The Traditional `Req/Ack` Handshake, It's More Complicated Than You Think!

Ben Cohen 9/1/2024

1.1 INTRODUCTION

The request/acknowledge (**`req/ack`**) paradigm serves as the cornerstone example in assertion literature and courses, effectively illustrating the fundamental concepts of antecedent-consequent relationships and vacuity. However, this seemingly straightforward example often conceals a multitude of potential hidden or unexpressed requirements. This paper aims to uncover these "other" requirements and demonstrate how to articulate them using SystemVerilog Assertions (SVA). This paper pursues two primary objectives:

1. To highlight the critical importance of requirements, emphasizing often misunderstood implications, and to underscore the necessity of formulating clear, unambiguous requirements
2. To place significant emphasis on the impact of assertion writing styles, exploring how different approaches can affect clarity, maintainability, and effectiveness. By delving deeper into the `req/ack` example, we will reveal the complexities that lie beneath its surface, providing valuable insights for both novice and experienced practitioners in the field of hardware verification.

1.2 WHAT'S MISSING FROM THE TEXTBOOK HANDSHAKE REQUIREMENT?

The textbook handshake typically looks as follows:

```
    ap_handshake: assert property(@ (posedge clk) req |-> ##[1:5] ack );
```

This assertion checks the requirement that if **`req==1,`** then within 1 to 5 cycles, **`ack==1.`** However, this requirement addresses only a single aspect of the handshake and doesn't cover important details about signal signatures and the environment. Specifically:

1) Are **`req`** and **`ack`** single bits or vectors?
2) Are these signals single pulses or do they hold until the handshake is completed?
    a. When is the handshake considered completed? when the **`ack`** is asserted?
3) If **`req`** and **`ack`** are single bits, are these signals shared internally within the client by multiple sub-clients, implying that for each sub-client **`req`** there is a corresponding sub-client **`ack`**? Or does a single server **`ack`** satisfy multiple sub-client **`req`**s asserted over many cycles?
4) Are these **`req/ack`** signals associated with tags, implying tagged responses? And in which order, first-come/first-served?
5) If the **`req`** signal is a vector, what priority algorithms should be used? Fixed priority or first-come-first-served?
6) Is it legal for a **`req`** to drop its assertion before the **`ack`** (i.e., change its mind)? What would be the outcome?
7) Is it legal for a server to provide an **`ack`** without a prior **`req`**?
8) Is there a handshake completion constraint specifying when it should be completed? For example, should the handshake be completed within a dynamic range constrained by a control signal?
9) Are the client(s) and server(s) operating on synchronous clocks? If not, clock domain crossing (CDC) must be considered.

1.3 EXAMPLE 1: SINGLE PULSE REQ, SINGLE PULSE ACK

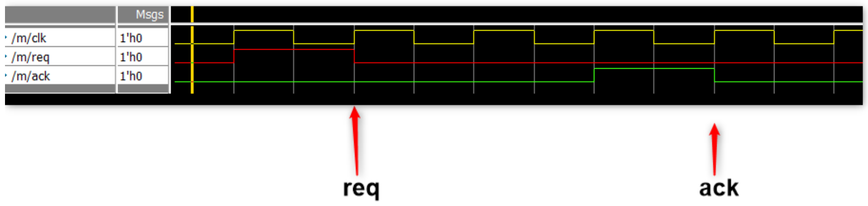
This example elaborates on the requirements for a simple handshake protocol, utilizing single-bit request (`req`) and acknowledge (`ack`) signals. It aims to provide clarity on the signal signatures and elucidate the precise timing relationship between these two signals. By delving into the specifics of this handshake mechanism, we achieve two crucial objectives:

- Greater clarity in signal behavior and interactions
- Significant reduction in requirement ambiguity

This refined approach allows for a more comprehensive understanding of the protocol, enabling more accurate implementation and verification. As we explore the nuances of this seemingly simple interaction, we'll uncover the importance of precise specification in hardware design and verification processes.

Requirements:

1. The client and server are synchronized to a master clock.
2. The request (**`req`**) and acknowledge (**`ack`**) signals are each single-bit signals.
3. The client asserts a single-pulse **`req`** signal when it needs access to a resource.
4. The client shall wait for a single-pulse **`ack`** signal. It expects the **`ack`** within 1 to 5 clock cycles, but not in the same cycle. Receipt of the **`ack`** completes the handshaking cycle.
5. The client shall not assert another **`req`** signal while it is waiting for the **`ack`** signal.
6. The server sets the **`ack`** signal for one pulse.
7. The server shall not provide an **`ack`** signal without receiving a **`req`** signal.



SVA Code (\reqack_example1.sv) https://www.edaplayground.com/x/hhz3

```
// 1. The client and server are synchronized to a master clock.
// 2. The request (req) and acknowledge (ack) signals are single-bit signals.
// 3. The client asserts a single-pulse req signal when it needs access to a resource.
 ap_req_pulse: assert property(@(posedge clk) req |=> !req);

// 6. The server sets the ack signal for one pulse.
 ap_ack_pulse: assert property(@(posedge clk) ack |=> !ack);

// 4. The client shall wait for a single-pulse ack signal. It expects the ack within 1 to 5 clock cycles,
//  but not in the same cycle. Receipt of the ack completes the handshaking cycle
 ap_req_ack: assert property(@ (posedge clk)
      $rose(req) |-> !ack ##[1:5] $rose(ack));

// 5. The client shall not assert another req signal while it is waiting for the ack signal.
 initial a_no_initial_ack: assert property (@(posedge clk) !ack[*1:$] intersect req[->1]);
 ap_no_req_till_ack: assert property(@(posedge clk) req |=> !req[*1:$] intersect ack[->1]);
 ap_no_ack_wo_req: assert property(@(posedge clk)  not (ack ##1 !req[*0:$] ##1 ack));

// 7. The server shall not provide an ack signal without receiving a req signal.
 bit past_req;
 always_ff @(posedge clk) begin
    if(req && !ack) past_req<=1;
    else if(ack) past_req <=0;
 end
 ap_req_ack2:    assert property (@(posedge clk) ack |->  past_req);
```

1.4 EXAMPLE 2: REQ HOLD UNTIL ACK, ABORT WHEN REQ DROPS OUT

This example introduces a more sophisticated interaction between the request (`req`) and acknowledge (`ack`) signals, building upon the foundation established in the previous scenario. The added complexity stems from a crucial requirement: the `req` signal must maintain its asserted state until the `ack` signal is asserted. An intriguing aspect of this protocol is the handling of premature `req` deassertion: If the `req` signal is deasserted before receiving the corresponding `ack`, the assertion is considered vacuous. This vacuity represents a form of protocol abortion, indicating an incomplete handshake.

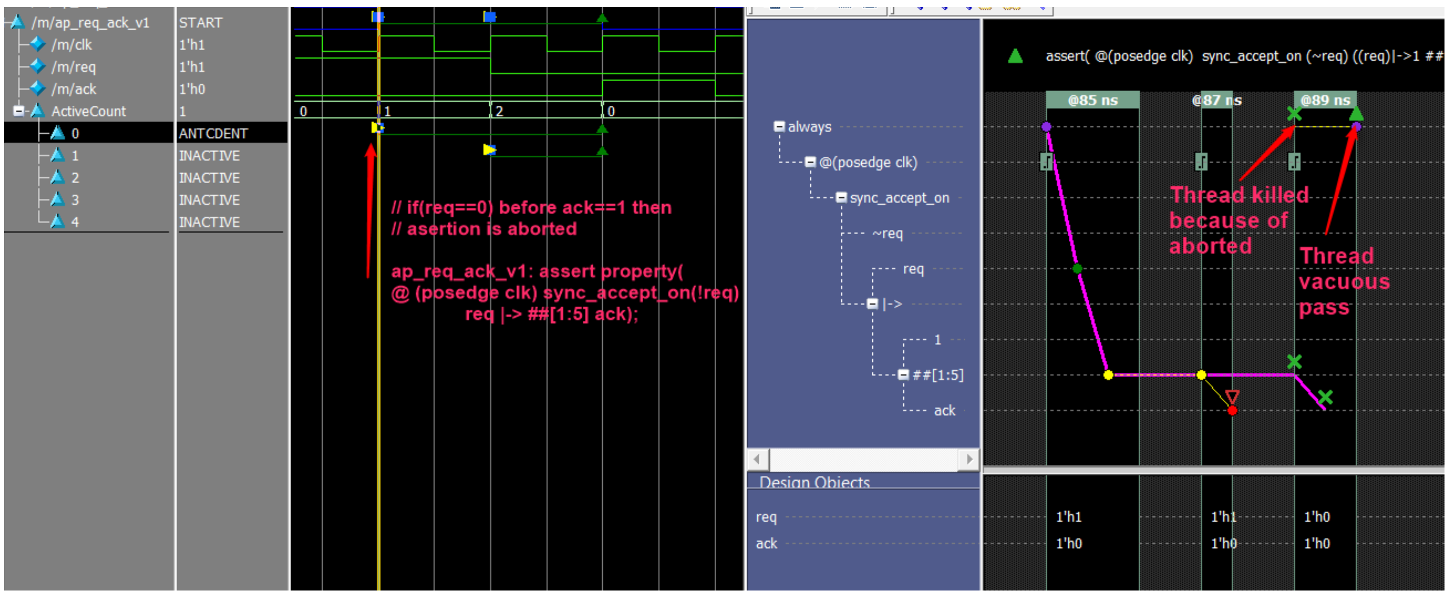
Requirements:

1. The client and server are synchronized to a master clock.
2. The request (req) and acknowledge (`ack`) signals are single-bit signals.
3. The client shall hold the req until it receives the one cycle ack signal.
4. The client shall expect the `ack` within 5 clock cycles.
5. The client may hold the `req` when it receives the `ack` if it wants another `ack`
6. If the client drops the `req` before getting the `ack`, then the pending request is cancelled and the `ack` should not be provided by the server.

Analysis and clarifications:
I initially added a requirement that if the server provides an `ack` without a `req`, then the client needs to provide a `req` to indicate an acceptance. But that could indicate a new request with the expectation of a new `ack`. This is a poor requirement, as it complicates the design and the verification processes. For this example, that item is dropped.

SVA Code: reqack_example2.sv https://www.edaplayground.com/x/inXp

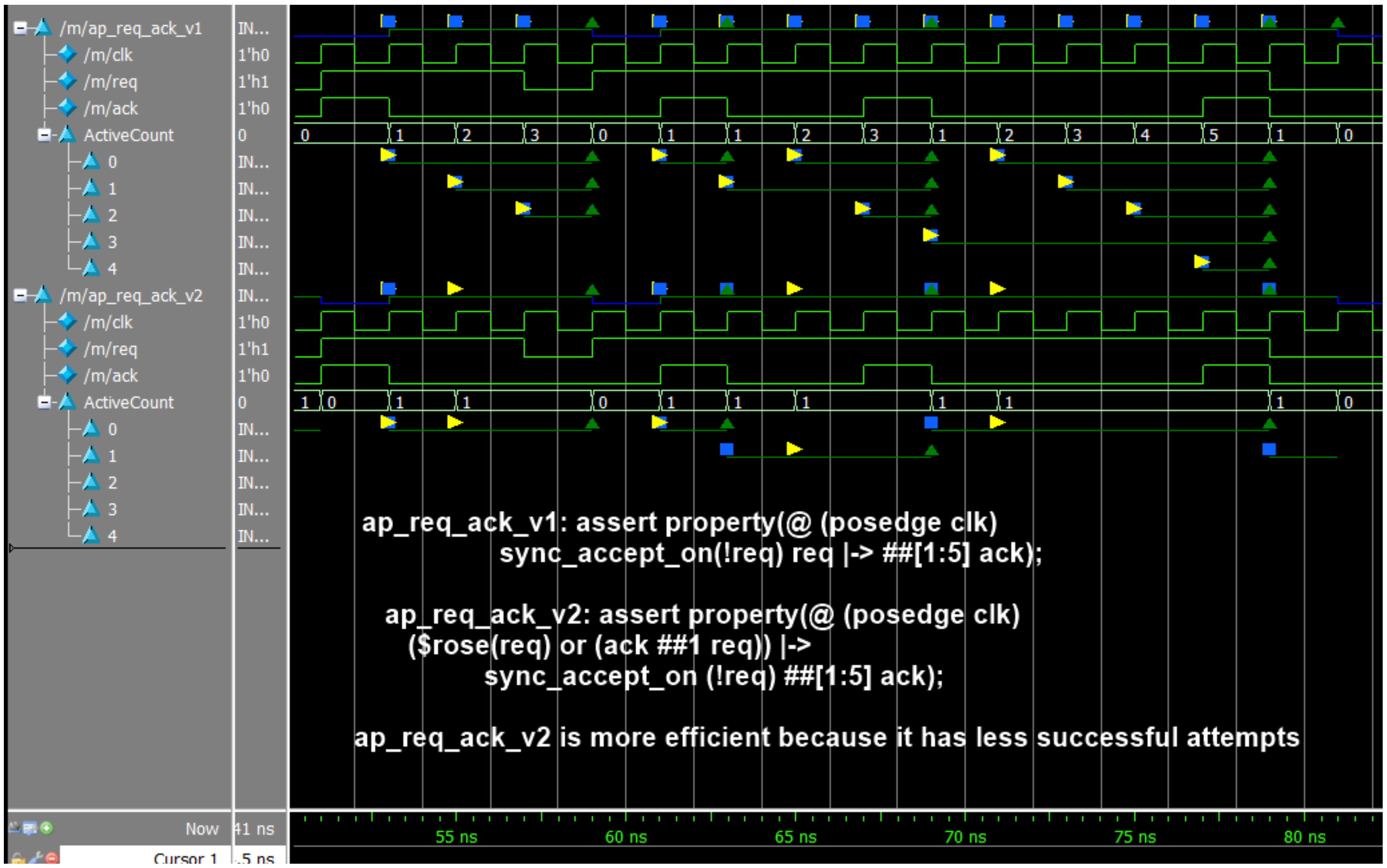
```
// Item 1, 2 are demonstrated in previous example
// Item 3,4, 5, 6
  // hold req until ack, abort on cancel of req (i.e., req==0 before ack)
  // There is a new thread at every attempt
  // All active threads satisfied with the ack
  // All active threads aborted if fell(ack) before ack
    ap_req_ack_v1: assert property(@ (posedge clk)
                     sync_accept_on(!req) req |-> ##[1:5] ack);
```



Thread viewer for ap_req_ack_v1 assertion

//More efficient approach with less successfully attempted threads:

```
    ap_req_ack_v2: assert property(@ (posedge clk)
       ($rose(req) or (ack ##1 req)) |->
                 sync_accept_on (!req) ##[1:5] ack);
```



```
ap_req_ack_v1: assert property(@ (posedge clk)
          sync_accept_on(!req) req |-> ##[1:5] ack);

ap_req_ack_v2: assert property(@ (posedge clk)
  ($rose(req) or (ack ##1 req)) |->
        sync_accept_on (!req) ##[1:5] ack);
```

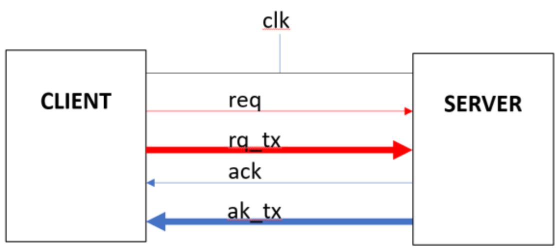
ap_req_ack_v2 is more efficient because it has less successful attempts

Simulation for ap_req_ack_v1 and ap_req_ack_v2 assertions

1.5 EXAMPLE 3: REQ WITH TRANSACTION ID, ACK FOR THE TRANSACTION

This example enhances the single-bit handshake protocol by incorporating additional identification elements. Specifically, it associates a client ID with the request and a server ID with the acknowledgement. This represents a more sophisticated model where the client manages multiple internal units, each potentially requiring access to the shared resource.

1. Requirements:
   1. Synchronization: The client and server must be synchronized to a common master clock to ensure proper timing and communication.
   2. Signal Types: Both the request (**req**) and acknowledge (**ack**) signals are single-bit signals, meaning they can only be in one of two states (0 or 1).
   3. Request Identification: Each request is uniquely identified by a transaction ID. This allows the **req** line to remain active for multiple clock cycles, representing different independent requests. A transaction ID may be associated with a sub-client user via internal client control logic, but it is not the identity of the user; it is more like a tracking number. Thus, the IDs represent unique transaction numbers that cycle after 16 transitions. The IDs have meaningful values only when their corresponding attachment control signals are active.
   4. Server Response: Upon processing a request, the server must provide an acknowledgment (**ack**) signal. This **ack** must be accompanied by the corresponding ID of the original request.
   5. ID Specification: The ID is defined as a four-bit vector, allowing for up to 16 unique identifiers.
   6. Response Time Limit: For each request ID, the client must receive the corresponding ack within a maximum of 5 clock cycles from the time of the initial request. This ensures timely communication between client and server.



Example 3 Architectural Diagram

CYCLE	1	2	3	4	5	6	7
req	0	1	1	0	1	0	0
q_tx	x	3	4	x	5	x	x
ack	0	0	0	1	0	1	1
k_tx	x	x	x	3	x	4	5

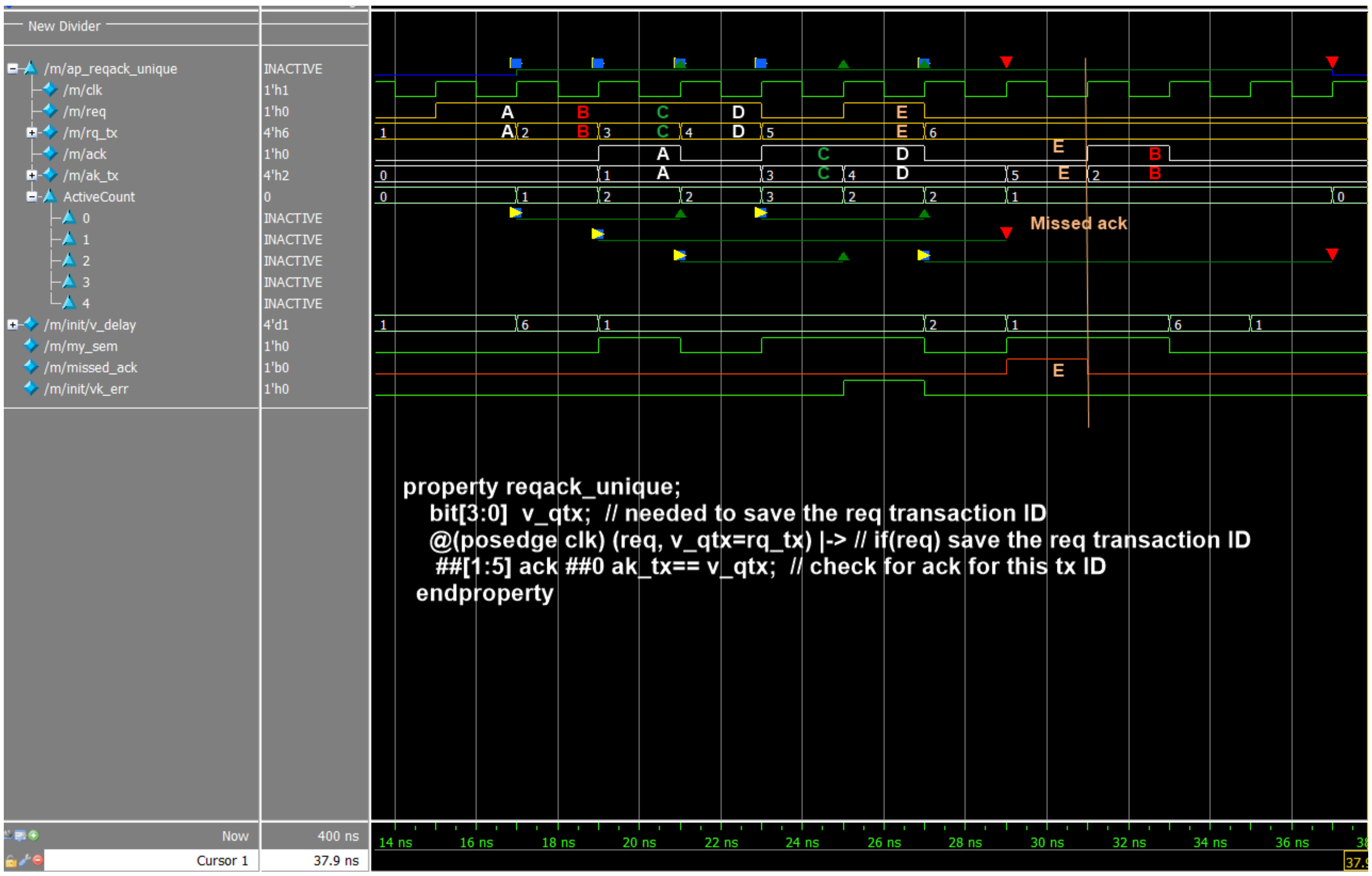
Example 3 Timing Diagram example

Code reqack_example3.sv https://www.edaplayground.com/x/tV5R

```
  property reqack_unique;
      bit[3:0]  v_qtx;  // needed to save the req transaction ID
      @(posedge clk) (req, v_qtx=rq_tx) |-> // if(req) save the req transaction ID
       ##[1:5] ack ##0 ak_tx== v_qtx;   // check for ack for this tx ID
  endproperty
  ap_reqack_unique: assert property(@(posedge clk) reqack_unique);
```

Notes:

1) The testbench models the client and server as automatic tasks forked at every cycle with weighted arguments.
   The client sends the `req` and the `rq_tx` . It also maintains the transaction counter (`tb_qtx`)
   `fork client(v_rq); join_none` // start the client
   `fork server(vk_err, vtx_err, v_delay); join_none` // start the server
2) The testbench includes random forced errors where the `ack` is suppressed by the server.
3) The testbench includes a semaphore to insure that only one of the forked tasks can drive the `ack` to the active state.



```
property reqack_unique;
    bit[3:0]  v_qtx;  // needed to save the req transaction ID
    @(posedge clk) (req, v_qtx=rq_tx) |-> // if(req) save the req transaction ID
     ##[1:5] ack ##0 ak_tx== v_qtx;  // check for ack for this tx ID
  endproperty
```

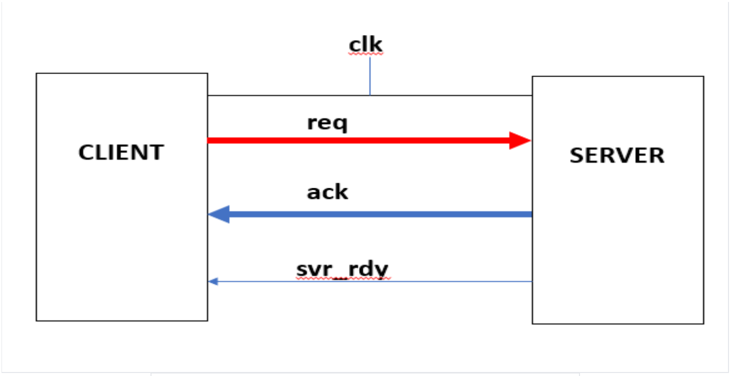
Example 3 Simultion Results

1.6 EXAMPLE 4: REQ WITH REQ / ACK ID FIXED PRIORITY

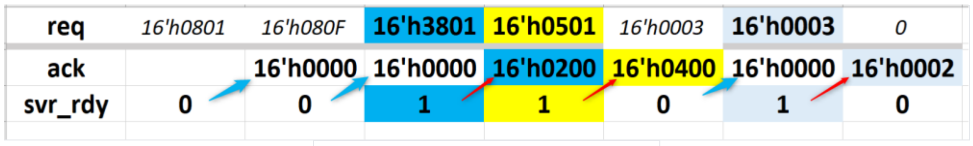
This example adds priority to the handshake protocol. Specifically, it associates a client ID with the 16-bit request lines and a server ID with the one-hot 16-bit acknowledgement.

Requirements:

1. Synchronization: The client and server must be synchronized to a common master clock to ensure proper timing and communication.
2. Signal Types: Both the request (req) and acknowledge (ack) signals are 16-bit. Any request line must present a valid value when the server ready signal (**`svr_rdy`**) is active. In this scenario, the acknowledgement (**`ack`**) shall provide a valid response in the cycle immediately following the active **`svr_rdy`** signal.
3. Request Priority: Bit 15 has the highest priority, and bit 0 has the lowest. Priority levels are fixed. Multiple request signals may be active in any cycle. However, the acknowledgement lines may be either all zeros or have only one active bit (logical 1).
4. Server Response: Upon processing a request, the server must provide a 1-bit acknowledgment (**`ack`**) signal based on the priority at the time of the request. This **`ack`** must be accompanied by the corresponding ID of the original request.
5. Response Time Limit: Since requests are prioritized, the highest valid request shall have its acknowledge signal active in the next cycle.



Example 4 Architectural Diagram



req	16'h0801	16'h080F	16'h3801	16'h0501	16'h0003	16'h0003	0
ack		16'h0000	16'h0000	16'h0200	16'h0400	16'h0000	16'h0002
svr_rdy	0	0	1	1	0	1	0

Example 4 Timing diagram

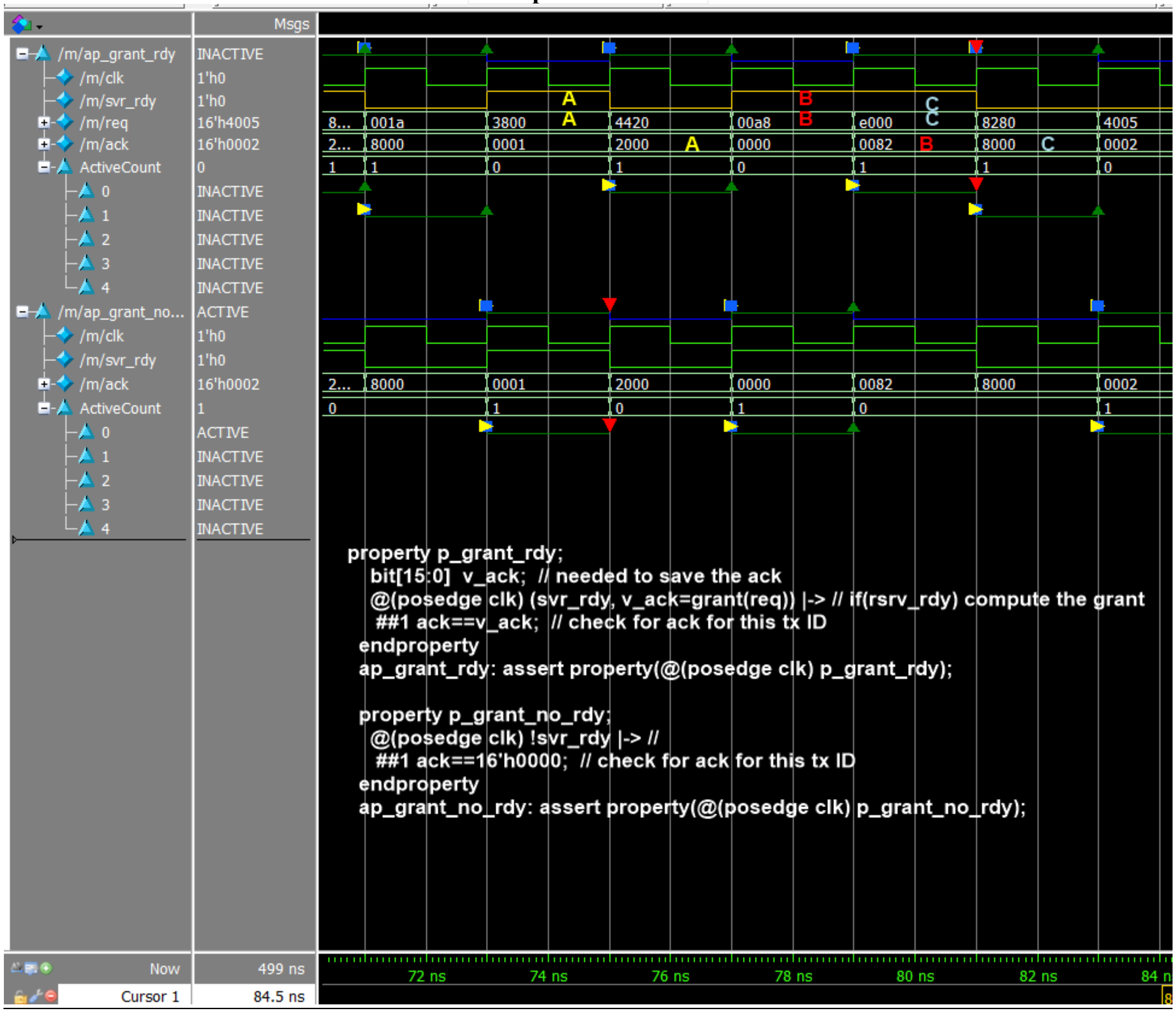
Code: reqack._example4.sv https://www.edaplayground.com/x/jPPd

```
function automatic bit[15:0] grant(bit[15:0] rqst); // Computes the priority
    bit[15:0] temp;
    temp=0;
     for (int i = 15; i>= 0; i--) begin : forlp1
      if(rqst[i]) begin : if1a
         temp[i]=1;
         return temp;
         break;
      end  : if1a
     end  : forlp1
     return temp;
  endfunction
```

```
property p_grant_rdy;
  bit[15:0]  v_ack;  // needed to save the ack
  @(posedge clk) (svr_rdy, v_ack=grant(req)) |-> // if(rsrv_rdy) compute the grant function
   ##1 ack==v_ack;  // check for ack for this req
endproperty
ap_grant_rdy: assert property(@(posedge clk) p_grant_rdy);

property p_grant_no_rdy;
  @(posedge clk) !svr_rdy |-> //
   ##1 ack==16'h0000;  // check for ack for this req
endproperty
ap_grant_no_rdy: assert property(@(posedge clk) p_grant_no_rdy);
```

Example 4 Simulation



```
property p_grant_rdy;
  bit[15:0]  v_ack;  // needed to save the ack
  @(posedge clk) (svr_rdy, v_ack=grant(req)) |-> // if(rsrv_rdy) compute the grant
   ##1 ack==v_ack;  // check for ack for this tx ID
 endproperty
 ap_grant_rdy: assert property(@(posedge clk) p_grant_rdy);

 property p_grant_no_rdy;
  @(posedge clk) !svr_rdy |-> //
   ##1 ack==16'h0000;  // check for ack for this tx ID
 endproperty
 ap_grant_no_rdy: assert property(@(posedge clk) p_grant_no_rdy);
```

1.7 EXAMPLE 5: ONE PULSE REQ, DELAYED ACK

Requirements:

1) Upon a single pulse **req**, an **ack** pulse shall occur within 3 to 5 cycles afterwards.
2) Following the **req**, that signal shall remain in the inactive level until the **ack** is active.
3) The **req** and **ack** shall never occur in the same cycle.

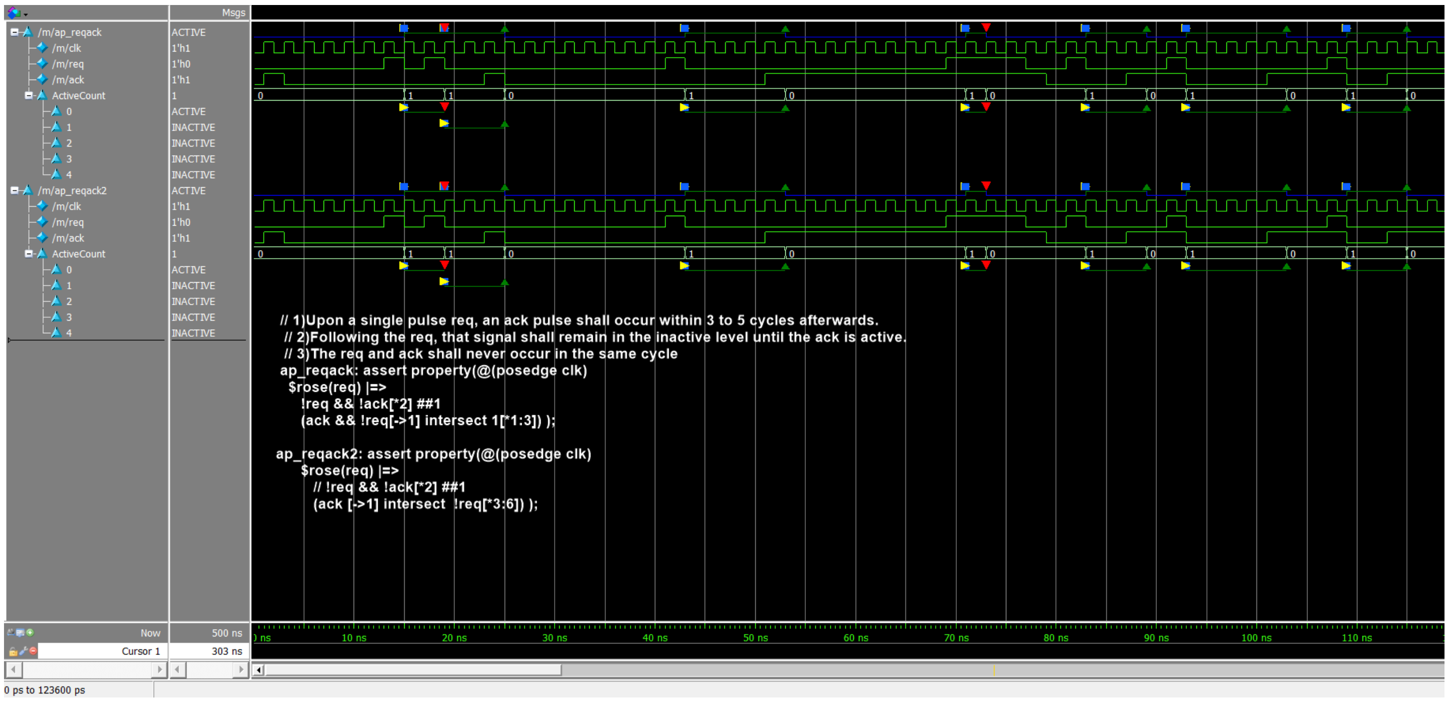
CYCLE	1	2	3	4	5	6	7
req	0	1	0	0	0	0	1
ack	0	0	0	0	0	1	0
		$rose(req) |=>	!req && !ack[*2] ##1		(ack && !req[->1])		
					<------------- 1[*1:3] ------------->		

Example 5 Timing Diagram

Code reqack_example5.sv https://www.edaplayground.com/x/CqKP

```
ap_reqack: assert property(@(posedge clk)
   $rose(req) |=>
      !req && !ack[*2] ##1  // walk-through style by sequence sections
      (ack && !req[->1] intersect 1[*1:3]) );

ap_reqack2: assert property(@(posedge clk)    // simpler coding
      $rose(req) |=>
         // !req && !ack[*2] ##1
         (ack [->1] intersect  !req[*3:6]) );
         // !req is repeated 3 to 6 times starting from cycle after the req
```



```
// 1)Upon a single pulse req, an ack pulse shall occur within 3 to 5 cycles afterwards.
 // 2)Following the req, that signal shall remain in the inactive level until the ack is active.
 // 3)The req and ack shall never occur in the same cycle
ap_reqack: assert property(@(posedge clk)
  $rose(req) |=>
    !req && !ack[*2] ##1
    (ack && !req[->1] intersect 1[*1:3]) );

ap_reqack2: assert property(@(posedge clk)
    $rose(req) |=>
      // !req && !ack[*2] ##1
      (ack [->1] intersect  !req[*3:6]) );
```

Example 5 Simulation

4.0 Conclusions and Recommendations

I initially set out to write a paper to showcase various styles of writing assertions for the simple request/acknowledge (req/ack) handshake protocol. This decision stemmed from observing the challenges users frequently encounter on forums. Despite my extensive experience as a design and verification engineer and my authorship of books on these subjects, including assertions, I was enlightened by several discoveries during this endeavor.

1. **Identifying Gaps in Requirements**: The process of writing assertions revealed unexpected gaps in the requirements. This insight underscores the critical role that assertions play in ensuring comprehensive and robust protocol design.
2. **Uncovering Deficiencies**: While crafting assertions to align with the specified requirements, I discovered a significant deficiency in the requirements themselves. This revelation emphasizes the necessity of thorough analysis and testing in protocol design, as even well-defined protocols can harbor overlooked issues.
3. **Assistance with Coding Syntax**: When it came to recalling forgotten language syntax or style—such as constraints, generation, and breaks—Perplexity.ai provided invaluable assistance. Its support in these areas streamlined the coding process and enhanced the overall quality of my work.
4. **AI's Role in Refining Language and Style**: In seeking help from Perplexity.ai to refine my English and writing style, I uncovered additional requirements that were either missing or needed further clarification. The AI's ability to beautifully enhance my English and composition was both impressive and somewhat intimidating. Moreover, the AI's capability to search for more information on the protocol and provide additional insights and necessary requirements was invaluable. This feature not only enriched my understanding of the protocol but also ensured that all aspects were thoroughly addressed, highlighting the AI's potential to augment human expertise with comprehensive research and analysis.
5. **Importance of Committee Review**: A crucial aspect that emerged during this process was the necessity of a committee review for both the requirements and the code. Such a review brings in diverse viewpoints, enhancing the overall quality and robustness of the protocol design. For this paper, I engaged in valuable interactions with LinkedIn users and collaborated closely with a colleague, which provided additional perspectives and insights. This collaborative approach aligns with established practices in various fields, where committee reviews are standard for ensuring thoroughness and identifying potential issues.

Through this journey, I gained a deeper appreciation for the intricate interplay between requirements, assertions, and the tools that support their development. This experience not only highlighted the importance of meticulous attention to detail but also demonstrated the value of leveraging AI and collaborative efforts to enhance both technical and linguistic aspects of my work.

Code examples: https://SystemVerilog.us/vf/reqack.tar

Ben Cohen
Ben@systemverilog.us

Link to the list of papers and books that I wrote, many are now donated.
http://systemverilog.us/vf/Cohen_Links_to_papers_books.pdf