University of Rhode Island
Dept. of Electrical and Computer Engineering
Kelley Hall
4 East Alumni Ave.
Kingston, RI 02881−0805, USA

Technical Report No. 022002−xxxx DRAFT − CONFIDENTIAL

The Partially Time−Tagged Memory System for the Levo High ILP Processor

Augustus K. Uht
Department of Electrical and Computer Engineering
University of Rhode Island

Email: mailto:uht@ele.uri.edu
Web: http://www.ele.uri.edu/~uht

February 27, 2002, Rev. 1

Abstract[1]

Levo is the High ILP Processor project at URI and Northeastern University. To date, IPC's of about 7−8 on average have been demonstrated in initial simulations. The beginning target IPC for Levo is in the 10's. In order to achieve such high performance, a high−bandwidth and low−latency memory system is required. The memory system is further constrained to allow concurrent memory operations whenever possible, to enhance performance, and to maintain the sequential execution consistency of the executing code. This paper proposes such a memory system; this system makes use of time tags to a limited degree. Multiple copies of memory data corresponding to different times in the nominal sequential order are held in the L1 D−Cache. They are used to keep a coherent main memory (L2) state, but are not read by the main part of Levo (the Execution Window, or E−window). Within the E−window, memory system time tags are used extensively. The entire Levo data memory system helps achieve the Levo system goals of high performance, relatively low cost and scalability (cost grows linearly or less with an increasing number of processing elements, e.g., adders).

[1] This work was partially supported by the National Science Foundation through grant MIP−9708183, and by the URI Office of the Provost. This work will be submitted for publication.

1Introduction

<TBD; see Abstract>

2Existing Levo Microarchitecture

See Figure 1 for a current view of the Levo block diagram. Missing is the Memory−Window (M−window) proper. This system has separate L1 caches, a unified L2 cache, a standard Main Memory, and bank interleaving throughout the M−window, and the Memory Forwarding Units (MFU) and memory busses in the Execution Window (E−window).

3Description of the new Partially Time−Tagged Memory System

The Levo block diagram does not change substantially, if at all. However, the microarchitecture and operation of the L1 D−cache and the MFU's change substantially, and this is where we focus our attention. (Note that we do not consider the Instruction Window (I−window) or the I−cache.) The L2 and higher cache and memory systems are unchanged (for now). For the purposes of clarity of discussion, the following description ignores the interleaving. It is to be assumed that it will be in the actual system, however.

3.1*L1 Microarchitecture*

There are two structures within the L1 cache: a standard cache structure of any appropriate type, and a *Time−Tagged Write Buffer (TTWB)*. The buffer has approximately the same number of entries as the E−Window and approximately the same dimensions: n rows by m+1 columns. The purpose of the buffer is to ensure that committed writes to the memory system occur in the proper order. The buffer entries can be set over a large period of time and out−of−order.

The buffer is organized as a row time tag (row−TT) by column time tag (col−TT) array of entries. (Figure to be supplied later; it's just a rectangle with boxes inside.) Col−TT −1 is on the left, and col−TT m−1 is on the right. As in the E−window, the columns are renamed and virtually shift to the left when column 0 instructions are finished executing and are to be committed.

Each entry in the buffer has the following components, corresponding to one memory store (ST) instruction or operation:

address A

datum D

valid bit V

written bit W

Initially the valid and written bits are cleared to 0. When a ST comes in from the L1/L0(MFU) bus (top horizontal busses in the figure), its address and datum are put into the buffer entry with the same time tag as the ST; the entry's valid bit is also set to 1. Once the entry has been written to the memory system, the W bit is set to 1.

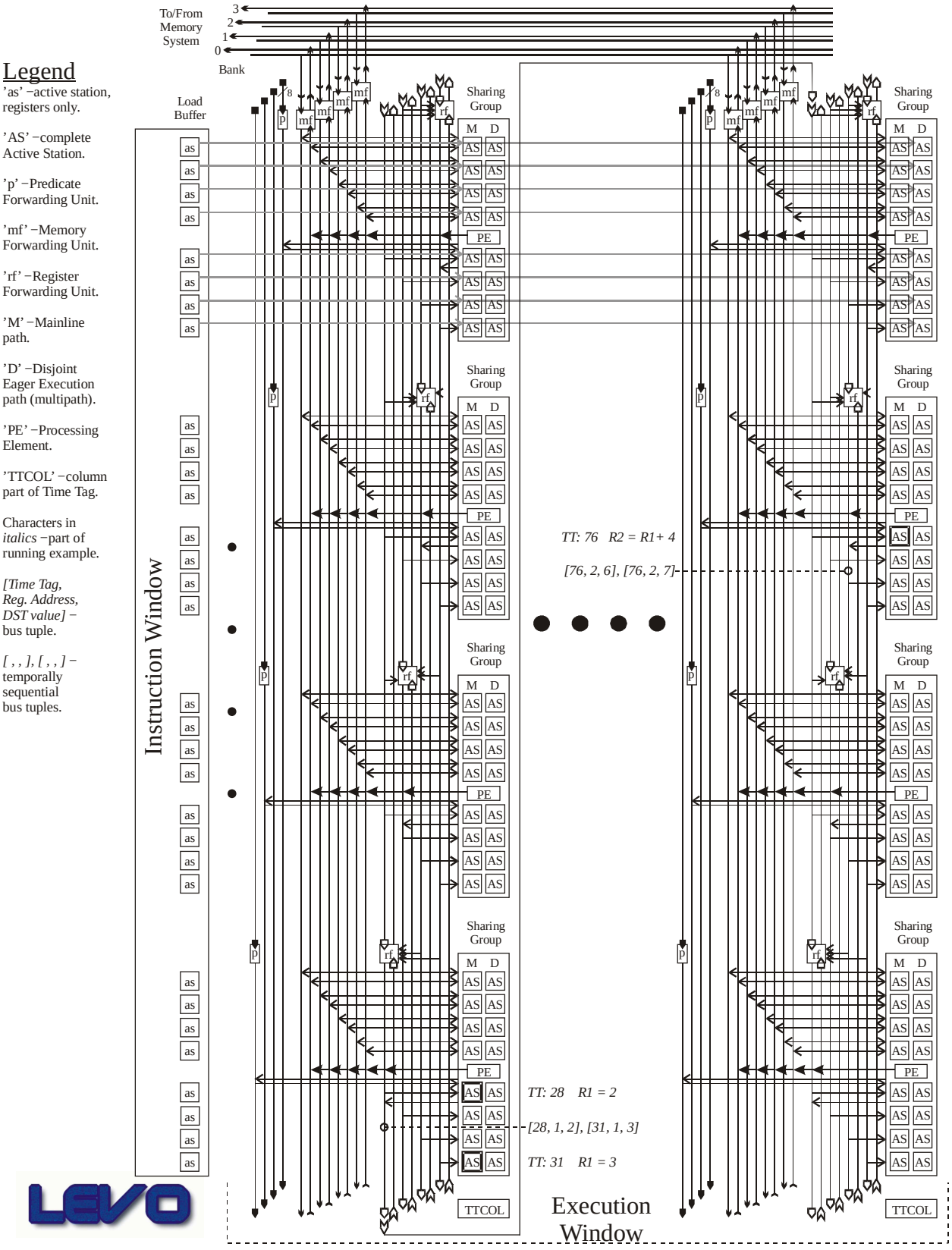

Figure 1. Current Levo block diagram.

3.2 *L1 Operation*

As stated above, when the buffer snoops a store on the L1/L0 bus, the store datum and address are placed into the buffer entry corresponding to the store's time tag. Lastly, the entry valid bit is set to 1 and the written bit is set to 0. Note that columns (or rows) can be snarfed in any order.

Each entry in the −1 col−TT column is written into the (std. L1) cache if the entry's W bit is 0 and its V bit is 1. After the write, the entry's W bit is set to 1 and its V bit is cleared to 0. The datum may also be written to main memory (L1+1) at the same time, especially if the write is a miss to the cache. The usual allocation policies for the cache can be used.

All memory loads (LD) are satisfied by the cache, not the buffer. This way loads to L1 always get the main memory (time −1) version of the datum. If there is a read miss, as usual a cache line will be replaced with the proper data from L1+1, and the load will be satisfied; that is, the datum, its address and time tag will be broadcast on the L1/L0 bus.

Note that the equivalent of a stream buffer may be necessary between the TTWB and the cache, so that the TTWB can unload its data quickly, and let the cache updates happen in the background.

We must also be concerned with speculative entries in the TTWB. Actually, every entry in the TTWB is speculative except for those in column −1. If a store operation is squashed in the E− window, as usual a nullify message is broadcast on the L1/L0 bus; this message contains the squashed datum's address and time tag. When the nullify message reaches the TTWB, the corresponding entry has its valid bit cleared to 0; this ensures that incorrect data will not pollute the cache or higher levels of the memory system.

Comments

Entries in the TTWB can be overwritten an unlimited number of times with other stores having the same time tag, with or without the same address. (The former is typically the case when a store's datum is recomputed. The latter is typically the case when a store is squashed due to either branch mispredictions or an address recomputation; normally this would be preceded by a nullify message.)

To emphasize: loads are satisfied only by the cache. Stores are handled by the TTWB and then possibly the cache, depending on the write policy.

To emphasize: there is NO necessary system−wide stall upon a read miss; except for the corresponding load instruction and its flow dependents, the rest of the system computes on, hopefully hiding the miss latency. Note that even the load may not be stalled, if value prediction is used (as Levo likely will do).

3.3 *MFU (L0)Microarchitecture (and Some Operation)*

Before proceeding with the L0's description, a brief review and clarification of the basic operation of memory instructions in the E−window is in order.

First, an executing store (address has already been calculated) broadcasts its datum, address and time tag on its column memory bus (vertical, in the figure). Loads with computed addresses snoop and snarf the store's datum and time tag in the usual data forwarding fashion.

If a load is ready to execute, it performs a backwarding operation on the memory bus, requesting its datum from temporally earlier stores, its earlier L0 cache, or other earlier location. When a location (store or cache) matches the address, it satisfies the load request with a standard store forwarding operation, and the load gets its datum.

Complications: Store data must be logically broadcast to all later instructions in the E−window. Similarly, loads must be able to go all the way back to main memory, if intermediate locations do not contain its datum.

While a store is normally thought of as traveling forward, to the end of the bus, it is also traveling (electrically) backwards to the other end of the bus, and hence to the L0 cache on its column. In order to keep propagation delays and system complexity down, the L0 cache at the top of the column passes on the store to the L1/L0 bus and broadcasts it on same. All other columns' L0 caches see this store, and will snarf the store if their column time tag is greater than that of the store; a snarf register is used in the L0 caches to keep the latest store datum with that address. In this way, the store datum finds its way to the next later column's L0 cache, assuming it meets the constraints above. Therefore, for any given address in any previous column, the later column holds the latest version (largest time tag) of the datum for that address.

Note that there is now no reason for a column memory bus to physically go from the bottom of the column to the top of the next column. The path is already there.

Once a column's L0 receives a store datum, there are a couple of possible actions. For now, in order to keep bus traffic down, we assume that the store is NOT broadcast on the later column's memory bus; it waits for a load request before broadcasting.

Every L0 cache is guaranteed to save and hold (not replace, except by overwriting with the same address), all stores from its immediately preceding cache (column). In this way, it is guaranteed that at least one L0 cache on the L1/L0 bus will hold an executed store. Note that if an L0 cache further away has room or the desire, it can also snarf the store. (The exact policy to use here will be a matter of system tuning.)

WRT a load request, if it is not satisfied by an earlier active station store in its column, and not by the L0 cache at the top of the column, then the load request is broadcast on the L1/L0 bus. All other L0 caches with earlier column time tags may satisfy the load request, as is done within a column. If no L0 cache has the datum, the request is satisfied by the L1 cache, which is temporally the earliest cache on the L1/L0 bus. Of course, the request to the L1 cache may miss, and hence go on to a higher level in the memory hierarchy, and so on. Once it finds its datum, and works its way back to the L1 cache, the datum is packaged as a special store operation and broadcast on the L1/L0 bus. The original load's L0 cache snarfs the store (as may any other L0 cache), sees that it is a special store, and thus broadcasts it on its column bus (forced broadcast). The load then snarfs the datum.

OK, back to the L0 microarchitecture. Effectively the above gives some of the requirements of an L0 cache. We'll now say how they are met.

Similarly to the L1 cache, there are two structures in an L0 cache: a mainly standard cache, and the MFU (L0) Previous Neighbor Buffer (MPNB). The MPNB is the entity that holds the latest stores from the immediately prior column until the MPNB column is committed. There are n entries in the buffer, one per possible store in the previous column. Each entry acts somewhat like a regular Active Station: it only holds the store datum with the latest time tag for any given address. That is, there is only one version of any address in the buffer. When an entry in the buffer is written or rewritten, it is also written into the other MFU structure, the cache; it is also marked as permanent in the cache and may not be replaced. Most reads to an L0 are satisfied by its cache. Any cache entry can be replaced, as needed, except for the buffer entries. (If the datum is needed again, it will just be a miss on the cache and be satisfied from earlier in the hierarchy.) The cache is written by the buffer and by all stores and virtual stores (from nullifies) from the L1/L0 bus. Each cache entry also contains a snarf/time tag field. The entry is overwritten by a store with the same address only if the time tag of the store is equal or greater than that in the snarf field; if that is the case, the snarf field is updated with the store's time tag.

For a typical cache, say direct−mapped to make the discussion easy, it is always possible for more than one buffer entry to map into the same cache entry. To handle such a case we assign each cache entry an M or "multi" bit. It is set when a second buffer entry tries to be in another buffer entry's cache location. Since the buffer entries are permanent, we can't replace the one that's already there (or rather, they both can't be there at the same time). A load request mapping to that cache location will check both the address, as usual, and the M bit. If the addresses match, the request is satisfied and that's all that needs to be done. If the addresses do not match, and the M bit is set, the load does an associative lookup on the buffer to see if its datum is there. (We try to avoid the associative lookup for speed reasons.)

Each cache entry also has a "load satisfied" or LS bit. As the name implies, when the entry is used to satisfy a following load, the entry's LS bit is set. If the entry is overwritten by a store with the same address, the entry's LS is examined: if cleared, nothing else is done, and the store waits for a load request. If the LS bit is set, the load now potentially has the wrong data, so the new store MUST broadcast regardless of whether a load request is received or not.

Note on virtual stores: The nullify command goes logically backwards and forces the previous store with the same address to re−broadcast; this in turn causes dependent loads to potentially re−execute (if the store data has changed). Note that in the worst case the nullify command will have to go all the way back to main memory to get the previous value. A nullify command also acts as a store with valid address and time tag, but unspecified data, hence a "virtual store". When broadcast on the L1/L0 bus, it invalidates any entry it has in the L1 buffer, as well as later loads with the same address.

(In a way, a nullify is also a virtual load, since it asks for data but doesn't snarf it. The virtual load also causes real store operations to go forward through the system.)

The above algorithms can likely be improved upon or at least tuned.

4 Other Thoughts

It may well be that the register filter units (RFU) could be combined with the MFU's, giving us a Unified Filtering Unit. Including Predicate Forwarding Units may also be possible. This would simplify the design, require less hardware, and be easier to implement and realize (including layout).

It is suggested that at least for the memory system, the forwarding and backwarding busses be replaced by common bi−directional busses. This may also help with hardware utilization, as well as simplify the layout, possibly reducing the size of the die.

The block diagram shown earlier should in no way be taken as a prototype of the physical layout. It is likely that an H−tree organization (possible at least with the memory system) can be used to dramatically decrease bus lengths, hence capacitance, and increase circuit speed, helping to keep the cycle time low.

5 Summary

A possibly novel (I haven't checked the literature yet) memory system for high−ILP machines has been proposed. Time tags are used throughout the lower levels of the memory system to enhance the parallelism in the machine while maintaining necessary dependencies.