

# Sun Microsystems Laboratories

**Subject:** Requeue State Diagram  
**Date:** September 6, 2008  
**From:** Ivan Sutherland  
**SML#:** 2008-is14

## References:

SML# 2008-is11: Description of a Dock Design, Ivan Sutherland, 29 July 2008

## PURPOSE

This memo describes how the Dock stage called “requeue” works. As its title implies, this memo is mainly concerned with the finite state machine embedded in the requeue stage.

## FUNCTION OF REQUEUE

The requeue stage is the junction between the epilog FIFO, called EPI, and the instruction execution ring. The requeue stage loads instructions from EPI into the ring or recirculates instructions already in the ring.

The requeue stage has two inputs and one output, namely the instruction execution ring. One of its inputs comes from the final stage of the instruction ring, namely the on deck stage called OD. Its other input is from EPI from which it gets fresh instructions to put into the ring.

Unlike the “demand merge” stages used in the switch fabric, the requeue stage is a “controlled merge” stage. It has internal state that directs it to take data from one of its inputs and to ignore or discard data from the other. This memo describes what that internal state does and how and when that internal state changes.

## STATES

The requeue stage has two kinds of internal states that I shall call “loading” and “circulating.” The loading states, 0 and 1, appear in the lower half of the state diagram. The notation “EPI --> ring” in the figure indicates that the requeue stage passes instructions from EPI into the ring. This loads the instruction ring.

---

Ivan told me to remove the confidentiality notice on this document --Adam

The circulating states, 2 and 3, appear in the upper half of the state diagram. The notation “EPI waits” in the figure indicates that in these states the requeue stage declines to take input from EPI, taking input exclusively from OD.

Another task of the requeue stage is to separate one-shot instructions from those marked to requeue. The on deck stage passes all instructions to the requeue stage whether or not they are marked for re-use. The requeue stage must discard the one-shot instructions. In all states it discards any one-shot instruction it gets from OD.

There is an additional distinction of internal states that I shall call “draining” and “using.” The draining states, 0 and 2, appear on the left of the state diagram. In these states the requeue stage discards instructions given to it by OD, thus draining the ring.

The using states, 1 and 3, appear at the right of the state diagram. In these states the requeue stage will pass instructions from OD through to the ring if possible, discarding only one-shot instructions from OD. In state 1 requeue cannot pass instructions into the ring because its ring output is in use by EPI. In this state it declines to take input from OD and OD waits.

## **WHEN TO CHANGE STATE**

When loading the ring, in states 0 and 1 shown at the bottom of the figure, the requeue stage is sensitive to a “tail” instruction. This special instruction marks the end of a sequence of instructions that will repeat. Upon receipt of a tail instruction from EPI, the requeue stage discards the tail instruction and changes from loading to circulating. This change appears in the state diagram as the two upward directed arrows marked “TAIL.”

Although the Outer Loop Counter, OLC is a part of the on deck stage, the requeue stage must act differently depending on the state of the OLC. The requeue stage must not access the OLC directly because the OLC lies in a separate asynchronous time domain. Instead, the on deck stage marks each and every instruction it sends to the requeue stage with a bit to indicate the state of the OLC at the time the instruction left OD. The requeue stage gets a single bit with every input from on deck indicating whether the OLC = 0, or at least, whether OLC was zero when on deck sent this instruction.

When draining the ring, in states 0 and 2 shown at the left of the figure, the requeue stage is sensitive to an instruction marked for requeue with a non-zero OLC. This instruction might be the “load OLC” instruction that set the OLC to a non-zero value. Alternatively, it might be a subsequent instruction if the load OLC instruction was a one-shot instruction. In either case, the requeue stage discards this instruction and

begins to recirculate. The horizontal arrows marked "OLC != 0 & requeue" in the figure indicate this change.

Please note that the requeue stage always discards the instruction that causes it to change state. It discards a tail instruction that moves it from loading to circulating. It also discards the first requeue instruction with non-zero OLC that moves it from draining to using.

There is only one way back from state 3. In state 3 the requeue stage circulates instructions from OD back into the ring if they are marked for re-use, discarding only one-shot instructions. The requeue stage examines each instruction from OD, seeking one marked as OD = 0. Such an instruction indicates that the dock has completed execution of the instruction loop and that the instructions in the ring are no longer valid.

## **IMPLICATIONS OF THIS DESIGN**

Null programs will cause deadlock. After receiving a tail instruction from EPI, the requeue stage enters state 2 or 3 in which it blocks further input from EPI. To avoid this state that will cause deadlock, the program after a tail instruction must at least make the OLC be non-zero and then return it to zero before there is another tail.

One can execute as long a program as one wishes while draining the ring. In state 0, the requeue stage will continue to accept and discard one-shot instructions from OD even while passing more instructions into the ring. The requeue stage leaves state 0 only in response to a tail instruction from EPI or an instruction marked for re-use from OD.

The requeue stage must discard all instructions that cause it to change state. Thus the first instruction marked for re-use after the OLC becomes non-zero will be lost. This is good if the instruction in question is a re-useable unconditional load OLC. Discarding such an instruction avoids making an infinite loop. However, discarding the first re-useable instruction may result in unexpected behavior if the programmer fails to mark an initial load OLC instruction for re-use, even though it won't be re-used. Such a failure will cause the first bone-fide instruction marked for re-use to vanish.

## **IS ARBITRATION REQUIRED?**

No. In states 0 and 1 the requeue stage takes data from EPI. The signal to leave these states comes from the EPI input, and is therefore in the same time domain as the current action of the requeue stage. Likewise in states 0 and 2 the requeue stage takes data from its OD input. The signal to leave these states comes from the OD input, and is therefore in the same time domain as the current action.

In state 0 the actions of copying input from EPI and discarding input from OD are concurrent. Thus the change from state 0 to state 1 might come anywhere in the cycle of copying EPI to output. Similarly, the change from state 0 to state 2 might come anywhere in the cycle of discarding an input from OD. Neither of these changes, however, affects the action of the cycle in progress, and so no arbitration is needed.

The change from state 3 back to state 0 does affect both actions. This change happens as a result of taking data from OD, and changes how the requeue stage will treat subsequent data from OD. That doesn't require arbitration. In the other timing domain, it merely reinstates the ability of the requeue stage to accept data from EPI. When this happens the EPI is either empty or has data waiting to deliver. Neither of these actions requires arbitration.

It's interesting to notice, once again, a state diagram with orthogonal and independent state transitions in one direction. In this case the transitions are marked "TAIL" and "OLC != 0 & requeue". Only after both transitions put the requeue stage in state 3 is it possible to make the diagonal transition back to state 0. The "outward" transitions from state 0 to states 1 or 2 are concurrent, but their sequence is of no concern. We need not consider their sequence because only when both outward transitions are complete does the requeue stage reach state 3 and only from state 3 is return to state 0 allowed.

# requeueStateDiagram

ies 6 September 2008

