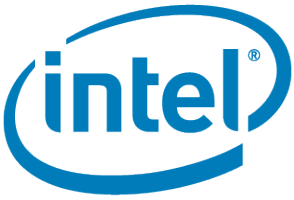


SIP Converged IOSF Trunk Clock Gating Methodology

September 2015

Intel Confidential

This document contains information on products in the design phase of development.

Contents

Overview

Trunk clock gating, as opposed to the internal local clock gating, refers to the action of shutting the clock to an IP from the source. The IOSF specification defines the basic interactions between the fabric and the agent, namely, `clkreq` and `clkack`, that allows trunk clock gating to the agent. Naturally, the provided provision is rather focused on how the fabric should coordinate the trunk clock gating to the agent, leaving trunk clock gating on the fabric itself to implementation-specific discussions.

Often times, unlike the IOSF specification's point of view, the fabric is not a single-unit entity when implemented. Rather, it tends to consist of multiple independent units (or, equivalently, IPs) to form the fabric as a fabric cluster. For instance, multiple instances of PSF can be used to form a hierarchical primary fabric cluster. In the same manner, it is common to deploy multiple sideband routers to form a sideband fabric cluster. In these cases, the IOSF term "fabric" collectively refers to the cluster that contains multiple instances of fabric IPs. To maximize the power savings, the trunk clock gating on the "fabric" should also work on the IP-level, as opposed to turning on and off the clock to the entire fabric cluster, the discussion of which is hardly touched upon by the IOSF specification.

The goal of this document is to specify the SIP converged IOSF trunk clock-gating methodology that enables the following:

- Agent trunk clock gating in the IOSF-compliant manner
- Unit-router-level trunk clock gating of fabric IPs
- Converged standardization applicable to both primary and sideband fabrics

The agent trunk clock gating is to be coordinated between the agent and the fabric per the IOSF standard. In the trunk clock-gating context, "fabric" refers to the entire cluster minus the agent. Particularly, in the subsequent discussions, it is essential to note that the "fabric" consist of two fundamental elements: routers and the Power Management Unit (PMU). An agent communicates with these two fabric components in an IOSF-compliant manner via the ISM (with the router) and `clkreq`/`clkack` signaling (with the PMU).

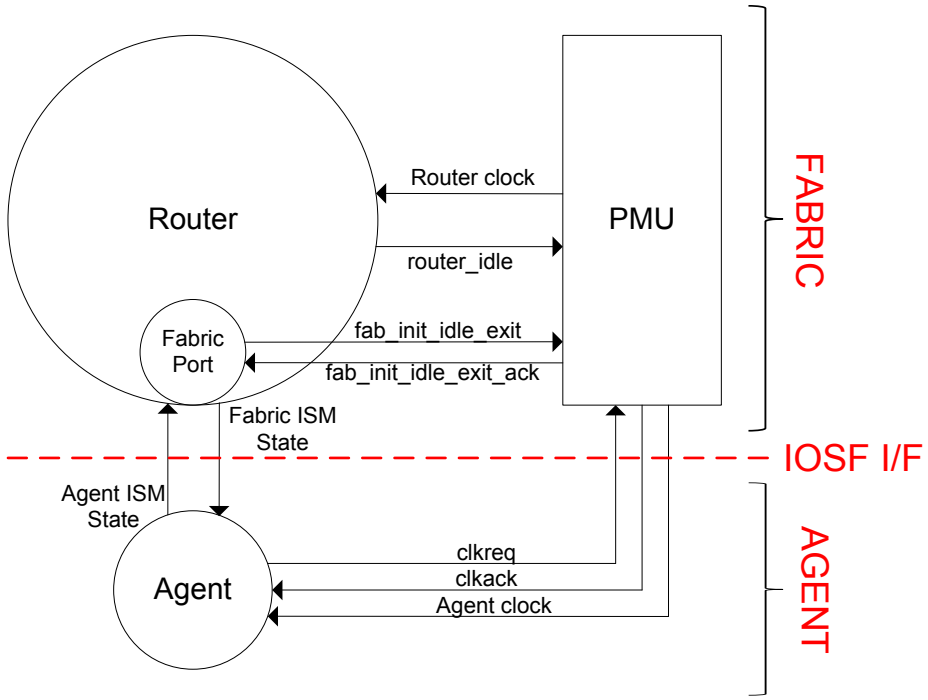
Within the fabric, the unit-router trunk clock gating is to be coordinated by the PMU. To enable this, there is an interface between the router and PMU that the proposed methodology standardizes. The router/PMU interface is not a part of the IOSF standard.

The following sections discuss how these components (agent, routers, and PMU) fit together to enable trunk clock gating. The signaling between router and PMU is also explained.

Methodology Details

Figure 1 illustrates how the agent, a router, and PMU form a coherent cluster that supports trunk clock gating. In the picture, most notably, the agent performs `clkreq/clkack` signaling directly with PMU, and also receives its trunk clock directly from PMU, which enables the router to stay mostly agnostic of all these trunk clock gating activities.

Figure 1. Router/Agent Trunk Clock Gating



The trunk clock gating of the router can be entirely subject to the trunk clock gating states of the agents that are attached to it. That is, when all the `clkreq` signals from the agents are de-asserted, PMU can make the decision to turn off the router trunk clock. Likewise, when the router trunk clock is turned off, as any of the agent's `clkreq` is asserted, the PMU also needs to turn on the PSF trunk clock as it returns a `clkack`.

There are two reasons why there should be communications between the router and the PMU. First, consider a case where the last agent sends the final transaction to the router and de-asserts its `clkreq` to the PMU. There is then a danger that the PMU may de-assert a `clkack` and turn off the router trunk clock all together, but prematurely, while the final transaction is still in the router. To prevent this, the router sends the signal, `router_idle`, to the PMU to indicate when it is safe to turn off the clock to the router. The `router_idle` signal is normally a flopped output that goes high when all the ISMs in the router are at IDLE and no pending transactions remain in the router's internal FIFOs.

The second scenario that necessitates the router/PMU communication involves the so-called "fabric-initiated idle exit." To understand this, let us assume that the router still has its trunk clock active, whereas the agent is trunk-clock-gated. Now, further assume that a transaction is injected into the router that targets the trunk-clock-gated agent. Per the IOSF standard, here is the sequence of activities that need to occur.

1. The PMU turn on the clock to the agent.

2. The router moves its fabric ISM to ACTIVREQ .

3. The agent moves its agent ISM to ACTIVEREQ, and, at the same time, asserts clkreq to the PMU.

4. When ready, the agent moves to the ACTIVE state.

5. The router then moves to the ACTIVE state and sends the transaction.

Obviously, to correctly perform the steps (1) and (2) above, in the prescribed order, it is necessary to have communications between the PMU and the router. So the following two additional steps are added: before (1), the router asserts `fab_init_idle_exit` to the PMU, and, after (1), the PMU asserts `fab_init_idle_exit_ack` back to the router.

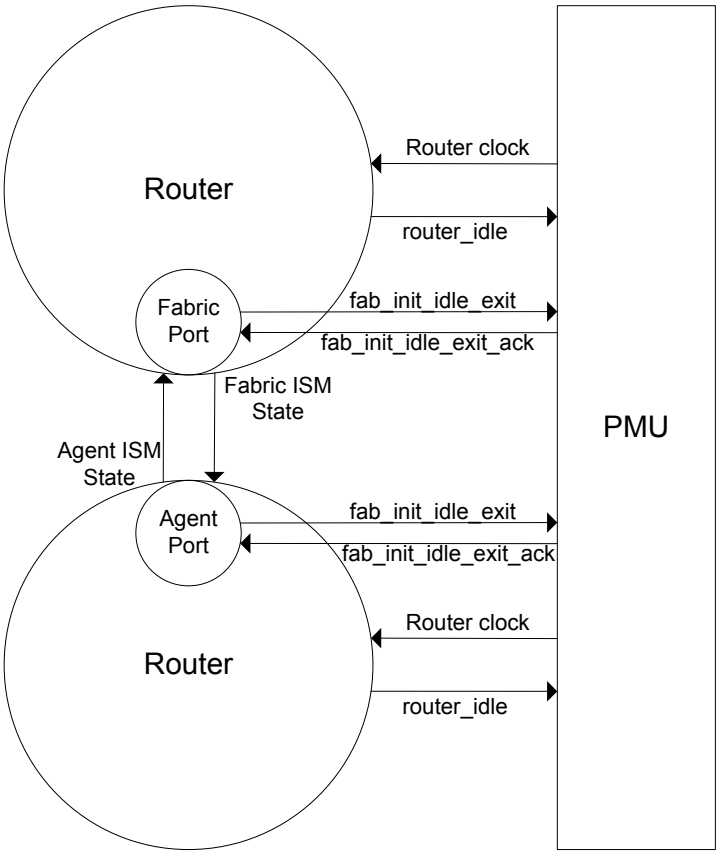
To recap, the `fab_init_idle_exit` and `fab_init_idle_exit_ack` are handshake signals that are used by the router to wake up the other side that it wishes to talk to. This is in contrast to clkreq/clkack, which are used by the agent when it needs to wake up both itself and the router that it wishes talk to. The following rules apply for the `fab_init_idle_exit/fab_init_idle_exit_ack` handshake.

- `fab_init_idle_exit` gets asserted when the corresponding ISM needs to exit the IDLE state.
- `fab_init_idle_exit` assertion is subject `fab_init_idle_exit_ack` being low.
- `fab_init_idle_exit` stays asserted until the ISM moves to IDLE, and gets de-asserted as the ISM enters the IDLE state.
- `fab_init_idle_exit_ack` assertion is subject to `fab_init_idle_exit` being high.
- `fab_init_idle_exit_ack` de-assertion is subject to `fab_init_idle_exit` being low.

Naturally, `fab_init_idle_exit` and `fab_init_idle_exit_ack` are per-port signals.

Figure 2 illustrates a router-to-router connection. Recall that this is entirely a within-fabric connection, where the IOSF `clkreq/clkack` protocol is not applicable. Rather, this connection relies solely on the `fab_init_idle_exit/fab_init_idle_exit_ack` signaling previously explained. For a router-to-router connection, one port must be an IOSF agent port whereas the other must be an IOSF fabric port. However, that distinction is purely mechanical and has no bearing on which one is semantically an agent port or which one is a fabric port. Rather, it is more appropriate to say that they are both fabric ports in behavior. Therefore, the agent port in a router-to-router connection does not implement the IOSF `clkreq/clkack` signaling, but implements `fab_init_idle_exit/fab_init_idle_exit_ack` signaling, just like the fabric port it talks to.

Figure 2. Router-to-Router Trunk Clock Gating



Trunk Clock-Gating Clustering Examples: Synchronous Cluster

The philosophy of the trunk clock-gating framework explained is two-fold. First, the agent sticks strictly to the IOSF standard, only using ISM and `clkreq/clkack` in a manner that is fully compliant with the specification. Second, all the information is readily available to the PMU so that it can make legitimate and intelligent trunk clock-gating decisions. With regard to this second point, this section provides a multi-PSF cluster trunk clock-gating example, to help develop insight.

The example, shown in Figure 3, consists of a two-router fabric integrating with three agents. Note that, even though the whole cluster (routers + agents) is using the same clock, clocks are separated into multiple trunks for each router and for each agent.

The following shows what the PMU should do with this cluster in brief:

1. The PMU can separately trunk-clock-gate agents, by simply looking at their `clkreq` signals.

2. The PMU can trunk-clock-gate Router 1 when `clkreqs` from Agent 1 and Agent 2 are de-asserted and `router_idle` from Router 1 is asserted.

3. The PMU can trunk-clock-gate Router 0 when Router 1 is trunk-clock-gated, Agent 0 `clkreq` is de-asserted, and Router 0 `router_idle` is asserted.

Suppose Router 1, Agent 1, and Agent 2 are trunk-clock-gated, and Agent 1 needs to send a transaction to Agent 2. The following should then occur:

1. Agent 1 asserts `clkreq`.

   The PMU then turns on the trunk clocks to both Agent 1 and Router 1.

2. Agent 1, upon seeing `clkack` asserted, injects the transaction into Router 1.

3. Router 1 figures that the transaction is intended to Agent 2, and asserts `fab_init_idle_exit` to the PMU on that port.

4. The PMU, upon seeing `fab_init_idle_exit` asserted, resumes the trunk clock to Agent 2, and returns `fab_init_idle_exit_ack` to Router 1.

5. Router 1 sends the transaction to Agent 2.

Suppose everything is trunk-clock gated. If, for example, agent 0 needs to send a transaction to Agent 1, then the following should occur:

1. Agent 0 will assert `clkreq`. The PMU then turns on the trunk clocks to both Agent 0 and Router 0.

2. Agent 0, upon seeing `clkack` asserted, injects the transaction into Router 0.

3. Router 0 figures that the transaction is intended for Router 1, and asserts `fab_init_idle_exit` to the PMU on that port.

4. The PMU, upon seeing `fab_init_idle_exit` asserted, resumes the trunk clock to Router 1, and returns `fab_init_idle_exit_ack` to Router 0.

5. Router 0, upon seeing `fab_init_idle_exit_ack` asserted, injects the transaction into Router 1.

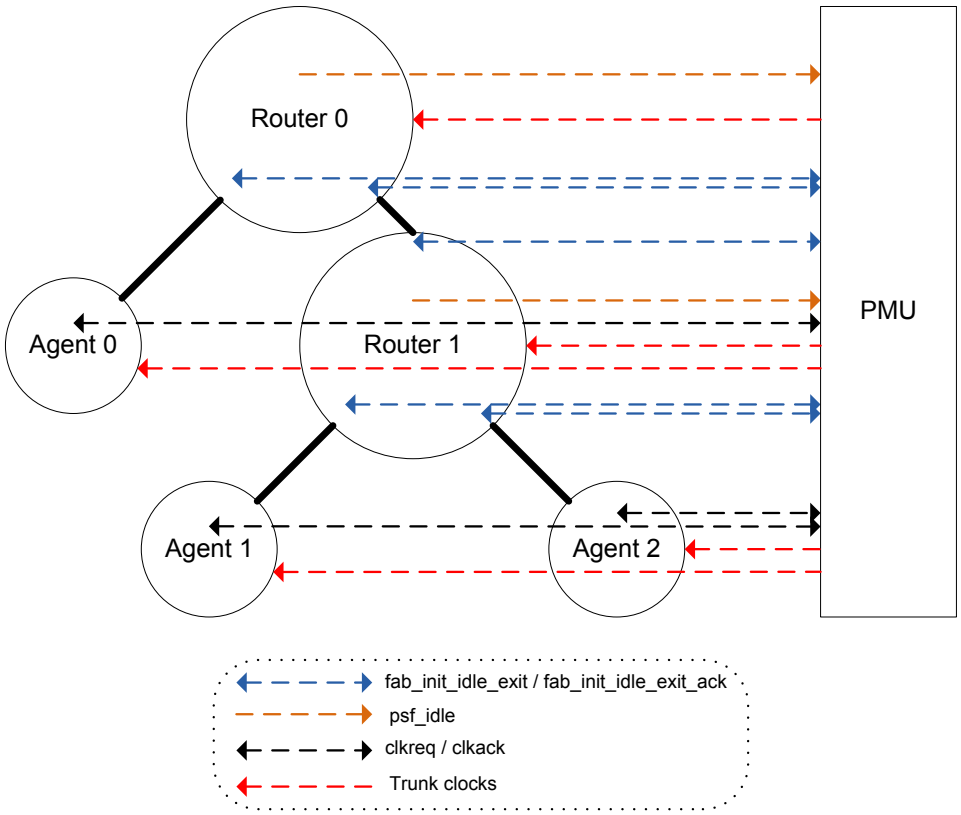
6. Router 1 figures that the transaction is intended to Agent 1, and asserts `fab_init_idle_exit` to the PMU on that port.

7. The PMU, upon seeing `fab_init_idle_exit` asserted, resumes the trunk clock to Agent 1, and returns `fab_init_idle_exit_ack` to Router 1.

8. Router 1 sends the transaction to Agent 1.

Note: The essential difference between `clkack` and `fab_init_idle_exit`: `clkack` is asking for the clock for the agent itself and the router that it is talking to, whereas `fab_init_idle_exit` is asking for the clock only of the router that it is talking to (because it already has its own clock running).

Figure 3. Synchronous Trunk Clock Gating Cluster Example



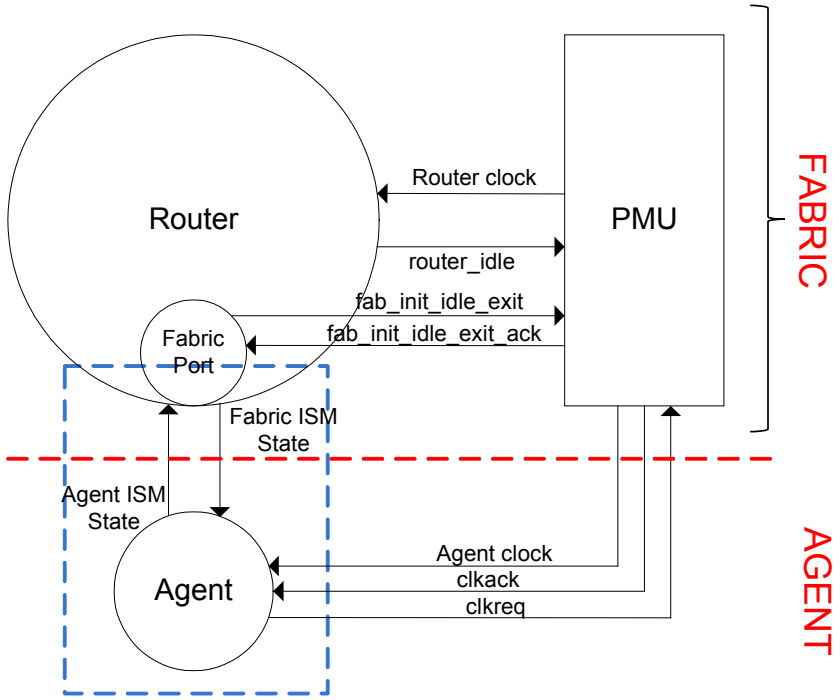
Dealing with Asynchronous Clock Domains

An agent is said to be on an asynchronous clock domain if its IOSF port connected to the router is on a different clock domain than the router's. An asynchronous agent leads to a multiple-clock domain router, where the router has a port (called an "asynchronous port") that belongs to the agent's IOSF port clock domain. Figure 4 illustrates such a configuration.

In the figure, note that the router's asynchronous port shares the same clock trunk as the asynchronous agent. How this trunk is turned on follows two possible scenarios:

- When the router initiates the `fabric-initiated idle exit`, it will assert `fab_init_idle_exit` to the PMU. The PMU then will resume the asynchronous trunk clock. That is, the PMU may have the asynchronous port clock turned off until the Router needs to access the port.
- When the agent asserts `clkack`, the PMU turns on the asynchronous clock. If the Router has been trunk-clock-gated, the PMU also resumes the router clock at the same time.

Figure 4. Router/Asynchronous Agent Trunk Clock Gating



Trunk Clock Gating Clustering Examples: Asynchronous Cluster

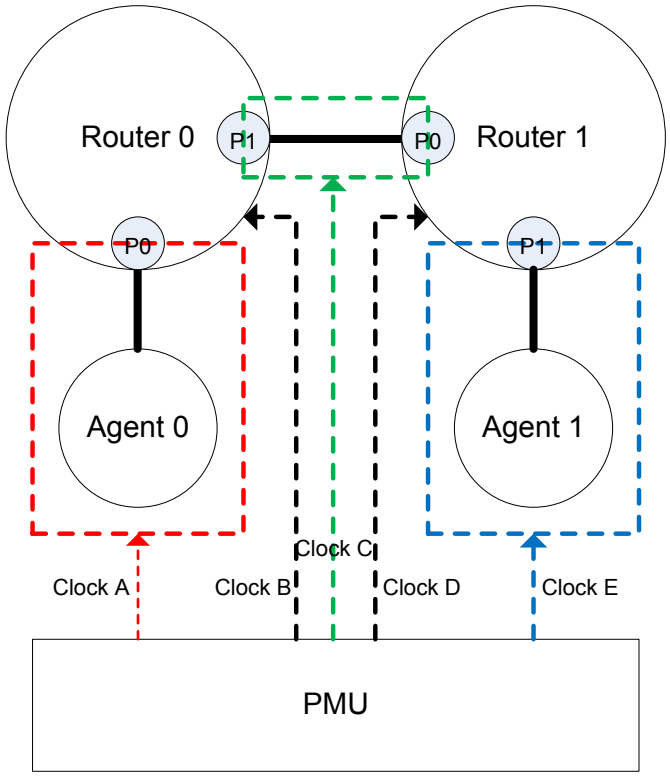
Figure 5 shows a fabric cluster that has asynchronous clock domains.

Note: Router 0 consists of three clock domains: Clock A, B, and C. Router 1 also consists of three clock domains: Clock C, D, and E.

Assume that all the clocks are initially turned off, and Agent 0 needs to send a transaction to Agent 1. Then, the following should occur.

1. Agent 0 handshakes with the PMU via `clkreq/clkack`, which will turn on Clock A and Clock B.
2. Agent 0 injects the transaction into Router 0.
3. Router 0 handshakes with the PMU via `fab_init_idle_exit/fab_init_idle_exit_ack` on P1, which will turn on Clock C and Clock D.
4. Router 0 injects the transaction into Router 1.
5. Router 1 handshakes with the PMU via `fab_init_idle_exit/fab_init_idle_exit_ack` on P1, which will turn on Clock E.
6. Router 1 sends the transaction into Agent 1.

Figure 5. Asynchronous Trunk Clock Gating Cluster Example



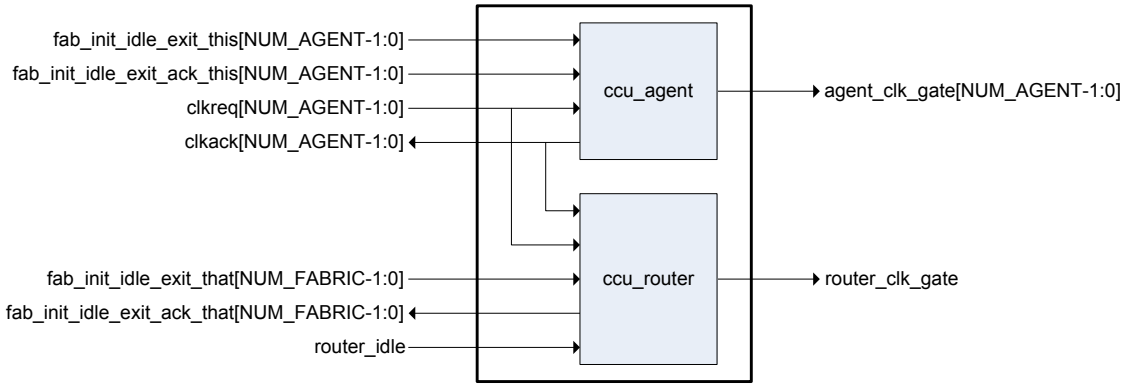
Clock Control Unit Design Example

Unlike the seemingly straightforward operations for the simplistic examples above, the task of designing a properly working PMU may be difficult, due to the simultaneous activities occurring on multiple ports. To help readers develop further insight into this topic, a simple Clock Control Unit (CCU) design example is presented in this section. Specifically, we focus on designing the CCU that controls one specific router and the agents that talk to that router for the following:

- When can the PMU gate the trunk clock for the router?
- When should the PMU resume the trunk for the router?
- When can the PMU gate the trunk clock for an agent?
- When should the PMU resume the trunk clock for an agent?

Naturally, therefore, CCU consists of two state machines: one to control the trunk clock of the router (ccu_router) and the other to control the trunk clocks of the agents on that router (ccu_agent). There will be one CCU for each router in the fabric cluster.

Figure 6. CCU Construction



As shown in Figure 6, the CCU deals with the following five groups' top-level I/O signals. (NUM_AGENT refers to the number of the ports on that router that talk to agents. NUM_FABRIC refers to the number of the ports on that router that talk to other routers.)

- `clkreq`[NUM_AGENT-1:0] inputs from the agents, and `clkack`[NUM_AGNET-1:0] outputs to the agents
- `fab_init_idle_exit_this`[NUM_AGENT-1:0] inputs from the router, and `fab_init_idle_exit_ack_this`[NUM_AGENT-1:0] outputs to the router.
- `fab_init_idle_exit_that`[NUM_FABRIC-1:0] inputs from other routers' ports that talk to the router under consideration, and corresponding `fab_init_idle_exit_ack_that`[NUM_FABRIC-1:0] outputs to those ports (coming from other CCU's)
- `router_idle` input coming from the router under consideration
- Trunk clock outputs to the router and to the agents: `clk_router` and `clk_agent`[NUM_AGENT-1:0]

In the Appendix, sample codes for `ccu_router` and `ccu_agent` are provided. There are a few points to note.

Note: Inputs to the logic are assumed to be synchronous to the clock of the logic itself. Those inputs need to be re-timed if they are from different clock domains.

Note: In the code, `fab_init_idle_exit_ack` and `clkack` are the simply one-clock delayed versions of `fab_init_idle_exit` and `clkreq`. This amounts to the CCU behavior that gates off the clock whenever possible. The end application can add system-specific behavior here. For example, to avoid a huge current spike, it may be necessary to stagger multiple clock gating events over time.

The code provided is missing two essential elements:

- The logic that actually implements clock gating cells.

Note: The provided code only outputs the control signals for those clock gating cells.

- The code is missing the logic that controls the trunk clock of the asynchronous ports that talk to other routers. (For example, Clock C in Figure 5.) One essential observation is that such a clock must be controllable from both sides. There can be three different cases for this.

1. Suppose a synchronous port is talking to a synchronous port on the other side. In this case, it is not necessary to implement additional logic, because the port clocks (which also are the router clocks) are to be controlled by the `router_clk_gate` signals. The clock-gate cell of each router will be separate.

2. Suppose an asynchronous port is talking to an asynchronous port, as in Figure 5. In this case, there should be a shared clock-gate cell that controls the shared port clock (Clock C in Figure 5). Then, there should be additional logic that turns on the clock when either of the ports has its `fab_init_idle_exit_ack` asserted.

Note: The additional logic is not responsible for generating `fab_init_idle_exit_ack` itself, as that signal is generated by the provided code.

3. Suppose an asynchronous port is talking to a synchronous port. In this case, the link is the port clock of the asynchronous port, and it is the router clock at the synchronous side at the same time. For this, there should be a shared clock-gate cell and its control logic. When `fab_init_idle_exit_ack` is asserted for the asynchronous port, the control logic should turn on the port clock, which turns on the synchronous port side automatically.

Note: The clock is also turned on when the synchronous-port router's CCU turns on the router clock. Therefore, the control of the clock-gate cell should be a form of the OR function to turn on the clock.

On the other hand, when `fab_init_idle_exit_ack` is asserted from the synchronous port side, it means that the synchronous port and the router that it belongs to, and the asynchronous port already have the clock (the shared clock-gate cell is already turned on). Therefore, the `fab_init_idle_exit` indication is only to turn on the router clock of the router that the asynchronous port belongs to, which is properly handled by the code provided.

Appendix: CCU example

```
module ccu
(
  clkreq,
  clkack,

  fab_init_idle_exit_this,
  fab_init_idle_exit_ack_this,

  fab_init_idle_exit_that,
  fab_init_idle_exit_ack_that,

  router_idle,

  fab_clk_gate,
  agent_clk_gate,

  rst_b,
  clk
);

parameter NUM_AGENT = 4;
parameter NUM_FABRIC = 4;

input logic [NUM_AGENT-1:0]          clkreq;
output logic [NUM_AGENT-1:0]         clkack;

input logic [NUM_AGENT-1:0]          fab_init_idle_exit_this;
input logic [NUM_AGENT-1:0]          fab_init_idle_exit_ack_this;

input logic [NUM_FABRIC-1:0]         fab_init_idle_exit_that;
output logic [NUM_FABRIC-1:0]        fab_init_idle_exit_ack_that;

input logic                          router_idle;

output logic                         fab_clk_gate;
output logic [NUM_AGENT-1:0]         agent_clk_gate;

input logic                          rst_b;
input logic                          clk;

ccu_router #(.NUM_AGENT(NUM_AGENT), .NUM_FABRIC(NUM_FABRIC)) i_ccu_router
(
  .clkreq (clkreq),
  .clkack (clkack),

  .fab_init_idle_exit (fab_init_idle_exit_that),
  .fab_init_idle_exit_ack (fab_init_idle_exit_ack_that),

  .router_idle (router_idle),

  .fab_clk_gate (fab_clk_gate),

  .rst_b (rst_b),
  .clk (clk)
);

ccu_agent #(.NUM_AGENT(NUM_AGENT)) i_ccu_agent
(
  .clkreq (clkreq),
  .clkack (clkack),

  .fab_init_idle_exit (fab_init_idle_exit_this),
  .fab_init_idle_exit_ack (fab_init_idle_exit_ack_this),

  .agent_clk_gate (agent_clk_gate),

  .rst_b (rst_b),
```

```
  .clk (clk)
);

endmodule

module ccu_router
(
  clkreq,
  clkack,

  fab_init_idle_exit,
  fab_init_idle_exit_ack,

  router_idle,

  fab_clk_gate,

  rst_b,
  clk
);

parameter NUM_AGENT = 4;
parameter NUM_FABRIC = 4;

input logic [NUM_AGENT-1:0]          clkreq;
input logic [NUM_AGENT-1:0]          clkack;

input logic [NUM_FABRIC-1:0]         fab_init_idle_exit;
output logic [NUM_FABRIC-1:0]        fab_init_idle_exit_ack;

input logic                          router_idle;

output logic                         fab_clk_gate;

input logic                          rst_b;
input logic                          clk;

//------------------------------------------------------------------------------
// fab_init_idle_ack signaling
//------------------------------------------------------------------------------

always_ff @(posedge clk or negedge rst_b)
 if (!rst_b) fab_init_idle_exit_ack <= '0;
 else fab_init_idle_exit_ack <= fab_init_idle_exit;

//------------------------------------------------------------------------------
// fabric clock gating control
//------------------------------------------------------------------------------

always_ff @(posedge clk or negedge rst_b)
 if (!rst_b) fabric_clk_gate <= 1'b1;
 else if (|clkack || |fab_init_idle_exit_ack) fabric_clk_gate <= 1'b1;
 else if (router_idle && !|fab_init_idle_exit && !|fab_init_idle_exit_ack && !|clkreq &&
!|clkack) fabric_clk_gate <= 1'b0;

endmodule

module ccu_agent
(
  clkreq,
  clkack,

  fab_init_idle_exit,
  fab_init_idle_exit_ack,

  agent_clk_gate,

  rst_b,
  clk
);
```

```
parameter NUM_AGENT = 4;

input logic [NUM_AGENT-1:0]          clkreq;
output logic [NUM_AGENT-1:0]         clkack;

input logic [NUM_AGENT-1:0]          fab_init_idle_exit;
input logic [NUM_AGENT-1:0]          fab_init_idle_exit_ack;

output logic [NUM_AGENT-1:0]         agent_clk_gate;

input logic                          rst_b;
input logic                          clk;

//------------------------------------------------------------------------------
// clkack signaling
//------------------------------------------------------------------------------

always_ff @(posedge clk or negedge rst_b)
 if (!rst_b) clkack <= '0;
 else clkack <= clkreq;

//------------------------------------------------------------------------------
// agent clock gating control
//------------------------------------------------------------------------------

always_ff @(posedge clk or negedge rst_b)
 if (!rst_b) agent_clk_gate <= 1'b1;
 else
  for (int i=0; i<NUM_AGENT; ++i)
   if (clkack[i] || fab_init_idle_exit_ack[i]) agent_clk_gate[i] <= 1'b1;
   else if (!clkack[i] && !fab_init_idle_exit_ack[i]) agent_clk_gate[i] <= 1'b0;

endmodule
```