CS-534: Packet Switch Architecture
Spring 2011

Department of Computer Science

Exercise Set 5:
Linked-List Queue Management

Assigned: Wed. 15 Mar. 2011 (week 5) - Due: Wed. 23 Mar. 2011 (week 6)

5.1 Pointer-Access Cost of Enqueue and Dequeue Operations

In section 3.3 slide 6, we saw the implementation of multiple queues using linked lists, so that they can share a given buffer memory space. The pointers are kept in separate memories, so that parallel accesses can be performed to the data and to the pointers. Oftentimes, the desired throughput corresponds to *one operation per clock cycle* --either one data segment (block) enqueue or one data segment dequeue per clock cycle. For each such operation, the required data memory throughput is one access (one address) per clock cycle --see exercises 3.1, 3.2. The throughput of the control (pointer) memories will often be adjusted to match the data memory throughput. When the required throughput exceeds one access (one address) per cycle, multi-ported memories will be needed for the corresponding pointers. The objective of this exercise is to determine the above throughput, i.e. *count* the number of required accesses per operation (enqueue or dequeue) to each pointer (control) memory. When counting, one has to be careful about what needs to be done in the "exceptional" cases, as outline below.

(a) Count the above numbers, for each of the pointer (control) memories (Empty, Head, Tail, NextPointer), in each of the following cases; the number of ports for each memory --in order to sustain the one-operation-per-cycle rate, is the worst of these numbers for that memory:

- **enqN** (Enqueue Normal): append a given *new* segment at the tail of a given queue *qID*. You don't know a priori whether the queue was empty or not, so you have to check for that, but assume that in this case the queue was **not** empty.
- **enqE** (Enqueue Exceptional): append a given *new* segment at the tail of a given queue *qID*. You don't know a priori whether the queue was empty or not, so you have to check for that, and assume that in this case the queue **was** empty.
- **deqN** (Dequeue Normal): remove the segment at the head of a given queue *qID*, and return the address of that segment. We know that the queue was not empty (the scheduler does not issue illegal or dummy operations), so you do not have to check for that. However, we do not know a priori whether the queue will become empty after we remove the head segment, so you have to check for that, but assume that in this case the queue does **not** become empty.
- **deqE** (Dequeue Exceptional): remove the segment at the head of a given queue *qID*, and return the address of that segment. We know that the queue was not empty so you do not have to check for that. However, we do not know a priori whether the queue will become empty after we remove the head segment, so you have to check for that, and assume that in this case the queue **does** become empty.

(b) If we wish to reduce the width of the Head/Tail memory by a factor of 2, we can store head and tail pointers "underneath" each other, in a memory that has half the original width; in that case, more accesses will likely be needed to that memory, but perhaps less than twice the original number? Count this new number. You now have to be careful about where the empty bit is stored: *(i)* in the "head/tail" memory, within the "head" word; *(ii)* in the "head/tail"

memory, within the "tail" word; *(iii)* in the "head/tail" memory, within both the head and tail words (two copies --always consistent or not necessarily always?); *(iv)* in a separate, single-bit-wide memory (addressed e.g. by the head/tail address divided by 2?). In the first three cases, do we need a separate write-enable signal for the empty-bit part of the memory? (note: if you want to modify only one of the two fields in a memory word but you know the value of the other, you don't need separate write-enalbles: you can always re-write the known value in the other field). There is a method to economize on the head/tail memory width by 1 bit: instead of storing an extra "empty" bit next to a k-bit pointer, reserve one of the values of this k-bit field --e.g. the 00...0 value (the "NULL" pointer)-- to symbolize an empty queue; in which of the above cases can we apply this method?

5.2 Queue Operation Latency

The above exercise concerned throughput; this exercise concerns *latency* of the enqueue and dequeue operations for linked-list queues implemented with multiple pointer memories as above. Independent accesses to distinct memory blocks can be performed in parallel; however, dependent accesses, where the result (read data) of one access is used as input to the next, must be performed in sequence. Count the number of clock cycles required to complete all accesses to all memories involved in an enqueue operation, and similarly for a dequeue operation; count this number separately *(i)* excluding the acceess to the data block, and *(ii)* including that data block access (assume that the entire data block, stored in a "wide" memory, is accessed in a single clock cycle). Assume that the enqueue and dequeue operations in the pipeline are independent from each other, thus you do not have to consider any bypasses (or stall cycles -- although, most likely, all dependencies can be resolved with bypasses, without ever needing to stall).

Assume that the empty bit and head and tail pointers for each queue are stored in a wide, *on-chip* SRAM, and are accessible in parallel (for a given queue), with a single-clock-cycle latency; this memory has as many ports as you calculated in the previous exercise. On the other hand, for the NextPointer memory, which is significantly larger, consider two cases: *(a)* the NextPointer memory is also on-chip, on the same chip, accessible with a single-clock-cycle latency; *(b)* the NextPointer memory is off-chip, in which case each chip-to-chip transfer costs an extra clock cycle in latency: reads have a 3-cycle latency (send address, read, return data), and writes have a 2-cycle latency (send address and data, write); in all cases, accesses are pipelined, and hence can be performed at a rate of one per clock cycle.

Up to the Home Page of CS-534

Last updated: 15 March 2011, by M. Katevenis.