Tagged Up/Down Sorter— A Hardware Priority Queue

SIMON W. MOORE AND BRIAN T. GRAHAM

University of Cambridge, New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK
Email: Simon.Moore@cl.cam.ac.uk, Brian.Graham@cl.cam.ac.uk

We present a hardware oriented priority queue algorithm requiring $\frac{n}{2}$ comparators and swappers to maintain an n item queue. It supports two operations, *insert* and *extract minimum* (or alternatively, *extract maximum*), both of which operate in a single cycle. Thus, sorting time is $O(n)$. Records with identical keys are always extracted in FIFO order of insertion. A formal proof of correctness of these sorting and FIFO characteristics is presented.

Received February 2, 1995; revised September 26, 1995

1. INTRODUCTION

A priority queue is an essential component in many software systems. This paper was motivated by the apparent lack of a priority queue algorithm that could be efficiently implemented in hardware. Such a device could be used in a wide range of applications from rapid scheduling (e.g. for multithreaded processors [1] or ATM network routers [2]) to event timers which can efficiently handle multiple events.

The next section presents a statement of objectives. Section 3 reviews background material starting with simple non-scalable solutions and a hardware view of the popular software priority queue, the heap sort. Then the hardware oriented rebound (Section 3.3) and up/down (Section 3.4) sorters are discussed because they provide inspiration for the tagged up/down sorter (Section 4). Whilst the tagged up/down sorter is a deceptively simple algorithm, it has a complex behaviour. To show that the algorithm conforms to the objectives a machine checked formal proof of correctness is given (Section 5). Finally implementation strategies are presented in Section 6 and conclusions are drawn in Section 7.

2. OBJECTIVES

There are two objectives:

1. To design a device which can perform an *insert* or *extract minimum* (or as a variant, an *extract maximum*) operation every clock cycle.
2. Records with identical keys should be extracted in FIFO order of insertion (particularly useful for some scheduling operations).

3. BACKGROUND

There are many hardware sorting techniques, of which most aim to sort a complete set of data in the minimum time using as little hardware as possible (e.g. Batcher sorting networks [3, 4], heap sort on a systolic array [5] and others [6]). Unfortunately these do not meet our first objective of single cycle insertion and extraction.

In order to meet the first objective it is essential that any number inserted must be compared (and possibly swapped) with the current minimum value. An obvious solution would be maintain a sorted list but this would require $n - 1$ compare and swap units to sort n numbers in a single cycle.

3.1. Non-scalable solutions

There are several non-scalable solutions. For example, if key size can be kept very small than a FIFO may be allocated for every possible key [7]. Sorting is then just a multiplexing operation. Alternatively, if there is a wide range of possible keys but only a small number of records then a parallel search through all the records, using a content addressable memory structure, could be used [8].

If fast insertion is required but slow extraction is sufficient then a priority packet queue may be used [9]. In such a scheme inserted keys are compared with the current minimum and the larger result is buffered. Extraction picks up the current minimum and then an exhaustive search of the buffer is performed to find the new minimum, either serially or with some degree of parallelism.

3.2. Variations on the heap sort

Typically software implementations of priority queues utilise the heap sort technique which takes $O(log(n))$ time to insert or extract [10]. Insertions are made at the bottom of the heap and the heap is then massaged into a correct ordering (this process is sometimes called the "heapify" function). Extraction of the minimum is from the top and the hole it leaves is filled by a value from the bottom followed by an invocation of the heapify function.

A hardware variant to meet the objectives would require insertion and extraction initiated from the top of the heap. A dedicated processing element (PE) could be placed at each level of the tree structure within the heap. Thus, large values (assuming an extract minimum is required) would ripple down through the levels dislodging even larger values and settling in their place.

The only problem now is to maintain a balanced heap in order to prevent the algorithm degenerating into a sorted list structure. One approach would be to maintain a count of the number of nodes below every node so that at each level of the tree a decision about which lower levels should store the next value. This appears to be inefficient in terms of storage but it should be noted that the number of PEs and the size of the counter for each node would grow as $O(log(n))$. Thus, in terms of silicon real-estate this would work well for large datasets. Unfortunately an insertion or extraction takes at least two cycles (read and examine followed by a write).

3.3. The rebound sorter

The rebound sorter was proposed by Chen *et al.* [6] and improved upon by Ahn and Murray [11]. Whilst it is unsuitable for our application it forms the basis for more suitable approaches.

The basic sorting element (see Figure 1) consists of two memory elements capable of storing one word, a comparator and various data paths. Incoming data consists of two words: a word of *key* and a word of associated *data* to form a record. Records are inserted key first followed by the data on the subsequent cycle. The comparator is used to compare keys stored in the *Ln* and *Rn* parts of the sorting element in order to determine the direction that the (*key*, *data*) pairs should take; this is known as the *decision cycle*. The values input in the following cycle will be the associated data so the decision made in the current cycle is used again in order that the data follows its key; this is known as the *continuation cycle*.

Figure 2 illustrates the sorting behaviour. The principle of the algorithm is that incoming values proceed down the left side until they rebound off the bottom (hence the name) or hit a larger value on the diagonally lower right.

It can be seen that records take two cycles to insert or extract and all of the insertions must take place followed by all the extractions. Thus, this algorithm does not meet our objectives. Furthermore, it should be noted that

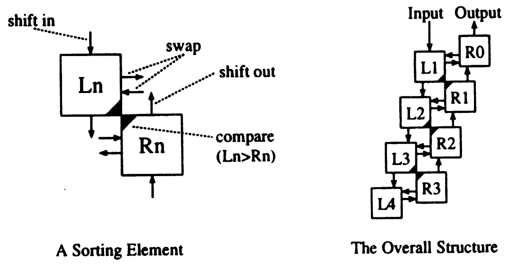

FIGURE 1. Structure of the rebound sorter.

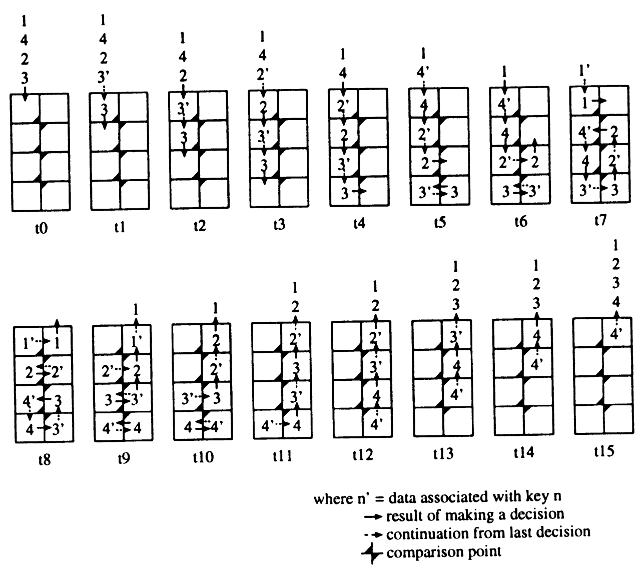
t0
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12
t13
t14
t15
where n' = data associated with key n
result of making a decision
continuation from last decision
comparison point

FIGURE 2. An example of a rebound sort.

$n - 1$ comparators are required to sort n records and that these comparators are only used every other cycle.

3.4. The up/down sorter

The up/down sorting algorithm was originally designed to be implemented in bubble memory technology [12]. It is constructed as a linear array of sorting elements in a similar manner to the rebound sorter described in the previous section. However, (*key*, *data*) pairs are inserted in parallel and the sorting element (see Figure 3) is more complex, primarily because of the implementation technology.

Initially all of the sorting elements contain infinity which may be indicated by the maximum possible number. An inserted value arrives in A_n. Simultaneously a copy of C_n is made to B_n, and D_n is transferred to A_{n+1}. A compare and steer operation takes place resulting in the maximum of A_n and B_n being transferred to D_n and the minimum of A_n and B_n transferred to C_n. Extraction similarly involves C_n being removed or transferred to B_{n-1}, D_n copied to A_n and C_{n+1} transferred to B_n followed by the compare and steer operation. An example is given in Figure 4.

Interestingly this algorithm allows insert and extract operations to be interleaved, and only requires $\frac{n}{2}$ sorting elements to sort n numbers. Unfortunately this implementation uses four storage areas per sorting element but if implemented in digital electronics this may be reduced

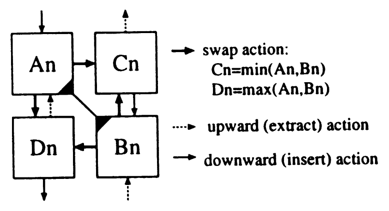

FIGURE 3. The up/down sorting element.

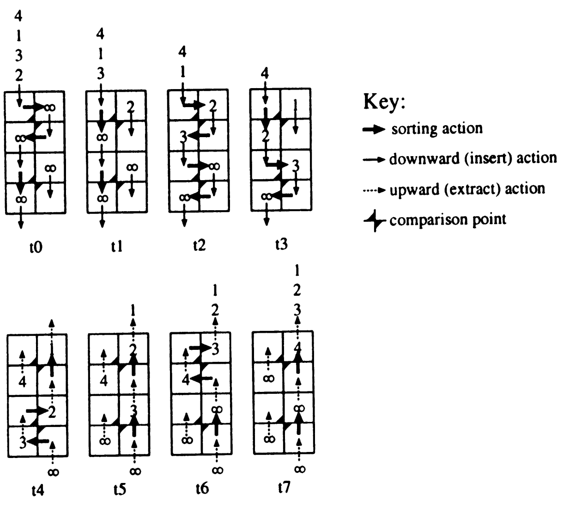

FIGURE 4. An example of an up/down sort.

to just two storage areas. The next section abstracts this algorithm, determines that FIFO ordering of identical keys is not maintained and suggests solutions. Then a clocked digital implementation is presented.

4. THE TAGGED UP/DOWN SORTER

4.1. Abstracting the up/down sorter algorithm

The up/down sorting element (see Figure 3) may be abstracted to two memory elements which may be swapped, and a comparator (see Figure 5). The algorithm may then be described as a two stage process:

1. Insert:

$$(L'o := \{key = new\_key, data = new\_data\}) \wedge (\forall n \geqslant 0.(L'_{n+1} := L_n))$$

or extract:

$$(extracted := R_0) \wedge (\forall n \geqslant 0.(R'_n := R_{n+1}))$$

2. Compare and swap:

$$\forall n \geqslant 0.(L'_n, R'_n) := \textit{if } (L_n.key < R_n.key) \textit{ then } (R_n, L_n) \textit{ else } (L_n, R_n)$$

where *key* is the key part of the record, *data* is the associated data part of the record, L_n, R_n are the current left and right records and L'_n, R'_n are the next left and right records.

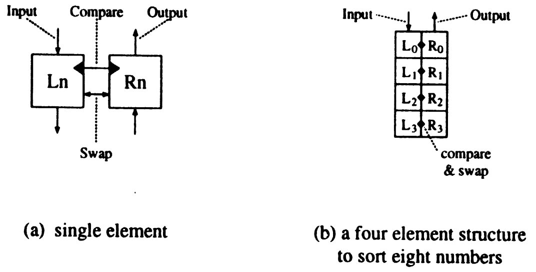

FIGURE 5. Abstraction of the up/down sorter.

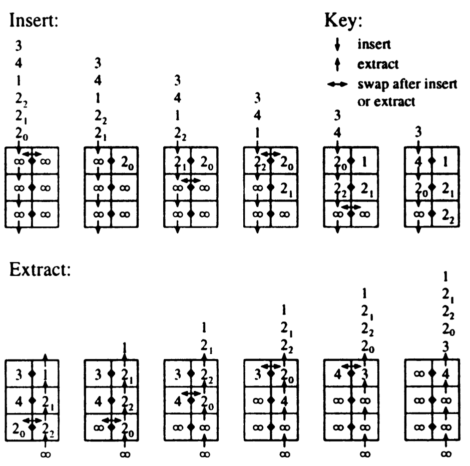

FIGURE 6. Ordering problem with the up/down sorter.

Whilst this algorithm sorts correctly, the FIFO ordering of records with identical keys is not maintained (see Figure 6). This problem arises when records on the right are swapped back to the left.

4.2. Ensuring FIFO ordering

FIFO ordering could be assured by associating an order of entry number with each record. However, a cleaner solution is to tag records by setting a single tag bit when they arrive on the right so that if they are swapped to the left they can be forced to swap back to the right on the next cycle. This works because once a record arrives on the right it must be sorted with respect to the other keys on the right. If a record gets swapped to the left, then on the next cycle (regardless of whether an *insert* or *extract* takes place) it will be compared with the right value which was previously physically below it. Thus the right key must be either greater than the one on the left or have the same key. However, the record on the right was inserted later than the record on the left. Therefore, a swap must be performed if the ordering on the right is to be maintained (see Figure 7 for an example). This is formally proved in the next section.

The tagged up/down sorting algorithm may thus be defined as a two stage process:

1. Insert

$$(L'_0 := \{key = new\_key, data = new\_data, tag = false\}) \wedge (\forall n \geqslant 0.(L'_{n+1} := L_n))$$

or extract:

$$(extracted := R_0) \wedge (\forall n \geqslant 0.(R'_n := R_{n+1}))$$

2. Compare and swap:

$$\forall n \geqslant 0.(L'_n, R'_n) := \textit{if } ((L_n.key < R_n.key) \vee L_n.tag) \textit{ then } (R_n, \{key = L_n.key, data = L_n.data, tag = true\}) \textit{ else } (L_n, R_n)$$

where *key* is the key part of the record, *data* is the

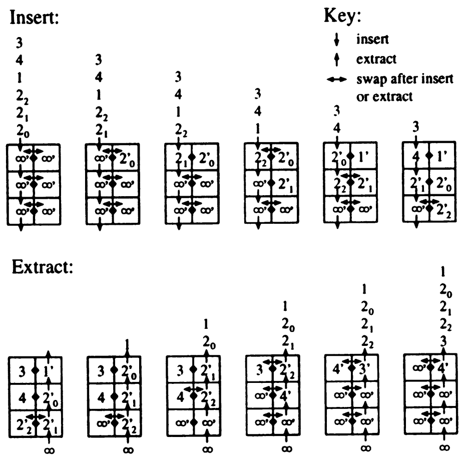

FIGURE 7. Using tagging to ensure FIFO ordering.

associated data part of the record, *tag* is the tag part of the record, L_n, R_n are the current left and right records and L'_n, R'_n are the next left and right records.

5. PROOF OF CORRECTNESS OF THE ALGORITHM

Formal verification of an algorithm entails making a formal definition of the algorithm and proving that, under particular constraints, certain necessary (and formally defined) properties are assured. These necessary properties give a more abstract specification of the behaviour of the algorithm. In this work, the key property is that the least record currently in the queue is the one returned by the extract operation. An invariant is defined to specify well-formed states of the queue and it is proved that key property holds for these states. The invariance is proved by showing it holds an on empty queue and is maintained by both insert and extract operations. The proof of these properties is realised using the HOL system [13], a high integrity machine-implementation of a classical higher order logic. The HOL system has been used often for reasoning about properties of hardware designs [14–18]. The scope of other applications can be gleaned from the proceedings of the annual HOL Users Group Workshops [19–22]. The use of this system lends a high level of assurance that the proof is valid.

Using formal methods demands a precise and complete specification of the algorithm's data structures and its operations and likewise the invariant and desired properties. Concepts expressed informally in natural language need translation into precise logical expressions, and these must be subjected to examination to confirm that they capture the intended concepts adequately. Hence in presenting the verification we dwell at length on the representation of the queue and the definition of the invariant.

5.1. Representing the queue and algorithm

Both insert and extract operations involve shifting one side of the stack one position relative to the other. We represent the nonempty locations of the queue as a pair of lists of records. Inserting a record consists of augmenting the left list with a new record at its head, while extraction removes the head record from the right list. Both are followed by a compare and swap function passing over the list pair. An informal proof by induction on the number of records in an infinite queue satisfies that the inserted records always cluster at the front of the queue, with no intervening empty locations, and thus the chosen representation is suitable.

A record datatype has a key field of type `:num` (natural numbers), a data field of arbitrary type, a tag field of type `:bool` and a timestamp field also of type `:num`. The timestamp serves only to record the order of insertion of records for specifying the required FIFO behaviour when two records have the same key, and is used by the algorithm specification. Record field selectors *key, time* and *tag*, and a tagging function `set_tag` are defined, all with the obvious meanings.

A few auxiliary functions are needed to define operations on lists and list pairs. We use the symbol ':' as the infix (cons) operator that adds a new element at the head of a list, and '::' for the infix append operator. The (higher order) function `Map` applies a function to every element in a list. The meaning of the function is expressed below, representing the primitive recursive definition that exists in the HOL system. Note that function application is indicated by juxtaposition, lists are enclosed by '[' and ']', elements are separated by ';', and the empty list is '[]'.

$$\texttt{Map}\, f[a_0;\ldots;a_n] = [f\,a_0;\ldots;f\,a_n]$$

Two Map functions for application to pairs of lists are given. `Map2_1` applies a function *f* to pairs of elements from each list, and builds a single list as a result. `Map2_2` is similar but returns instead a pair of lists. Both functions have an additional function-valued argument *g* which is applied to the remaining elements of the longer list, should the lists be unequal in length. (In our application, one list may be at most one longer than the other.) A minor variant of the former named `Map2_1a` (not shown) supplies the remainders of both lists as an added argument to the function *f*.

$$(\texttt{Map2\_1}\ fg([l_0;l_1;\ldots;l_n],[r_0;r_1;\ldots;r_n;r_{n+1}]) = [f(l_0,r_0);f(l_1,r_1);\ldots;f(l_n,r_n)] :: (g[r_{n+1}])$$

$$(\texttt{Map2\_2}\, fg([l_0;l_1;\ldots;l_n],[r_0;r_1;\ldots;r_n;r_{n+1}]) = ([l'_0;l'_n;\ldots;l'_n] :: ll, [r'_0;r'_1;\ldots;r'_n] :: rr)$$

$$\text{where } (l'_0,r'_0) = f(l_0,r_0), (l'_1,r'_1) = f(l_1,r_1), \ldots, (l'_n,r'_n) = f(l_n,r_n), (ll,rr) = g[r_{n+1}]$$

We define two distinct ordering relations. `below` is used by the algorithm to determine when a swap should be performed. It considers only the keys of the records, and the tagged property of the left one, in accordance with the algorithm description. `BELOW` represents the ordering relation of the specification, and rather than tags, uses the timestamps to decide the ordering of records with identical keys. A lower time is interpreted as an earlier insertion. BELOW is transitive.

$$\texttt{below}\ (a,b) \overset{\text{def}}{=} (a.key < b.key) \vee (a.tag)$$
$$\texttt{BELOW}\ (a,b) \overset{\text{def}}{=} (a.key < b.key) \vee ((a.key = b.key) \wedge (a.time < b.time))$$

The `Compare_and_swap` function maps down the pair of lists, swapping the records (and tagging the right ones) as required so the record on the right is `below` the record on the left for every pair of records. If the lists are of unequal length, the remaining element is moved to the right and tagged. (The λ symbol identifies a local function whose arguments end with a '.'. `Compare_and_swap` is an example of a 'curried' higher order function, where the list pair argument is not shown in the definition.) Notice that the timestamp of records does not affect the behaviour of the function.

```
Compare_and_swap $\overset{\text{def}}{=}$
  Map2_2
    $(\lambda(a,b).(\mathit{if}(\texttt{below}(a,b))\mathit{then}(b,\text{set\_tag}\ a)$
                                  $\mathit{else}\,(a,b)))$
    $(\lambda[x].([],[\text{set\_tag}\,x]))$
```

`Insert` adds a new record on the left, while `Extract` removes the head element on the right list, thus both cause the two lists to shift one position relative to the other, always shifting the left side down relative to the right.

$$\texttt{Insert}\ x(ll,rr) \overset{\text{def}}{=} \texttt{Compare\_and\_swap}\ ((x:ll),rr)$$
$$\texttt{Extract}(ll,(r:rr)) \overset{\text{def}}{=} (r, \texttt{Compare\_and\_swap}\ (ll,rr))$$

The definitions of both operations combine the required shifting of the lists and the application of the `Compare_and_swap` function. `Extract` returns both the extracted record as well as the diminished queue.

5.2. Defining an invariant

The invariant captures many intuitions about the queue operation. These include that all records on the right are ordered, that tagged records on the left are BELOW the next lower record on the right, and that pairs of records at each level are ordered. Additionally, the right side has as many or just one more record than the left side, untagged records in the left must have a later timestamp than every lower record with the same key in either side and every record on the right is tagged. Formulating precise definitions of these properties relies on the `Map` functions described above.

The records on the right are ordered, with the least at the head. The predicate compares successive records on the right by using two copies of the right list, offset by one position. Note the expression $(\lambda x.[])$ is a function which returns the empty list [] when applied to any argument, thus ignoring the extra element in the longer list argument. The predicate `All` holds when applied to a list, all members of which are the value `T`, the HOL constant for *true*. `TL` returns the remainder of a list without the head element.

$$\texttt{rt\_Ordered}\ rr \overset{\text{def}}{=} \texttt{All}\ (\texttt{Map2\_1 BELOW}\ (\lambda x.[])(rr, \texttt{TL}rr))$$

If a record on the left side is tagged, then it is BELOW the record on the right one level deeper in the queue. This together with the previous invariant captures the property referred to in Subsection 4.2 that once a record arrives on the right it is sorted with respect to the keys of other records on the right. The $\Rightarrow$ symbol represents implication.

```
lt_Tagged $(ll,rr) \overset{\text{def}}{=}$
  All (Map2_1 $(\lambda(a,b).\,a.tag \Rightarrow \texttt{BELOW}\ (a,b))$
    $(\lambda x[])(ll, \texttt{TL}\ rr))$
```

Each pair of records at the same depth in the queue is ordered, with the one on the right BELOW the left one.

```
pair_Ordered $(ll,rr) \overset{\text{def}}{=}$
  All (Map2_1 $(\lambda(a,b).\ \texttt{BELOW}\ (b,a))(\lambda x.[])(ll,rr))$
```

Every untagged record on the left that has the same key value as a record located deeper on either side, must have a later timestamp. This uses the Map variant `Map2_1a` , which includes the remaining parts of the lists as arguments to the first function argument. This allows the relation to consider all records deeper in the queue.

```
lt_UnTagged $(ll,rr) \overset{\text{def}}{=}$
  All
    (Map2_1a
    $(\lambda(a,b)(aa,bb).$
    $\neg(a.tag \Rightarrow$
    All  (Map $(\lambda c.(a.key = c.key)$
       $\Rightarrow (c.time < a.time))aa)\wedge$
    All  (Map$(\lambda c.(a.key = c.key)$
       $\Rightarrow (c.time < a.time))bb))$
    $(\lambda x.[])$
    $(ll,rr))$
```

The right side is either the same length as or one longer than the left side. `Len` gives the length of a list.

```
Lengths$(ll,rr) \overset{\text{def}}{=}$
  (Len $ll$ =Len $rr$)$\vee$
  (Len $ll + 1$ =Len $rr$)
```

Lastly, every record on the right is tagged.

$$\texttt{rt\_Tagged}\ rr \overset{\text{def}}{=} \texttt{All}\ (\texttt{Map}\ (\lambda a\,.\,a.tag)rr)$$

The invariant is the conjunction of the six conditions.

```
Invariant $(ll,rr) \overset{\text{def}}{=}$
      rt_Ordered $rr$ $\wedge$ lt_Tagged $(ll,rr)\wedge$
  pair_Ordered $(ll,rr)\wedge$ Lengths $(ll,rr)\wedge$
   lt_UnTagged $(ll,rr)$ $\wedge$ rt_Tagged $rr$
```

Two more predicates are required. The timestamp must reflect the order of insertion of records. Thus the timestamp of an inserted record must be later than that of every record with the same key already in the queue.

Also records are not tagged prior to insertion.

$$\begin{aligned}&\texttt{load\_constraint}\ x(ll, rr) \overset{\text{def}}{=}\\ &\quad \neg(x.tag) \wedge\\ &\quad \texttt{All}\ (\texttt{Map}\ (\lambda a.(x.key = a.key)\\ &\qquad \Rightarrow (a.time < x.time))ll) \wedge\\ &\quad \texttt{All}\ (\texttt{Map}\ (\lambda a.(x.key = a.key)\\ &\qquad \Rightarrow (a.time < x.time))rr)\end{aligned}$$

Finally, the Least predicate expresses that a record is the minimum with respect to the BELOW ordering for all records in the queue.

$$\begin{aligned}&\texttt{Least}\ x(ll, rr) \overset{\text{def}}{=}\\ &\quad \texttt{All}\ (\texttt{Map}\ (\lambda a.\texttt{BELOW}\ (x, a))ll) \wedge\\ &\quad \texttt{All}\ (\texttt{Map}\ (\lambda a.\texttt{BELOW}\ (x, a))rr)\end{aligned}$$

5.3. Results

The correctness result comprises three theorems. (Note that HOL theorems are identified by the ⊢ symbol.) These theorems express that the invariant holds on an empty queue, it is preserved through both queue operations, and it assures the extracted record is the least.

$$\vdash \texttt{Invariant}\ ([], []) \tag{1a}$$

$$\begin{aligned}&\vdash \texttt{load\_constraint}\ x(ll, rr) \Rightarrow\\ &\quad \texttt{Invariant}\ (ll, rr) \Rightarrow\\ &\qquad \texttt{Invariant}\ (\texttt{Insert}\ x(ll, rr))\end{aligned} \tag{1b}$$

$$\begin{aligned}&\vdash \texttt{Invariant}\ (ll, (r : rr)) \Rightarrow\\ &\quad \forall\, removed\ rest.\\ &\quad ((removed\ rest) = \texttt{Extract}\ (ll, (r : rr))) \Rightarrow\\ &\qquad (removed = r) \wedge (\texttt{Least}\ removed(ll, rr)) \wedge\\ &\qquad (\texttt{Invariant}\ rest)\end{aligned} \tag{1c}$$

Theorem (1a) follows immediately from the invariant definition. We prove an intermediate result to assist the proof of the other theorems. This result shows a shift followed by a Compare_and_swap operation maintains the invariant on the rest of the queue.

$$\begin{aligned}&\vdash \texttt{Invariant}\ (ll, (r : rr)) \Rightarrow\\ &\quad \texttt{Invariant}(\texttt{Compare\_and\_swap}\ (ll, rr))\end{aligned} \tag{2}$$

The proof is by successive list inductions, first on *ll* then on *rr*. The two base cases are solved using the Lengths constraint to derive that the queue is empty or has one element. The result follows immediately in both cases. For the case when both *ll* and *rr* are non-empty, the proof can be reduced to five conditions involving the front elements of each list, and the maintenance of the invariant on the Compare_and_swap'ed remainders of the list. The latter is solved by the inductive hypothesis and the fact that if the invariant holds on a queue then it also holds on the remaining queue after the top level is removed. The five conditions arise from the invariant clauses (aside from Lengths, which is satisfied by the invariant on the remaining queue).

- rt_Ordered requires that the new top right record be ordered with respect to the next right record. Both the possible next right records are ordered with respect to the original top right record by the antecedent condition. If no swap occurs it is satisfied. If a swap occurs at the top, the new record is BELOW the one formerly on the right, and the transitivity of BELOW assures the requirement is met.
- lt_Tagged requires that if a swap occurs at the top, the record ending up on the left must be ordered with respect to the record one level lower on the right (if nonempty). This follows from the antecedent condition, by pair_Ordered or rt_Ordered, depending on whether or not the lower records are swapped.
- pair_Ordered requires the top pair to be ordered, and this is satisfied by the definition of Compare_and_swap and the antecedent.
- lt_UnTagged requires that if no swap occurs at the top, then the top left record must have a later timestamp than all the records lower in the queue on either side with the same key. This follows immediately from the same invariant clause in the antecedent of the implication.
- rt_Tagged requires the record ending up on the right to be tagged, and this follows immediately from the definition of Compare_and_swap.

Theorem (1b) is proved by a case split on the structure of *rr*. If it is [], by Lengths so is *ll*, and the result is immediate. In the other case we split the proof into requirements on the heads of the list and the rest, with the latter solved by theorem (2). The argument for the separate requirements parallels those for theorem (2), with the added premises that the new record is not tagged, and has a later timestamp than any record with the same key already in the queue.

The next theorem assures that the top right record is the Least with respect to the BELOW ordering.

$$\vdash \texttt{Invariant}\ (ll, (r : rr)) \Rightarrow \texttt{Least}\ r(ll, rr) \tag{3}$$

Since the right side is ordered, and record pairs at each level are ordered, the result follows by the transitivity of BELOW.

Theorem (1c) combines the results from theorems (2) and (3), and the definition of Extract.

The proofs of all theorems have been completed using the HOL system. The informal proof sketches presented above outline the reasoning behind the mechanical proofs, and reflect the sequence of proof steps (tactics) applied. We submit that the formal proof lends a very high assurance of the validity of the proof. Such assurance cannot replace peer review in evaluating results, but has been effective in discovering omissions, in informal proofs, developed prior to and along with the formal proof. The invariant was strengthened in response to each omission, and the final definition was marked version number five. This demonstrates the practical advantage of machine-checked proof, even when the subject is a relatively simple system.

We observe that the result verifies the abstract

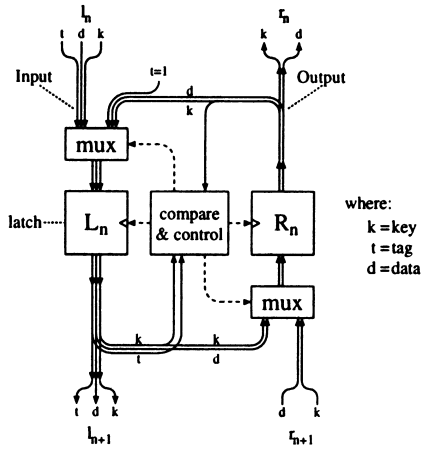

FIGURE 8. Two stage tagged up/down sorting element.

algorithm. The results can be applied to the verification of a concrete design by incorporating limits on the number of records, thus constraining both insert and extract operations, and defining an abstraction from the loaded cells of a queue implementation to the list pair representation. This remains as future work.

6. CLOCKED DIGITAL IMPLEMENTATION OF THE TAGGED UP/DOWN SORTER

A naïve two step clocked digital design is presented in Figure 8. Values are shifted in and compared in the first cycle and if the comparison requires it, a swap is performed on the next cycle. However, this may be reduced to a single cycle process by redirecting the inputs and outputs of the latches rather than swapping the values between the latches (see Figure 9).

6.1. Controlling the single cycle design

The two crossbars are controlled by x which maps A_n to the left and B_n to the right if $x = true$, otherwise A_n maps to the right and B_n to the left. Insertion and extraction are controlled by *insert* and *extract* signals which are

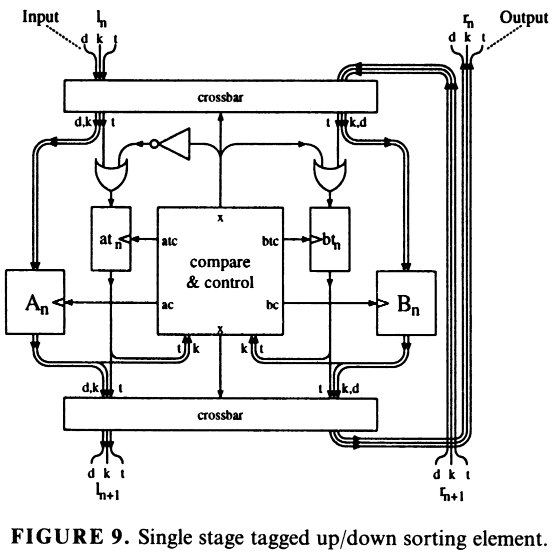

FIGURE 9. Single stage tagged up/down sorting element.

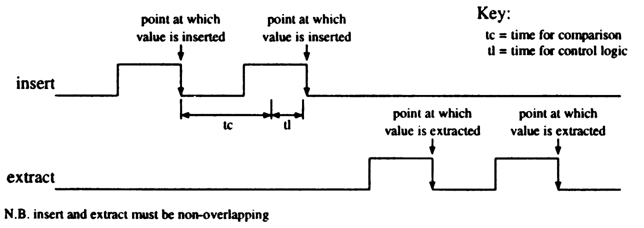
point at which value is inserted
point at which value is inserted
Key:
tc = time for comparison
tl = time for control logic
insert
tc
tl
point at which value is extracted
point at which value is extracted
extract
N.B. insert and extract must be non-overlapping

FIGURE 10. Example timing for the single stage tagged up/down sorter to perform two inserts followed by two extracts.

mutually exclusive. An insert or extract is performed by pulsing the appropriate control line high (see Figure 10) which clocks the required latches for A_n, B_n and x on the falling edge. *ac*, *atc*, *bc* and *btc* control the clocking of the (A_n, at_n) and (B_n, bt_n) latches. If $x = true$ then the left value is in A so *ac* and *atc* will be clocked if *insert* is pulsed. If $x = false$ then the right value is in A so *ac* and *atc* will be clocked if *extract* is pulsed; however, to force tagging of right hand side values, *atc* will also be clocked if *insert* is pulsed and the OR gate arrangement into at_n will set the tag bit. Thus, the tag is set on the following cycle. The corresponding logic is required for B, but with $\neg x$.

The control equations for the one step control logic are defined by (assuming that the flip-flops are negative edge-triggered, i.e. they latch on the falling clock edge):

let $x =$ control for crossbars:
$(l_{out}, r_{out}) = if\ (x)\ then\ (l_{in}, r_{in})\ else\ (r_{in} l_{in})$
$oldx = x$ latched on the falling edge of $insert \vee extract$
$A_n.key =$ key part of record in latch A_n
$A_n.tag =$ tag part of A_n
$B_n.key =$ key part of record in latch B_n
$B_n.tag =$ tag part of B_n
$insert =$ insertion control signal
$extract =$ extraction control signal
N.B. $insert \wedge extract = false$

$$x = ((oldx \wedge (A_n.key = B_n.key)) \vee (A_n.key > B_n.key) \vee (\neg\, oldx \wedge B_n.tag)) \wedge \neg(oldx \wedge A_n.tag) \quad (1)$$

$$ac = (x \wedge insert) \vee (\neg x \wedge extract) \quad (2)$$
$$bc = (\neg x \wedge insert) \vee (x \wedge extract) \quad (3)$$
$$atc = insert \vee (\neg x \wedge extract) \quad (4)$$
$$btc = insert \vee (x \wedge extract) \quad (5)$$

6.2. Discussion of the operation of the single cycle design

First we consider the insert operation. The record to be inserted is presented at l_n (see Figure 9), the *insert* signal is pulsed *true* and *extract* remains *false*. The latch used to hold the record will depend upon the value of x which is determined by the contents of (A_n, at_n), (B_n, bt_n) and *oldx* before the insert takes place. If, for example, we take $x = true$ (so the A_n and at_n latches are holding the left record) and perform an insert (so $extract = false$) then the control equations (2) through (5) become:

$$ac = atc = btc = insert$$
$$bc = false$$

Thus, since $x = true$ the new record (at l_n) will be placed on the inputs of A_n and at_n which will be latched into place on the falling edge of the *insert* signal by *ac* and *atc* (the original record in (A_n, at_n) being propagated to the next sorting element). We can also see that latch B_n is not clocked because *bc* remains *false* but that the tag bit is set by the OR gate arrangement into bt_n and the clocking signal *btc*.

On the falling edge of *insert* the current value of $x = true$ is transferred to the variable *oldx* and the next value of x is calculated from (1):

$$x = ((A_n.k = B_n.k) \vee (A_n.k > B_n.k)) \wedge \neg A_n.t$$

Thus, the crossbar only causes a swap (x goes from *true* to *false*) if $(A_n.k < B_n.k) \vee at_n$ which conforms with the algorithm in section 4.2. Furthermore, it should be noted that if a swap has occurred then (B_n, bt_n) has been remapped from the right output (r_n) to the left output (l_{n+1}) and that this record has been correctly tagged. Likewise, if we had started with $x = false$ then similar conformation would be obtained.

Now consider the extract operation. If we start with $x = false$ (so the A_n and at_n latches are holding the right record and r_{n+1} is the input) then equations (2) through (5) become:

$$ac = atc = extract$$
$$bc = btc = false$$

Thus, r_{n+1} will be latched into A_n, and the tag bit at_n will be set, on the falling edge of *extract*. At this point the value of *oldx* will be set to *false* resulting in the following calculation of the next value for x:

$$x = (A_n.k > B_n.k) \vee \neg B_n.t$$

It can be seen that the conditions for x to change from *false* to *true*, thereby causing a swap, correspond with the specification in Section 4.2 (remembering that $oldx = false$ so B_n was mapped onto the left and A_n onto the right). Furthermore, the value shifted in has correctly had its tag set.

6.3. Implementing the single cycle design

A single cycle implementation has been produced based upon the schematic of Figure 9 and the control equations in Section 6.1. Mentor Graphics' GDT ECAD system was used with ES2's 1 μm two layer metal CMOS technology files.

The size of the key and length of the sorting structure were varied to assess scalability. Results showed that performance remained virtually constant as the length grew, the only difficulty being efficient distribution of *insert* and *extract* signals for very long structures. Performance decreases with key size due to the comparators. However, careful comparator design reduces this to $O(log(key\_size))$. Silicon area grows almost linearly with the length and key size.

A detailed analog simulation was undertaken on a sorter with 8-bits of key and data and a length of 8 (i.e. it can sort up to 16 records). The automatically routed design, using minimum size transistors, consumer a silicon area of 5.7 mm^2 without pads. The cycle time is approximately 10 ns. It is anticipated that a full custom implementation using dynamic logic would reduce the silicon area and improve the cycle time.

7. CONCLUSIONS

We have presented the algorithm, formal proof of correctness and clocked digital implementation for the tagged up/down sorter. The algorithm requires just $\frac{n}{2}$ comparators in order to sort n records. We have fulfilled our objectives of single cycle *insert* and *extract* operations. Furthermore, *extract* always removes the record with the least key, and in the case of repeated keys FIFO ordering is maintained.

A formal verification of the operating properties of the single cell design described in Section 6.1 has been completed but it not presented. Future work will include extending this verification to an n-element sorter implementation. We are also interested in exploring self-timed implementations of the tagged up/down sorter.

ACKNOWLEDGEMENTS

We wish to thank (in alphabetical order) Mr P. V. Byrne, Dr P. Curzon, Dr M. J. C. Gordon, Dr E. A. Hyden, Dr D. R. McAuley, Dr P. Robinson, Professor D. J. Wheeler, Dr W. Wong and other colleagues at the University of Cambridge Computer Laboratory for their support. Many thanks to the referees for their comments. This work was supported in part by the UK EPSRC in the form of a studentship (91307980) and a grant (GR/J 11140).

REFERENCES

[1] Moore, S. W. (1994) *Multithreaded Processor Design.* Technical Report 358, University of Cambridge, Computer Laboratory.

[2] The ATM Forum (1993) *ATM User-Network Interface Specification.* Prentice-Hall, Englewood Cliffs, N.J.

[3] Batcher, K. (1968) Sorting networks and their applications. *AFIPS Spring Joint Comp. Conf.* **32,** 307–314.

[4] Dijkstra, E. W. (1987) A heuristic explanation of Batcher's baffler. *Sci. Comp. Program.* **9,** 213–220.

[5] Lin, Y.-C. (1993) On balancing sorting on a linear array. *IEEE Trans. Parallel Distributed Syst.* **4,** 566–571.

[6] Bitton, D., DeWitt, D. J., Hsiao, D. K. and Menon, J. (1984) A taxonomy of parallel sorting. *Comput. Surv.* **16,** 287–318.

[7] Heinze, G. C., Palmer, R., Dresser, I. and N. Leister (1992) A three chip set for ATM switching. In *Proc. IEEE*

Custom Integrated Circuits Conf. 14.3.1–14.3.4. IEEE Press, New York.

[8] Hsu, T. C. and Kung, L. Y. (1989) A hardware mechanism for priority queue. *Comp. Architecture News*, **17**, 162–169.

[9] Picker, D. and Fellman, R. D. (1995) A VLSI priority queue with inheritance and overwriting. *IEEE Trans. VLSI Syst.* **3**, 245–253.

[10] Cormen, T. H., Leiserson, C. E. and Rivest, R. L. (1991) *Introduction to Algorithms*. MIT Press, Cambridge, MA.

[11] Ahn, B. and Murray, J. M. (1989) A pipelined, expandable VLSI sorting engine implemented in CMOS technology. *IEEE Int. Conf. Circuits Syst.* **3**, 134–137.

[12] Lee, D. T., Chang, H. S. U. and Wong, C. K. (1981) An on-chip compare/steer bubble sorter. *IEEE Trans. Comp.* **C30**, 396–404.

[13] Gordon, M. J. C. and Melham, T. F. (1993) *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic*. Cambridge University Press, Cambridge.

[14] Gordon, M. J. C. (1986). Why higher-order logic is a good formalism for specifying and verifying hardware. In Milne, G. and Subrahmanyam, P. A. (eds), *Formal Aspects of VLSI Design,* p. 153–177. North Holland, Amsterdam.

[15] Graham, B. T. (1992) *The SECD Microprocessor: A verification Case Study*. Kluwer International Series in Engineering and Computer Science, Kluwer, Boston, MA.

[16] Cohn, A. H. (1988) A proof of correctness of the VIPER microprocessor: the first level. In Birtwistle, G. and Subrahmanyam, P. A. (eds), *VLSI Specification, Verification and Synthesis* p. 27–71 Kluwer, Norwell, MA. Also University of Cambridge, Computer Laboratory, Technical Report 104.

[17] Cohn, A. J. (1989) A proof of correctness of the VIPER microprocessors: the second level. In Birtwistle, G. and Subrahmanyam, P. A. (eds), *Trends in Hardware Verification and Automated Theorem Proving*, pp. 1–91. Springer-Verlag, New York.

[18] Melham, T. (1993) *Higher Order Logic and Hardware Verification. Cambridge Tracts in Theoretical Computer Science 31*. Cambridge University Press, Cambridge.

[19] Archer, M., Joyce, J. J., Levitt, K. N. and Windley, P. J. (eds) *Proc. 1991 Int. Workshop on the HOL Theorem Proving System and its Applications*. IEEE Computer Society Press, New York.

[20] Claesen, L. J. M. and Gordon, M. J. C. (eds) (1992) *Higher Order Logic Theorem Proving and its Applications: Proc. IFIP TC10/WG10.2 Workshop*, volume A-20 of *IFIP Trans.* North-Holland/Elsevier, Amsterdam.

[21] Joyce, J. J. and Seger, C.-J. H. (eds) (1994) *Proc. 6th Int. Workshop on Higher Order Logic Theorem Proving and its Applications (HUG '93), LNCS 780*. Springer-Verlag, Berlin.

[22] Melham, T. G. and Camilleri, J. (eds) (1994) *Proc. 7th Int. Workshop on Higher Order Logic Theorem Proving and Its Applications, LNCS 859*. Springer-Verlag, Berlin.