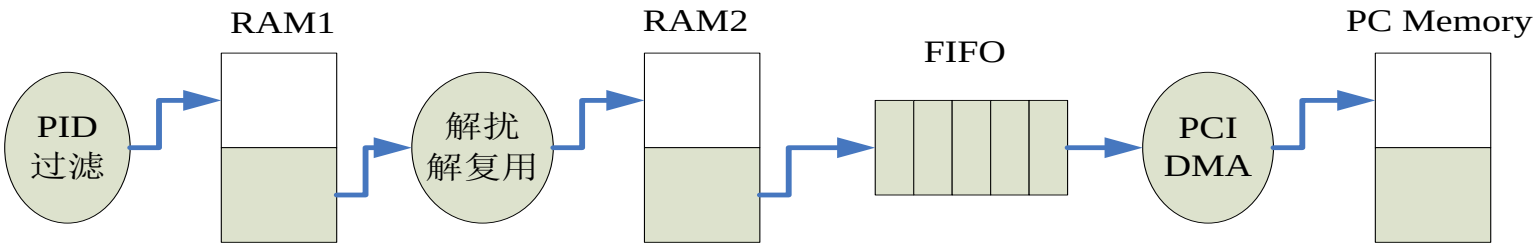
Reset毛刺消除


glitch
delayed
reset
rst_n
drst_n
frst_n
filtered
masterrst_n
clk
drst_n
delay
rst_n
frst_n
Low-asserted asynchronous reset
Delayed reset
De Morgan equivalent "or"-gate
Filtered reset

课程安排

- 亚稳态的基本概念
- 慢时钟信号进入快时钟域的处理方法
- 快时钟信号进入慢时钟域的处理方法
- 异步复位（Reset）路径处理方法
- **设计实例**

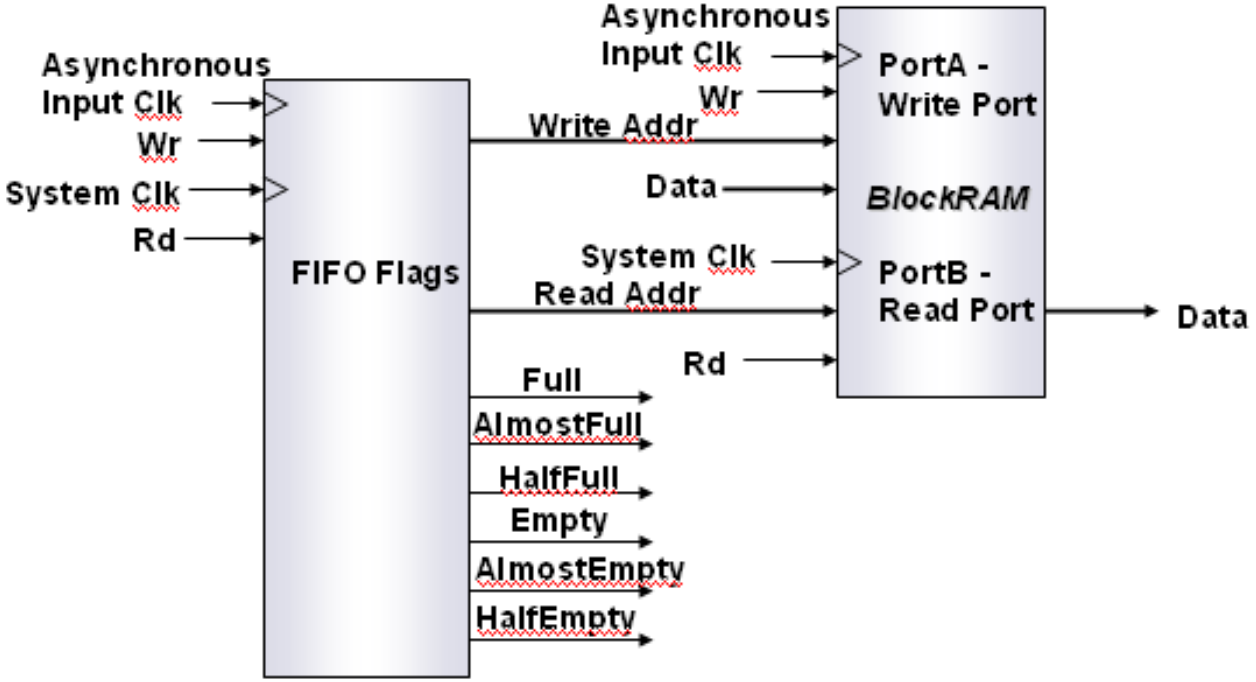
数据接口－采用DPRAM



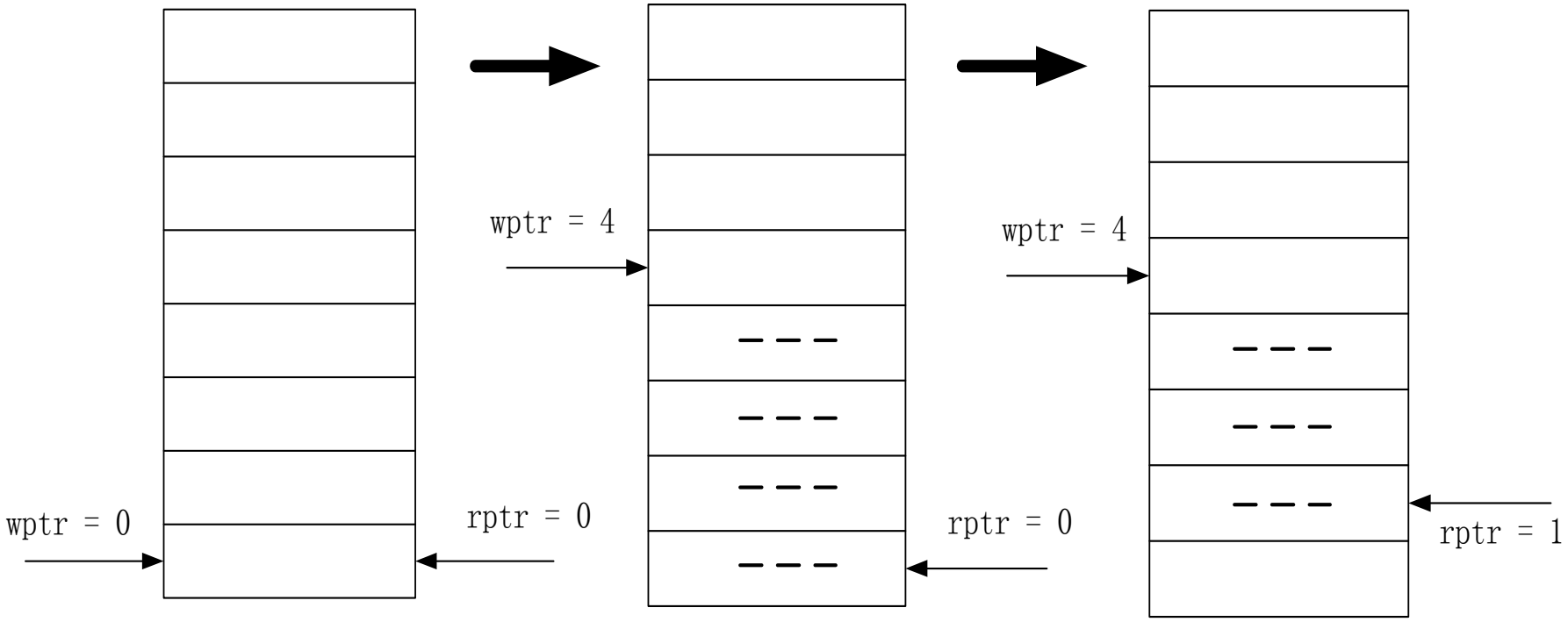
- 以定长包为单位的处理
- 提高数据处理的并行性
- 提高数据传输速率
- 隔离时钟域之间的冲突
  - 66MHz， 33MHz, 8MHz
  - 双口RAM与同步器

多时钟系统设计的经典案例-异步FIFO

- Use a FIFO to cross domains



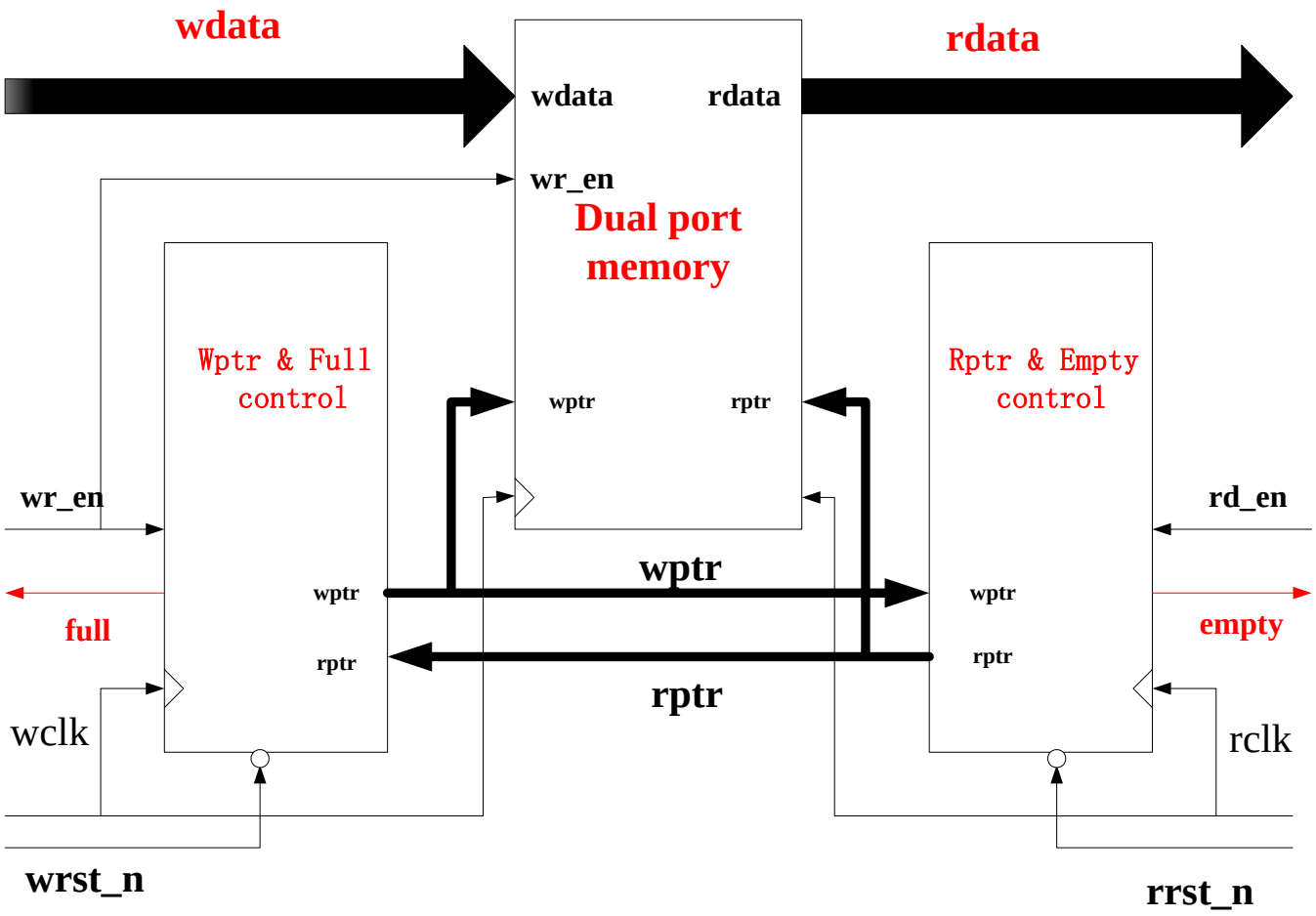
What is FIFO



Initial state

4 data are written in fifo, while one data has been read

FIFO Structure



Asynchronous Signals in FIFO Design

- FIFO can not be read when there is no data in it - empty
- FIFO can not be written when the number of data in it reaches FIFO depth – full
- FIFO full or empty signals are generated by 2 addres s pointers, wptr and rptr, which are generated from 2 different clk domains, wclk and rclk respectively.
  - wptr and rptr are asynchronous signals

FIFO Design: Asynchronous Case – cont.

- Suggestion:
  - FIFO pointers implemented as Gray-code counter which only change one bit at a time
    - Example of binary and Gray codes:

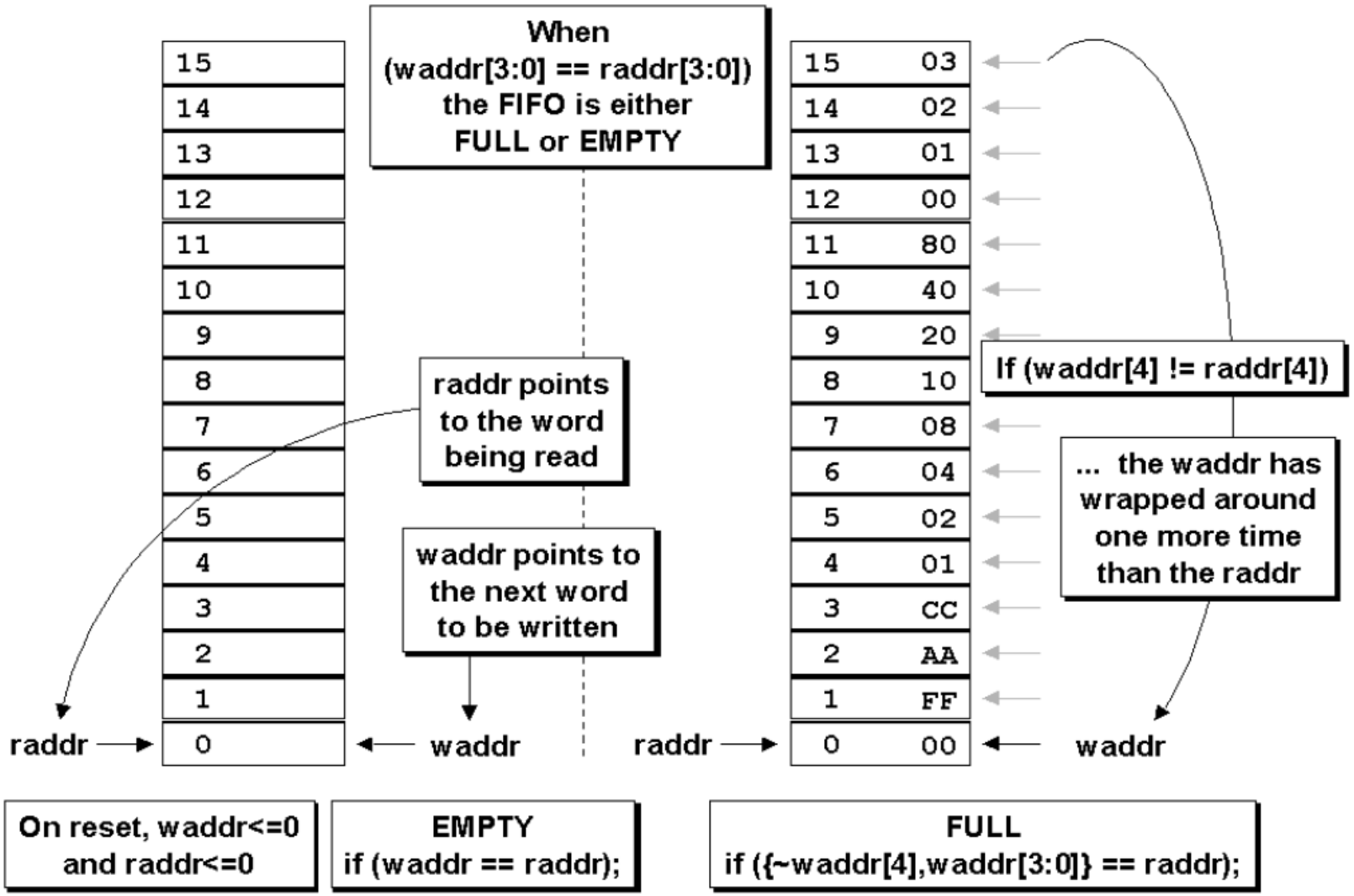
      Binary code: 00, 01, 10, 11

      Gray code: 00, 01, 11, 10
    - Will either be the old value or the new value if asynchronous signal co mes in the middle of a Gray code counter transition
  - The Gray-code pointers are synchronized by the clock from th e opposite clock domain to generate FIFO full /empty flags

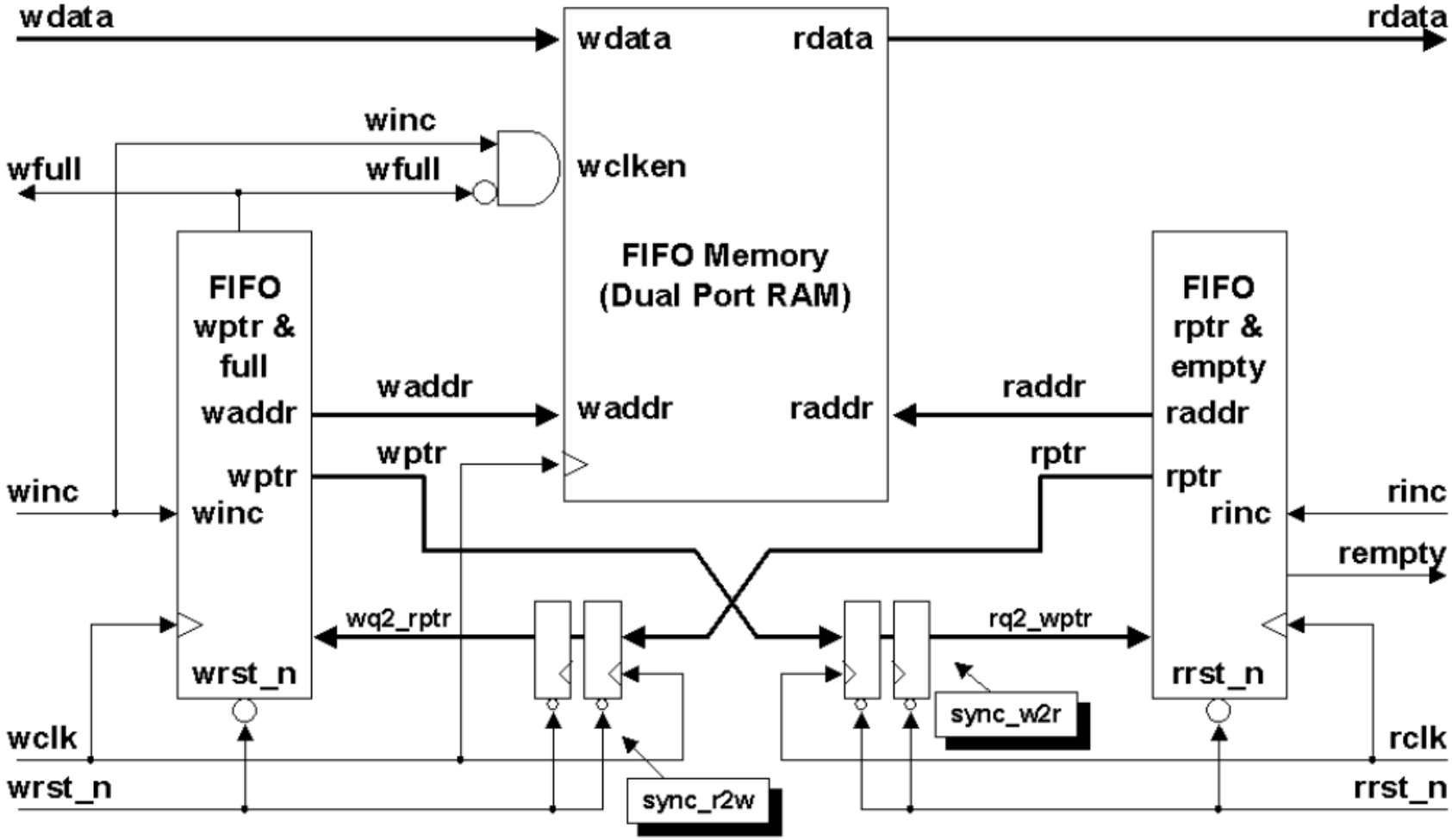
wptr == rptr → FIFO empty

wptr + 1 == rptr → FIFO full

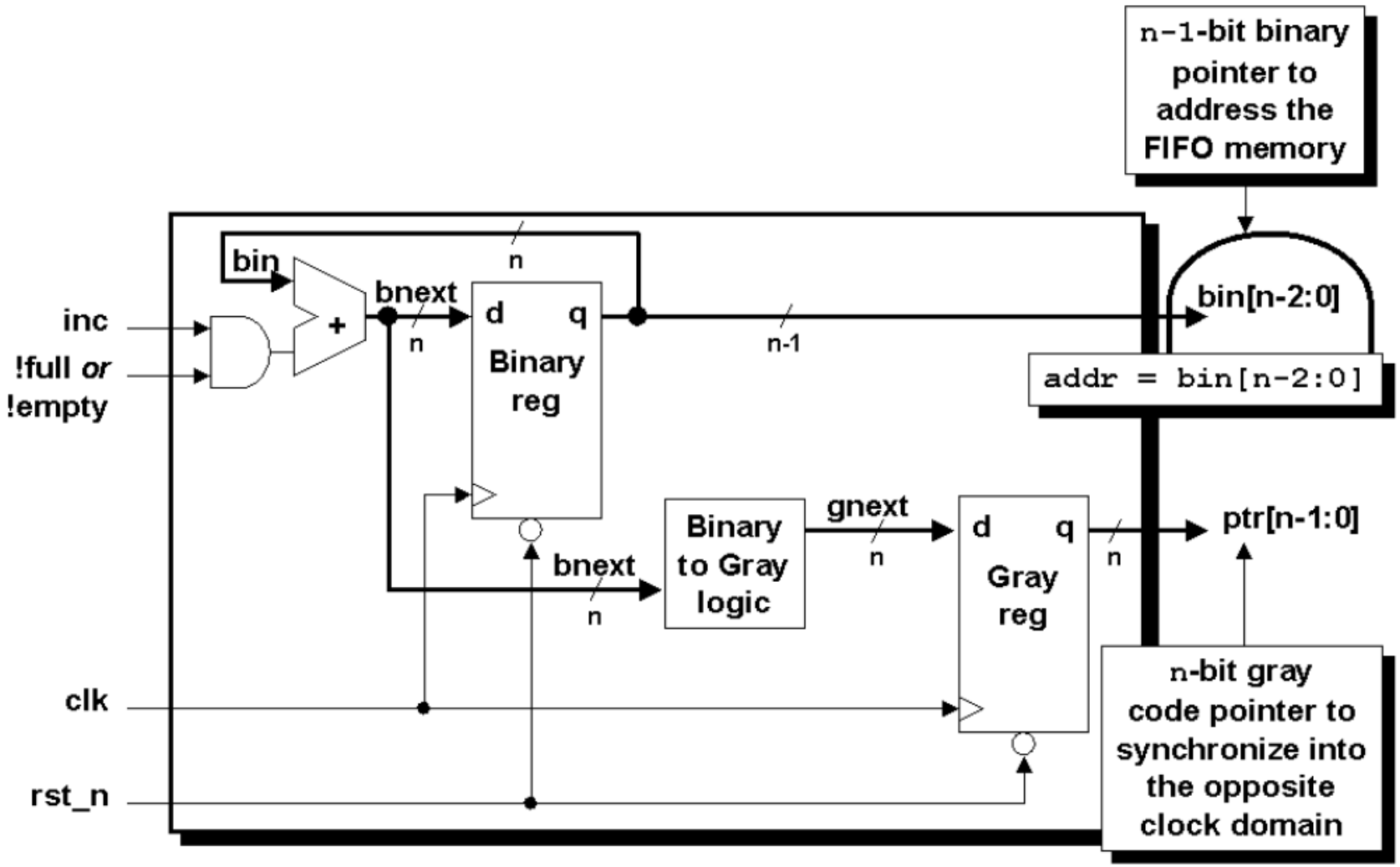
FIFO Design: Asynchronous Case – cont.



Asynchronous FIFO Architecture



Gray Code Counter



RTL Code for Asy_FIFO(1)

Top Mod

```
module fifo1 #(parameter DSIZE = 8,
               parameter ASIZE = 4)
  (output [DSIZE-1:0] rdata,
   output             wfull,
   output             rempty,
   input  [DSIZE-1:0] wdata,
   input              winc, wclk, wrst_n,
   input              rinc, rclk, rrst_n);

  wire   [ASIZE-1:0] waddr, raddr;
  wire   [ASIZE:0]   wptr, rptr, wq2_rptr, rq2_wptr;

  sync_r2w      sync_r2w  (.wq2_rptr(wq2_rptr), .rptr(rptr),
                           .wclk(wclk), .wrst_n(wrst_n));

  sync_w2r      sync_w2r  (.rq2_wptr(rq2_wptr), .wptr(wptr),
                           .rclk(rclk), .rrst_n(rrst_n));

  fifomem #(DSIZE, ASIZE) fifomem
                          (.rdata(rdata), .wdata(wdata),
                           .waddr(waddr), .raddr(raddr),
                           .wclken(winc), .wfull(wfull),
                           .wclk(wclk));

  rptr_empty #(ASIZE)     rptr_empty
                          (.rempty(rempty),
                           .raddr(raddr),
                           .rptr(rptr),  .rq2_wptr(rq2_wptr),
                           .rinc(rinc), .rclk(rclk),
                           .rrst_n(rrst_n));

  wptr_full  #(ASIZE)     wptr_full
                          (.wfull(wfull), .waddr(waddr),
                           .wptr(wptr), .wq2_rptr(wq2_rptr),
                           .winc(winc), .wclk(wclk),
                           .wrst_n(wrst_n));
endmodule
```

RTL Code for Asy_FIFO(2)

FIFO memory buffer

```
module fifomem #(parameter  DATASIZE = 8, // Memory data word width
                 parameter  ADDRSIZE = 4) // Number of mem address bits
  (output [DATASIZE-1:0] rdata,
   input  [DATASIZE-1:0] wdata,
   input  [ADDRSIZE-1:0] waddr, raddr,
   input                 wclken, wfull, wclk);

  `ifdef VENDORRAM
    // instantiation of a vendor's dual-port RAM
    vendor_ram mem (.dout(rdata), .din(wdata),
                    .waddr(waddr), .raddr(raddr),
                    .wclken(wclken),
                    .wclken_n(wfull), .clk(wclk));
  `else
    // RTL Verilog memory model
    localparam DEPTH = 1<<ADDRSIZE;
    reg [DATASIZE-1:0] mem [0:DEPTH-1];

    assign rdata = mem[raddr];

    always @(posedge wclk)
      if (wclken && !wfull) mem[waddr] <= wdata;
  `endif
endmodule
```

RTL Code for Asy_FIFO(3)

Read-domain to write-domain synchronizer

```
module sync_r2w #(parameter ADDRSIZE = 4)
   (output reg [ADDRSIZE:0] wq2_rptr,
    input      [ADDRSIZE:0] rptr,
    input                   wclk, wrst_n);
   reg [ADDRSIZE:0] wq1_rptr;

   always @(posedge wclk or negedge wrst_n)
     if (!wrst_n) {wq2_rptr,wq1_rptr} <= 0;
     else         {wq2_rptr,wq1_rptr} <= {wq1_rptr,rptr};
endmodule
```

RTL Code for Asy_FIFO(4)

Write-domain to read-domain synchronizer

```
module sync_w2r #(parameter ADDRSIZE = 4)
   (output reg [ADDRSIZE:0] rq2_wptr,
    input      [ADDRSIZE:0] wptr,
    input                   rclk, rrst_n);
   reg [ADDRSIZE:0] rq1_wptr;

   always @(posedge rclk or negedge rrst_n)
     if (!rrst_n) {rq2_wptr,rq1_wptr} <= 0;
     else         {rq2_wptr,rq1_wptr} <= {rq1_wptr,wptr};
endmodule
```

RTL Code for Asy_FIFO(5)

Read pointer & empty generation logic

```
module rptr_empty #(parameter ADDRSIZE = 4)
  (output reg                 rempty,
   output     [ADDRSIZE-1:0] raddr,
   output reg [ADDRSIZE  :0] rptr,
   input      [ADDRSIZE  :0] rq2_wptr,
   input                     rinc, rclk, rrst_n);

  reg  [ADDRSIZE:0] rbin;
  wire [ADDRSIZE:0] rgraynext, rbinnext;

  //-------------------
  // GRAYSTYLE2 pointer
  //-------------------
  always @(posedge rclk or negedge rrst_n)
    if (!rrst_n) {rbin, rptr} <= 0;
    else         {rbin, rptr} <= {rbinnext, rgraynext};

  // Memory read-address pointer (okay to use binary to address memory)
  assign raddr     = rbin[ADDRSIZE-1:0];

  assign rbinnext  = rbin + (rinc & ~rempty);
  assign rgraynext = (rbinnext>>1) ^ rbinnext;

  //---------------------------------------------------------------
  // FIFO empty when the next rptr == synchronized wptr or on reset
  //---------------------------------------------------------------
  assign rempty_val = (rgraynext == rq2_wptr);

  always @(posedge rclk or negedge rrst_n)
    if (!rrst_n) rempty <= 1'b1;
    else         rempty <= rempty_val;
endmodule
```

RTL Code for Asy_FIFO(6)

Write pointer & full generation logic

```
module wptr_full  #(parameter ADDRSIZE = 4)
  (output reg                 wfull,
   output     [ADDRSIZE-1:0] waddr,
   output reg [ADDRSIZE  :0] wptr,
   input      [ADDRSIZE  :0] wq2_rptr,
   input                     winc, wclk, wrst_n);

  reg  [ADDRSIZE:0] wbin;
  wire [ADDRSIZE:0] wgraynext, wbinnext;

  // GRAYSTYLE2 pointer
  always @(posedge wclk or negedge wrst_n)
    if (!wrst_n) {wbin, wptr} <= 0;
    else         {wbin, wptr} <= {wbinnext, wgraynext};

  // Memory write-address pointer (okay to use binary to address memory)
  assign waddr = wbin[ADDRSIZE-1:0];

  assign wbinnext  = wbin + (winc & ~wfull);
  assign wgraynext = (wbinnext>>1) ^ wbinnext;

  //------------------------------------------------------------------
  // Simplified version of the three necessary full-tests:
  // assign wfull_val=((wgnext[ADDRSIZE]    !=wq2_rptr[ADDRSIZE]  ) &&
  //                   (wgnext[ADDRSIZE-1]  !=wq2_rptr[ADDRSIZE-1]) &&
  //                   (wgnext[ADDRSIZE-2:0]==wq2_rptr[ADDRSIZE-2:0]));
  //------------------------------------------------------------------
  assign wfull_val = (wgraynext=={~wq2_rptr[ADDRSIZE:ADDRSIZE-1],
                                  wq2_rptr[ADDRSIZE-2:0]});

  always @(posedge wclk or negedge wrst_n)
    if (!wrst_n) wfull  <= 1'b0;
    else         wfull  <= wfull_val;
endmodule
```

结论

- 掌握控制信号在异步时钟域的处理方法
- 掌握数据信号在异步时钟域的处理方法
- 了解异步Reset的处理方法