### 4.3.1 Multiclocked Sequences and Properties

🕮 **Rule**: *[1] Multiclocked sequences are built by concatenating singly clocked subsequences using the single delay concatenation operator ##1 or the zero-delay concatenation operator ##0. The single delay indicated by ##1is understood to be from the end point of the first sequence, which occurs at a tick of the first clock, to the nearest strictly subsequent tick of the second clock, where the second sequence begins. The zero delay indicated by ##0 is understood to be from the end point of the first sequence, which occurs at a tick of the first clock, to the nearest possibly overlapping tick of the second clock, where the second sequence b*

🕮 **Rule**: Properties can use `|->, |=>` : Multiclocked properties can use the overlapping `|->` or non-overlapping implication `|=>` operators to create a multiclocked property from an antecedent sequence and a consequent property. The `|=>` or the `|->` operators synchronize the last expression of the antecedent clocked with the antecedent clock and the first elements of the consequent property being evaluated clocked with the consequent clock. The synchronization is the same as the one used with `##1` (for the `|=>`) and `##0` (for the `|->`) operators.
Consider the following two assertions (***/ch4/4.3/mclk2.sv***)

```
ap0: assert property(@(posedge clk1) $rose(a) |-> @(posedge clk2)  b);
ap1: assert property(@(posedge clk1) $rose(a) |=> @(posedge clk2)  b);
```

| Condition | $rose(a) | @(**posedge** clk1) **$rose**(a) \|-> @(**posedge** clk2) b) | @(**posedge** clk1) **$rose**(a) \|=> @(**posedge** clk2) b) |
|---|---|---|---|
| **At time t1**<br>**posedge** clk1 is true event<br>**posedge** clk2 is true event | True | **b is evaluated at t1** | **b is evaluated at** @(**posedge** clk2) after t1 |
| **At time t2**<br>**posedge** clk1 is true event<br>**posedge** clk2 is false | True | **b is evaluated at** @(**posedge** clk2) after t2 | **b is evaluated at** @(**posedge** clk2) after t2 |

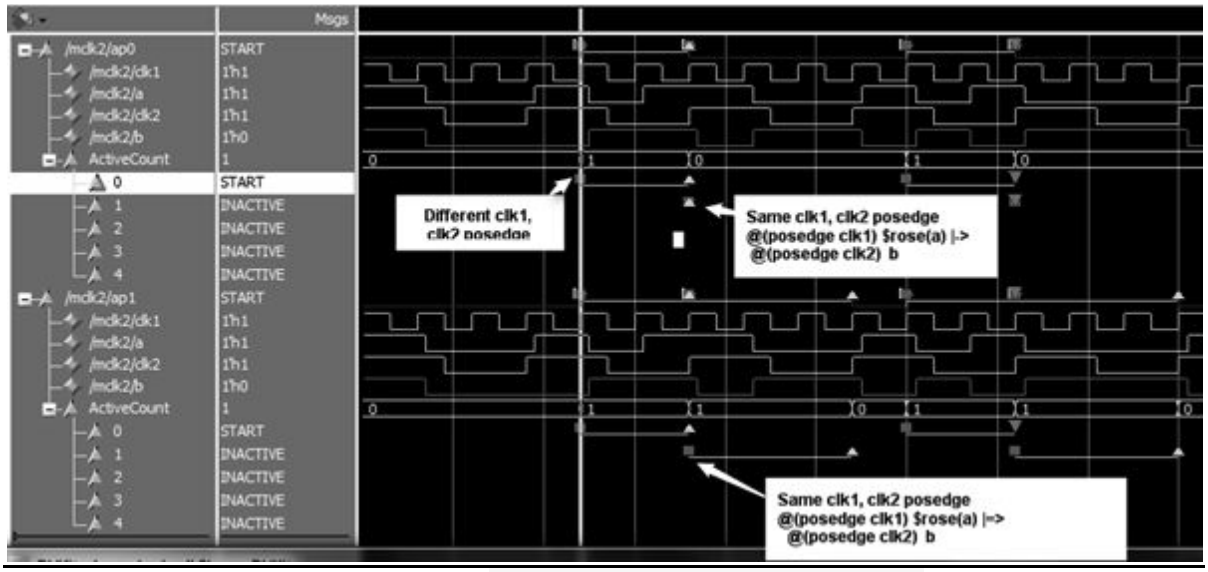



**Figure 4.3.1-1 Multiclocked property** */ch4/4.3/mclk2.png* */ch4/4.3/mclk2.sv*

🕮 **Rule**: `##1, ##0` concatenation: Multiclocked subsequences can only be combined with the `##1` or `##0` operators. The use of the `and`, `or`, `throughout`, `within`, or `intersect` operator would be illegal to combine multiclocked subsequences. Thus,

```
@(posedge clk1) a  ##2 @(posedge clk2) b // 💣 ##2 is illegal
    ##1 @(posedge clk3) (c##1 d)
    intersect   @(posedge clk4) e ##1 f; // 💣 intersect is illegal for multiclocked subsequences
```

🕮 **Rule**: Unique leading clocking event: After expansion and considerations of clock flow, a concurrent assertion must have a unique leading clocking event. This provides the basis for the start of a new attempt (if there was no unique leading clock then the statistics would be meaningless.) The leading clocking event can be explicit, or inferred from the default clocking event or derived from contextually inferred clocking event. Thus,

```
module lce;  // /ch4/4.3/lce.sv
    bit clk, clk0, clk2, a, b, c, d, e;
    initial forever #10 clk=!clk;
    // Illegal, no leading clocking event, clock does not flow through
    ap_and2_seq_ERROR: assert property(   // 💣
        (@ (posedge clk) a ##1 b) and // (a ##1 b) has a leading clocking event
        (c ##2 d) ); // 💣 illegal, (c ##2 d) has no leading clocking event here

  // Illegal, no unique leading clocking event,  both sequences start concurrently
    ap_and2_seq_ERROR2: assert property(   // 💣
       (@ (posedge clk) a ##1 b) and
       (@ (posedge clk2) c ##2 d) );   // has multiple leading clocks for its maximal property

    // OK, one leading clocking event: posedge clk0
    ap_and2ab: assert property(@ (posedge clk0) 1|-> // ✓
        (@ (posedge clk) a ##1 b) and
       (@ (posedge clk2) c ##2 d) );

    // Illegal, no unique leading clocking event, both properties start concurrently
    ap_and2_Prop_ERROR: assert property(    // 💣
       (@ (posedge clk)  1 |-> a ##1 b) and // A variable could have been used instead of "1"
       (@ (posedge clk2) 1 |-> c ##2 d) );

     // OK, Same leading clocking event,  both properties start concurrently
    ap_and2_Prop_OK_same_clk: assert property(   // ✓
       (@ (posedge clk) 1 |-> a ##1 b) and
       (@ (posedge clk) 1 |-> c ##2 d));


    // OK, one leading clocking event
    ap_and2_Prop_OK: assert property(@ (posedge clk0) 1 |-> // ✓
       (@ (posedge clk) 1 |-> a ##1 b) and
       (@ (posedge clk2) 1 |-> c ##2 d) );
endmodule : lce
```

🕮 **Rule**: No empty match matches in the clock boundary The maximal (i.e., top) singly clocked subsequences of a multiclocked sequence are required to admit only nonempty matches. Thus, you cannot concatenate subsequences with an empty match in multiclocked sequences. For example:

```
ap_mult_clk: assert property(   // ch4/4.3/lce.sv
       @ (posedge clk1) a[*0:1] ##1  // 💣 illegal a[*0] is empty
       @ (posedge clk2) b[*1:2] ##1 c);
 // The LHS of '##' operator, across which the clock flow changes, cannot match the empty word.
```

Above is equivalent to:

```
(@(posedge clk1) a[*0] ##1 @(posedge clk2) b[*1] ##1 c) or  // 💣 a[*0] is empty
(@(posedge clk1) a[*1] ##1 @(posedge clk2) b[*1] ##1 c) or // OK
(@(posedge clk1) a[*0] ##1 @(posedge clk2) b[*2] ##1 c) or // 💣 a[*0] is empty
(@(posedge clk1) a[*1] ##1 @(posedge clk2) b[*2] ##1 c) ;  // OK
```

### 4.3.2 Clocking Rules in Assertions

🕮 **Rule**: Clocking in a property propagates to verification directives: A clocking event specified inside a PROPERTY declaration is propagated to the enclosing verification directive (e.g., **assert, cover, assume**). Thus,

```
property p_with_one_clock; @(posedge clk) a |=> b; endproperty : p_with_one_clock
ap_with_one_clock : assert property (p_with_one_clock);  // ✓
```

### 4.3.3 Clock Flow

🕮 **Rule:** Flow through `|->, |=>` **:** When no explicit clock event is specified in an implication operator, the clock from the end point of the antecedent is understood to flow across the operator. Thus, the following to property expressions are identical:

```
@(posedge clk0) a0 |-> @(posedge clk0) a1 ##1 @(posedge clk2) a2;  // ✓
@(posedge clk0) a0 |-> a1 ##1 @(posedge clk2) a2;  // ✓
```

`clk0` implicitly flows across the implication operator. `a1` is clocked with

🕮 **Rule:** The clock definition can change when crossing `|->, |=>` operators.

```
@(posedge clk0) a0 |-> @(posedge clk2) a1;  // ✓
@(posedge clk0) a0 |=> @(posedge clk2) a1;  // ✓
```

🕮 **Rule:** Clock flow can change in conditional branch of **`if`:** A property expression of the form

```
if ( expression_or_dist ) property_expr [ else property_expr ]
```

Clock flow can change in the conditional branch of **`if`** property operator (i.e., from the Boolean condition in the **`if`** statement to the beginnings of the **`if`** and **`else`** clause properties). Thus:

```
ap_ifOK: assert property (@ (posedge clk)
          if (a) @ (posedge clk1) b|=> c    // ✓ b, c tested at clk1
          else d ##1 e);                    // ✓ "a", "d", "e" tested at current (posedge clk)

ap_ifOK2: assert property ( @ (posedge clk)
          if (a) b|=> @ (posedge clk1) c    // ✓ "a", "b","d" tested @ (posedge clk)
          else   d ##1 @ (posedge clk2) e); // ✓ " c" @ (posedge clk1), e @ (posedge clk2),
```

🕮 **Rule**: *[1] A clocking event in the declaration of a sequence or property does not flow out of an instance of that sequence or property*. However, the clock flows across elements of same sequence


**clk trapped in sequence**

**Clocking event within sequence or property does NOT flow out**

For example, assuming no default clocking:

```
sequence q_ef;   @ (posedge clk) e ##[1:5] f;   endsequence : q_ef
ap_q_ef_a: assert property (q_ef ##1 a); // 💣 Illegal, clk does not flow into "a"
                                         // (i.e., no clocking event).
ap_ok: assert property(@ (posedge clk)
                       e ##[1:5] f ##1 a); // ✓ clock flows across elements of same sequence
ap_error: assert property (not q_ef);  // 💣 Illegal. Clocking event does not flow out of an
  // instance of the sequence q_ef. Thus, the not property operator has no leading clocking event.
ap_qWith_one_clock : assert property (q_ef);  // ✓ leading clocking event specified inside a
                  // named sequence is propagated to the enclosing assertion statement
```

🕮 **Rule:** Clocking event is trapped in parenthesized sequence: *[1] The scope of a clocking event flows into parenthesized subexpressions and, if the subexpression is a sequence, also flows left-to-right across the parenthesized subexpression. However, the scope of a clocking event does not flow out of enclosing parentheses.* The standard also states that when sequence instances are flattened, the resulting expression that is returned is enclosed in parenthesis; therefore clocks do not flow out of sequence instances either. In the following example, the parentheses are within a sequence:


```
@(posedge clk1) w ##1 (r  ##1 @(posedge clk2)  y    )     |=> m;
```

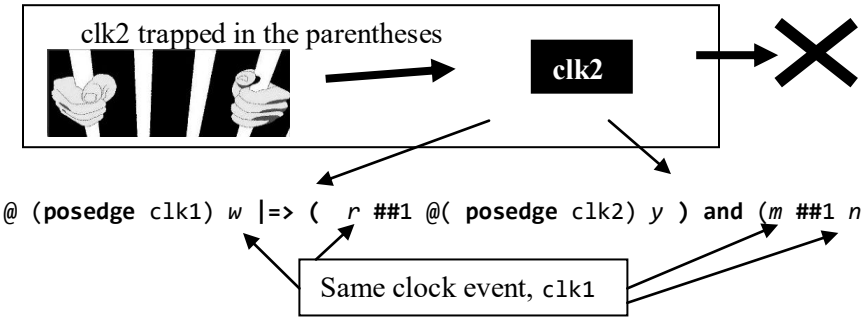
Same clocking event , `posedge clk1`

Is equivalent to:

```
@(posedge clk1) w ##1
                (@(posedge clk1) r ##1
                 @(posedge clk2)  y    |=>
                                        @(posedge clk1) m;
```

`w, r, m` are clocked at `posedge clk1` and `y` is clocked at `posedge clk2`. Clock `posedge clk1` flows across `##1`, over the parenthesized subsequence `(r ##1 @( posedge clk2) y)`, and across the non-overlapping implication `|=>` operator. Clock `posedge clk1` also flows into the parenthesized subsequence, but it does not flow through `@(posedge clk2)`. Clock `posedge clk2` does not flow out of its enclosing parentheses; thus it does not flow into `m`.

Consider the following example where the parentheses are within a property:




```
@ (posedge clk1) w |=> (  r ##1 @( posedge clk2) y ) and (m ##1 n)
```

Same clock event, `clk1`

`w, r, m, n` are clocked at `posedge clk1`, and `y` is clocked at `posedge clk2`. Clock `clk1` flows across the non-overlapping operator `|=>`, distributes to both operands of the property `and` operator, and flows into each of the parenthesized subexpressions. Within `(r ##1 @(posedge clk2) y)`, `clk1` flows across `##1` but does not flow through `@(posedge clk2)`. Clock `posedge clk2` does not flow out of its enclosing parentheses. Within `(m ##1 n)`, `posedge clk1` flows across the `##1` delay.