RTL Coding Guideline

Chapter 1

Coding Style For Better Synthesis

Before giving further explanation and examples of both blocking and nonblocking assignments, it would be useful to outline eight guidelines that help to accurately simulate hardware, modeled using Verilog. Adherence to these guidelines will also remove 90-100% of the Verilog race conditions encountered by most Verilog designers.

- **When modeling sequential logic, use nonblocking assignments.**
- **When modeling latches, use nonblocking assignments.**
- **When modeling combinational logic with an always block, use blocking assignments.**
- **When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.**
- **Do not mix blocking and nonblocking assignments in the same always block.**
- **Do not make assignments to the same variable from more than one always block.**
- **Use $strobe to display values that have been assigned using nonblocking assignments.**
- **Do not make assignments using #0 delays**

1.1 Blocking assignments

The blocking assignment operator is an equal sign ("="). A blocking assignment gets its name because a blocking assignment must evaluate the RHS arguments and complete the assignment without interruption from any other Verilog statement. The assignment is said to "block" other assignments until the current assignment has completed. The

one exception is a blocking assignment with timing delays on the RHS of the blocking operator, which is considered to be a poor coding style. **A problem with blocking assignments occurs when the RHS variable of one assignment in one procedural block is also the LHS variable of another assignment in another procedural block and both equations are scheduled to execute in the same simulation time step**.

```
module fbosc1 (y1, y2, clk, rst);
output y1, y2;
input clk, rst;
reg y1, y2;
always @(posedgeclk or posedgerst)
if (rst) y1 = 0; // reset
else y1 = y2;
always @(posedgeclk or posedgerst)
if (rst) y2 = 1; // preset
else y2 = y1;
endmodule
```

1.2 Nonblocking assignments

The nonblocking assignment operator is the same as the less-than-or-equal-to operator ("<="). A nonblocking assignment gets its name because the assignment evaluates the RHS expression of a nonblocking statement at the beginning of a time step and schedules the LHS update to take place at the end of the time step.

Execution of nonblocking assignments can be viewed as a two-step process:

- Evaluate the RHS of nonblocking statements at the beginning of the time step.
- Update the LHS of nonblocking statements at the end of the time step.

Nonblocking assignments are only made to register data types and are therefore only permitted inside of procedural blocks, such as initial blocks and always blocks. Nonblocking assignments are not permitted in continuous assignments

1.3 The Verilog "stratified event queue"

the "stratified event queue" is logically partitioned into four distinct queues for the current simulation time and additional queues for future simulation times.

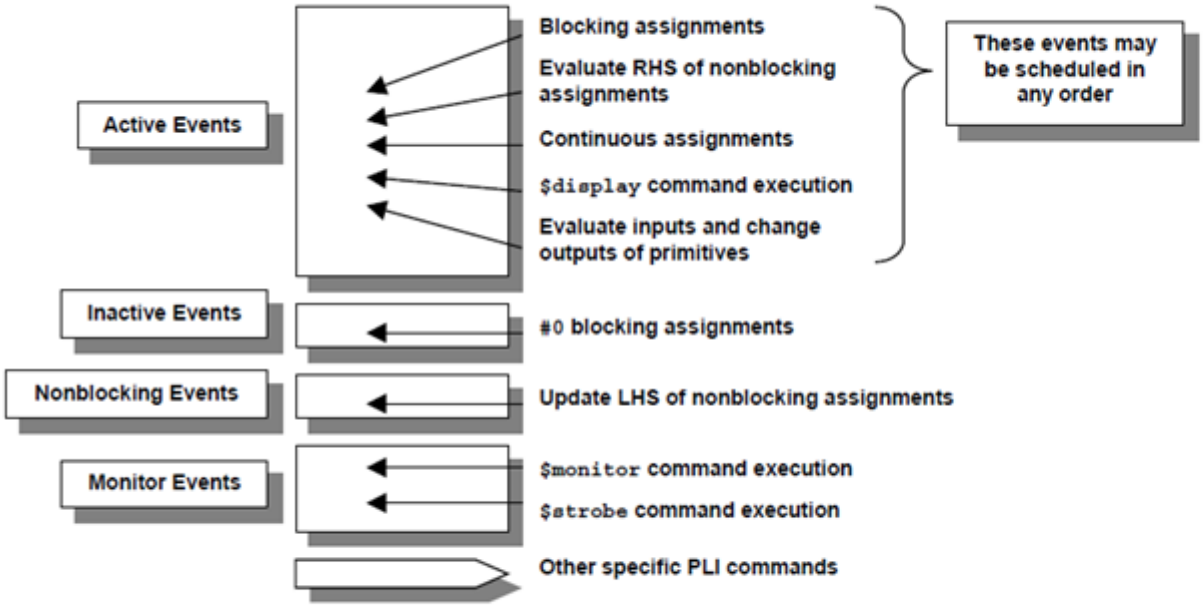

Figure 1 - Verilog "stratified event queue"

Figure 1.1: Verilog "stratified event queue".

The **active events** queue is where most Verilog events are scheduled, including blocking assignments, continuous assignments, $display commands, evaluation of instance and primitive inputs followed by updates of primitive and instance outputs, and the evaluation of nonblocking RHS expressions. The LHS of nonblocking assignments are not updated in the active events queue.
Events are added to any of the event queues but are only removed from the active events queue. Events that are scheduled on the other event queues will eventually become "activated," or promoted into the active events queue.

The practice of making #0-delay assignments is generally a flawed practice employed by designers who try to make assignments to the same variable from two separate procedural blocks, attempting to beat Verilog race conditions by scheduling one of the assignments to take place slightly later in the same simulation time step. **Adding #0-delay assignments to Verilog models needlessly complicates the analysis of scheduled events**.

1.4 Self-triggering always blocks

In general, a Verilog always block cannot trigger itself. Consider the oscillator example in Example 1.1. This oscillator uses blocking assignments. Blocking assignments evaluate their RHS expression and update their LHS value without interruption. The blocking assignment must complete before the @(clk) edge-trigger event can be scheduled. By the time the trigger event has been scheduled, the blocking clk assignment has completed; therefore, there is no trigger event from within the always block to trigger the @(clk) trigger.

Listing 1.1: Non-self-triggering oscillator using blocking assignments

```
module osc1 (clk);
output clk;
reg clk;
initial #10 clk = 0;
always @(clk) #10 clk = !clk;
endmodule
```

In contrast, the oscillator in Example 1.2 uses nonblocking assignments. After the first @(clk) trigger, the RHS expression of the nonblocking assignment is evaluated and the LHS value scheduled into the nonblocking assign updates event queue. Before the nonblocking assign updatesevent queue is "activated," the @(clk) trigger statement is encountered and the always block again becomes sensitive to changes on the clksignal. When the nonblocking LHS value is updated later in the same time step, the @(clk)

is again triggered. **osc2 example is self triggering(which is not necessarily a recommended coding style)**.

Listing 1.2: Self-triggering oscillator using nonblocking assignments

```
module osc2 (clk);
outputclk;
regclk;
initial #10 clk = 0;
always @(clk) #10 clk<= !clk;
endmodule
```

1.5 Combinational logic - use blocking assignments

The code shown in Example 1.3 builds the y-output from three sequentially executed statements. Since nonblocking assignments evaluate the RHS expressions before updating the LHS variables, the values of tmp1 and tmp2 were the original values of these two variables upon entry to this always block and not the values that will be updated at the end of the simulation time step. The y-output will reflect the old values of tmp1 and tmp2, not the values calculated in the current pass of the always block

Listing 1.3: Bad combinational logic coding style using nonblocking assignments

```
module ao4 (y, a, b, c, d);
output y;
input a, b, c, d;
reg y, tmp1, tmp2;
always @(a or b or c or d) begin
tmp1 <= a & b;
tmp2 <= c & d;
y <= tmp1 | tmp2;
end
endmodule
```

The code shown in Example 1.4 is identical to the code shown in Example 1.3, except that tmp1 and tmp2 have been added to the sensitivity list. As describe in section 1.2, when the nonblocking assignments update the LHS variables in the nonblocking assign update events queue, the always block will self-trigger and update the y-outputs with the newly calculated tmp1 and tmp2 values. **y-output value will now be correct after taking two passes through the always block. Multiple passes through an always block equates to degraded simulation performance and should be avoided if a reasonable alternative exists (use blocking statements for combinational modeling)**.

Listing 1.4: combinational logic coding style using nonblocking assignments

```
module ao5 (y, a, b, c, d);
output y;
input a, b, c, d;
reg y, tmp1, tmp2;
always @(a or b or c or d or tmp1 or tmp2) begin
tmp1 <= a & b;
tmp2 <= c & d;
y <= tmp1 | tmp2;
end
endmodule
```

NOTE

- Using the $display command with nonblocking assignments does not work
- "Making multiple nonblocking assignments to the same variable in the same always block is defined by the Verilog Standard. The last nonblocking assignment to the same variable wins"

1.6 FSM

A common classification used to describe the type of an FSM is Mealy and Moore state machines[2] [3]. A Moore FSM is a state machine where the outputs are only a function of the present state. A Mealy FSM is a state machine where one or more of the outputs is a function of the present state and one or more of the inputs.

1.6.1 Binary Encoded or Onehot Encoded?

Common classifications used to describe the state encoding of an FSM are Binary (or highly encoded) and Onehot.

A binary-encoded FSM design only requires as many flip-flops as are needed to uniquely encode the number of states in the state machine. The actual number of flip-flops required is equal to the ceiling of the log-base-2 of the number of states in the FSM.

A onehot FSM design requires a flip-flop for each state in the design and only one flip-flop (the flip-flop representing the current or "hot" state) is set at a time in a onehot FSM design. For a state machine with 9- 16 states, a binary FSM only requires 4 flip-flops while a onehot FSM requires a flip-flop for each state in the design (9-16 flip-flops).

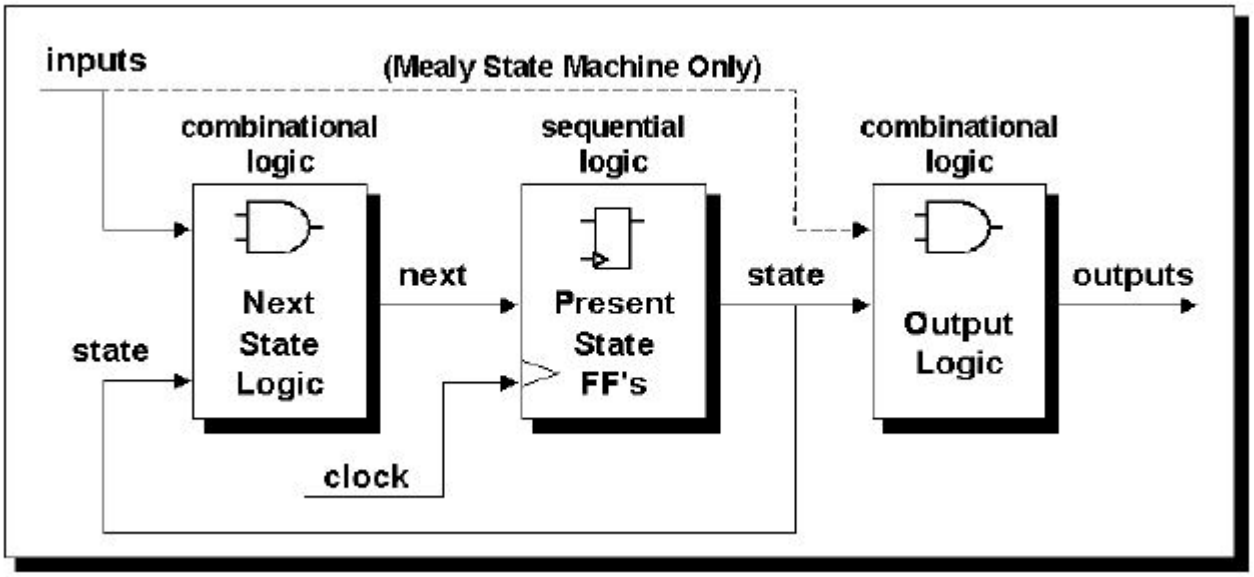

Figure 1.2: Finite State Machine (FSM) block diagram.

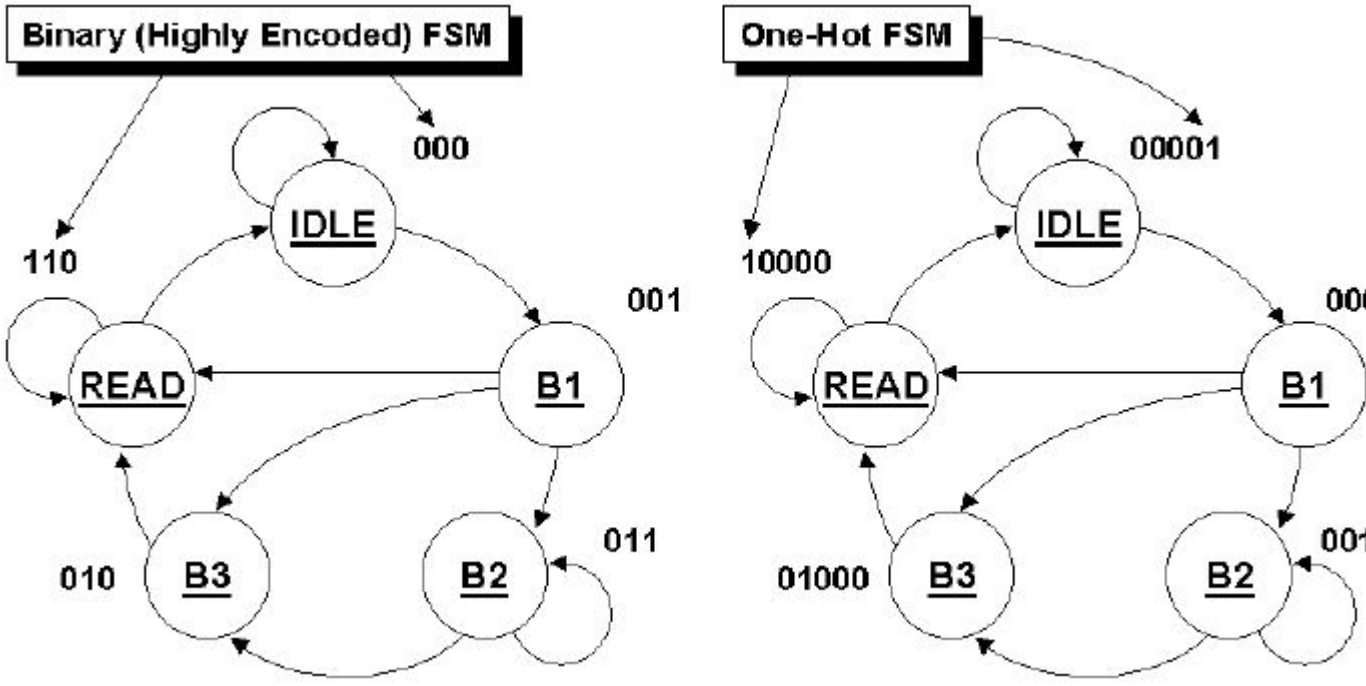

Figure 1.3: FSM encoding.

FPGA vendors frequently recommend using a onehot state encoding style because flip-flops are plentiful in an FPGA and the combinational logic required to implement a onehot FSM design is typically smaller than most binary encoding styles. Since FPGA performance is typically related to the combinational logic size of the FPGA design, onehot FSMs typically run faster than a binary encoded FSM with larger combinational logic blocks[4].

1.6.2 One Always Block FSM Style (Not Recommended)

One of the most common FSM coding styles in use today is the one sequential always block FSM coding style. For most FSM designs, the one always block FSM coding style is more verbose, more confusing and more error prone than a comparable two always block coding style.

1.6.3 Two Always Block FSM Style

One of the best Verilog coding styles is to code the FSM design using two always blocks, one for the sequential state register and one for the combinational next-state and combinational output logic.

Listing 1.5: fsm design - two always block style

```
module fsm_cc4_2
(output reg gnt,
input dly, done, req, clk, rst_n);
parameter [1:0] IDLE = 2'b00,
BBUSY = 2'b01,
BWAIT = 2'b10,
BFREE = 2'b11;
reg [1:0] state, next;
always @(posedge clk or negedge rst_n)
if (!rst_n) state <= IDLE;
else state <= next;
always @(state or dly or done or req) begin
next = 2'bx;
gnt = 1'b0;
case (state)
IDLE : if (req) next = BBUSY;
else next = IDLE;
BBUSY: begin
gnt = 1'b1;
if (!done) next = BBUSY;
else if ( dly) next = BWAIT;
else next = BFREE;
```

```
end
BWAIT: begin
gnt = 1'b1;
if (!dly) next = BFREE;
else next = BWAIT;
end
BFREE: if (req) next = BBUSY;
else next = IDLE;
endcase
end
endmodule
```

FSM Coding Notes

- Parameters (Parameters are constants that are local to a module) are used to define state encodings instead of the Verilog 'define macro definition construct. After parameter definitions are created, the parameters are used throughout the rest of the design, not the state encodings.
- The sequential always block is coded using nonblocking assignments.
- The combinational always block sensitivity list is sensitive to changes on the state variable and all of the inputs referenced in the combinational always block.
- Assignments within the combinational always block are made using Verilog blocking assignments.
- Default output and next state assignments are made before coding the case statement as shown in 1.5. This eliminates latches and reduces the amount of code required to code the rest of the outputs in the case statement and highlights in the case statement exactly in which states the individual output(s) change).
- Assignments within the combinational always block are made using Verilog blocking assignments.

1.6.4 Onehot FSM Coding Style

Efficient (small and fast) onehot state machines can be coded using an inverse case statement; a case statement where each case item is an expression that evaluates to true or false.

Listing 1.6: fsm design -onehot style

```
module fsm_cc4_fp
```

```
(output reg gnt,
input dly, done, req, clk, rst_n);
parameter [3:0] IDLE = 0,
BBUSY = 1,
BWAIT = 2,
BFREE = 3;
reg [3:0] state, next;
always @(posedge clk or negedge rst_n)
if (!rst_n) begin
state <= 4'b0;
state[IDLE] <= 1'b1;
end
else state <= next;
always @(state or dly or done or req) begin
next = 4'b0;
gnt = 1'b0;
case (1'b1) // ambit synthesis case = full, parallel
state[IDLE] : if (req) next[BBUSY] = 1'b1;
else next[IDLE] = 1'b1;
state[BBUSY]: begin
gnt = 1'b1;
if (!done) next[BBUSY] = 1'b1;
else if ( dly) next[BWAIT] = 1'b1;
else next[BFREE] = 1'b1;
end
state[BWAIT]: begin
gnt = 1'b1;
if (!dly) next[BFREE] = 1'b1;
else next[BWAIT] = 1'b1;
end
state[BFREE]: begin
if (req) next[BBUSY] = 1'b1;
else next[IDLE] = 1'b1;
end
endcase
end
endmodule
```

1.6.5 Registered FSM Outputs

synthesis results by standardizing the output and input delay constraints of synthesized modules [5].

FSM outputs are easily registered by adding a third always sequential block to an FSM module where output assignments are generated in a case statement with case items corresponding to the next state that will be active when the output is clocked.

Listing 1.7: fsm design -three always blocks w/registered outputs

```
module fsm_cc4_fp
(output reg gnt,
input dly, done, req, clk, rst_n);
parameter [3:0] IDLE = 0,
BBUSY = 1,
BWAIT = 2,
BFREE = 3;
reg [3:0] state, next;
always @(posedge clk or negedge rst_n)
if (!rst_n) begin
state <= 4'b0;
state[IDLE] <= 1'b1;
end
else state <= next;
always @(state or dly or done or req) begin
next = 4'b0;
gnt = 1'b0;
case (1'b1) // ambit synthesis case = full, parallel
state[IDLE] : if (req) next[BBUSY] = 1'b1;
else next[IDLE] = 1'b1;
state[BBUSY]: begin
gnt = 1'b1;
if (!done) next[BBUSY] = 1'b1;
else if ( dly) next[BWAIT] = 1'b1;
else next[BFREE] = 1'b1;
end
state[BWAIT]: begin
gnt = 1'b1;
if (!dly) next[BFREE] = 1'b1;
else next[BWAIT] = 1'b1;
end
state[BFREE]: begin
if (req) next[BBUSY] = 1'b1;
else next[IDLE] = 1'b1;
end
endcase
end
```

```
endmodule
```

One or Two or Three always blocks for FSM??

- Use a two always block coding style to code FSM designs with combinational outputs. This style is efficient and easy to code and can also easily handle Mealy FSM designs.
- Use a three always block coding style to code FSM designs with registered outputs. This style is efficient and easy to code.

1.7 Clock Domain Crossing

1.7.1 Metastability

Metastbility refers to signals that do not have stable 0 or 1 states for some duration of time at some point during normal operation of a design. In a multi-clock design, **metastability cannot be avoided but the detrimental effects of metastability can be neutralized**.

Figure 1.5 shows a synchronization failure that occurs when a signal generated in one clock domain is sampled too close to the rising edge of a clock signal from a second clock domain. Synchronization failure is caused by an output going metastable and not converging to a legal stable state by the time the output must be sampled again.

1.7.2 Why is metastability a problem?

metastable output that traverses additional logic in the receiving clock domain can cause illegal signal values to be propagated throughout the rest of the design. Since the CDC signal can fluctuate for some period of time, the input logic in the receiving clock domain might recognize the logic level of the fluctuating signal to be different values and hence propagate erroneous signals into the receiving clock domain.

1.7.3 Synchronizers

There are two scenarios that are possible when passing signals across CDC boundaries, and it is important to determine which scenario applies to your design:

- It is permitted to miss samples that are passed between clock domains.
- Every signal passed between clock domains must be sampled.

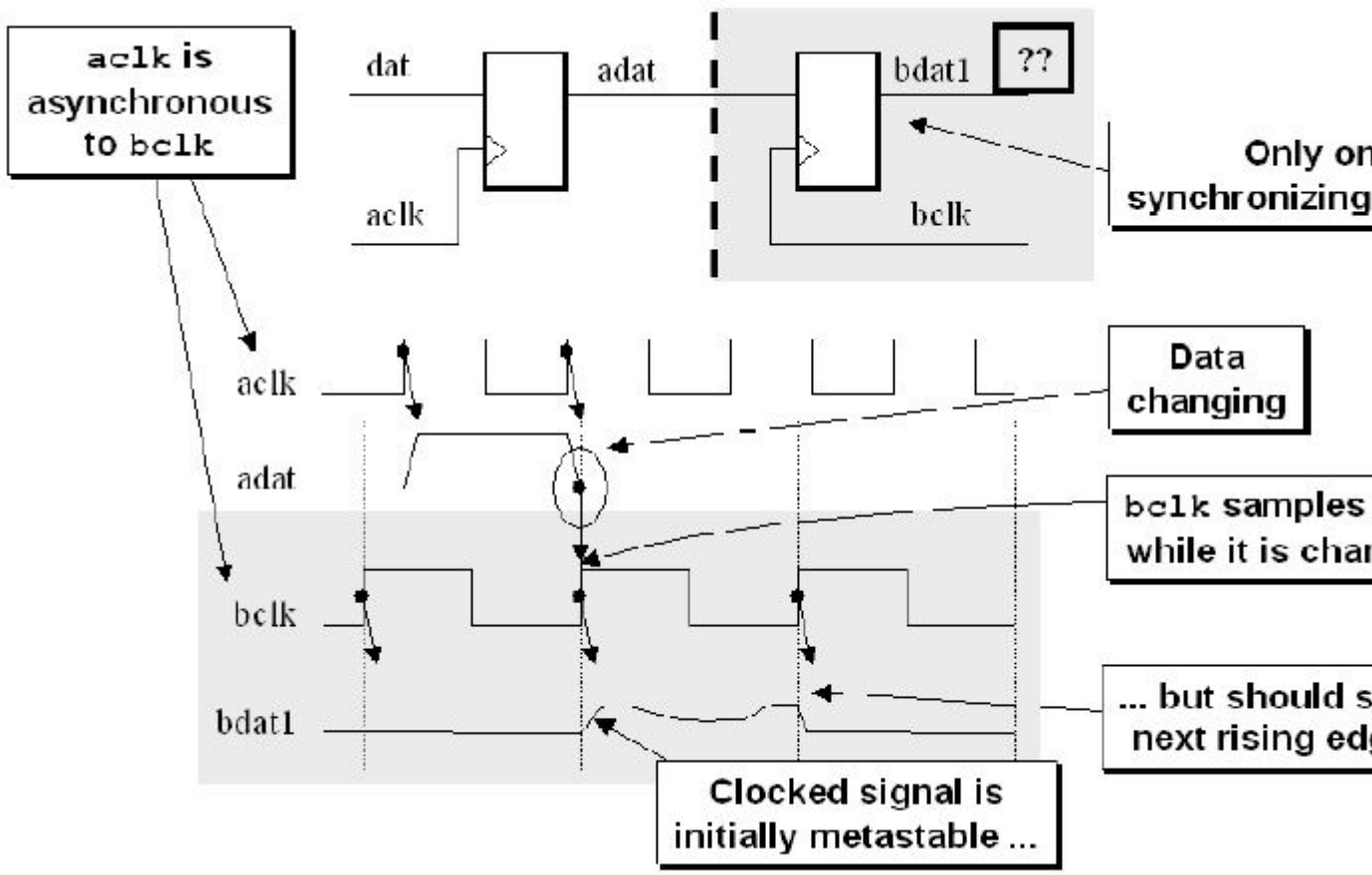

Figure 1.4: Asynchronous clocks and synchronization failure

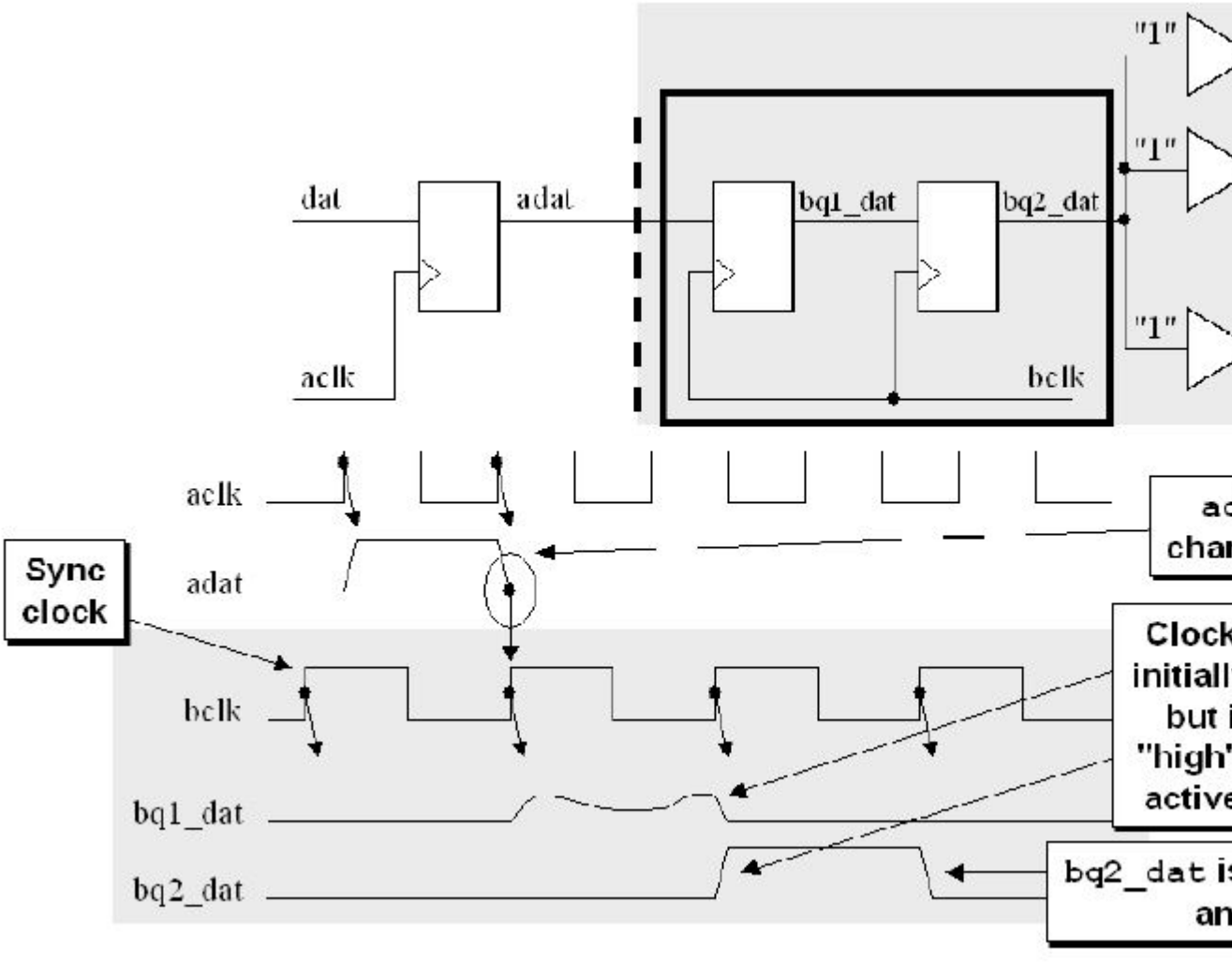

Figure 1.5: Asynchronous clocks and synchronization failure

1.7.4 Two flip-flop synchronizer

"A synchronizer is a device that samples an asynchronous signal and outputs a version of the signal that has transitions synchronized to a local or sample clock."
The simplest and most common synchronizer used by digital designers is a two-flip-flop synchronizer as shown in Figure 3.

References

[1] http://www.sunburst-design.com/papers/

[2] William I. Fletcher, An Engineering Approach To Digital Design, New Jersey, Prentice-Hall, 1980.

[3] Zvi Kohavi, Switching And Finite Automata Theory, Second Edition, New York, McGraw-Hill Book Company, 1978.

[4] The Programmable Logic Data Book, Xilinx, 1994, pg. 8-171.

[5] Clifford E. Cummings, "Coding And Scripting Techniques For FSM Designs With Synthesis-Optimized, Glitch- Free Outputs," SNUG'2000 Boston (Synopsys Users Group Boston, MA, 2000) Proceedings, September 2000.