The P3Q (Parameterized, Pipelined, Priority Queue)

An example of a highly parameterized synthesizable hardware component with complex control, and its solution in BSV

Introduction

The P3Q (Parameterized, Pipelined, Priority Queue) is an example of a highly parameterized synthesizable hardware component with complex control. The description here is a simplified version of a real-world IP block, part of a memory controller. In the next section we describe the specifications of the component, and the rest of this document is a tutorial exposition to a solution using BSV (Bluespec SystemVerilog). After reading the specification, we invite the reader to attempt a solution in any language of their choice, before studying the BSV solution. We would be very interested in seeing comparative solutions in other languages. We hope the reader will be persuaded that BSV offers a way to describe *synthesizable* hardware at a very high level, with a degree of abstraction and safety comparable to the most advanced languages.

The BSV source code for the full solution is displayed in an appendix of this document, and is also available in machine-readable form along with distributions of this document.

The specification of the P3Q

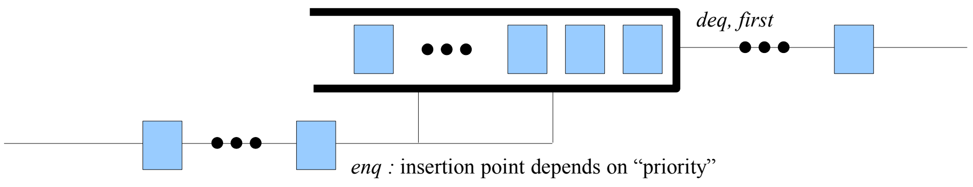

Figure 1The parameterized, pipelined, priority queue (P3Q)

The P3Q, illustrated in Figure 1, is a queue with a normal dequeue operation, i.e., one dequeues the item at the head of the queue. However, the enqueue operation is not necessarily at the tail. The items in the queue are maintained in a sorted order according to some notion of "priority" between items (a total ordering), and when a new item is enqueued, it must be inserted in its proper position according to this ordering.

The functional specs of the operations on the queue are quite conventional:

- *enq:* insert a new item into the queue, with its position in the queue determined by its priority relative to the items already in the queue. The *enq* operation can only occur if the queue is not full.
- *deq:* discard the item at the head of the queue. The *deq* operation can only occur if the queue is not empty.
- *first:* non-destructive examination of the head of the queue, i.e., returns the value of the item at the head of the queue, without dequeuing it. This is legal only when the queue is not empty.
- *clear:* Discard all items currently in the queue, making it empty. This is legal even if the queue is already empty.

The performance specs are also not unusual:

- *Throughput:* it must be possible to *enq* on every clock, and to examine *first* and *deq* on every clock (provided the not-full and not-empty constraints, respectively, allow it).
- *Minimum latency:* There is a minimum of 1-tick latency through the queue, i.e., an item enqueued on one clock is not available for *first* or *deq* until the next clock at the earliest.
- *Maximum latency:* When *enqs* and *deqs* are continuous, the latency of an item should be no more than 1+ the number of items ahead of it in the queue (taking into account also insertions ahead of it in the queue).
- *Clear supersedes:* The *clear* operation can be called on any clock. If *enq* and/or *deq* are attempted on the same clock, the *clear* operation supersedes, i.e., the queue is empty at the next clock.

The degree of parameterization, described below, is quite unusual. The design can have many *instances*, and each instance may specify different values for these parameters.

- *Size (capacity):* The capacity of the queue (the number of items it holds when full).
- *Item type:* The data type of the items held in the queue. Preferably, this is more than just the bit-width of the items. For example, in a queue instance that holds IPv4 (Internet Protocol v4) addresses, it should be impossible to enqueue or dequeue an int, even though they both happen to be represented in 32 bits.
- *Bit representation of items:* Even for a given item data type, it should be possible to customize the precise bit representation (e.g., little-endian vs. big-endian, ordering of fields in a struct).
- *Priority function:* For each item type and bit representation, it should be possible to customize the ">=" priority comparison operator which is used to determine where an item is inserted in the queue.
- *Pipelined or not:* The queue is expected to be instantiated in other designs running at various clock speeds. In some cases, the entire *enq* operation, which is the most expensive operation, can be completed in one clock. In other cases, it will be necessary to pipeline the operation, taking the decision about where to insert on one clock, and performing the actual insertion on the next clock. Nevertheless, the external performance specs remain unchanged. In particular the maximum-latency spec remains: an *enq* on one clock should be available for *first* and *deq* on the next clock, if it is the only item in the queue. Thus, a pipelined implementation will need some sort of "bypassing" logic to meet this maximum latency spec.

Invitation

At this point, before reading the Bluespec SystemVerilog solution below, we invite the reader to attempt a solution in any language of their choice, including software programming languages.

We note that succinct, clear solutions may be written in some software programming languages meeting the functional specs and some of the parameterization specs (capacity, item type, priority function). But usually a solution in a software programming language:

- will have nothing to say about the performance specs, since there is typically no notion of clocks, or even an abstract notion of time to express concepts like latency, throughput or simultaneity;
- may have nothing to say about customizable bit representations for item types;
- will have nothing to say about pipelining, unless it is a parallel programming language;
- and, most importantly, is unlikely to be synthesizable into quality hardware.

The P3Q in BSV

The P3Q solution is implemented in BSV in several files, each one containing a BSV *package.* The following is a list, which will be explored in more detail below:

PriQ.bsv	The P3Q module (polymorphic in the item data type, bit representation and priority ordering)
ItemQueue.bsv	An example item data type, showing how to customize bit representation and priority ordering
Config.bsv	Configuration parameters for ItemQueue

The discussion below is not intended to be a detailed tutorial on BSV. Rather, it is meant to be just enough commentary so that the reader gets a plausible idea of what is going on in the code. The goal is to persuade the reader that these facilities of BSV provide a way to express synthesizable designs at a very *high level*, even though this way is radically different from traditional methods associated with the terms "behavioral synthesis" or "high-level synthesis".

Interface type declaration for the P3Q

We use the standard BSV FIFO interface for the P3Q. The interface is described in detail in the BSV Reference Guide in sections 12.6 and C.1.3, and is also shown below in Figure 2. A BSV interface declaration is a *type* declaration because it represents generic information about the external view of a module. There can be more than one module implementation that presents/offers/implements/exports the same interface—indeed the BSV library provides many FIFO modules with the same FIFO interface.

The type FIFO#(a) is a parameterized type, i.e., the type *FIFO* parameterized with the type variable *a*. Such types are also called *polymorphic* types. This type can be read as "a queue that contains items of some as yet undetermined type *a*". The type variable *a* is analogous to C++ *templates*, i.e., it allows us to write generic code. BSV polymorphic types have a stronger notion of type-checking. For example, (a) it ensures that, in a particular queue, all elements have exactly the same type, and (b) it ensures that, for example, in a context expecting FIFO#(int), it is impossible to supply a FIFO#(IP_address) or a FIFO#(Bool).

A BSV interface definition contains *method* prototypes, corresponding to the operations or transactions that can be performed on the module. In this case, there are four methods corresponding to the four queue operations in the spec. The *enq* method takes an argument *x* of type *a*, and is of Action type, i.e., it returns no result, but causes something to happen inside the queue. The *deq* method and *clear* methods are also of type Action, i.e., they cause something to happen inside the queue; neither method takes any arguments. The *first* method takes no arguments, but returns a value of type *a*. Not being of type Action, it is guaranteed by BSV's type semantics to have no side effect, i.e., it is a purely combinational function (it may depend on, but cannot modify any internal state in the queue).

```
interface FIFO #(type a);
    method Action enq (a x);
    method Action deq;
    method a first;
    method Action clear;
endinterface: FIFO
```

Figure 2 The FIFO interface type (in the BSV library)

PriQ.bsv: the P3Q module (polymorphic in the item type, bit representation and priority ordering)

We now turn our attention to the file PriQ.bsv, which implements the actual queue module. We first explain the concepts of the queue, and then walk through the code (shown in full in the Appendix). Figure 3 shows the main storage for a queue of capacity *n.* It consists of an array of *n* registers, each holding an item's data plus a Valid/Invalid bit. We do not use the traditional "circular buffer with head and tail pointers" representation. Instead, these registers conceptually form a shift register with a variable insertion point. When there are *j* items in the queue, they occupy locations 1..*j*, with the head at index 1 and tail at index *j*. Their tag bits are Valid, and the remaining registers *j+1* through *n* have Invalid tag bits. Thus, the queue is empty if register 1's tag is Invalid, and is full if register *n'*s tag is Valid. The *first* operation examines register 1 (if Valid). The *deq* operation shifts all items in the range *n..1* to the right. The *clear* operation will set all the tags to Invalid. The *enq* operation is of course the most complex. Depending on the priority of the item being enqueued, it will occupy some register *k* in the range 1..*j+1,* and existing items in the range *k..j* will have to be shifted left.

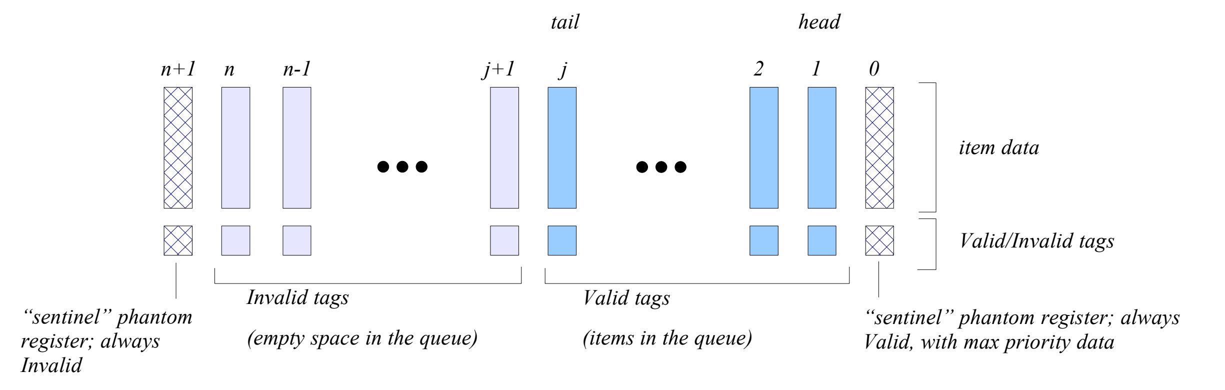
tail
head
n+1
n
n-1
j+1
j
2
1
0
item data
Valid/Invalid tags
"sentinel" phantom register; always Invalid
Invalid tags
(empty space in the queue)
Valid tags
(items in the queue)
"sentinel" phantom register; always Valid, with max priority data

Figure 3 The internal registers holding the queue

We will find it convenient to imagine two "phantom" sentinel registers at indexes 0 and *n+1*, respectively. Register 0 will always be Valid, and will always hold some dummy data with maximum priority value, and register *n+1* will always be Invalid (and its data is irrelevant). Because BSV uniformly treats registers just like modules, it will be easy to express these phantom registers in the code so that they look and behave just like registers, but produce no hardware (just wires that are driven by constant values).

In the code, and in the text, the terms *left* and *right* refer to the orientations implied by Figure 3.

Pipelining

In the pipelined version of the queue, we will have two stages, illustrated in Figure 4. In the first stage, we compute a set of *moves* for each of the queue registers, and also hold any data to be enqueued in a lastV register. In the second stage, we actually perform the moves on all the registers and load the lastV data into its proper place.

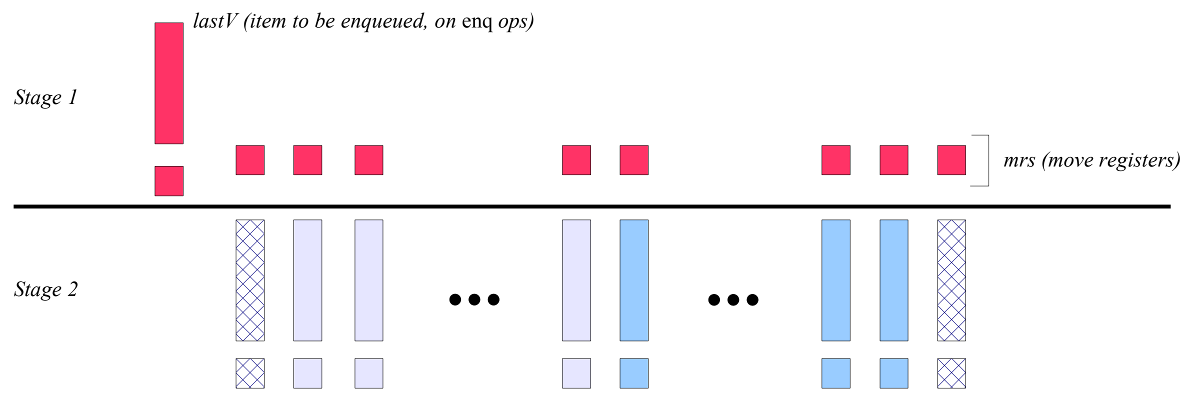

Figure 4 Pipeline stages (when the queue is pipelined)

Code walk-through

We are now ready to walk through the code in the file PriQ.bsv (also shown in the appendix). The "package-endpackage" brackets at the top and bottom of the file are for grouping together related definitions and controlling visibility of names. The file defines a package PriQ.

The two "import" statements just bring in definitions from the RWire and FIFO packages (in the BSV library).

"typedef enum {LEFT, .." defines a new type, Move, representing the kinds of "moves" we wish to perform on the queue registers. A data item should either be shifted in from the left, remain unchanged, be shifted in from the right, or be replaced by a newly enqueued item. The "deriving (Eq, Bits)" phrase just instructs the BSV compiler to define the bit representation and equality operation ("==") on this type in the obvious canonical way.

The "module mkConstReg ... endmodule" section is the module we will use to define the sentinel phantom registers. In BSV, registers are just primitive modules, i.e., they are instantiated just like modules, and they are operated with interface methods, just like modules. In particular, registers have _read and _write methods. Further, registers, like all modules in BSV, can be polymorphic and are strongly typed. In this case, the module interface is Reg#(a), indicating that it presents an interface of a register containing data of type *a*, which is a type parameter. The #(a x) phrase is a parameter to the module itself, and indicates the constant value *x* of type *a* that the register will contain. The body of the module just contains the definitions of the _read and _write methods. The former always returns the value *x*, and the latter performs no action, i.e., ignores its argument *y* of type *a*.

The section "module mkSizedPriQ ... endmodule: mkSizedPriQ", constituting the rest of the file, is the definition of the P3Q. The FIFO#(qe) phrase indicates the type of the interface that this module implements (offers, exports, presents). The FIFO type was discussed earlier. In this case, we are specifying that we are defining a polymorphic queue, where the items have some as yet unspecified type qe.

The phrase "provisos (...)" asserts some constraints on the type variable qe. The Bits#(qe, sqe) proviso asserts that the type qe must be representable in bit vectors of width sqe, because we need to store them in registers. The Ord#(qe) proviso asserts that the >= operation must be defined on items of type qe – this represents the priority ordering on items. The Bounded#(qe) proviso asserts that qe must have smallest and largest elements called *minBound* and *maxBound*, respectively, according to the >= ordering; we will load the phantom register at index 0 with the maxBound value. Thus, these provisos/assertions allow us to write the PriQ code in a completely generic way, parameterized by any specific details about the type of the items in the queue, their bit representation, or their priority ordering. We can deal with those concerns entirely separately, and we shall see an example later, when we study the ItemQueue.bsv file.

The #(Bool pipelining, Integer n) parameters to the module specify whether the module is to be pipelined or not, and the capacity *n* of the queue, respectively. All such module parameters are *static* parameters, i.e., they only affect the elaboration of the module.

The section: Reg#(Maybe#(qe)) queue [n+2];

describes the *n+2* registers of Figure 3, i.e., it declares an array called *queue* of *n+2* register interfaces. The register contents have type "Maybe#(qe)". The Maybe type is a user-definable type, but because it is very useful and frequently used, it is available from the standard BSV library. It is an example of a so-called "tagged union" type, and bundles together the idea that such a register either contains an Invalid bit, or contains a Valid bit along with data of type qe. Strong-typing ensures that data can only be read when it contains a Valid bit, and that it is impossible to read junk data when it contains the Invalid bit.

The immediately following for-loop initializes the queue array. Note that the 0'th and *n+1*'st locations are initialized using the mkConstReg module that we defined earlier, the former with Valid and maxBound, and the latter with Invalid. The rest of the entries are initialized with ordinary registers (using the mkReg module), each initialized to Invalid so that the queue is initially empty. The fact that we can mix standard registers with mkConstReg instantiations works smoothly precisely because registers are treated uniformly as modules.

The section: Reg#(Move) mrs [n+2]; Reg#(Maybe#(qe)) lastV = ?;

describes the registers of stage 1 of the pipeline in Figure 4. The register lastV has type Maybe#(qe) because it will only hold a Valid value if there is an *enq* operation in progress, otherwise it will be Invalid. The "?" initializer just indicates that we don't care what the initial value is in this register.

The next section: if (pipelining) begin ... end

initializes these stage 1 registers, if the *pipelining* Boolean parameter is true. This is an optional static elaboration activity, analogous to a "generate if" in Verilog. If the *pipelining* parameter is False, none of these registers get created, and there is zero hardware cost.

We next define the function *do_moves()* that creates a giant action combining a number of smaller actions representing all the moves on all the registers. The argument "Move ms[ ]" is an array of Move values, one per register. The argument "Maybe #(qe) v" represents a new value being enqueued, if Valid, and is Invalid otherwise. The for-loop in the function is just a static-elaboration representing *n* copies of the function body (analogous to a Verilog "generate for"). At each index, it performs the appropriate action on the register depending on whether the corresponding move is LEFT, RIGHT, REPLACE or SAME. This function will be called from the main "process_requests" rule described later.

The next function, *current_entry()*, encapsulates some details of "bypassing" in the case of pipelining. Without pipelining, the current *i*'th entry in the queue is of course the contents of register *i*, reflected in "return self" at the bottom of the function. However, in the case of pipelining, we must also take into account the moves that have been queued up for execution in the next cycle in stage 2 of the pipeline, because the "current entry" is, logically, based on the state after those moves. The function predicts what that state will be, by looking at the moves registers, and returns the appropriate value.

After a few utility definitions (firstElement, lastElement, notEmpty, notFull), the next major definitions are the two "RWires" rw_enq and rw_deq. These are present because of the requirement that the *enq* and *deq* methods may be invoked (by the external environment) on the same cycle. The exact state changes to be effected depend on whether or not they are being invoked simultaneously, and thus are not conveniently coded within either the *enq* method or the *deq* method body separately. Instead, the *enq* and *deq* methods, as can be seen near the end of the file, simply use these two RWires to indicate (a) whether they are being invoked or not, and (b) in the case of *enq*, to communicate the value of the item being enqueued. The single rule "process_requests", then, does all the work based on this information.

At this point we take a small diversion to discuss a few general BSV topics: rules, rule semantics, mapping of rules to clocked hardware, and the role of RWires. In BSV, all *behavior* is expressed in terms of rules (in this PriQ module, there is one rule, process_requests, to be described later). Each rule contains a *condition* (which can be a composite of several sub-conditions) and an *action* (which can be a composite of several sub-actions). The behavior of a system is defined using Rule semantics: execute one rule at a time, at each step picking any rule whose condition is true and performing its action. In particular, when coding with Rules, reasoning about correctness is not complicated by concerns of concurrency – each rule can be considered in isolation. In the rule semantics, there is no notion of clocks, just rule "steps/executions/firings". "Methods" in BSV, like the *enq* and *deq* methods of the PriQ module, are just parts of a rule, i.e., they can be considered as components of the rules in the environment of the module from which these methods are invoked.

The Bluespec synthesis tool will produce clocked hardware where typically the effect of many rules will occur on each clock, but it is always consistent with the Rule semantics, i.e., the net state change on each clock is always equivalent to a logical sequence of rule firings. Thus, the hardware can only reach states predicted by the rule semantics, thereby preserving any correctness properties established in the rule semantics. Thus, we get highly concurrent implementations, with the simplicity of reasoning about correctness without concurrency.

RWires are a bridge from the Rule semantics into the clocked hardware implementation, because certain specs ("simultaneous execution in the same clock") are only meaningful in the clocked hardware. In the Rule semantics, an RWire looks like a register: it is written ("wset" method) in one rule r1, and read ("wget" method) in a later rule r2. The implementation guarantees that, if r1 and r2 are executed in the same clock, then (a) this fact can be detected by r2 using the wget method, (b) r1 can communicate data to r2 using wset and wget, and (c) all this is implemented without any state elements, i.e., just using wires. Specifically, if rule r1 invokes rw_enq.wset(x), and rule r2 invokes rw_enq.wget, then (a) r2 reads a Valid result if r1 and r2 are being invoked in the same clock, else it reads Invalid, and (b) r2 reads the value of x as part of the Valid result.

We use these properties to communicate from the *enq* and *deq* methods into the rule called process_request, i.e., we communicate whether or not these methods are being invoked and, in the case of *enq,* we also communicate the value being enqueued. Returning to the code, we see that RWire rw_enq carries a value of type qe (the value being enqueued), and that RWire rw_deq does not carry any value of interest, and hence uses the type Bit#(0) (zero-width bit-vector).

The next four functions encapsulate what must be done at each register when there is no method call on a clock, when there is only a *deq* call, when there is only an *enq* call, and when *enq* and *deq* are called simultaneously. Each function takes the index *i* of a register and returns the Move necessary for that register. The "neither" and "deq_only" functions are straightforward.

The "enq_only" function also takes the value of the new item, v, as an argument. It uses the case expression to consider the current item (*i*) and the item to its right (*i-1*). If the right item is Invalid, then we are in the empty-space region of the queue, and nothing needs to be done (SAME). If the current item is Invalid and the right item is Valid, then the right item is the tail of the queue. We compare it with the new item v, deciding either to REPLACE the current item (grow new tail), or shift the right item to the left (RIGHT). The final clause of the case expression holds in the interior of the queue when the current item and right item are both Valid. In this case we compare the new item with both of them, and decide on the appropriate move. The sentinel registers allow us to write the same code for all the registers, without having to treat registers 1 and *n* as special cases. The "both" function that follows, is similar.

Note that in these functions, the comparisons are done using the >= operator, even though we have yet to specify the item type *qe*. The proviso "Ord#(qe)" at the top of the module guarantees that the >= operator is defined on this type, and the Bluespec tool will automatically plug in the correct definition of >= for each particular type *qe* at which this queue is instantiated.

We now come to the "rule process_requests ... endrule: process_requests" construct, which is a single rule that does all the work in this module. First, we define two arrays for holding Moves for stage 1 and stage 2. Note: these arrays are not state elements, they are just arrayed variables; and like all variables in BSV, they never represent hardware state, just a way to name intermediate values in computations. State elements, including registers, are created in BSV *only* through explicit module instantiation. The next two variables, stage1_v and stage2_v are used to name the incoming *enq* item, if any (Valid if present, Invalid otherwise).

The next phrase, "let the_fun = (case ... endcase)" tests the two RWires to determine whether an *enq,* a *deq,* neither, or both, are being requested, and picks the appropriate function from the previously defined family of four functions. The immediately following for-loop then applies this function at each index to compute all the moves. Note that the "enq_only" and "both" functions were defined with two arguments. Here, they are applied first to the argument *v*, and then to the index *i*. BSV functions can be applied to arguments one at a time in this way.

The next phrase shows what happens with pipelining and without. With pipelining, the just-computed moves are simply stored in the move registers "mrs", and the previous values in those registers are read out into the stage2_moves array. Similarly, the incoming value is stored in the lastV register, and the previous value is read out and called stage2_v. Without pipelining, stage2_moves are the same as stage1_moves, and stage2_v is the same as stage1_v. Note that the "if (pipelining) ... else ..." phrase is executed as part of static elaboration (like "generate if" in Verilog), i.e., for each instance of this queue we will either get the pipelined version or the non-pipelined version, never both. In other words, there is zero hardware cost for this parameterization. Similarly, statements like "stage2_moves[i] = stage1_moves[i]" are simply static renamings of the same wires; there is no hardware cost nor actual data movement involved.

The final line of the rule calls "do_moves" to perform all the actions. Again the "function call" here of type Action is a "call" during static elaboration; the Bluespec tool just produces the hardware to do the work, and it does not represent any "call" as such in the hardware. This style of encapsulating the work of the rule into the "do_moves" function is possible because Action is a first-class type in BSV, and it is possible to write arbitrary computations (including recursive computations) returning an Action.

The rule itself is preceded by the attribute "(* no_implicit_conditions, fire_when_enabled *)". This is an assertion made by the designer that nothing should prevent this rule from firing on every clock (because the rule itself, using RWires, can decide whether it needs to do any work on each clock). This assertion is statically verified by the Bluespec tools. (In general, a rule might not fire either because some condition or sub-condition of the rule is false, or because it is prevented from firing by some other more "urgent" rule that accesses a common resource.)

The final parts of the module are the "method ... endmethod" definitions of the *enq*, *deq*, *first*, and *clear* methods of the module interface. These are all straightforward. The only points of interest are the fact that the *enq* method is conditioned on the queue not being full, and the *deq* and *first* methods are conditioned on the queue not being empty. These "implicit conditions" of methods ensure that the environment cannot call these methods until the queue signals that it is legal to do so.

Finally, we note again that this entire module has been written generically, without any commitment to item type, its precise bit representation, or its priority ordering. All these details will be plugged in automatically by the Bluespec tool when instantiated for particular item types.

ItemQueue.bsv and Config.bsv: a specific item type, bit representation and priority ordering, and a P3Q for this item type

The file ItemQueue.bsv shows an example of a specific item type with customizable bit representation and priority ordering, and an instantiation of a P3Q for these kinds of items. For convenience, we have pulled out some key configuration parameters of ItemQueue into a separate package called Config in the file Config.bsv. We will refer to both files in the following discussion (both files are shown in full in the Appendix).

In ItemQueue.bsv, after importing a few packages, we define the Item type. This is a made-up example, where an Item contains 3 fields called tag, line and offset – the intention is to suggest fields of an address given to a cache memory, but those semantics are not important for this discussion. The first three typedefs define three types called Tag, Line and Offset, which are just bit-vectors of width TAG_SIZE, LINE_SIZE and OFFSET_SIZE, respectively, which are defined in Config. The uppercase first letter in the names Tag, Line and Offset are significant; BSV's convention is that all typenames begin with an uppercase letter.

Immediately following this is the type definition for the *Item* type: a struct (a record) containing the three requisite fields. The fieldnames begin with lowercase letters.

Custom bit representation

In order to get a representation of any type *t* in *n* bits, it must be made a member of the "Bits#(t,n)" typeclass. Specifically, this means that the compiler must be given definitions of two functions called pack and unpack that convert from the abstract type to the bit representation and vice versa. Most of the time, we simply append a "deriving (Bits)" clause to the type's typedef, which makes the compiler define a canonical bit representation (simply the concatenation of the bit representations of the three fields). Instead, in the next section, we define a custom bit representation for the Item type by explicitly specifying to the compiler the pack and unpack functions that make Item a member of the Bits typeclass.

Let us imagine that we will want one of 6 possible bit representations, representing the concatenation of the three fields in their 6 possible permutations. Exactly which one we want will be controlled by a parameter, *item_packing*, defined in Config.bsv.

The header of the section, "instance Bits#(Item, sz)" declares that Items can be represented in *sz* bits, and the next provisos line simply specifies that *sz* is just the sum of the sizes of the field sizes. Thus, we do not have to calculate *sz*, the compiler does it for us.

The pack() function that follows first defines 6 pack functions for the 6 possible pack permutations. Let us take a look at one of them, say *p231*. We exploit a convenient predefined type in the BSV library, the 3-tuple (which is just a generic record of 3 fields). Given three values *x*, *y* and *z*, we build a 3-tuple containing the values *y*, *z* and *x* (the 231 permuation), and then we use the 3-tuple *pack()* function to produce the bits. The inner call to *pack()* is not a recursive call. The typeclass mechanism is BSV's systematic mechanism for *overloading*. The inner call to *pack()* is thus the pack() function for 3-tuples, not the *pack()* function being defined here for Items.

The "match ..." line is an example of pattern matching. The argument *a* to the *pack()* function is pattern-matched to an Item type, allowing us in one shot to bind the names *x*, *y* and *z* to the values of the three fields tag, line and offset.

Finally, the case statement in the pack function picks the right pack function depending on the desired encoding represented in the parameter item_packing.

Note: although we are defining 6 packing functions, and choosing one of them in the case statement, this is all resolved during *static elaboration*, i.e., in a given implementation, the item_packing parameter will have a particular value, and the case statement will be completely resolved to pick just one packing function. Thus, there is no hardware overhead implied by this flexible parameterization.

Similarly, the *unpack()* function that follows first defines the 6 possible unpackings, and then invokes the right one from a case statement. Again, in each unpacking, we first use the *unpack()* defined on 3-tuples to just directly unpack the bit-vector into its 3 logical subfields, and then build the corresponding Item.

One final point on bit representations, not illustrated in this code. Suppose the Item type was embedded within some larger struct type TLarger, containing a field *f1*, an Item field, and another field *f2*. Suppose we used "deriving (Bits)" to instruct the compiler to pick a canonical bit representation for TLarger. The compiler would define *pack()* and *unpack()* for TLarger, and the bit representation would be the concatenation of the representations for *f1*, the Item field and *f2*. However, note that for the Item field, the compiler would use the *pack()* and *unpack()* functions we defined above, i.e., the Item field would continue to have its custom representation, even though it is embedded in a canonical representation at the next level up.

Custom priority ordering

We again use the BSV typeclass overloading mechanism to define a custom priority ordering for the Item type, in the next section. In the typedef for Item, had we just appended a "deriving (Ord)" clause, we would have instructed the compiler to define a canonical ordering on Items, which would have used a lexicographic ordering on the three fields (in turn using the ordering on bit fields to compare fields).

Suppose we wish to define ordering to be based on the lexicographic ordering of the line field followed by the tag field (and the offset field plays no role in ordering).

The header of the section, "instance Ord#(Item)" declares that Items are in the Ord typeclass, and the section defines the four operators <, <=, > and >= required for the Ord typeclass. For each of these operators, we construct 2-tuples containing the line and tag fields in that order, and then we use the corresponding <, <=, > and >= operators which are defined on 2-tuples (again, the inner call to these operators is not a recursion, but an application of the operator for a different type).

An important point to note is that these operators work on the abstract Item types, i.e., they do not have to be changed when we change the custom bit representation in the previous section–the compiler will do the right thing.

The next section, "instance Bounded(...)" similarly declares Item to be a member of the Bounded typeclass, which means we provide definitions for two identifiers called *minBound* and *maxBound* for the Item type. Recall that our P3Q module uses maxBound for a virtual sentinel "highest priority" item just above the head of the queue. The definitions just exploit the *pack()* and *unpack()* functions we defined earlier, to produce Items where all bits are 0 and all bits are 1, respectively.

A P3Q for this Item type

The final few lines of the ItemQueue.bsv file show a specialization of the generic mkSizedPriQ module into a module specifically for the Item type. This module has the (* synthesize *) attribute, namely it can be synthesized by the BSC compiler into hardware (Verilog RTL). The generic module could not be synthesized because the compiler does not know how to size the queue items, how to represent a max priority item, how to compare their priorities, etc., how deep to make the queue, etc., but once we provide the specific Item type and the *pipelining* and *queue_length* parameters, the compiler has all the information it needs to generate hardware.

Synthesis

In about 2-3 days, the P3Q design described above, and an accompanying testbench, were written by a BSV expert and the design was synthesized without problems at 400 MHz with a 0.18 micron library, using the Magma ASIC synthesis tools.

Summary

This white paper has shown how designs can be written in Bluespec SystemVerilog (BSV) succinctly and correctly. The attributes of BSV that enable this include:

- A high-level description of behavior*: Rules*
- A high-level way to modularize behavior with consistent, transactional semantics: *Interface Methods*
- *Polymorphism*, or the ability to parameterize with types
- A mechanism, *typeclasses,* which provides a systematic, semantically well-founded, user extensible mechanism for overloading, which allows, among other things,
  - A systematic way to define custom bit representations
  - A systematic way to define the meanings of common operators like <, <=, > and >=, and the meaning for concepts like "lower bound" and "upper bound" for new types
- A mechanism to parameterize designs with circuit fragments, such as the ordering function
- A seamless way to mix in complex, parameterized static elaboration with behavioral descriptions

Many of these capabilities exist in high-level programming languages, and even in some so-called hardware *modeling* languages. But in BSV, these descriptions also have clear hardware semantics, and the Bluespec tool provides a demonstrated route to synthesizing such descriptions into quality hardware.

[Document revision: 4 May 2009 23:00h]

Appendix: The file PriQ.bsv

```
// Copyright 2005-2009 Bluespec, Inc.  All rights reserved.

package PriQ;

// NOTE that in the following discussion we shall use the following
// conventions:
// 1.  The head of the queue (the next to be dequeued) is at the right-hand
// end, and has the lowest index (1).
// 2.  A higher priority is indicated by the priority value being higher.
// Entries with higher priority come out earlier.

import RWire::*;
import FIFO::*;

// The type of "moves".  The deriving clause tells the compiler to use the
// obvious definitions of equality of moves and their representation in bits.

typedef enum {LEFT, SAME, RIGHT, REPLACE} Move // LEFT means "receives from
           deriving (Eq, Bits);                // left" etc.


// We shall need a couple of "constant" registers, which always return their
// initial contents, and ignore any attempt to overwrite them.  This is the
// module that constructs them; note that they involve no state, only wires.

module mkConstReg#(a x)(Reg#(a));
   method a _read();
      return x;
   endmethod
   method Action _write(a y) = noAction;
endmodule

// This is the main module definition.  The parameter pipelining is a flag to
// tell whether we wish the implementation to be pipelined, n is the depth of
// the queue to be made, and qe is the type of the elements.  The provisos
// ensure (1) that values of this type can be represented as bits (so that
// they can be stored in registers); (2) that they can be ordered (so that we
// can use the <, >, etc. operators on them); (3) that they have "max" and
// "min values (because we shall want to use one of these in one of the end
// sentinel constant-registers).

module mkSizedPriQ#(Bool pipelining, Integer n)(FIFO#(qe))
   provisos (Bits#(qe,sqe), Ord#(qe), Bounded#(qe));

   // Note: a value of type "Maybe#(a)" is either "Invalid", or "Valid(x)",
   // where x is a value of type a.

   // We declare an array of registers to hold the queue, with an extra
   // constant sentinel register at each end.  The ordinary registers contain
   // Invalid if they are empty, and Valid(v) if they contain the value v.  The
   // constant register at the low end is always non-empty (and its contents
   // will be >= to anything it is compared with); the one at the high end is
   // always empty.
   Reg#(Maybe#(qe)) queue[n+2];
   for (Integer i = 0; i<n+2; i=i+1)
      begin
        let m = (i==0 ? mkConstReg(tagged Valid maxBound) :
                i>n  ? mkConstReg(tagged Invalid) :
                mkReg(tagged Invalid)); // the actual registers are all
```

```
                                 // empty to start with.
     let entry <- m;   //instantiate an appropriate register, and
     queue[i] = entry; //initialise the array entry with it.
   end


// Next we declare the interfaces of the pipeline registers, in case we
// need them: they consist of an array of registers storing moves, and one
// possibly to store a value to be inserted into the queue.  (The "?" is to
// prevent irritating warnings from the tool, which otherwise notices that
// lastV is not initialized when pipelining is False; initializing it here,
// even to the undetermined value "?", keeps the tool quiet.)
Reg#(Move) mrs[n+2];
Reg#(Maybe#(qe)) lastV = ?;

// If we are pipelining we instantiate all these registers; if we are not,
// we shall not use them and we needn't bother.
if (pipelining)
   begin
     for (Integer i = 0; i<n+2; i=i+1)
        begin
           let m = (i==0 ? mkConstReg(SAME) :
                   i>n  ? mkConstReg(SAME) :
                   mkReg(SAME));
           let move_reg <- m;
            mrs[i] = move_reg;
        end
     lastV <- mkRegU;
   end

// This defines the action which moves elements around in the queue.  Its
// arguments are the list of moves required, and maybe a value to be
// enqueued.
function Action do_moves(List#(Move) ms, Maybe#(qe) v);
   action
     for (Integer i = 1; i <= n; i = i+1)
        begin
           let self  = asReg (queue[i]);
           let left  = asReg (queue[i+1]);
           let right = asReg (queue[i-1]);
           case (ms[i])
             LEFT:    self <= left;
             RIGHT:   self <= right;
             REPLACE: self <= v;
             SAME:    noAction;
           endcase
        end
   endaction
endfunction

// This finds the current entry for a particular index.  If we are
// pipelining, the definition takes account of the stored moves, reading
// the appropriate slot; if not, the definition is of course trivial.
function Maybe#(qe) current_entry(Integer i);
   let self  = queue[i];
   let left  = queue[i+1];
   let right = queue[i-1];
   if (pipelining)
     begin
        let m = mrs[i];
        case (m)
           LEFT:    return left;
           SAME:    return self;
           RIGHT:   return right;
```

```
            REPLACE: return lastV;
         endcase
      end
   else return self;
endfunction

// These next definitions refer to the first and last "real" elements of
// the queue, and use them to test for emptiness and fulness of the entire
// queue.
let firstElement = current_entry(1);
let lastElement  = current_entry(n);
let notEmpty = (isValid(firstElement));
let notFull  = (!isValid(lastElement));

// NOTE: the BSV tool schedules many rules and methods in each clock cycle,
// but the overall effect is always as if they happened strictly
// sequentially within the cycle.  (This makes it much easier to recognise
// resource conflicts, rather than debugging a race condition.)  An RWire
// is implemented purely with wires, but looks rather like a register,
// except that its contents always revert to "Invalid" at the end of each
// clock cycle.  Here we use them for communication between the interface
// methods and the processing rule (which will be executed "later" in the
// cycle).

// If the value in this RWire is not Invalid, it indicates that we have
// accepted an "enq" call a little earlier in the cycle, in which case it
// holds Valid(the value enqueued):

RWire#(qe) rw_enq <- mkRWire;

// If the value in this RWire is not Invalid, it indicates that we have
// accepted an "deq" call a little earlier; the actual Valid value is
// irrelevant, so no bits are reserved for it:

RWire#(Bit#(0)) rw_deq <- mkRWire;

// This next group of four functions computes the move required to deal
// with the various combinations of enqueue and dequeue requests.  Their
// arguments are the value to be enqueued (if there is one), and the index
// of the queue slot concerned.  The queue is sorted in descending order of
// priority (right to left), using the >= ordering.

// If neither an enqueue nor a dequeue request has been received, all the
// elements stay the same.
function Move neither(Integer i) = SAME;

// If only a dequeue is to happen, each element is replaced by the one to
// its left.
function Move deq_only(Integer i) = LEFT;

// If only an enqueue is to happen, the new element must be slotted in at
// the appropriate place; all to the left of it are replaced by the one to
// their right, while all to the right of it stay the same.
function Move enq_only(qe v, Integer i);
   let self = current_entry(i);
   let right =current_entry(i-1);
   return (case (tuple2(self,right)) matches
           {.*, tagged Invalid}:
                      return SAME;
           {tagged Invalid, tagged Valid .p}:
                            return ((p>=v) ? REPLACE : RIGHT );
           {tagged Valid .s, tagged Valid .p}:
                            return ((s>=v) ? SAME :
                                         (p>=v) ? REPLACE : RIGHT );
         endcase);
endfunction

// Finally comes the function to handle simultaneous enqueue and dequeue.
function Move both(qe v, Integer i);
   let self = current_entry(i);
   let left  = current_entry(i+1);
   return (case (tuple2(left,self)) matches
           {.*, tagged Invalid}:
                      return SAME;
           {tagged Valid .l, tagged Valid .s}:
                             return ((l>=v) ? LEFT :
                                    (s>=v || i==1) ? REPLACE : SAME );
           {tagged Invalid, tagged Valid .s}:
                             return ((s>=v) ? REPLACE : SAME );
         endcase);
endfunction

// This is the rule which does all the work.  It is essential that it fires
// on every clock, so we specify attributes which the tool will check (and
// give an error report if they are not satisfied).

(* no_implicit_conditions, fire_when_enabled *)
rule process_requests;
   // We define two arrays of moves: the first will be initialised with the
   // moves required by the incoming commands, while the second will be
   // used to control the queue elements.
   Move stage1_moves[n+1];
   Move stage2_moves[n+1];

   // These next two variable hold incoming queue entries: the first is
   // initialised from the enq method; the second will be stored in the
   // appropriate queue element.
   Maybe#(qe) stage1_v = rw_enq.wget;
   Maybe#(qe) stage2_v;

   // We now choose the appropriate function from the group defined above,
   // by testing the values on the RWires.  NOTE that in BSV arguments can
   // be supplied to functions in stages.  Here enq_only and both needed
   // two arguments; we give them one here, and then they require just one
   // more, exactly like the other two functions.
   let the_fun =
       (case (tuple2(rw_enq.wget, rw_deq.wget)) matches
          {tagged Valid .v, tagged Invalid}:  return enq_only(v);
          {tagged Invalid,  tagged Valid .*}: return deq_only;
          {tagged Valid .v, tagged Valid .*}: return both(v);
          {tagged Invalid,  tagged Invalid}:  return neither;
      endcase);

   // This loop applies the chosen function (in parallel) to all the
   // slots, initializing the array of moves
   for (Integer i = 1; i<=n; i=i+1)
     action
        stage1_moves[i] = the_fun(i);
     endaction

   // If we are pipelining, we read the stage_ values from their registers,
   // and store the stage_1 values in their place.  If not pipelining, we
   // copy the stage_1 values to their stage_2 counterparts.
   if (pipelining)
      action
         for (Integer i = 1; i<=n; i=i+1)
            action
               (mrs[i]) <= stage1_moves[i];
```

```
                let m2 = mrs[i];
                stage2_moves[i] = m2;
             endaction

          lastV <= stage1_v;
          stage2_v = lastV;
       endaction
    else
       begin
          for (Integer i = 1; i<=n; i=i+1)
             stage2_moves[i] = stage1_moves[i];
          stage2_v = stage1_v;
       end

    // Now that all the stage_2 values are set, we apply them to the queue.
    do_moves(stage2_moves, stage2_v);
 endrule: process_requests

 // Finally come the methods.

 // The enq method merely sets the relevant rwire (assuming that it will be
 // processed "later" in the cycle, by the rule defined above).
 method Action enq(x) if (notFull);
    rw_enq.wset(x);
 endmethod

 // The "deq" method sets the relevant rwire; the value set is irrelevant.
 method Action deq() if (notEmpty);
    rw_deq.wset(?);
 endmethod

 // The first value is in the first queue element, if it's not empty:
 method first() if (firstElement matches tagged Valid .v);
    return (v);
 endmethod

 // The clear method empties each element, either immediately or (if
 // pipelining) by setting the pipeline registers appropriately.
 method Action clear();
    if (pipelining)
     action
        for (Integer i = 1; i<=n; i=i+1)
           (mrs[i]) <= REPLACE;
        lastV <= tagged Invalid;
     endaction
    else
     action
        Move ms[n+1];
        for (Integer i = 1; i<=n; i=i+1)
           ms[i] = REPLACE;
        do_moves(ms, tagged Invalid);
     endaction
 endmethod

endmodule: mkSizedPriQ

endpackage: PriQ
```

Appendix: The file Config.bsv

```
// Copyright 2005-2009 Bluespec, Inc.  All rights reserved.

// Configuration parameters
// Simply change these definitions and recompile to get a different
// point in the parameterization space

package Config;

// ----------------------------------------------------------------
// The bit widths of the Item fields
typedef  16  TAG_SIZE;
typedef   5  LINE_SIZE;
typedef   9  OFFSET_SIZE;

// ----------------------------------------------------------------
// A code specifying the field ordering
//     tag=1, line=2, offset=3
// so  123 specifies the order (tag,line,offset),
// and 312 specifies the order (offset,tag,line), and so on

Integer item_packing = 321;

// ----------------------------------------------------------------
// Specify whether the P3Q should be pipelined or not

Bool pipelining = True;

// ----------------------------------------------------------------
// Specify the capacity of the P3Q

Integer queue_length = 4;

// ----------------------------------------------------------------

endpackage: Config
```

Appendix: The file ItemQueue.bsv

```
// Copyright 2005-2009 Bluespec, Inc.  All rights reserved.

package ItemQueue;

// This package shows a made-up example of an "Item" type,
// just to demonstrate customizable bit representation and ordering.

// It also shows an instantiation of a P3Q containing such items.

// An item is a struct (record) with three fields: tag, line, offset.
// (suggesting an address for a cache)
//
// Parameterizations (customizations):
// - bitwidth of the fields
// - bit representation customizable to 6 choices, representing
//   all possible concatenation orders of the three fields
//       i.e., {tag,line,offset}, {tag,offset,line}, ...
// - item ordering, based on certain bits of the fields

import FIFO::*;
import PriQ::*;

// ----------------------------------------------------------------
// Bit widths are defined in the Config package (in Config.bsv)

import Config::*;

// ----------------------------------------------------------------
// The Item type

typedef  Bit#(TAG_SIZE)      Tag;
typedef  Bit#(LINE_SIZE)     Line;
typedef  Bit#(OFFSET_SIZE)   Offset;

typedef struct {
   Tag      tag;
   Line     line;
   Offset   offset;
} Item;

// ----------------------------------------------------------------
// Customized bit representation for Item type

// Rather than using "deriving (Bits)" in the typedef of Item,
// which would result in a canonical representation (concatenation
// of tag, line and offset in that order), we explicitly define the way in
// which the type Item is an instance of the Bits typeclass, by
// providing explicit definitions of the "pack" and "unpack" functions.
// The Bluespec tool automatically uses these representations
// wherever necessary.

// The particular packing is chosen by case-switch on an Integer
// parameter item_packing, which is defined in the Config package.

instance Bits#(Item, sz)
   // assert TAG_SIZE + LINE_SIZE + OFFSET_SIZE = sz
   provisos (Add#(TAG_SIZE, LINE_SIZE, k), Add#(k, OFFSET_SIZE, sz));

   function pack (a);
```

```
      function p123(x,y,z) = pack(tuple3(x,y,z));
      function p231(x,y,z) = pack(tuple3(y,z,x));
      function p312(x,y,z) = pack(tuple3(z,x,y));
      function p132(x,y,z) = pack(tuple3(x,z,y));
      function p321(x,y,z) = pack(tuple3(z,y,x));
      function p213(x,y,z) = pack(tuple3(y,x,z));

      match tagged Item {tag: .x, line: .y, offset: .z} = a;

      case (item_packing)
       123: return p123(x,y,z);
       231: return p231(x,y,z);
       312: return p312(x,y,z);
       132: return p132(x,y,z);
       321: return p321(x,y,z);
       213: return p213(x,y,z);
       default: return(error("invalid item_packing value"));
      endcase
   endfunction

   function unpack(unpacked_bits);

      function u123(bs);
       match {.x,.y,.z} = Tuple3#(Tag,Line,Offset)'(unpack(bs));
       return (Item {tag: x, line: y, offset: z});
      endfunction
      function u231(bs);
       match {.y,.z,.x} = Tuple3#(Line,Offset,Tag)'(unpack(bs));
       return (Item {tag: x, line: y, offset: z});
      endfunction
      function u312(bs);
       match {.z,.x,.y} = Tuple3#(Offset,Tag,Line)'(unpack(bs));
       return (Item {tag: x, line: y, offset: z});
      endfunction
      function u132(bs);
       match {.x,.z,.y} = Tuple3#(Tag,Offset,Line)'(unpack(bs));
       return (Item {tag: x, line: y, offset: z});
      endfunction
      function u321(bs);
       match {.z,.y,.x} = Tuple3#(Offset,Line,Tag)'(unpack(bs));
       return (Item {tag: x, line: y, offset: z});
      endfunction
      function u213(bs);
       match {.y,.x,.z} = Tuple3#(Line,Tag,Offset)'(unpack(bs));
       return (Item {tag: x, line: y, offset: z});
      endfunction

      case (item_packing)
       123: return u123 (unpacked_bits);
       231: return u231 (unpacked_bits);
       312: return u312 (unpacked_bits);
       132: return u132 (unpacked_bits);
       321: return u321 (unpacked_bits);
       213: return u213 (unpacked_bits);
       default: return(error("invalid item_packing value"));
      endcase
   endfunction
endinstance

// ------------------------------------------------------------------
// Item ordering

// Rather than using "deriving (Ord)" in the typedef of Item,
// which would result in a canonical ordering (compare items as bit
// vectors), we explicitly define the way in which the type Item is
// an instance of the Ord typeclass, by providing explicit definitions
// for the comparison operators.
// The Bluespec tool automatically uses these definitions of comparison ops
// wherever one uses <, <=, >, >= on values of this type.

// Let's assume that ordering is based on the tag field and on
// certain upper bits of the line field (suggesting a "page" address).

instance Ord#(Item);
   function \< (e1,e2);
      let t1 = tuple2(e1.line, e1.tag);
      let t2 = tuple2(e2.line, e2.tag);
      return (t1 < t2);
   endfunction
   function \<= (e1,e2);
      let t1 = tuple2(e1.line, e1.tag);
      let t2 = tuple2(e2.line, e2.tag);
      return (t1 <= t2);
   endfunction
   function \> (e1,e2);
      let t1 = tuple2(e1.line, e1.tag);
      let t2 = tuple2(e2.line, e2.tag);
      return (t1 > t2);
   endfunction
   function \>= (e1,e2);
      let t1 = tuple2(e1.line, e1.tag);
      let t2 = tuple2(e2.line, e2.tag);
      return (t1 >= t2);
   endfunction
endinstance

// ----------------
// For use in P3Qs, we also need a definition of a "maximum" priority Item.

// Rather than using "deriving (Bounded)" in the typedef of Item,
// which would result in a canonical definition of "maxBound" (bit-vector
// with all 1s), we explicitly define the way in which the type Item is an
// instance of the Bounded typeclass, by providing explicit definitions of
// the "minBound" and "maxBound" identifiers.
// The Bluespec tool automatically uses these representations
// wherever necessary.

instance Bounded#(Item);
   minBound = unpack (0);
   maxBound = unpack (-1);
endinstance

// ----------------------------------------------------------------
// Having defined the item type, we can define a module to produce the
// particular kind of P3Q that we want, and it can be synthesized to RTL.
// (Note that the definition of mkSizedPriQ in PriQ.bsv cannot be synthesized,
// it is not tied down to a particular entry type -- so, for example, it is
// not known at that stage how wide the ports must be.)

(* synthesize *)
module mkQueue(Q#(Item));
   let queue <- mkSizedPriQ (pipelining, queue_length);
   return queue;
endmodule

endpackage: ItemQueue
```