# The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor

Daniel Leibholz and Rahul Razdan
Digital Equipment Corporation
Hudson, MA 01749

## Abstract

*This paper describes the internal organization of the 21264, a 500 MHz, Out-Of-Order, quad-fetch, six-way issue microprocessor. The aggressive cycle-time of the 21264 in combination with many architectural innovations, such as out-of-order and speculative execution, enable this microprocessor to deliver an estimated 30 SpecInt95 and 50 SpecFp95 performance. In addition, the 21264 can sustain 5+ Gigabytes/sec of bandwidth to an L2 cache and 3+ Gigabytes/sec to memory for high performance on memory-intensive applications.*

## Introduction

The 21264 is the third generation of Alpha microprocessors designed and built by Digital Semiconductor. Like its predecessors , the 21064 [1] and the 21164 [2], the design objective of the 21264 team was to build a world-class microprocessor which is the undisputed performance leader. The principle levers used to achieve this objective were:

- A cycle time (2.0 ns in 0.35 micron CMOS at 2 volts) was chosen by evaluation of the circuit loops which provide the most performance leverage. For example, an integer add and result bypass (to the next integer operation) is critical to the performance of most integer programs and is therefore a determining factor in choosing the cycle time.
- An out-of-order, superscalar execution core was built to increase the average instructions executed per cycle (ipc) for the machine. The out-of-order execution model dynamically finds instruction-level-parallelism in the program and hides memory latency by executing load instructions that may be located past conditional branches.
- Performance-focused instructions were added to the Alpha architecture and implemented in the 21264. These include:
  - ⇒ Motion estimation instructions accelerate CPU-intensive video compression and decompression algorithms.
  - ⇒ Prefetch instructions enable software control of the data caches.
  - ⇒ Floating point square root and bidirectional register file transfer instructions (integer-to-floating point) enhance floating point performance.
- High-speed interfaces to the backup (L2) cache and system memory dramatically increase the bandwidth available from each of these sources.

The combination of these techniques delivers an estimated 30 SpecInt95, and over 50 SpecFp95 performance on the standard SPEC95 benchmark suite and over 1600 MB/s on the McCalpin STREAM benchmark. In addition, the dramatic rise in external

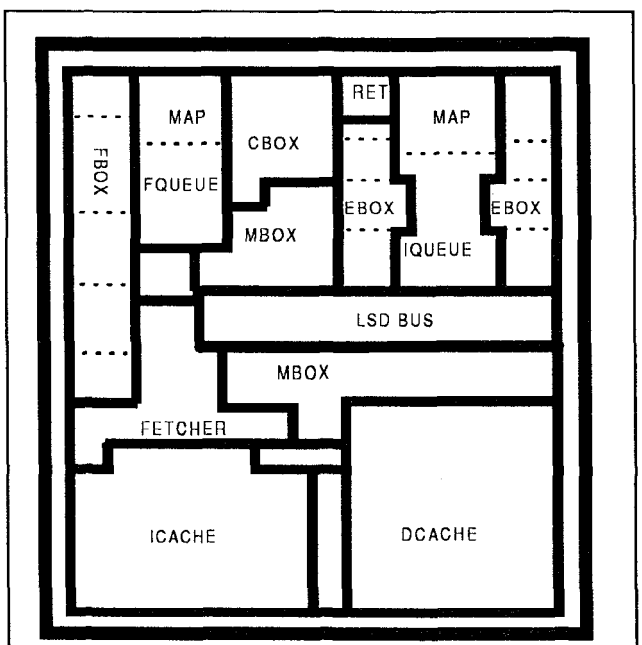


Figure 1. 21264 Floorplan

pinbus bandwidth allows the 21264 to perform well in large applications whose performance is dominated by system memory bandwidth.

## Processor Overview

Figure 1 shows a floorplan of the 21264. The die is approximately 3 $cm^2$ in 0.35-micron, six-layer-metal CMOS and contains 15 million transistors. The 21264 is comprised of ten major sections. These are the fetch unit (*Fetcher*), the register mapper (*Mapper*) and issue unit (*Issuer*), the integer execution unit (*Ebox*), the floating-point execution unit (*Fbox*), the instruction retire unit (*Retirator*), the memory unit (*Mbox*), the cache and system control unit (*Cbox*), and finally the on-chip instruction and data caches (*Icache* and *Dcache* respectively). Instructions are processed by the CPU as follows:

- The *Fetcher* fetches four instructions per cycle from the Icache based on a predicted program counter (PPC). The PPC is generated by a combination of cache line, set, and branch prediction algorithms described below.
- The *Mapper* implements a register-renaming scheme to dynamically map the architected register specifiers into a larger pool of physical registers. The renaming eliminates write-after-write and write-after-read dependencies and allocates physical space for uncommitted results. The mapped instructions are loaded into separate integer and floating point instruction queues.
- The integer *Issuer* selects four data-ready instructions from among twenty queue locations and issues them to the execution units. Likewise, the floating point *Issuer* selects two data-ready instructions from among fifteen queue locations and issues them to the floating point execution units.
- The register file read occurs in stage 4 of the pipeline and execution in the *Ebox* and *Fbox* begins in stage 5. For single-cycle operations, such as integer add and shift, results may be available for execution via bypass busses in the next cycle. For multiple-cycle operations, execution proceeds to completion and results are bypassed around the register files.
- Memory operations are performed by the *Ebox* and *Mbox*. For load instructions, the effective virtual address is calculated in the *Ebox* in stage 5 and the *Dcache* is read in stage 6. Data is broadcast to the functional units in stage 7 and is available for execution by a consumer instruction in stage 8. Store instructions broadcast data from the register file to the *Mbox* in stage 7 where it is locally queued until the store instruction is committed. This data can be supplied to load instructions (bypassed around the Dcache) in advance of the store commitment.
- Memory references which miss the internal caches and data evicted from the *Dcache* are forwarded to the *Cbox*, which controls a second-level board cache and the system interface. The *Cbox* handles all off-chip transactions and implements several coherency protocols.
- Instructions are committed by the *Retirator* when they, and all previously fetched instructions, are past the point at which exceptions or mispredicts are detected. They can be retired in advance of results being calculated or fetched from memory. As they are committed, the *Retirator* frees registers that are no longer needed.

The following sections describe each of the functional units within the 21264 in greater detail.

## Fetcher

The *Fetcher* retrieves a 16-byte-aligned group of four 32-bit instructions each cycle from a 64KB, 2-way associative *Icache*. The fetcher uses a variety of branch and jump prediction techniques to generate a predicted path of execution for the pipeline. In the event of an exception or mispredict, the fetcher redirects the execution path and restarts the pipeline.

The 21264 employs a hierarchical, *tournament* branch prediction algorithm [3] in which two predictions, *Local* and *Global*, are generated and maintained in parallel and the prediction to use for a particular instance of a branch is dynamically chosen between the two based on a *Choice* predictor (figure 2). In all, about 36K bits of branch history information is stored, resulting in mispredict rates of about 7-10 mispredicts per one thousand instructions on the SPECInt95 benchmarks.

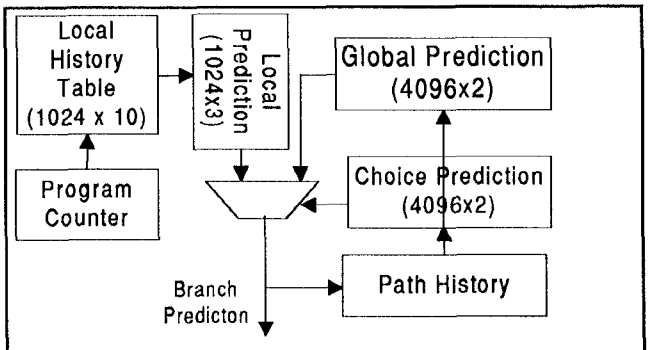


Figure 2. Local, Global, and Choice Predictors

*Local Branch Predictor*

The local branch predictor bases predictions on the past behavior of the specific branch instruction being fetched. It maintains a PC-indexed history table of branch patterns which, in turn, index a table of prediction counters, which supply the prediction as the MSB of the referenced counter. The history table records the last 10 taken/not-taken branch decisions for 1K branches (indexed by 10 bits of the program counter). As branch history is accumulated, a given history table entry may, in succession, index a different prediction counter. For example, a branch that is taken on every third iteration will generate, in succession, taken/not-taken patterns of 0010010010, 0100100100, and 1001001001 (assume "taken" is denoted as "1" and "not-taken" as "0"); treating these patterns as integers, they will reference the 142d, 292d, and 585d saturating counters in succession, providing the appropriate predictions on each iteration to sustain the pattern. When the branch is issued, resolved, and committed, the history table is updated with the true branch direction and the referenced counter is incremented or decremented (using saturating addition) in the direction which reinforces the prediction.

*Global Branch Predictor*

The global branch predictor bases its prediction on the behavior of the branches that have been fetched prior to the current branch. Consider the following code sequence:

```
loop:
        //modify and b
        if (a == 100) {...}      //1
        if (b % 10 == 0) {...}   //2
        if (a % b == 0) {...}    //3
        branch to loop
```

Prediction based on program flow would conclude that if the first two branches were taken then the third branch should be predicted-taken.

The predictor maintains a silo of 13-bit vectors of branch predictions and indexes a table of prediction counters which supply the prediction in the MSB of the addressed counter. If a mispredict occurs, the branch prediction vector is backed up and corrected.

*Choice Predictor*

The choice of global-versus-local branch prediction is made dynamically on a path-based predictor which decides which predictor to use based on the past correctness of choice. It is trained to select the global predictor when global prediction was correct and local prediction was incorrect.

*Prefetching and Bubble Squashing*

In order to sustain a high fetch bandwidth, the 21264 minimizes fetch penalties due to taken branches or jumps by loading the predicted target address into the Icache along with the instructions. Accompanying each group of four instructions in the Icache is a pointer to the predicted next cache line and set; this is used to index the Icache in the next cycle. The address predictor is trained by the branch predictor to predict branch targets and is trained by the Ibox to recognize subroutine call sites. In addition, the fetcher prefetches up to four cache blocks from the second-level cache and system as it fetches down a predicted path.

## Register Mapper

Each of the four fetched instructions has (up to) two source register specifiers and one destination register specifier. Consider the following group of instructions:

```
(0) ADD r0, r1 ⇒ r2
(1) SUB r4, r4 ⇒ r1
(2) ADD r2, r2 ⇒ r3
(3) MUL r0, r0 ⇒ r2
```

If write-after-write dependencies are eliminated, (0) and (3) can be executed out-of-order. By eliminating write-after-read dependencies, (0) and (1) can be executed out-of-order; and (2) and (3) can be executed out-of-order. (2) must obey a read-after-write dependency and be executed after (0). Register renaming, assigning a unique storage location for uncommitted state associated with each write-reference of a register specifier, has an implementation advantage in that a specific physical register file entry can hold the results of an instruction before and after the instruction is committed. Once the result of an instruction is calculated and written, it does not have to be transferred to an architecture file when it is committed.

The register mapper is comprised of two identical structures: one maps integer register specifiers and the other maps floating point specifiers. Each mapper has three functional components: a content-associative-memory (CAM) array, a map silo, and a free register generator. The CAMs determine which physical registers were most recently allocated to the source and destination register specifiers of the four instructions to be mapped. The free register generator picks four unallocated physical registers to be assigned to the destination registers. The CAM outputs and intra-instruction information (such as

producer/consumer relationships within the four instructions) are combined to assign either previously allocated registers or the registers supplied by the free register generator to the source specifiers.

The resulting four register maps are saved in the map silo, which, in turn, provides information to the free register generator as to which registers are currently allocated. Finally, the last map that was created is used as the initial map for the next cycle. In the event of a branch mispredict or trap, the CAM is restored with the map silo entry associated with the redirecting instruction.

## Issue Queue

Each cycle, up to four instructions are loaded into the two issue queues. The floating point queue (Fqueue) chooses two instructions from a 15-entry window and issues them to the two floating point pipelines. The integer queue (Iqueue) chooses four instructions from a 20-entry window and issues them to the four integer pipelines (figure 3).

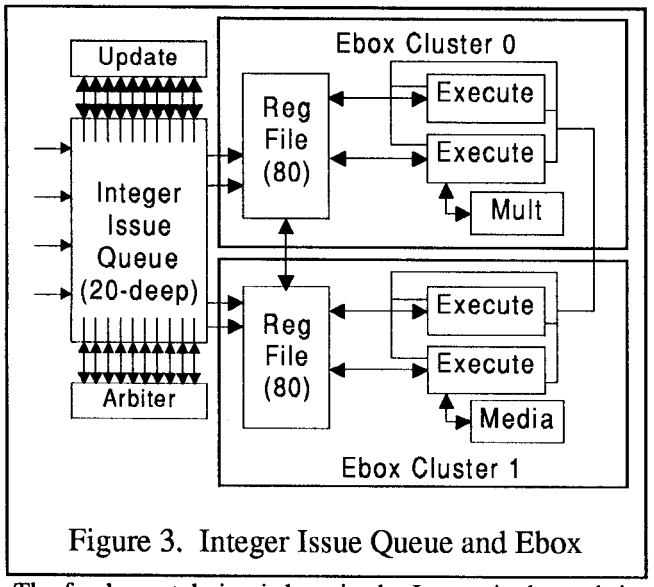


Figure 3. Integer Issue Queue and Ebox

The fundamental circuit loop in the Iqueue is the path in which a single-cycle producing instruction is *granted* (issued) at the end of one issue cycle and a consuming instruction requests to be issued at the beginning of the next cycle (e.g. Instructions (0) and (2) in the mapper example). The *grant* must be communicated to all newer consumer instructions in the issue window.

The issue queue maintains a register scoreboard, based on physical register number, and tracks the progress of multiple-cycle (e.g. integer multiply) and variable cycle (e.g. memory load) instructions. When arithmetic result data or load data is available for bypass, the scoreboard unit notifies all instructions in the queue.

The queue arbitration cycle works as follows:

1. New instructions are loaded into the "top" of the queue.
2. Register scoreboard information is communicated to all queue entries.
3. Instructions that are data-ready request for issue
4. A set of issue arbiters search the queue from "bottom" to "top", selecting instructions that are data-ready in an age-prioritized order and skipping over instructions that are not data-ready.
5. Selected instructions are broadcast to the functional units.

In the next cycle a queue-update mechanism calculates which queue entries are available for future instructions and squashes issued instructions out of the queue. Instructions that are still resident in the queue shift towards the bottom.

## Ebox and Fbox

The Ebox functional unit organization was designed around a fast execute-bypass cycle. In order to reduce the impact of the large number of register ports required for a quad-issue CPU and to limit the effect on cycle time of long bypass busses between the functional units, the Ebox was organized around two *clusters* (see figure 3). Each cluster contains two functional units, an 80-entry register file, and result busses to/from the other cluster. The lower two functional units contain one-cycle adders and logical units; the upper two contain adders, logic units, and shifters. One upper functional unit contains a 7-cycle, fully pipelined multiplier; the other contains a 3-cycle motion video pipeline, which implements motion estimation, threshold, and pixel compaction/expansion functions. The two integer unit clusters have equal capability to execute most instructions (integer multiply, motion video, and some special-purpose instructions can only be executed in one cluster).

The execute pipeline operation proceeds as follows:

- Stage 3: Instructions are issued to both clusters.
- Stage 4: Register files are read.
- Stage 5: Execution (may be multiple cycles)
- Stage 6: Results are written to the register file of the cluster in which execution is performed and are bypassed into the next execution stage within the cluster.
- Stage 7: Results are written to the cross-cluster register file and are bypassed into the next execution stage in the other cluster.

The one-cycle cross-cluster bypass delay resulted in a negligible performance penalty (about 1% on SPECInt95) but reduced the operand bypass bus length by 75%.

*Floating Point Execution Unit*

The floating point pipe execution units are organized around a single 72-entry register file. One unit contains a 4-cycle fully pipelined adder and the other contains a 4-cycle multiplier. In addition, the adder pipeline contains a square-root and divide unit. The Fbox pipeline operation is similar to the Ebox pipeline except the execution stage is elongated and there is only one cluster.

*Memory Operations*

The lower two integer functional unit adders are shared between ADD/SUB instructions and effective virtual address calculations (register + displacement) for load and store instructions. Loads are processed as follows:

- Stage 3: Up to two load instructions are issued, potentially out-of-order.
- Stage 4 and 5: Register file read and displacement address calculation.
- Stage 6A and 6B: The 64KB 2-way, virtually indexed, physically tagged data cache is accessed. The cache is phase-pipelined such that one index is supplied every 1 ns (assuming a 2ns cycle time). Most dual-ported caches impose constraints on the indices that are supplied each cycle to avoid bank conflicts. Phase-pipelining the cache avoids these constraints.
- Stage 7: A 128-bit load/store data bus (the LSD bus) is driven from the cache to the execution units. The cache data reaches both integer unit subclusters at the same time -- consuming instructions can issue to any functional unit 3 cycles after the load is issued. Cache data takes an additional cycle to reach the floating point execution unit.

## Mbox

The memory instruction pipeline discussed above is optimized for loads/stores which hit in the Dcache and do not cause any address reference order hazards. The Mbox detects and resolves these hazards and processes Dcache misses.

*Hazard Detection and Resolution*

As discussed earlier, out-of-order issued instructions can generate three type of hazards read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW). Register renaming resolves WAW and WAR for references to the architectural register specifiers, and the Ibox queue resolves the RAW dependencies. The Mbox must detect and resolve these hazards as they apply to references to memory. Consider the following series of memory instructions which reference address (A):

```
(0) LD  Memory(A) ⇒ R1
(1) ST  R2 ⇒ Memory(A)
(2) LD  Memory(A) ⇒ R3
(3) ST  R4 ⇒ Memory(A)
```

Assume that address (A) is cached in the Dcache. If (0) and (1) issue out-of-order from the Iqueue, R1 will incorrectly receive the result of the store. If (1) and (2) are issued out-of-order, R3 will incorrectly receive the value before the store, and, finally, if (1) and (3) issue and complete out-of-order, the value stored to location (A) will be R2 instead of R4.

The datapath which the Mbox uses to resolve these hazards is shown in figure 4. Since loads and stores can dual-issue, the data-path receives two effective addresses

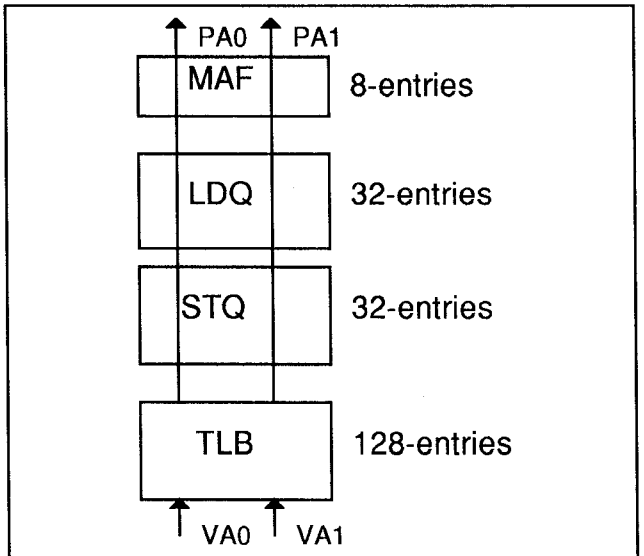


Figure 4. Mbox Address Datapath

per cycle (VA0 and VA1) from the Ebox adders. It first translates them to physical addresses (PA0 and PA1) using a dual-ported, 128-entry, fully associative translation lookaside buffer (TLB). The physical addresses travel over the three key structures in the Mbox: the LoaD Queue (LDQ), the STore Queue (STQ), and the Miss Address File (MAF).

The 32-entry LDQ contains all the in-flight load instructions, and the 32-entry STQ contains all the in-flight store instructions. The MAF contains all the in-flight cache transactions which are pending to the backup

cache and system. Each entry in the MAF refers to a 64-byte block of data which is ultimately bound for the Dcache or the Icache.

The instruction processing pipeline described above extends to the Mbox as follows:

- Stage 6: The Dcache tags and TLB are read. Dcache hit is calculated.
- Stage 7: The physical addresses generated from the TLB (PA0, PA1) are CAMed across the LDQ, STQ, and MAF.
- Stage 8: Load instructions are written into the LDQ; store instructions are written into the STQ, and, if the memory reference missed the Dcache, it is written into the MAF. In parallel, the CAM results from the preceding cycle are combined with relative instruction age information to detect hazards. In addition, the MAF uses the result of its CAMs to detect the loads and stores which can be merged into the same 64-byte cache block.
- Stage 9: The MAF entry allocation in stage 8 is validated to the system interface, and the MAF number associated with this particular memory miss is written into the appropriate (LDQ/STQ) structure. This MAF number provides a mapping between the merged references to the same cache block and individual outstanding load and store instructions.

Given these resources and within the context of the Mbox pipeline, memory hazards are solved as follows. RAW hazards are discovered when an issued store detects that a younger load to the same address has already issued and delivered its data. In this event, the CPU is trapped to the store instruction, and instruction flow is replayed by the Ibox. This is a potentially common hazard, so, in addition to trapping, the Ibox is trained to issue that load in-order with respect to the prior store instruction.

WAR hazards are discovered when an issued load detects an older store which references the same address; the CPU is trapped to the load address. Finally, WAW hazards are avoided by forcing the STQ to write data to the Dcache in-order. Thus, stores can be issued out-of-order and removed from the Iqueue, allowing further instruction processing, but the store data is written to the Dcache in program order.

*Dcache Miss Processing*

If a memory reference misses the Dcache and is not trapped or merged in stage 8 of the Mbox pipeline, a new MAF entry is generated for the reference. The Cbox finds the block in the L2 cache or main memory, and delivers the data to the Mbox. As the data is delivered, it must be spliced into the execution pipeline so that dependent instructions can be issued. The fill pipeline proceeds from the Cbox as follows:

1. The Cbox informs the Mbox and the rest of the chip that fill data will be available the Load Store Bus (LSD) in 6 cycles.
2. The Mbox receives the fill command plus the MAF number and CAMs the MAF number across the LDQ. The loads which referenced this cache block arbitrate for the two load/store pipes.
3. The Ibox stops issuing load instructions for one cycle because the LSD bus has been scheduled for fill data.
4. Bubble Cycle.
5. Fill data arrives at the 21264 pins. The Ibox ceases issuing loads for one cycle because the Dcache has been scheduled for a fill update.
6. The Ibox issues instructions which were dependent on the load data because the data will be available for bypass on the LSD bus.
7. The fill data is driven on the LSD bus and is ready to be consumed.
8. The fill data and tag are written into the Dcache.

Since the cache block is 64-bytes, and the 21264 has two 64-bit fill pipelines, it takes 4 transactions across these pipelines to complete a valid fill. The tag array is written in the first cycle so newly issued loads can "hit" in the Dcache on partially filled cache blocks. After the cache block is written, the MAF number is CAMed across both the LDQ and STQ. In the case of the LDQ, this CAM indicates all the loads which wanted this cache block but were not satisfied on the original fill. These loads are placed in "retry" state and use the Mbox retry pipeline to resolve themselves. In the case of the STQ, all stores which wanted this block are informed that the block is now in the Dcache and in the appropriate cache state to be writeable. When these stores are retired, they can be written into the Dcache in-order. In some cases, the Cbox delivers a cache block which can be consumed by loads but is not writeable. In these cases, all the stores in the STQ are also placed in the "retry" state for further processing by the Mbox retry pipeline. The Mbox retry pipeline starts in the first cycle of the fill pipeline, and similar to a fill, reissues loads and stores into the execution pipeline.

## Cbox

The Cache Box (Cbox) controls the cache subsystem within the 21264 microprocessor and has two primary tasks. First, it cooperates with the external system to

maintain cache coherence across multiple processors. Second, it schedules data movement of cache blocks to optimize overall performance on its two major interfaces, a 128-bit data bus to the board level cache and a 64-bit data bus to the system interface. These major busses and their associated address busses are shown in figure 5.

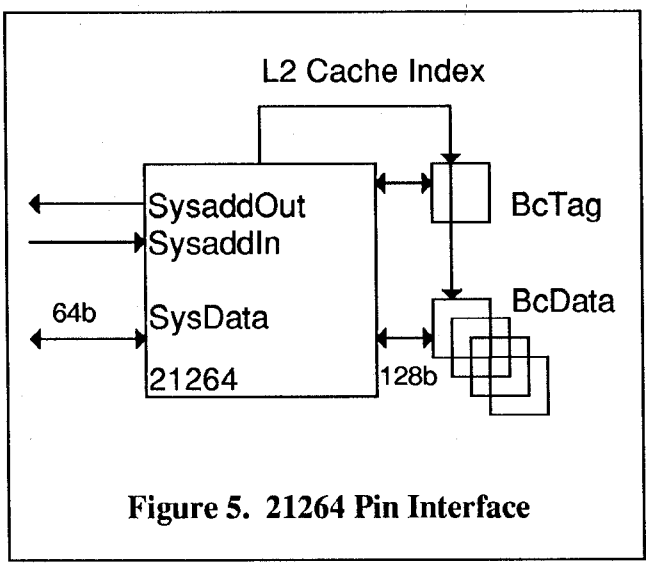


**Figure 5. 21264 Pin Interface**

*L2 Cache Interface*

The 21264 Cbox can support board level cache sizes ranging from 1MB to 16 MB with frequencies ranging from 1.5X to 8X the processor frequency. Thus, system designers can build a cache system optimized for performance by using 200 MHz Late-Write Synchronous SRAM parts or optimized for cost, by using slower PC-style SRAMs. In addition the 21264 supports dual-data 333 MHz. With these components, the 21264 can reach over 5 Gigabytes/sec of bandwidth to its L2 cache.

The pipeline to the L2 cache extends the CPU pipeline as follows:

- Stage 8: The Mbox delivers the address to the Cbox
- Stage 9: The address is being driven out on the pins on the L2 cache Index wires.
- Stage 10+: A programmable number of cycles later (based on L2 cache latency), the data is delivered from the L2 cache. Seven CPU cycles before the data is on the LSD bus, the Cbox informs the Mbox of the fill using the fill pipeline described earlier.

When the fill data is on the LSD bus, the result of the L2 cache tag comparison is known, and L2 cache hit is generated. If the block hits, the Valid status bit to be written into the Dcache tag array is asserted.

In the next cycle, the Dcache tag array is written with the appropriate status, and the Dcache data arrays are written with the fill data. In parallel, the victim data for the fill is extracted from the Dcache to the Victim Data File (VDF), and the fill data is also written to another entry in the VDF. If data extracted from the Dcache was dirty or recently modified, it must be written back to the L2 cache. If the data read from the L2 cache was dirty and missed, it must be victimized out to main memory.

In the case of a L2 cache miss, the transaction is queued in the MAF for the system interface along with any associated victimization commands.

*System Interface*

The 21264 Cbox can support system interface bandwidth from 1.5X to 8X the processor cycle time. On arrival of fill data, the Cbox accumulates two data words on the pads before streaming the data onto the LSD bus in the processor core using the same fill pipeline as is used for L2 cache processing. In addition, the system interface can fill the cache block in any cache state, and even cancel the fill before it completes.

Victim processing is under the control of the external system. Some systems choose to accumulate VDF transactions to minimize turnaround bubbles on the system data bus. Other systems map cache blocks to the same DRAM bank so that VDF writes are always bank hits.

*Cache Coherence Protocol*

The Cbox uses a write-invalidate cache coherence protocol to support a shared memory multiprocessing. This protocol uses three bits to indicate the status of a cache block -- Valid(V), Shared(S), and Dirty(D). The cache block states generated by these bits (VSD) are:

- Not valid. (INVALID)
- Exclusive clean (CLEAN)
- Exclusive dirty (DIRTY)
- Shared (SHARED)
- Shared dirty (DSHARED)

Loads in any particular processor can legally consume data for cache blocks which are in the CLEAN, DIRTY, SHARED, and DSHARED states. Stores in any particular processor can legally write data only in the DIRTY state. Since stores are only allowed to proceed on the exclusive DIRTY state, the external system is expected to invalidate any other cached copies of this memory block. In addition to these three states, the Cbox employs an additional state bit (the *Modify* bit) which tracks the coherence of a cache block in the Dcache with the same block in the L2 cache. This bit is

used to filter traffic between the processor and its board level cache. This technique is especially effective for code streams which initialize a cache block and then primarily perform reads to the block.

The Cbox must service transactions from two sources. The first source of transactions are loads and stores from this processor, which generate MAF entries to be serviced by the Cbox. The generation of these entries has been discussed in the previous sections. The other source of transactions are the loads and stores from other processors which generate system probes using the SysAddIn pins. These probes entries, which are requests from the other processor for the cache block and follow a very similar pipeline as the one utilized by MAF entries, are stored in an 8-entry queue (ProbeQ). The 21264 provides a rich set of probe primitives to build high performance system protocols ranging from cost-focused uniprocessors, to mid-range filter (duplicate tag) based 4-8 processor systems to performance focused directory based large (over 32 processor) servers.

These probe primitives are composed of two basic components --- data movement and next-state. The data-movement options are:

- NOP → data is not required for this block
- ReadifHit → data is required for any hit
- ReadifDirty → data is required only if block dirty
- ReadAnyway → data is required for hit (hit likely)

The next-state options are:

- NOP → do not change state
- Invalid → invalidate this block in the local cache
- Shared → set the next-state to shared
- Dshared → set the next-state to dirty-shared
- T1 → if block is clean set it to clean/shared
  if block is dirty set it to clean/shared
- T2 → if block is clean set it to clean/shared
  if block is dirty set it to dirty/shared
- T3 → if block is clean set it to clean/shared
  if block is dirty set it to invalid
  if dirty/shared set it to clean/shared

In addition to these probe primitives, systems can also control:

- Notification of dirty AND clean victim. For example, the notification of clean victims (when a processor disposes of a clean block) is useful for directory-based systems which must use directories to filter probe traffic to a given processor.
- Internal update of status bits for store processing. If a store instruction accesses a cache block in its own cache which is in an inappropriate cache state, systems can selectively allow the processor to automatically update the cache state to DIRTY.

*Example Protocol*

Given these basic primitives, consider a simple dual or quad processor system with no probe filtering (i.e. no duplicate tags or directories). In this case,

1. All transactions from any of the processors will be seen as probes in all the other processors. Since there is no probe filtering, the processor can internally move from the CLEAN state to the DIRTY state.
2. Loads from any processor will generate T2 probes with ReadIfDirty data movement. Thus, if the block was found CLEAN, it would transition to SHARED, and if it was found dirty, it would transition to DSHARED. The responsibility of victimization would be left with the probed processor. T1 probes could also be used if memory was to be updated concurrently with the delivery of data to the other processor.
3. Stores from any processor will generate Invalid probes with ReadIfDirty data movement.

Loads and stores to different addresses will miss in the other processors' caches and the data will be provided by main memory. Two or more loads referencing the same block will serialize at the system interface. The first load will miss in all the other caches and fill from main memory. Subsequent loads will use the probe-hit response to fill the block shared into their local caches, and the probe command itself (T2) will transition the block from CLEAN to CLEAN/SHARED.

Two or more stores to the same cache block will serialize at the system interface. The first store will get its data from memory and miss in all the other caches. All subsequent stores will hit dirty in the appropriate cache and get an updated version of the block. The cache block will move between processors until all stores to that location are satisfied.

Finally, a load and a store to the same location will serialize at the system interface. If the load reaches the serialization point first, its data will be filled from memory. The store will also obtain its data from memory, and in the process invalidate the cache block in the first processor. If the store reaches the serialization point first, it will get the cache block and perform the store. The load will get the updated block and be filled SHARED. The processor which issued the store will transition to DIRTY/SHARED based on the T2 probe, and must eventually victimize the cache block back to main

memory. Directory or duplicate-tag based protocols can be built using these primitives in a similar fashion.

## References

[1] D. Dobberpuhl *et al*,. "A 200-MHz 64-bit Dual Issue CMOS Microprocessor," Digital Technical Journal, vol. 4, no. 4, 1992.

[2] J. Edmondson *et al.*, "Superscalar instruction execution in the 21164 Alpha Microprocessor," IEEE *Micro*, vol. 15, no. 2, Apr. 1995.

[3] S. McFarling, "Combining Branch Predictors," Technical Note TN-36, Digital Equipment Corporation Western Research Laboratory, June 1993. <www.research.digital.com/wrl/techreports/abstracts/TN-36.html>

## Acknowledgments

The authors acknowledge the contributions of the following individuals: J. Emer, B. Gieseke, B. Grundmann, J. Keller, R. Kessler, E. McLellan, D. Meyer, J. Pierce, S. Steely, and D. Webb.