A Pipelined Asynchronous Cache System

Mika Nyström, Andrew M. Lines, Alain J. Martin

Abstract—
We present the design of a pipelined cache system for use in an asynchronous MIPS R3000-compatible processor. Although caches have been used in a few asynchronous processors (AMULET2e, TITAC-2), these rely on timing assumptions and delay matching or differ little from conventional synchronous designs. Our cache is designed as a distributed message passing system and implemented with full-custom quasi delay-insensitive circuits. This design achieves timing robustness, low latency, and high average-case throughput by making minimal assumptions on signal delays. Each cache is partitioned into 16 separately pipelined cells. These cells are connected via a bus system that operates at about twice the throughput of the cells. Consecutive accesses to different cells occur at the bus throughput, while consecutive accesses to the same cell are limited by the cell throughput. The cache architecture includes two 4-kB direct-mapped write-through caches for instructions and data. The design has been fabricated (as part of Caltech's MiniMIPS asynchronous processor). Using a ten-transistor unit cell, the entire cache system requires around 1.5 million transistors and occupies approximately 46 mm^2 in HP's 0.6-μm CMOS process. Simulations indicate an expected throughput of 320 million loads per second per cache at 25°C and 3.3 V with about 2.1 W power per cache; the cache system has been tested at speeds up to 180 MHz at the nominal voltage.

Keywords— **Asynchronous VLSI, quasi delay-insensitive, asynchronous SRAM, pipelined caches.**

I. Introduction

The MiniMIPS processor designed at Caltech is a test-bed for novel design ideas and methodologies in asynchronous quasi-delay-insensitive (QDI) design[3], [2], [1]. The QDI framework is derived from delay-insensitive (DI) design with the addition of weak but necessary timing assumptions called "isochronic forks" [3]. The goal of the Asynchronous MIPS project is to achieve the highest possible level of performance in a given fabrication technology (0.6-μm CMOS for the first version). The MiniMIPS is a simplified R3000-compatible processor, simple enough for us to complete the design with limited design resources yet powerful enough to demonstrate the potential of the design method. The main differences compared to the standard R3000 are the omission of kernel/user mode, memory translation, partial-word operations, the inclusion of on-chip four-kilobyte (kB) instruction- and data-caches, and slightly different interrupt semantics. The first prototype "MiniMIPS" is entirely binary-compatible with the R3000 at the user level (with partial-word operations emulated through the use of software traps or eliminated by recompilation). The overall architecture of the processor is discussed elsewhere [1].

M. Nyström and A. J. Martin are with the Computer Science Department of the California Institute of Technology, Pasadena, CA 91125, U.S.A. A. M. Lines was with the Computer Science Department of the California Institute of Technology; he is now with Fulcrum Microsystems, Inc., Calabasas Hills, CA 91301.

In order to achieve our goals, we have had to expand the current state of the art in asynchronous design tools, architecture, and microarchitecture. Not the least of the challenges stemmed from designing the memory system for the MiniMIPS. The goal was to execute code and perform load/store operations at high throughput even through program branches while maintaining a reasonably simple cache architecture. The design also had to be robust enough not to jeopardize the functionality of the MiniMIPS processor as a whole.

A. Problem overview

In the first prototype, we allowed ourselves the use of high-speed SRAM as off-chip "main" memory. Yet, achieving reasonable performance depended on managing the relatively high latency of accessing even on-chip SRAM level-one (L1) caches. We approached this problem at several levels: we developed a simple single-instruction branch-prediction scheme that hides part of the latency of fetching the target instruction of a branch (not described in this paper); we used a pipelined cache architecture that dispatches two memory operations within the latency of a single lookup in the cache memory array; and we developed an interleaved asynchronous memory architecture that allows the use of half-throughput (or slower) memory without sacrificing the overall throughput of the memory system.

B. The QDI design style

The MiniMIPS is entirely designed using the Martin synthesis method [2]. The processor uses four-phase (return-to-zero) signaling throughout, with data encoded as dual-rail, quad-rail (two bits combined into a single one-hot code), or using other delay-insensitive codes [11]. The communication-based implementation that results from this design method is amenable to expressing many concurrency constructs because communication actions are "blocking" so that explicit synchronization variables become unnecessary—the synchronizations are implied by the communication themselves. Function units compute when their inputs are available and their outputs are ready to be received. No process[1] needs to wait for an external, extraneous synchronization signal (such as a clock), but processes automatically wait until the units to which they communicate their outputs are ready to receive inputs.

The asynchronous timing discipline is especially simplifying in a memory system application since latencies are quite variable, which can result in "logjams" in proces-

[1]We use the terms "process" and "function unit" interchangeably and also make reference to the "programs" executed by these processes, meaning their logical behavior, as opposed to the software "program" executed by the CPU.

sor pipelines, and the design method handles these conditions implicitly. Furthermore, handshake-based process decomposition makes the system modular since the physical depth of pipelines is invisible in the high-level architecture of the processor. All processes in the cache control and cache core are entirely deterministic so that the sequence of actions performed by these units is determined only by the data and not by timing.

The price we pay for the modularity of the design method is that it is not possible to know when a certain piece of data is at a certain stage in a pipeline—we cannot "count clock cycles," so we need to generate explicit control information that travels with the data. This is necessary, for instance, in any situation that involves two data streams that are processed out of order, and a great deal of the complexity of the cache system derives from handling these kinds of data streams. In simple terms, the hazards that appear in an asynchronous pipeline are not determined by the physical pipeline topology as in the usual synchronous case but by the logical behavior of the function units (or the number of *tokens* present in the pipeline.) Hence, even though extra control signals may need to be added in the asynchronous case, the system stays modular. In a normal synchronous implementation of the same function using cycle counting, the FSMs implementing the out-of-order operations would need global timing information, reducing the modularity of the system. We suggest that the modularity advantage outweighs the cost of any extra control signals. Of course a synchronous system can also be structured around FIFOs with handshake signals, but as we shall see, once the handshake signals are there, the clock becomes unnecessary and in fact serves little purpose other than to slow down the circuits.

C. *Organization of this paper*

In this paper, we examine the design of the MiniMIPS cache system, giving an overview of the MiniMIPS overall memory system, then paying special attention to the high-level cache architecture, and finally to the design of the core memory arrays themselves.

II. MiniMIPS Memory System Overview

The MiniMIPS memory system consists of three parts: instruction cache (I-cache), data cache (D-cache), and memory interface. The machine employs a "Harvard" architecture; in other words, the I- and D-caches are wholly independent—the hardware does not keep them consistent. The two caches access main memory through the memory interface for cache refills and write-throughs (for the D-cache). From the point of view of the memory interface, I- and D-cache requests may occur completely asynchronously. An arbiter is used to interleave accesses from the I-cache and D-cache nondeterministically.[2] An overview of the memory system is shown in fig. 1.

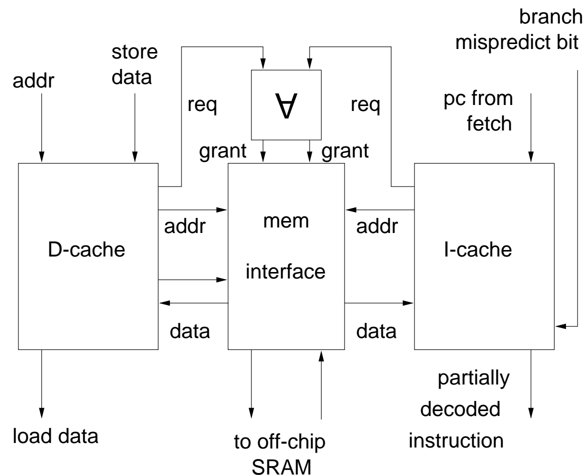

Fig. 1. Overview of the MiniMIPS memory system.

The main memory used for the MiniMIPS prototype is standard "asynchronous" SRAM. The off-chip memory system is not QDI but uses a more traditional "bundled data" asynchronous approach (this is the only part of the MiniMIPS that does not use delay-insensitive codes). Proper off-chip memory timing is achieved with delay lines that can be tuned to the delay of the memory modules used. For accessing other types of devices, e.g., EPROMs, UARTs, there are provisions for separate adjustable delay lines, which may even have address-dependent delays to allow complex off-chip address-decoding circuitry. For each off-chip memory access, the proper delay line is selected based on the address of the access.

III. High-level Cache Architecture

The two caches of the MiniMIPS are designed so as to be very similar, to allow the reuse of significant portions of the design between the I- and D-caches. The caches are designed so that each has a "control" section (part of the main datapath of the processor, see figure in [1]) and a 1024-line (4-kB) cache array. Each cache line consists of a word (32 bits) of data and 16 bits of tags. The tags are generated from bits 12 through 27 of the address. Address bit 31 is hard-wired to zero in the cache core; this technique is used to implement an "uncacheable" two-gigabyte segment since requests for addresses with bit 31 set always miss. Refills are performed in blocks of four lines or words; this is done in parallel as the off-chip memory bus is 128 bits wide (not dual-rail). It may not at first seem sensible to use single-word cache lines if they are going to be refilled four words at a time, but this is done to allow straightforward interleaved access to the tags. The MiniMIPS does not support partial-word operations in hardware and therefore no

[2]By guaranteeing that the arbiter is at least weakly fair and that we have finite buffering in the processor, it is possible to write data from the D-cache into main memory and then execute the data later, without hazards. The circuit implementation of the arbiter used in the MiniMIPS grants requests in the order they were received, unless the requests arrived very close to each other, in which case we do not know—or care—in what order the requests are granted; this means that if the I-cache requests access to the memory, the D-cache will be granted access at most once before the I-cache request is serviced (and vice versa).

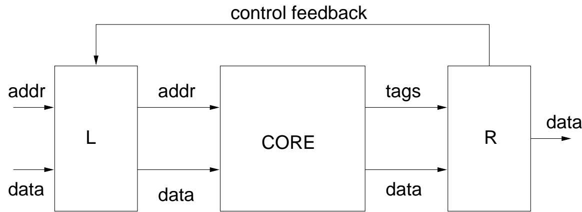

Fig. 2. High-level structure of the MiniMIPS cache system. This depicts the structure of the D-cache; in the I-cache, there is no store data going into the *L* or *CORE* process.

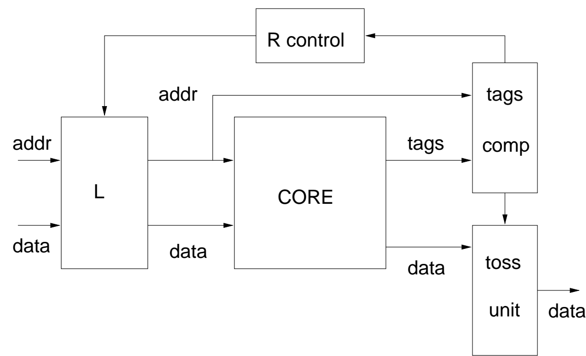

Fig. 3. Decomposed structure of the MiniMIPS cache system.

mechanism is provided for accessing smaller regions than whole words. The MiniMIPS, for reasons of simplicity and R3000-compatibility, has direct-mapped caches, and the D-cache is write-through.

A. Decomposition of the Cache System

We first decomposed the cache system into the two parts "core" and "control." We further decomposed the control part into several units, each with specific tasks. Taking our inspiration from a picture of the cache system as a pipeline (with feedback), we call the control unit that issues addresses and operations to the cache core L and the control unit that is responsible for tags comparisons, scheduling refills, and returning the results of cache operations R; see fig. 2. The L process receives its instructions either from the instruction decode (in the case of the D-cache) or from the instruction fetch unit (in the case of the I-cache, which receives no data since it cannot write).

The R process is further decomposed into a "toss" unit that discards data resulting from a cache miss (i.e., the tags are checked in parallel with the readout of the data from the cache core), a tags comparator, and a control process. This decomposition is shown in fig. 3. In the I-cache, a "predecode" unit is inserted between the cache core and the toss unit. This allows each instruction to be partially decoded before it is known whether or not it resulted from a cache hit. This speculative mechanism reduces the total latency of decoding a MiniMIPS instruction by about a nanosecond, a significant amount in a simple, single-delay-slot ("two-PC") architecture. The toss unit in the I-cache also interfaces with the instruction-fetch unit's branch-prediction mechanism; the I-toss unit is responsible for discarding the instruction corresponding to incorrectly predicted branches.

Each one of the units in fig. 3 is actually further pipelined into several smaller units, allowing much higher throughput than if these units were to be implemented directly. This is possible because of the modularity afforded by the design style mentioned in section I-B. Thus, from the point of view of the main pipeline, the deeply pipelined caches behave entirely as

```
repeat forever {
       receive (op,address);
       if (op=read) send (mem[address]);
       else {receive (data); mem[address]:= data;}
       } ,
```

except that there is buffering, so that multiple `receive`s may (but are not required to) occur before the first `send.`

The ultimate physical depth of the pipelines was chosen to be approximately eight stages per token, using the methods described in [5], [9].

A more detailed description of the process decomposition that led to the MiniMIPS cache control is given in [4].

B. Pipelining the cache system

The main challenge in the architecture of the cache system is hiding the latency of the cache lookup and tags comparison. Our performance goal of 320 MHz at room temperature has to be fulfilled with a cache core latency of about 3 ns and 1 ns for tags comparison plus at least 0.5 ns control overhead. It is clear that a solution in which process L executes

> ...
> *issue address*$(*)$; *wait for hit/miss from* $R(\dagger)$; *(if a miss was received, handle refill); issue next address;* ...

will not suffice, since it takes at least 5.3 ns to go from $(*)$ to $(\dagger)$, implying a maximum throughput of 188 MHz (provided that the four-phase return to zero occurs in parallel).

Note that although the normal time between the issuing of address 1 and that of address 2 to the cache core is about an instruction delay and the refill data on a miss will appear from off chip much later (perhaps ten instruction delays for the prototype MiniMIPS), the extra refill delay is invisible in both the program (formal) specification and in the circuit implementation since it is handled by a blocking communication.

C. Pipelining cache reads

We kept the cache latency from resulting in a throughput bottleneck by pipelining the cache aggressively and by adding concurrency. The L process now issues two operations before receiving the result of first operation. We describe the pipelining of the cache core itself (that makes this scheme possible) in section V. Now L behaves as follows:

issue address 1;
issue address 2;
wait for hit/miss 1;
issue address 3;
wait for hit/miss 2
...

Our design requires that results of memory lookups be output in the same order that the addresses are dispatched; therefore cache misses need to be handled by refills before the results of the next lookup can be processed. Because of the FIFO nature of the channels used in the implementation, L cannot easily control outputting the data immediately since it normally does not receive the result of a cache access until after it has dispatched the following cache operation. Thus the outputting of refilled data is handled entirely in process R; process L is left responsible for initiating the write into the cache core (this way we avoid having to control the cache core from more than one place). Process L now executes

issue address 1 as a read;
issue address 2 as a read;
receive miss for 1;
re-issue address 1 as a refill; (‡)
receive hit/miss for 2;
...

and process R executes

receive missed read on 1, discard output;
refill address 1, output value from memory;
receive read on 2;
....

It is clear that process L needs storage to be able to recall old addresses; this is handled by a linear FIFO. More importantly, we see that on a refill, the sequence of operations in L does not match that in R. The cache core provides the buffering required for storing the refill data until the L process dispatches the refill address at (‡).

D. Pipelining cache writes

Memory stores are translated to cache and main memory writes. The addition of writes to the caches introduces the possibility of *structural hazards*. To see why, consider a program that reads a value from memory and then immediately afterwards writes to the same *refill block* of memory. Suppose the first read is a cache miss; then process L will execute (we assume both the read and write are to the same address for clarity)

issue address 1 as read;
issue address 1 as write (with data);
receive miss for address 1;
re-issue address 1 as refill;

It is clear that what happens is that in the final state, the value that was written to the cache core will be overwritten by the old value from main memory, so that subsequent reads from address 1 will yield incorrect data.

There are numerous possible solutions to the write hazard. We could simply destroy the contents of the block (note that the value stored in main memory is correct), or we could stall the cache system so that writes are not allowed to proceed until all preceding reads have been resolved[6]. Neither of these solutions seemed attractive, the first requiring extra hardware (the MiniMIPS caches do not have valid bits), and the second incurring a substantial performance penalty since memory writes are a non-negligible proportion of total program operations (some 7% of representative program instructions are memory writes [8]). However, only very few of them—those where consecutive reads and writes refer to the same four-word stretch of memory—would actually *need* to be handled by the mechanism.

We chose to handle the write hazards "optimistically"; in other words, we dispatch all operations assuming they will succeed, and correct the situation later if that turns out to have been too optimistic. If so, we repeat the cache write as follows (from the vantage point of L):

issue address 1 as read;
issue address 1 as write (with data);
receive miss for address 1;
re-issue address 1 as refill;
re-issue address 1 as write (reissue data, too);

This solution requires that L be able to recall the data used in the last store (a one-place buffer is used for this), but in return the mechanism is only exercised when a conflict is detected. The R process detects that a cache miss has been followed by a store to the same block. For layout reasons, R is made a bit pessimistic, sometimes reporting conflicts even though the writes are to different pages of memory. This is because R only compares the block offset within the page to detect possible conflicts, not the entire address, but this is still correct because false positives simply incur an unnecessary cache write and do not lead to data inconsistencies.

E. Eliminating read thrashing

Owing to the delayed nature of the cache reads, there is a possibility of unnecessary refills due to accesses to consecutive memory addresses. For instance, assume words at two

consecutive addresses 1 and 2 are to be read (as might be a particularly frequent occurrence in the I-cache), and both miss; the sequence of operations from the vantage point of L is:

issue address 1 as read;
issue address 2 as read;
receive miss for address 1;
re-issue address 1 as refill;
receive miss for address 2;
re-issue address 2 as refill;

In this case, since refill blocks are actually four words long, if address 1 is in the beginning or middle of a refill block, the data required by the second read will already be refilled by the block containing address 1, and the second refill will be unnecessary. To eliminate the unnecessary memory access, we reuse the comparator used to detect conflicts between refills and subsequent writes and instead have L execute the sequence

issue address 1 as read;
issue address 2 as read;
receive miss for address 1;
re-issue address 1 as refill;
retry address 2 as read;

If address 2 still misses (it might, because we do not check every bit of the address, as mentioned in section III-D), we refill address 2 on the second try. Because the second failed attempt will result in a refill, the procedure always makes progress.

F. Extending the pipelining scheme

While the MiniMIPS processor does not include memory translation or partial-word operations, supporting these sorts of memory accesses was one of the main original motivations for the described architecture. The addition of memory translation can be accomplished by inserting a translation lookaside buffer (TLB) that translates the page number in parallel with the cache core lookup. This allows memory translation to be added to the memory system with essentially zero performance cost. Similarly, partial-word operations can be supported by adding a single-word buffer so that data read from the cache core may be merged with data from the main processor pipeline. Again, this can be implemented by extending the functionality of the R process in a way that is not visible (or costly) under normal circumstances but only when it is used—a decided advantage of the asynchronous implementation. These issues are explored in greater detail elsewhere [4].

IV. Pipelined Microarchitecture

Each one of the units of the MiniMIPS was pipelined until it could meet the throughput target for the entire processor. This means that the functional decomposition described in the preceding paragraphs was continued until each unit consisted of generally no more than a single stage of CMOS domino logic followed by inverters. This pipelining technique does not increase the forward latency, since no external latches are introduced, and the number of transitions on the forward path is unchanged. More pipelining does somewhat increase the area and energy, as more control circuitry is added to handle the larger number of handshakes. The area overhead for our deep pipelining is usually approximately 50% over the basic domino logic blocks. However, we save area by omitting any clock generation, distribution, or gating.

Furthermore, most of the units were "byte-sliced" into four (for the 32-bit datapath) separate datapaths that are controlled independently through a QDI copy tree (which adds moderately more area). This "two-dimensional" pipelining (described elsewhere [1]) was a key feature in making it possible to achieve the ambitious throughput target.

V. Cache Core Architecture Overview

The specification of the core was kept simple to separate the problems of managing the cache protocol and designing a dense and fast array. Each of the D-cache and I-cache cores is organized into 1024 lines of 48 bits each, 16 of which are interpreted as tags in the control, but are otherwise identical to the 32 data bits. Therefore, the total storage is 6 kB per cache. The 50% tags overhead was mainly a result of a desire to stay as compatible as possible with standard MIPS R3000 implementations and is not required for an asynchronous implementation.

The D-cache core handles only three operations: load, store, and refill. The requested operation is encoded as an instruction to the cache core. Included with this instruction is a 10-bit address, derived directly from bits 2–11 of the full address. For a load, the specified line of the cache is read and sent out on the load channel. For a store, a 48-bit number is received from the store channel and written to the specified line. For a refill, the least significant two bits of the address are ignored, and a four-word block is read from the refill input channel and stored in parallel. The I-cache core is identical, except that it lacks the store operation and channel. For concreteness, all subsequent discussion will be of the D-cache core.

VI. Cache Core Tree Structure

The 1024 lines are accessed via a three-stage tree. The first stage interleaves between four 1-kB cache rows. Within each 1-kB cache row, the next stage of the tree interleaves between four cells. Each of the 16 cells has 64 lines of 48 bits. All stages of the tree are pipelined to maintain high throughput.

A. The bit cell

To store a single bit, we used the ten-transistor (two pMOS and eight nMOS) SRAM circuit of Fig. 4 and Fig. 5. Although about 50% larger in area than the standard six-transistor circuit, it avoids charge from the read

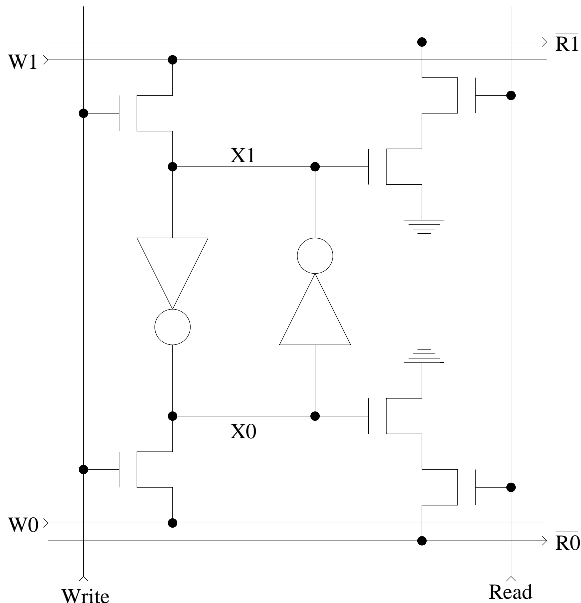

Fig. 4. Schematic of one-bit cache cell.

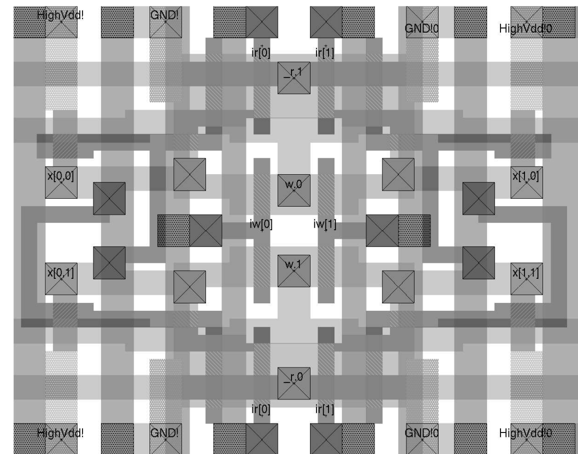

Fig. 5. layout of 1 bit x 2 lines (52λ x 71λ).

busses backwashing onto the state node. Also, since it has separate read and write dual-rail data lines, the read and write circuitry are more separable. In this design, the bit-line voltage swings are rail-to-rail. The conservatism of the design was due to a concern for reliability and first-silicon success of the entire MiniMIPS project; in this light, we accepted the area and power costs of the larger cell and full voltage swings. There are no fundamental issues that preclude the use of a standard six-transistor cell with reduced voltage swings and sense amplifiers in an asynchronous processor such as the MiniMIPS.

To improve the latency and throughput, we provide a different Vdd for the cross coupled inverters of the state bit. This “HighVdd” may be set higher than the normal Vdd (5 V versus 3.3 V). Provided that the p-transistors are weak, the lower-voltage write lines can still set the state bit. The greater gate voltage speeds up the read transitions. The HighVdd dissipates power only when a state bit is written, and it is a negligible contributor to overall power. HighVdd also allows us to adjust the latency of the cache without affecting its overall throughput, which allows us to determine whether the latency or throughput of the cache system is on the MiniMIPS critical path.

B. The 64-line cell

The lowest level of the core tree is a 64-line by 48-bit cell. It is composed of a 64-by-48 array of state bits, read circuitry, write circuitry, and decoding circuitry, as shown in Fig. 6. The decoding circuitry reads a six-bit address and a one-bit read/write instruction off its input channel. It demultiplexes this into a one-of-128 code. This code drives separate read and write line selects. The decode circuitry is a separate pipeline stage from the read or write itself.

For a read, the selected line of state bits pulls down one of two read bus rails, which causes the read circuitry to send the result on the output channel. No sense amps or other analog circuitry is used to inspect the read busses. Once the busses have dropped by a transistor threshold, the output begins to switch. Since these busses are only 64-way, the read delay is only about 1 ns. A 48-bit completion tree is used to detect the completion of the read. During the reset phase of the handshake, the read bus lines are precharged. A 64-way OR tree detects the neutrality of the read selection on the reset phase.

For a write, the data is latched onto a dual-rail bus, and the state bits of the selected line are set via n-transistor pass gates. These pass gates are the only pass gates used in the MiniMIPS, and also have the only on-chip transitions that are not acknowledged in a QDI fashion. Instead, the 48 data bus pairs and the 64 write selections are checked for validity (which takes several logic transitions), and the state bit is assumed to have changed by the time this is detected. During the reset phase of the handshake, the dual-rail write bus is kept stable (i.e., it is a bistable device, not a return to zero variable). This is so the data and selection may go neutral in parallel, without the danger of erasing the state bit via the pass gates.

It is possible to make the cache write fully QDI by pulling down with one write line while leaving the other floating high. Once the correct value is written, the floating line will be pulled down by the bit cell, thus indicating the completion of the write. Although this does not change the state bit circuitry, the extra write circuitry and delay required to do this was considered excessive in comparison to a modest timing assumption.

The 64×48-line cell can read continuously at 175 MHz with 0.1 W of power, and writes at 190 MHz with 0.07 W power (according to SPICE simulations).

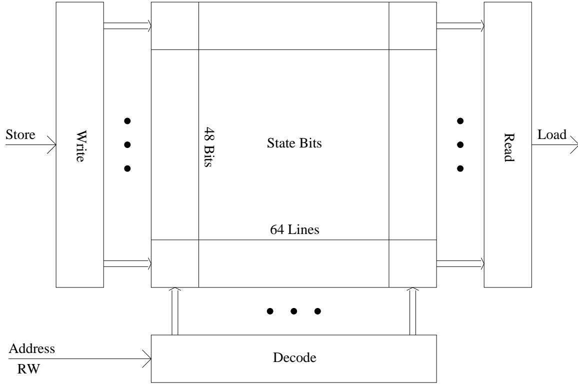

Fig. 6. Block diagram of 64-line by 48-bit cache cell.

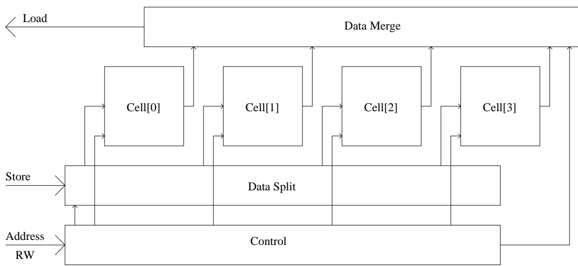

Fig. 7. Block diagram of the 1-kB cache row.

C. The 1-kB cache row

Four 64×48 cells are composed with bus circuitry to construct a 1-kB cache, as shown in Fig. 7. At this level, the only operations are read/write, and the address is eight bits long. The bus circuitry consists of a "data split" process, a "data merge" process, and a "control" process. The control uses the least significant two bits of the address to send the remaining address bits and the read/write bit to the correct 64×48 cell. It also uses the read/write bit and the least significant two address bits to tell either the "data split" or the "data merge" with which of the four 64×48 cells to communicate.

In order to increase the throughput substantially, the 48-bit data path is broken into six 8-bit sections. Each section receives a buffered copy of the control and has its own completion and acknowledge circuitry. This is a standard technique used throughout the MiniMIPS to avoid the cycle time penalty of broadcasting and completing across a large datapath. The 64×48 cells themselves deal with all 48 bits at once for reasons of greater density. However, to give the busses sufficiently high throughput, they must be decomposed.

All of the busses have sufficient slack (pipelining) so that a different 64×48 cell can be accessed while the last one is still in its reset cycle. If different 64×48 cells are accessed sequentially, their reading and resetting operations will run in parallel. The 1-kB row busses can read at 315 MHz with 0.6 W power, and write at 336 MHz with 0.6 W power.

D. The 4-kB cache

Four 1-kB cache rows are composed with another layer of bus circuitry to construct the 4-kB cache, as shown in Fig. 8 and Fig. 9. These busses are analogous to those of the 1-kB row except for the addition of a four-word parallel refill operation.

The "address bus" uses the least significant two bits of the address to deliver the read/write bit and next eight bits of the address to the correct row. In the case of a refill, it broadcasts the address to all four rows and instructs them to write. The datapath part of the cache control sends additional control directly to the "load bus" and "store bus."

The "load bus" merges up the data from the four rows in the case of a load. For a refill, the load bus receives a copy of the four-word refill from the memory unit and selects which word to send out the load channel (to the R process). This effectively bypasses the cache so that the refilled word is available as quickly as possible to the rest of the processor. This is necessary since L is dealing with the cache operation between the attempted read and the refill.

The "store bus" splits the store data and tags to the selected row in the case of a store. Since the cache is write-through, it also sends a copy of the store data to the memory unit, which passes through a write buffer. This buffer can hold from two to eight outstanding stores to memory, depending on their addresses. For a refill, the "store bus" receives a second copy of the four-word refill data (with tags added to each word by the memory unit) and sends them in parallel into the four cache rows.

As with the 1-kB caches, the "load bus" and "store bus" are decomposed into six eight-bit segments, which reduces their critical paths and increases the throughput.

The bus wires on the "load bus" and "store bus" extend nearly the entire height of the chip (22,000 λ[10]). For this length, wire resistance becomes quite significant, and it would probably be impossible directly to drive the whole length at high throughput. To solve this problem, a set of inverters was placed at the end of the cache core region, so that the signals were repeated up into the datapath portion of the memory system. At the end of the busses, an extra pipelining buffer was added to decouple the bus cycle from the cache control circuitry. With these improvements and substantial transistor width optimization, the busses of the 4-kB cache load at 320 MHz with 1.31 W of power and store at 280 MHz with 0.81 W power.

VII. Cache Core Performance

The latency of a load through the entire 4-kB cache core is only 14 transitions, or about 3.8 ns. Of these transitions, the slowest are the 64-way read in the cell and the merge and load busses.

When different rows are accessed sequentially (i.e., the addresses differ in the bottom two bits), the cache core can load at the throughput of the outer busses, or 320 MHz with 2.1 W of power. If the same row is accessed sequentially, but the cells differ (that is, the addresses differ in

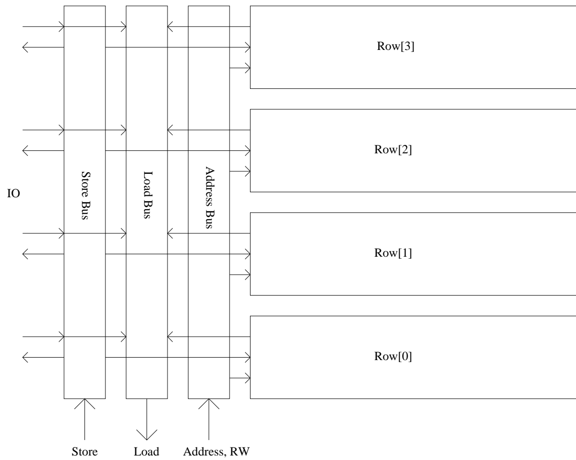

Fig. 8. Block diagram of the 4-kB data cache.

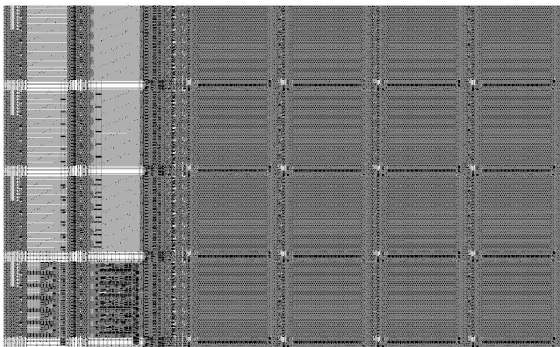

Fig. 9. Magic layout of the D-cache core. Metal 3 is not shown.

bits 2–3), the throughput is limited by the 1-kB busses, to 315 MHz. Only if successive accesses are to the same cell (when the least significant four bits are the same) will the throughput be limited to 175 MHz. In all cases, the power varies linearly with the throughput.

Assuming a random distribution of addresses, we may expect an average case load throughput of 303 MHz. The worst-case throughput is 175 MHz. For the I-cache, assuming that a branch or jump occurs once in every six instructions, we expect an average instruction fetching throughput of 317 MHz with a worst-case throughput of 226 MHz, since the MIPS ISA ensures that branches and jumps can occur at most every other instruction. Since the MIPS Level 1 ISA also requires a shadow after every load instruction, the latency of the D-cache is not likely to be on the critical path of normal user code unless addresses accessed sequentially line up in unfortunate ways (assuming no cache misses, of course).

VIII. Performance

A. Transistor counts and layout area

The ten-transistor SRAM cells for the data and tags of each 4-kB cache contribute 492,000 transistors. The D-cache core contains 215,000 additional transistors and occupies 19.4 mm^2 in total. The I-cache core contains 189,000 additional transistors and occupies 17.1 mm^2. The I- and D-cache control sections in the processor datapath consist of approximately 40,000 and 51,000 transistors, respectively, and take up 3.7 and 5.5 mm^2 in HP's 0.6-μm process. These transistor counts are measured from extracted layout, and thus over-count folded gates and include all the "staticizers" (weak cross-coupled inverters used to make dynamic nodes pseudo-static).

Standard MOSIS logic layout rules (`scmos-sub`) were used for the design, and the area of the cache core is dominated by wiring, mainly for the dual-rail read and write channels. The entire cache system requires around 1.5 million transistors (about 15 total transistors per data bit) and occupies approximately 46 mm^2. The size of the control logic is basically independent of the sizes of the caches; thus, a larger cache would have negligible area devoted to control logic.

B. Simulation, fabrication, and measurements

Analog simulations of the cache system indicate an average case throughput (ignoring cache misses) of 320 million loads per second per cache at 25°C and 3.3 V with about 2.1 W per cache power consumption. The access time of the cache core is about 3.8 ns. The standby power of a cache that is not being used is negligible. Performance of cache misses is almost entirely limited by the speed of the off-chip memory interface (the MiniMIPS uses a straightforward, but antiquated, asynchronous SRAM main memory system).

All the units described in this paper have been simulated both digitally and using the BSIM2 model on circuits extracted from the layout. The design was submitted for fabrication in the fall of 1998 in HP's CMOS14B 0.6-μm CMOS process via MOSIS.

Silicon was received in January of 1999. The MiniMIPS processor operates over a very wide voltage range (less than one volt to over six volts), but performance has been disappointing. Our target had been 280 MHz at 25°C, but a design error (a polysilicon wire used to drive a heavily loaded node in the instruction-fetch unit) and process detuning by HP only allowed the processor to operate at 180 MHz. By adjusting HighVdd (see section VI-A), we have determined that the instruction cache is not critical at this speed, but that is unfortunately the extent to which we have been able to verify our performance predictions for the cache system. Given our information about the process and our lab data, our best guess is that the I-cache would be able to operate at roughly 250 MHz (down from 320 MHz) on its own at 25°C and 3.3 V and at roughly 220 MHz at a more reasonable junction temperature of 75–85°C. The D-cache is likely to be about 20 MHz slower

because of the extra complexity of the control logic, but in actual code, this difference would most likely be absorbed by slack (buffering) between the various units of the MiniMIPS. Our power-delay product estimates were found to be very accurate for the processor as a whole, and we have no reason to believe otherwise for the cache system. We are currently investigating a die shrink of the MiniMIPS to a 0.25-μm or smaller technology.

C. Comparing with other designs

TITAC-2 is a MIPS R2000-based microprocessor designed in the "Scalable-Delay-Insensitive" framework[12], but its cache system is quite different from the MiniMIPS. The TITAC-2 processor uses an ASIC SRAM macro cell for its cache array and implements an 8-kB I-cache (no D-cache) with an eight-word line with a sequential fill. It uses a mechanism similar to the MiniMIPS main memory interface with a bundled data interface and an off-chip tunable delay line to properly time the signals to the SRAM macro. The TITAC-2 caches are aimed at a lower performance level than the MiniMIPS, about 65 MHz (52 VAX MIPS) in 0.5-μm CMOS.

The AMULET2e cache architecture is unconventional[13]. It uses a very highly (64-way) associative content-addressable memory (CAM) to store tags and RAM only for the data; this is due to a desire to be compatible with the ARM family of processors. The AMULET2e cache is truly a fully self-timed design, but like the TITAC-2's cache, it is aimed at a substantially lower performance level, with a cycle time of 16–20 ns in a 0.5-μm CMOS process. Our caches are about twice as big as the AMULET2e caches, but have five or six times higher throughput. More recently, the AMULET group has done work on studying higher-level cache protocols in asynchronous systems [15]; our work has focused more on high-performance circuits, pipelining techniques, and microarchitecture.

The Digital Equipment Corporation (DEC) Alpha 21064 is a high-performance 64-bit synchronous processor fabricated in an "enhanced" 0.75-μm CMOS process[14]. Its caches use a six-transistor SRAM bit with a local interconnect layer (said to save 20% area). Normalizing for feature-size differences, our ten-transistor SRAM cell is 2.65 times larger. This is partly due to the four extra n-transistors, but also partly due to wiring constraints, which could be resolved with more layers or a local interconnect. The Alpha 21064 runs at 200 MHz in the enhanced 0.75-μm technology, while our simulations in 0.6-μm CMOS run at 320 MHz and the silicon runs at least 180 MHz. The MiniMIPS has only a single delay slot, while the 21064 has two (not architecturally visible), so our cache latency may be better in proportion to our cycle time. Obviously, to compete with this design, we would need to redo our core cells to save area. This would be straightforward with a more highly optimized fabrication technology and a six-transistor unit cell.

IX. Summary and Conclusions

We believe that our experience with the QDI L1 cache system for the MiniMIPS shows that the asynchronous, communications-oriented design style is effective in achieving excellent average-case performance with relatively little design complexity. While the asynchronous design method's automatic power-down is a well-known advantage for low-power design, our focus in designing the MiniMIPS memory system has been more on performance and simplicity from the point of view of synchronization, even in the presence of variable delays due to the latency limitations of off-chip memory and the data-dependent throughput limitations of the cache core itself. In contrast to arithmetic/logical operations that can be handled with relative ease in both the synchronous and asynchronous processors, the latency variability of memory operations can be large and data-dependent, especially if partial-word operations are allowed. A synchronous memory system implementation in this environment would incur either large worst-case penalties or a great deal of design complexity in scheduling pipelined operations, whereas our asynchronous implementation elegantly handles much of this complexity with blocking communications.

This design represents the first of its kind, a QDI message passing implementation of a L1 cache system for a high speed asynchronous CPU. Many of the design decisions were constrained by the desire for compatibility with the now-obsolete R3000 cache architecture and by zealous avoidance of race conditions, interference, or low-voltage signaling. For future designs, it would be easy to use a six-transistor SRAM, sense amplifiers, or even a one-transistor or three-transistor DRAM cell. These changes would only alter the core 64×48 cells, leaving the bus and control architecture the same. Future designs from our group will utilize more advanced cache architectures, such as set-associativity, a write-back policy, shared tags, and a larger line. With these changes, area savings of 50% over the current design would be feasible without significant performance penalty.

The caches for the MiniMIPS processer demonstrate that the QDI design style is now sufficiently developed to implement all essential features of a modern microprocessor without using a global clock, and without making assumptions about the delays of gates. The MiniMIPS cache system outperforms its asynchronous competitors, and it is also competitive in performance, although not yet in area, with synchronous processors in similar fabrication technologies.

X. Acknowledgements

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency (DARPA) and monitored by the Office of Army Research. Mr. Nyström was also supported in part by an Okawa Fellowship, and Mr. Lines was supported in part by an Intel Graduate Fellowship. We thank Rajit Manohar for reviewing the manuscript and Marcel van der Goot for suggesting

the reuse of the conflict comparator to reduce cache thrashing.

References

[1] A.J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. Cummings, and T.K. Lee. "The Design of an Asynchronous MIPS R3000 Processor," *Proceedings of the 17th Conference on Advanced Research in VLSI.* Los Alamitos, Calif.: IEEE Computer Society Press, 1997.

[2] A.J. Martin. "Synthesis of Asynchronous VLSI Circuits," in *Formal Methods for VLSI Design,* J. Staunstrup, Ed. North-Holland, 1990.

[3] A.J. Martin. "The limitations to delay-insensitivity in asynchronous circuits," *Sixth MIT Conference on Advanced Research in VLSI,* W.J. Dally, Ed. Cambridge, Mass.: MIT Press, 1990.

[4] Mika Nyström. "Pipelined asynchronous cache design." M.S. Thesis, Caltech CS-TR-97-21, 1997.

[5] Andrew Lines. "Pipelined asynchronous circuits." M.S. Thesis, Caltech CS-TR-95-21, 1995.

[6] H.P. Hofstee. Personal communication, September 1997.

[7] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Englewood Cliffs, N.J.: Prentice-Hall, 1992.

[8] John Hennessey and David Patterson. *Computer Architecture: A Quantitative Approach,* 1st ed. San Mateo, Calif.: Morgan-Kaufmann, 1990.

[9] Ted E. Williams. *Self-Timed Rings and their Application to Division.* Ph.D. Thesis, Computer Systems Laboratory, Stanford University, 1991.

[10] Carver Mead and Lynn Conway, *Introduction to VLSI Systems.* Reading, Mass.: Addison-Wesley, 1980.

[11] Charles L. Seitz. "System timing," chapter 7 in [10].

[12] A. Takamura, M. Kuwako, M. Imai, T. Fuji, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya. "TITAC-2: An asynchronous 32-bit microprocessor based on Scalable-Delay-Insensitive model."

[13] J.D. Garside, S. Temple, and R. Mehra. "The AMULET2e cache system." In *Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems.* Los Alamitos, Calif.: IEEE Computer Society Press, 1996.

[14] Dileep Bhandarkar. *Alpha Implementations and Architecture: complete reference and guide.* Newton, Mass.: Digital Press, 1996.

[15] D. Hormdee and J.D. Garside. "AMULET3i cache architecture." In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems,* pp. 152–161. Los Alamitos, Calif.: IEEE Computer Society Press, March 2001.