‘time slot blade’: A scheduling mechanism to avoid transaction level race condition

MediaTek Inc.

Soya Zhang, Chenghuan Li, Yunyang Song, Zhijun Fu

Author

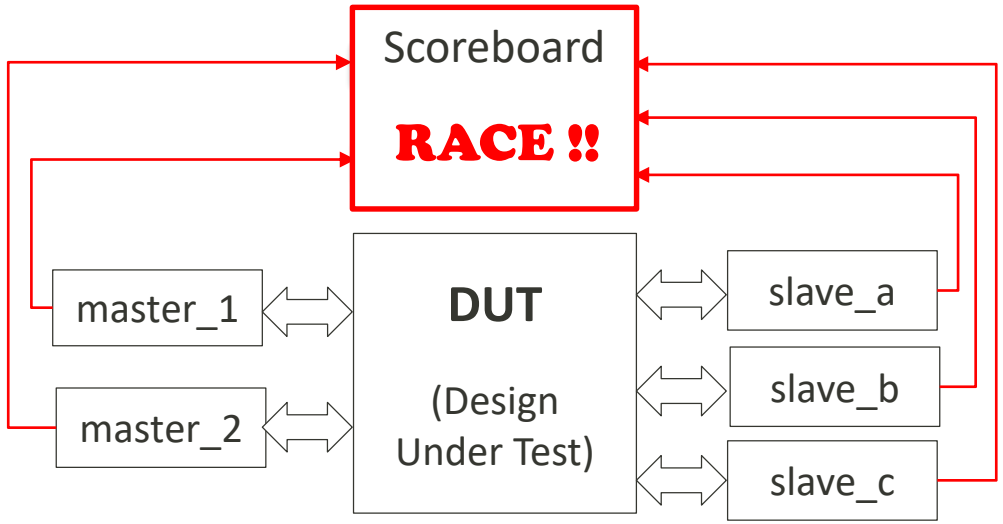
- Soya Zhang
  - MediaTek (Beijing) Inc
  - soya.zhang@mediatek.com
- Chenghuan Li
  - MediaTek (Beijing) Inc
  - chenghuan.li@mediatek.com
- Yunyang song
  - MediaTek (Beijing) Inc
  - yunyang.song@mediatek.com
- Zhijun Fu
  - MediaTek (Hefei) Inc.
  - zhijun.fu@mediatek.com

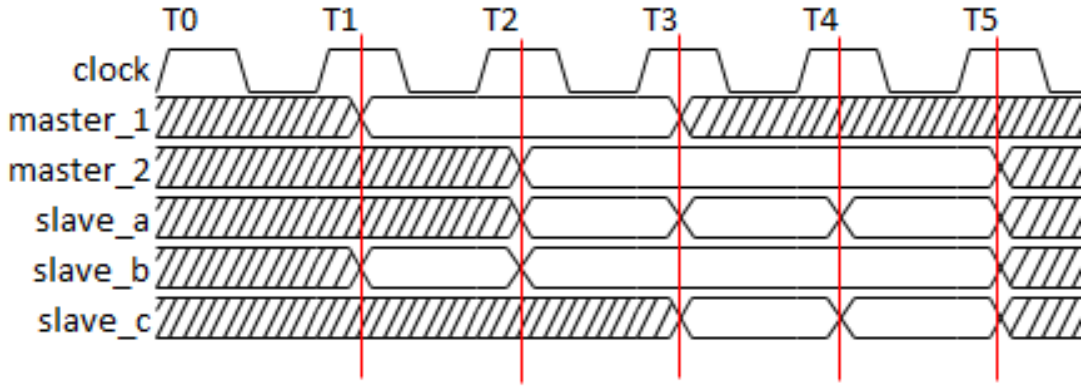
Outline

- Motivation
- Existing solutions
- Main Idea
- Solutions and Evidence
- Scalability and Advantage
- Limitation and solution
- Summary

Motivation

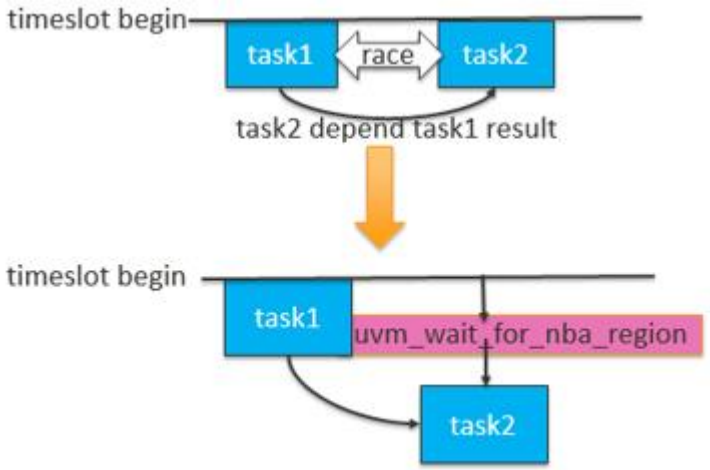
- In complex design verification, couples of UVM agents need be hooked up. These agents convert interface signal toggles into transaction, then deliver transaction to upper level components.
- When multiple transaction is collected on the same clock cycle, there might be race condition, which requires us to find out an robust way for transaction level scheduling



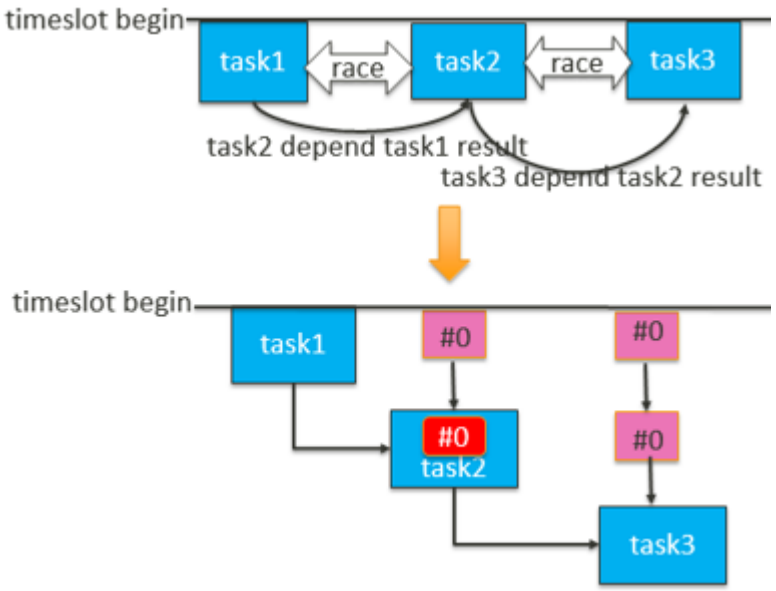


Existing solution

- UVM BCL provides uvm_wait_for_nba_region()
  - limitation: can only process two concurrent events



- Use '#0'
  - limitation: the number of delta cycles(#0) is arbitrary, which is difficult to maintain



Main Idea

- IEEE Standard 1800 defines SystemVerilog scheduling semantics, which groups the scheduled events into ‘time slot’, and each ‘time slot’ is divided into multiple regions
- Here we focus on following 4 regions
  - **Active** : simulation event evaluation
  - **Inactive** : explicit #0
  - **NBA** : non-blocking assignment
  - **Observed**: event’s triggered status is set and persists
- The main idea is to provide a mechanism to generate firmly isolated ‘time slots’
  - **‘Working Space’** = Active (NBA) region
  - **‘blade’** = ‘Inactive + Observed’ region

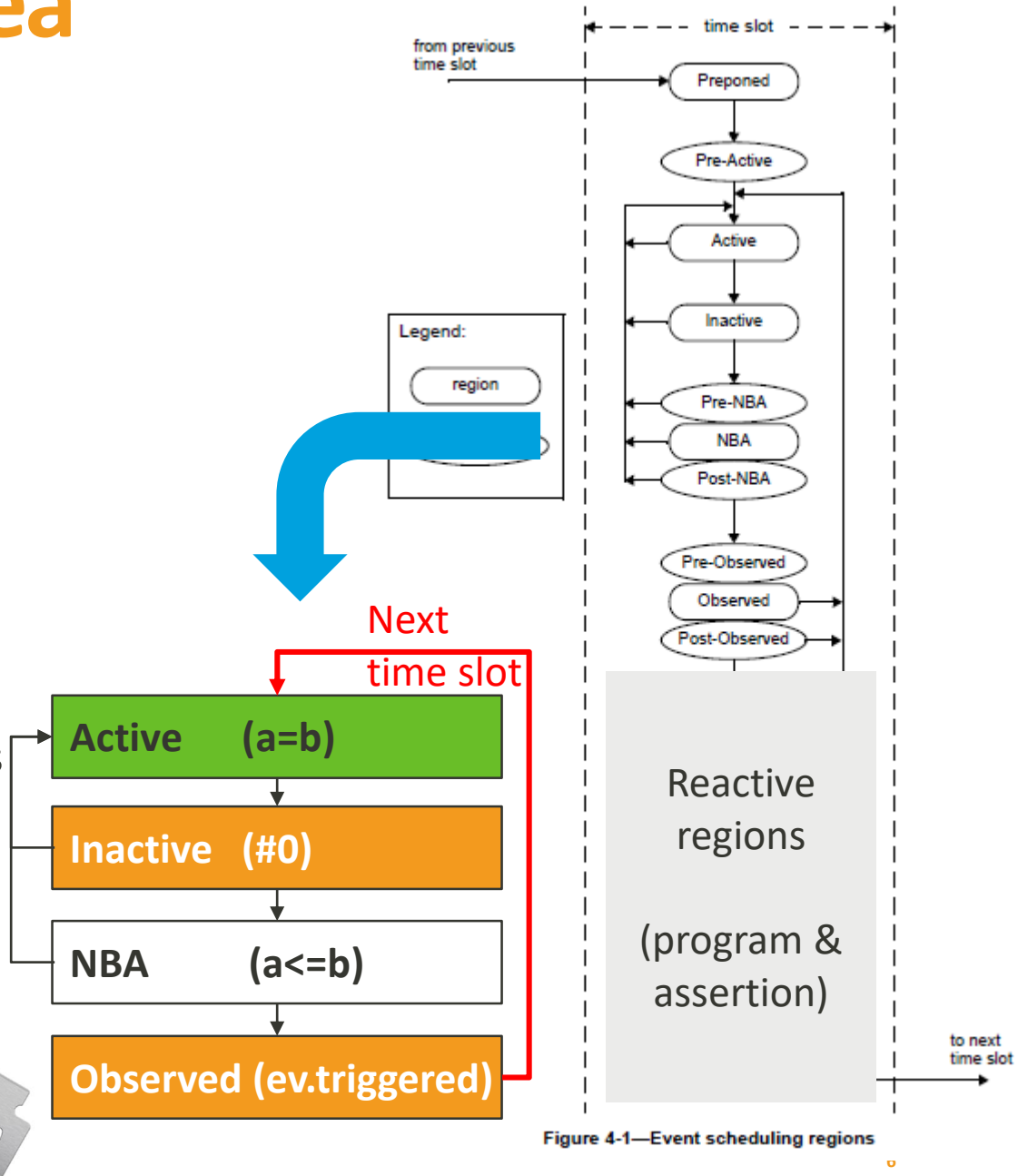
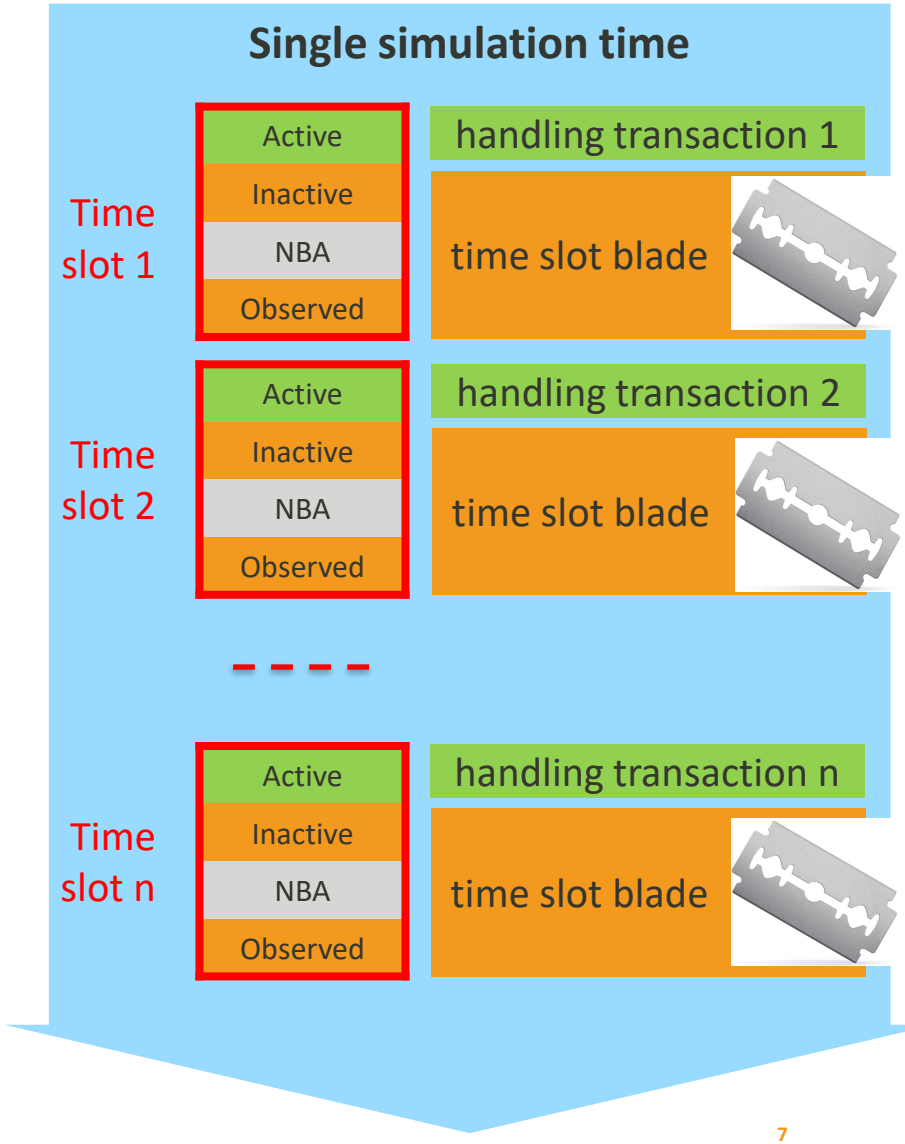


Figure 4-1—Event scheduling regions

Solution

- time slot ‘blade’

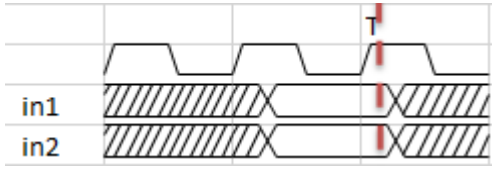
```
event   ev_start [int][`MAX_STAGE]
event   ev_done[int][`MAX_STAGE]
static   int  id_cnt ;

task  wait_time_slot_stage (int stage);
   id_cnt++;

   →ev_start [id_cnt][stage] ;   // trigger ‘start’ event to register task scheduling
   repeat (n * 100)  #0 ;   //  insert inactive #0 to make sure all ‘start’ events
                            //  are registered before being checked
   wait_all_prev_stage_done (stage); // wait all previous stages tasks are done by checking
                                     // registered ‘start’ & ‘done’ events ‘triggered’ status
   → ev_done [id_cnt][stage] ;  // trigger ‘done’ event to indicate registered task is finished
endtask

task wait_all_prev_stage_done (int stage )
   foreach(ev_start[idx]) begin
     for(int lv=0;lv<stage-1;lv++) begin
        if(ev_start[idx][lv].triggered)
          wait(ev_done[idx][lv].triggered);
     end
   end
endtask
```

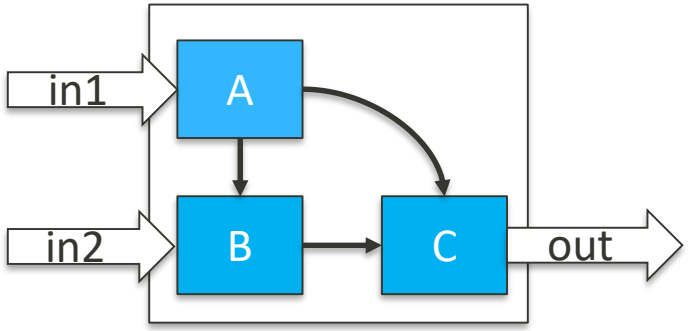


Usage example

- Scenario
  - Transaction from interface in1/in2 can change concurrently
  - model B depends on A; model C depends on A & B

  ➔Model A/B/C should be updated sequentially



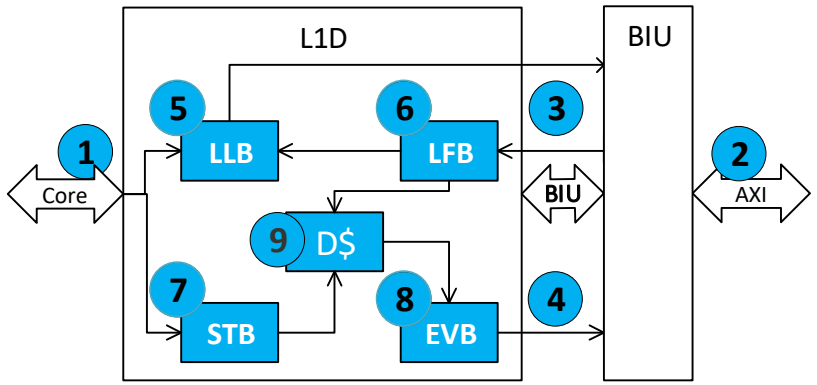


stage	task		
1	Update A		
2		Update B	
3			Update C

```
task  update_stage1();
  // received in1 transaction
   wait_time_slot_stage(1);
   //  update model A
endtask

task  update_stage2();
   // received in2 transaction
   wait_time_slot_stage(2);
   //  update model B
endtask

task  update_stage3()
   wait_time_slot_stage(3);
   //  update model C
endtask
```

Evidence

- The method shown above is deployed on a cache design verification environment
  - Transaction are collected from **9** different interfaces
  - Every ‘simulation time’ is divided into to **11** stages for transaction scheduling
  - The solution works fine on 2 different simulators w/o any porting effort



Stage	Checker	D$	LLB	STB	EVB
0					
1			V: CIU valid=1	V: split_req=1 (W/CW)	+1T
2	invd_hit()		U: ZAC hit LFB	V: ZAC hit cache	
3	evict_hit()				
4		D$ access		N: D$ access(WWW)	D$ access (WWR)
5	store_hit()				
6	Check D$ access		D$ access(WWW)	N : BIU WR N: D$ access(XWW) U: D$ access(XWR)	
7	load_hit()				
8	checkout				
10	post_checkout				

Simulator results

```
task  run();
  while(1) begin
      wait_time_slot_stage(10);
      #1us;
      run_cnt++;
      if(run_cnt>run_cnt_max) break;
   end
endtask
```

loop count	stage number	simulator		
		A	B	C
10k	no schedule	0.65s	0.01s	0.1s
	10	78.29s	1.9s	0.3s
	20	153.2s	3.44s	0.5s
100k	no schedule	0.646s	0.04s	0.1s
	10	9713.6s	17.33s	2.3s
	20	17882s	34.09s	4s

- The solution can work in different simulators. If event number is doubled, the simulation time will be doubled

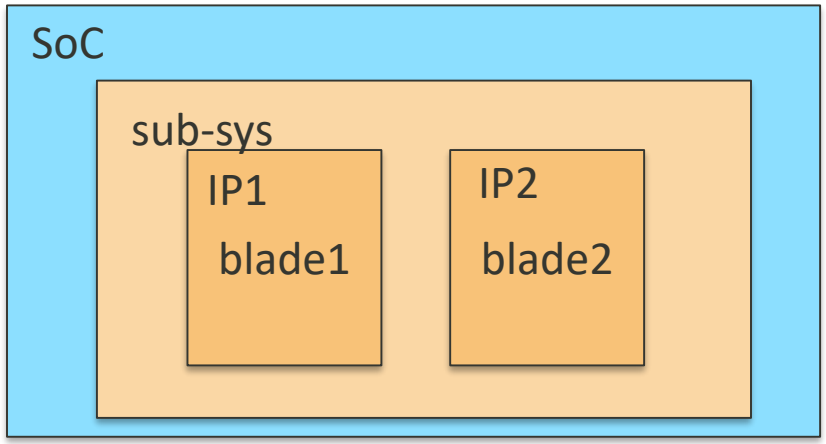
advantage

- **Scalable:** can have arbitrary number of stages

```
task  update_stage();
   // received transaction
   wait_time_slot_stage(m+1);
   //process_transaction();
endtask
```

stage	task			
1	A			
.....		.....		
m			m	
m+1				m+1

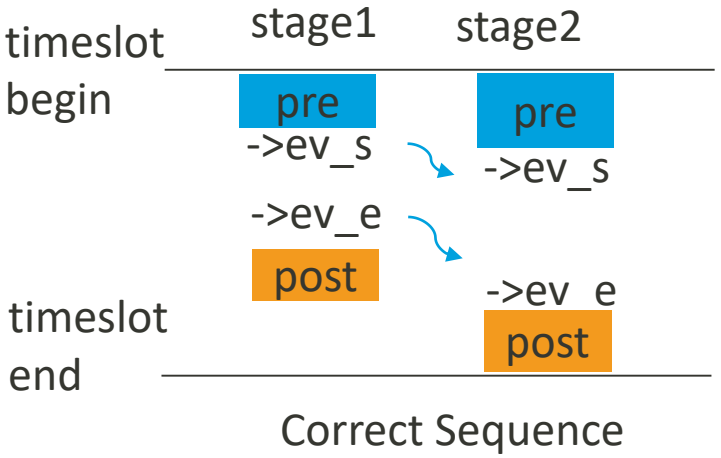
- **Portable:** ease for 'IP → Sub-system → SoC' integration
  - Stage scheduling can be easily aligned among different DV environments



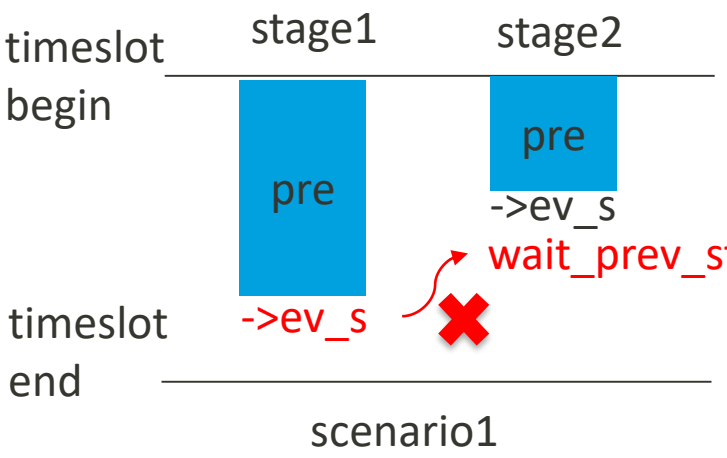
Limitation

- Functional limitation
  - not full handshake, less reliable but high efficiency
    - scenario1: previous stage pre-process spend long cpu time
    - scenario2: previous stage post-process spend long cpu time

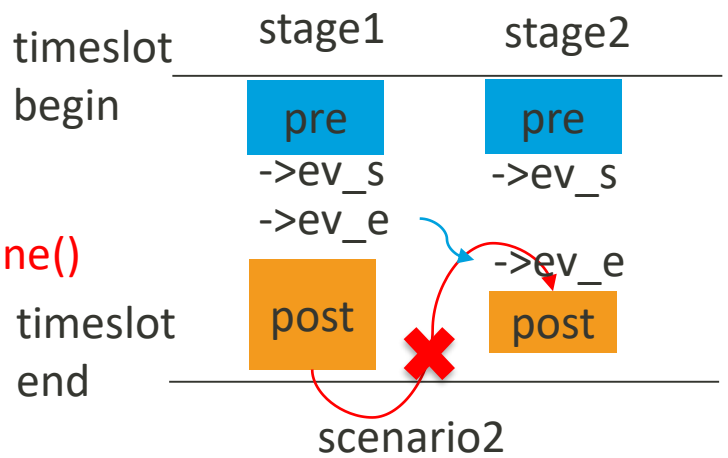
```
task  update_stage(int n);
   // pre_process();
   wait_time_slot_stage(n);
   // post_process();
endtask
```



Correct Sequence



scenario1



scenario2

- Usage restriction
  - need make sure ‘pre_process()’ and ‘post_process()’ do not have code that consume time (#0, etc.)

Reliability mechanism

- Divide start/end event trigger into separated tasks
  - User code must guarantee that start/end event is triggered in pair. If not it may cause deadlock

```
task  update_stage(int n);
   // pre_process();
   wait_time_slot_stage(n);
   // post_process();
endtask
```

```
task  update_stage(int n);
    trigger_stage_start_event(n);
    // pre_process();
    // post_process();
    trigger_stage_end_event(n);
endtask
```

Summary

- Race condition might happen in transaction level models when verify complex design with multiple interfaces.
- This paper introduce a way to fix the race condition. We call it ‘time slot blade’.
- The solution is proven on a cache design verification with 2 different simulators.