Pipelined Asynchronous Cache Design

Thesis by

Mika Nyström

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California, U.S.A.

1997
(Revised April, 1999)

Caltech Computer Science Technical Report
caltechCSTR:2001.009

Abstract

This thesis describes the development of pipelined asynchronous cache memories. The work is done in the context of the performance characteristics of memories and transistor logic of a late 1990's high-performance asynchronous microprocessor.

We describe the general framework of asynchronous memory systems, caching, and those system characteristics that make caching of growing importance and keep it an interesting research topic. Finally, we present the main contribution of this work, which is a latency-tolerating asynchronous cache micro-architecture suitable for asynchronous microprocessors.

In Chapter Two, we present a case study of the Level 1 data and instruction caches for the Caltech MiniMIPS asynchronous microprocessor, currently under development at Caltech. The implementation is quasi-delay-insensitive in 0.6 micron scalable CMOS rules, with a logic latency of approximately 2 nanoseconds.

Acknowledgments

I wish to thank the past and present members of the Asynchronous VLSI Group at Caltech for many stimulating discussions: Paul Pénzes, Robert Southworth, Marcel van der Goot, Tony Lee, Peter Hofstee, José Tierno, Uri Cummings, and especially Andrew Lines, whose comments inspired much of the work described in this thesis and Rajit Manohar, whose help with TEXnicalities, among many other things, made it at all possible. Last but not least, thanks to Alain Martin for being my advisor and putting up with the many odd ideas I have had.

The work described in this thesis was sponsored by the Defense Advanced Research Projects Agency and monitored by the Office of Army Research. It was also supported in part by an Okawa Foundation Fellowship. This work would not have been possible without the work of many hundreds of programmers who indirectly supported it with free software such as Berkeley's `magic` and BSD operating system, Donald Knuth's TEX, and too many minor utility programs to mention.

Contents

Chapter One.

Strategies for Pipelined Asynchronous Caching

1.1. Introduction

This Thesis discusses aspects of the design of the memory system of *asynchronous* processors. In the most basic meaning of the word, asynchronous processors are simply those that operate without the benefit of a single global clock signal to which all actions are synchronized. In our work, however, we study the *quasi-delay-insensitive* (QDI) subclass of asynchronous circuits[14]. QDI circuits are those whose correct operation is independent of delays in operators or wires, with the single exception that it is postulated that an electric signal may be copied to multiple destinations in such a way that the maximum difference in arrival time, or "skew," of the signal at the destinations is small compared to the gate delays in the system. (These copying wires, or *isochronic forks*, may be identified, and the designer may wish to spend effort to ensure that the postulated isochronicity condition holds for them.)

1.1.1. Asynchronous Advantages

Asynchronous design promises to bring many improvements to VLSI systems design over the currently widespread synchronous fashion. Chief among these is:

> *Average-case instead of worst-case performance.*
> Asynchronous systems can be made to perform "as well as they can" for any given set of inputs rather than needing to be artificially constrained by a fixed clock.

In asynchronous circuits, the timing rôle of the clock is taken over by locally generated handshake signals. In our design style, these signals carry information that is used to provide automatic flow control. This leads to another promise of asynchronous design, less often stated but nonetheless important:

> *Modularity in design.*
> The use of handshake signals encodes relevant state in the unit interfaces. This allows the use of modular interfaces, as opposed to the complex pipeline control logic often seen in synchronous designs.

The average-case performance and modularity properties act together to make it possible for the asynchronous circuit designer to invent large high-performance systems with a relatively modest amount of effort.

1.1.2. Modern Memory Systems

In recent years, manufacturing technology has made it possible to fabricate extremely fast microprocessors (the current (1997) state of the art commercial synchronous CMOS designs operate with a clock frequency of one half gigahertz) and very large dynamic random access memories based on trench capacitors and a single transistor per bit (currently commercially available at about 70 million bits per chip). However, there is a fundamental physical tradeoff in the design of memories. It is not possible to design an extremely dense DRAM that has arbitrarily short access time, and as a result DRAM (speed) performance has not been improving at the same rate as CPU logic performance.

The reason that it has been possible to increase CPU speeds at the same time as DRAM densities to the extent seen during the 1980's and 1990's lies in *caching.* Most computer programs exhibit a great deal of locality of reference (both in terms of instructions and data), and this property allows the use of a small, fast memory to improve the average access time to memory by caching often-used data.

Current high-performance microprocessors are still more aggressive in terms of the gap between processor speed and memory latency than a simple caching system would allow. To allow processor instruction rates to escalate past the ability of an on-chip SRAM cache, a prefetch buffer is used. The prefetch buffer is used to compensate for latency in the level-one (L1) cache itself.

1.1.3. The Caltech MiniMIPS Processor

The Caltech MiniMIPS processor [11] serves as a convenient example of a high-performance microprocessor. From the point of view of the cache system, the only thing about the MiniMIPS processor that is "unconventional" is its asynchronous interface. The MiniMIPS processor is designed very aggressively with respect to its cache subsystem; the processor is targeted at 300 MHz operation (in 0.6μm Scalable CMOS) with an L1 cache access time of three nanoseconds (this is the access time of the SRAM array itself, excluding the control logic presented in this work). This means that under the best possible conditions, it takes almost a full instruction cycle to fetch a new instruction from the cache.

The Caltech MiniMIPS processor architecture has been designed to take advantage of the two asynchronous design promises. In this Thesis, we shall demonstrate how we use low-latency design techniques to make possible a 300 MIPS single-scalar processor and how we design a cache subsystem for such a microprocessor in a reassuringly modular way.

1.2. Organization of Thesis

This Thesis consists of two major parts, organized in Chapters. In Chapter One, we describe the memory hierarchy of modern digital computers, and we present a high-level picture of a pipelined asynchronous cache architecture. The goal is to show systematically how to arrive at the final design from an initial specification, and we do this in three steps.

Step 1. A simple sequential specification for our cache is presented. This specification simply describes the input/output behavior of the cache—the final decisions on the ordering and implementation of the actions are left for later.

Step 2. We show how to decompose the sequential specification into multiple implementable Communicating Hardware Processes. This step in itself does not introduce pipelining or concurrency, but by specifying which processes are independent, we set the stage for the introduction of concurrent actions.

Step 3. Finally, we reorder the communication actions of the processes generated in Step 2. This introduces the desired concurrency.

Chapter Two applies the lessons of the first Chapter to a real-world example: The Caltech MiniMIPS processor cache system.

1.3. The Memory Hierarchy

In a von Neumann* computer, the central processing unit (CPU) carries out computations by executing a program stored in "memory," and the state of the computations is also stored in this memory. In all modern computers, a deep *memory hierarchy* may be identified, running from the fastest, most scarce (and most expensive on a per-bit basis) downwards, with implementation technologies in current vogue with associated approximate access times and sizes (in machine words) in parentheses:

- Processor registers (specially designed registers, $\approx$1 ns, about 2^5).
- Level-one (L1) cache (fast static RAM, 5 ns, about 2^{12}).
- Level-two (L2) cache (static RAM, 10–20 ns, commonly omitted, about 2^{14}).
- Level-three (L3) cache (static RAM, only in recent use, about 2^{18}).
- Main memory (dynamic RAM, 70 ns, about 2^{26}).
- Virtual memory (movable-head magnetic disk, 10 ms, about 2^{30}).

* This is in contrast with a "Harvard" machine that stores its program and data in separate memories. Since the von Neumann (or "stored-program") concept has come to dominate completely, a Harvard machine is contemporarily meant to be one that simply has independent data and instruction caches. Therefore it is not a contradiction to say that the MiniMIPS is both a von Neumann and a Harvard machine[2].

- Off-line storage (tape or optical, seconds to minutes, basically unlimited).

Clearly, there exists an inverse relationship between speed of access and amount of storage in any economical system. Many computer systems do not implement all the levels of the memory hierarchy indicated here, whereas other systems introduce more levels in the memory hierarchy by such methods as "near" and "far" memory accesses in multiprocessors or multicomputers. It is also important to emphasize that the implementation of the memory hierarchy is not entirely left to hardware. The (s)lower levels of the memory hierarchy tend to be implemented by system-level software. While we shall focus almost exclusively on hardware implementations of the highest levels of the memory hierarchy, the reader should bear in mind that many of the complications that we mention to occur in practical systems (e.g., exception handling mechanisms) are often due to attempts to implement efficiently the lower levels of the memory hierarchy. Expanding the memory hierarchy is a focus of much current research in computer systems.

1.3.1. Caches

In its most basic terms, a cache is a small, fast memory that is used to apply knowledge about a program's patterns of reference to memory in order to avoid the latency of often referring to memory locations stored too far down the memory hierarchy. In these terms, each one of the levels of the memory hierarchy can be fairly seen as a cache for the next level down the list.

The primary focus of this work is on the L1 cache subsystem. The L1 cache is especially interesting from a hardware design point of view because of its extreme demands on throughput and latency. The L1 cache needs to operate at the full throughput of the processor, and its latency is very important because the total average latency of the memory system is dependent mostly on the latency of the L1 cache.

The cache design that we develop does not attempt to help along the L1 cache latency by using a prefetch buffer (except, in a sense, in combination with the MiniMIPS branch prediction mechanism). Rather than that, we face the problem head-on and use a low-latency asynchronous design approach and internal pipelining of the cache to minimize total latency while maintaining acceptable throughput. The performance of the cache memory array itself is taken as given.*

1.4. The Design Problem

To describe our algorithms, we shall use a language related to Hoare's Communicating Sequential Processes (CSP), named Communicating Hardware Processes (CHP) [13].

* The MiniMIPS cache memory array was designed by Mr. Andrew Lines [6].

In CHP, the program for a simple read-only memory would be

$$
\begin{array}{l}
MEM \equiv \\
*[\ MEM\_ADDR?addr;\ \ MEM\_DATA!mem[addr]\ ]\ .
\end{array}
$$

Simply stated, the purpose of a cache memory is to store often-used data in faster storage. We may write the program of a cache in CHP as:

$$
\begin{array}{l}
SIMPLECACHE \equiv \\
*[\ READ?addr; \\
\quad q := \ \mathit{find\_entry}(addr); \\
\quad [\ q = \text{"}not\ \ found\text{"} \ \longrightarrow \ MEM\_ADDR!addr; MEM\_DATA?x; \\
\qquad\qquad DATA!x, cache\_entry[\mathit{refill\_policy}(addr)] \ := \ x \\
\quad [\!]\ q \neq \text{"}not\ \ found\text{"} \ \longrightarrow \ DATA!cache\_entry[q] \\
\quad] \\
]
\end{array}
$$

This program searches for an entry corresponding to the requested address *addr*—this is done by the function *find_entry*. If *find_entry* finds a match, that value stored at the matching location of the cache array is immediately returned to the requestor. If not, the address is forwarded to the main memory system and the value found in main memory is returned to the requestor at the same time as it is stored in the cache array for future use. What remains is to specify the caching policy, set by *refill_policy* and maintained by *find_entry*. It should be clear that if we insert the *SIMPLECACHE* between its user (the process that communicates on *READ* and *DATA*) and the *MEMORY*, the behavior from the point of view of a delay-insensitive user will be unchanged. By implementing the smaller *cache_entry* array with faster and more expensive memory than the larger *mem* array, we may thus hope to improve average performance without changing the functionality.

We are going to restrict ourselves mostly to caches that are *direct-mapped,* i.e., caches for which the function *find_entry* can be implemented as a tags comparison from a single array position, *cache_entry* can be implemented as a memory read* from a single array position, and *refill_policy* is the identity function. Direct-mapped cache systems as a rule offer slightly lower hit rates than set-associative cache systems—we pick a direct-mapped cache architecture for simplicity. Furthermore, we do not consider the possibility of improving performance by sometimes *not* writing the value read from memory to the cache array. On the other hand, we shall refer to designs that read more than one entry (or cache *line*—the "atomic" unit of cache storage associated with a single tags entry) from memory on refills. The unit of data read in on refills is called a cache *block;* this must be at least one

* We shall interchangeably use the terms "load" and "read" as well as the terms "store" and "write" since we are describing our design in the context of load/store ("RISC") architectures.

line, but it may be chosen larger to improve performance. Finally, shared-memory issues that arise when multiple independent writers are accessing the same memory are outside the scope of this Thesis. (In the MiniMIPS processor, these issues are handled by the use of "uncacheable" memory.)

The addition of the ability to write to the memory complicates things somewhat, so that now

$$
\begin{array}{l}
MEMORY \equiv \\
*[\ MEMOP?m, MEM\_ADDR?addr; \\
\quad [\ m = \text{"}read\text{"} \longrightarrow MEM\_DATA!mem[addr] \\
\quad [\!]\ m = \text{"}write\text{"} \longrightarrow MEM\_WDATA?mem[addr] \\
\quad] \\
\ ]
\end{array}
$$

and

$$
\begin{array}{l}
CACHE \equiv \\
*[\ CACHEOP?c, ADDR?addr; \\
\quad [\ c = \text{"}read\text{"} \longrightarrow \\
\qquad q := find\_entry(addr) \\
\qquad [\ \ q \neq \text{"}not\ found\text{"} \longrightarrow DATA!cache\_array[q] \\
\qquad [\!]\ q = \text{"}not\ found\text{"} \longrightarrow MEM\_ADDR!addr, MEMOP!c; MEM\_DATA?x; \\
\qquad\qquad\qquad DATA!x, cache\_entry[index(addr)] := x \\
\qquad] \\
\quad [\!]\ c = \text{"}write\text{"} \longrightarrow WRITE?x, MEMOP!c, MEM\_ADDR!addr; \\
\qquad cache\_entry[index(addr)] := x, MEM\_WDATA!x \\
\ ]]\ .
\end{array}
$$

We have chosen a simple *write-through* cache architecture for simplicity (it is also required for the MiniMIPS by the MIPS Level 1 Instruction Set Architecture (ISA) [4]). The simplicity of the write-through cache is based on the invariant that after each cache operation has completed, the contents of the cache mirrors main memory, i.e., that any datum in the cache is identical to that located at the address in main memory cached by that datum. This allows, for instance, making space in the cache for new data by simply discarding old data, rather than having to remember to write back dirty lines. We shall refer to this invariant as the *write-through cache invariant.*

1.5. Pipelining the Cache

The basic impediment to high cache performance (other than rudely inefficient user programs) is the relatively long latency of the static RAM array forming the cache core. A common approach for dealing with unacceptably high latencies is to add pipelining, thereby allowing multiple operations to be in progress simultaneously. Unfortunately, pipelining cannot be applied directly in this case because of the possibility of data dependencies between successive cache operations—dependencies that are particularly troublesome since the cache potentially involves shared state between these operations.

The extra work necessary to pipeline a simple asynchronous cache system is the primary subject of this Thesis. The general approach is simple: We decompose the control process that we have wrapped around the cache core into two parts, one that issues the commands to the cache core (we call this the L process) and one that reads and acts upon the results of the cache reads (we call this the R process). We now have a loop consisting of the L, $CORE$, and R processes (see Figure 1).

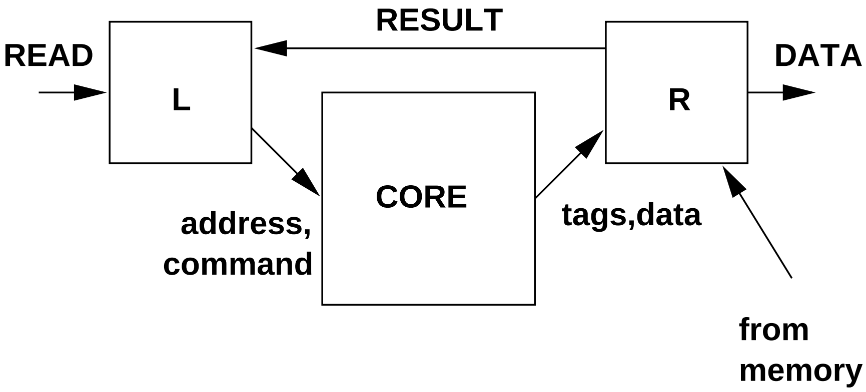

Figure 1. Simplified view of the loop consisting of the two control processes and the cache core. The L process receives commands from the CPU, processes them, and issues commands to the cache core. The R process receives results from the cache core and sends the results back to the CPU, as appropriate. The R process is also responsible for communicating with main memory.

1.5.1. Logical and Physical Pipelining

By examining the way a four-phase handshake is converted to a circuit implementation, it is seen to be possible to move the production of the result early enough in the handshake (see, e.g., the final section of this Thesis on low-level implementation details) that the pipelining comes at essentially zero forward latency cost compared to *unpipelined* logic.* (And of course it reduces the backward latency since buffering tends to decouple the handshakes in the system.) This means that

* The *forward* latency of a unit is the time taken for it to produce its outputs given that all its inputs have arrived; the *backward* latency is the time taken for it to acknowledge its inputs [19].

we are free to insert pipelining anywhere we deem it suitable—in the MiniMIPS this (for the most part) means that any logic that cannot comfortably be done in a single stage of CMOS logic (due to fanin or fanout restrictions or other physical considerations) is split and converted into multiple *pipeline* stages. In other words, almost *every* stage of logic is a separate pipeline stage. This is what we refer to as *physical pipelining.* Under certain conditions (which amount to the maintenance of deterministic operation), adding physical pipelining or *slack* to a system does not change its input-output behavior; in a system that satisfies these conditions it is thus unnecessary to involve the physical pipelining in the highest-level specifications, which makes it possible to defer some of the performance-enhancing design decisions. (Certain sufficient conditions for this to be true are easily verifiable, e.g., the lack of *probes* [15],[10],[3].)

In synchronous systems, the process of pipelining may introduce hazards. The reason for this is that instead of the acknowledgement signals used in QDI asynchronous design, the clock is abused as a global synchronization signal on the algorithmic level, at great expense to the modularity of the design. In the QDI style, the hazards do not appear at the level of physical pipelining. Instead, they appear at the level of CHP description when communication actions in a pipeline are out of step—i.e., when the number of outstanding writes into the pipeline exceeds the number of completed reads by more than one.* This we term *logical pipelining.* The addition of logical pipelining is the addition of concurrency to a system; this potentially changes the input/output behavior of the system and can be made visible in a high-level specification, permitting designers clearly to argue the concurrency issues at this level rather than at the level of individual gates. The abstraction barrier engendered by the separation of logical and physical pipelining is one of the most important properties of our style of QDI design.†

1.5.2. A Few Words on Pipelining and CHP

While the pipeline stage handshakes (i.e., synchronization behavior expressed in terms of handshaking expansions, "HSE"[13]) used throughout the design were

---

* Equivalently, we could say that there are multiple *tokens* in the pipeline at once. Even when it is not quite the case that one end of the pipeline is several full handshakes ahead of the other end, pipelining still helps performance by overlapping the different phases of a single handshake.

† It would not be impossible to achieve the same separation of concerns in a synchronous system. However, since it is usually not necessary to use handshakes in synchronous systems, this opportunity is often overlooked in favor of the "simpler" (but much less modular) approach of computing the information that might be carried by handshake signals with extra logic. Many clock-driven systems do use handshake signals when standard interfaces are to be determined or for communication over longer distances.

all chosen from a set of only three (and the overwhelming majority of the processes used just one of those), some of these processes communicated in unusual ways. It is a easy to realize that communication in QDI asynchronous systems is easiest to handle when it is unconditional, i.e., when a process executes an endless loop of

$$*[L \ ; \ R] \ ,$$

where L and R may be complex communication actions involving functions of data values, etc., but where the communications that occur are always on the *same* channels. The reason this is so is that the completion and acknowledgement networks involved reduce to simple collections of Muller C-elements rather than having the more complex logic needed to decide whether or not to acknowledge certain inputs dependent on data values.

The complications necessary to bring about conditional communications are covered in [7]. The cache controllers in the MiniMIPS presented ample opportunity to use processes with complex conditional communication patterns. We introduce two new CHP constructs better to explain these communication patterns: the *value probe* and the *channel peek.*

1.5.2.1. The value probe

We write the value probe as

$$\overline{A, B : P(A, B)}$$

which consists of a list of channel references A, B,... separated by a colon from a predicate on the values on those channels and any other program variables. If any of the channels in the list of channels is not yet defined, the value of the probe is **false**. The communication from the point of view of the sender does not complete. Note that the syntax is defined so that

$$\overline{A} \ \equiv \ \overline{A : \mathbf{true}} \ .$$

The value probe is usually used for convenience to group together control values on a channel. It is equivalent to using multiple channels.

1.5.2.2. The channel peek

We also introduce the channel peek

$$A¿x \ ,$$

which has the same semantics as the receive $A?x$ except that the value is not removed from the channel and the communication does not complete from the point of view of the sender. Note that it *does* block until there is a defined value on the channel, as opposed to the value probe. The channel peek is used to avoid reading and storing values that are going to be used repeatedly.*

* The syntax used for these additions to the CHP language was suggested by Mr. Matt Hanna.

1.5.3. Pipelining the Cache Loop

Each element in the cache loop (including the cache core) is made highly pipelined in the physical sense. As long as the control program executes cache operations in sequence with the associated refill/use decisions, however, this pipelining is effectively wasted. We remedy this by adding logical pipelining of the actions at the CHP level. As in the other parts of the MiniMIPS design, here we find that the low-level physical pipelining is almost orthogonal to the logical pipelining of data that occurs when we do several things at once at the algorithmic level.

We shall investigate the pipelined cache design in two stages: First we shall decompose the cache by factoring out a *CORE* process, corresponding to Mr. Lines' static RAM array. Second, we (without motivation) decompose the cache control into L and R processes, and then we shall finally introduce the pipelining by allowing the L–$CORE$–R loop to process more than one operation at a time (whereupon our decision to decompose into L and R will become motivated).

1.5.4. Decomposing the Cache

We begin by decomposing the sequential program given in the previous section for *CACHE* by extracting the cache core array itself. The programs assume that the address *addr* can be partitioned into two parts, $\mathit{offset}(\mathit{addr})$ and $\mathit{tag}(\mathit{addr})$. For our immediate purposes, the nature of these functions is not important, but conventionally, the offset part is some number of the least significant bits of the address while the tag part is the remaining most significant part of the address—we shall use the notations $\mathit{addr.offset}$ and $\mathit{addr.tag}$ for clarity. The choice of these functions affects performance parameters such as hit rate, line contention, and convenience of implementation.

A reasonable program for the cache *CORE* is (Note that $\mathit{addr.tag}$ is unused except for the refill and write cases.)

$$
\begin{array}{l}
\mathit{CORE} \equiv \\
*[\ \ CC?c, CA?\mathit{addr}; \\
\quad \mathit{offset} := \mathit{addr.offset}; \\
\quad [c = \text{"}\mathit{read}\text{"} \longrightarrow \ \ \mathit{RTAG}!\mathit{tags}[\mathit{offset}], \mathit{CDATA}!\mathit{data}[\mathit{offset}] \\
\quad [\!]\, c = \text{"}\mathit{write}\text{"} \longrightarrow \ \ \mathit{tags}[\mathit{offset}] := \ \ \mathit{addr.tag}, \mathit{CW}?\mathit{data}[\mathit{offset}] \\
\quad [\!]\, c = \text{"}\mathit{refill}\text{"} \longrightarrow \ \ \mathit{tags}[\mathit{offset}] := \ \ \mathit{addr.tag}, \mathit{CW}?\mathit{data}[\mathit{offset}] \\
\quad] \\
\ ] \quad .
\end{array}
$$

1.5.5. The cache control

By factoring out the cache *CORE*, the remainder of the cache becomes a single control process, which we call *CONTROL*, and write as

$$
\begin{array}{l}
CONTROL \equiv \\
*[\ CACHEOP?c, ADDR?addr; \\
\quad [\ c = \text{"}read\text{"} \ \longrightarrow \\
\qquad CC!\text{"}read\text{"}, CA!addr; \\
\qquad RTAG?t, CDATA?c; \\
\qquad [\ \ t = addr.tag \ \longrightarrow \ DATA!c \\
\qquad [\!]\ t \neq addr.tag \longrightarrow \ MEM\_ADDR!addr; \ MEM\_DATA?x; \\
\qquad\qquad\qquad\qquad DATA!x; CW!x, CC!\text{"}write\text{"}, CA!addr \\
\qquad] \\
\quad [\!]\ c = \text{"}write\text{"} \ \longrightarrow \ WRITE?x; \\
\qquad CC!\text{"}write\text{"}, CA!addr, CW!x \\
\quad]] \ .
\end{array}
$$

The reader will recognize this as the process quotient of $CACHE$ over $CORE$.

1.5.6. Factoring out the L process

After extracting the cache $CORE$, the cache control process $CONTROL$ is split into two parts, one (called L) concerned with issuing commands to the $CORE$ and the other (called R) concerned with reading the results of the commands, including tags comparisons.

The purpose of the L process is to issue commands to the cache core and to copy the write data to the main memory. It corresponds to the generation of addresses and commands into the *cache_array* entry in the sequential specification.

By referring to the sequential specification, we see that the addresses and commands generated into the cache core depend on the results of previous operations; since the *in_cache* function is not implemented in process L, we introduce a channel $RESULT$ that informs L. In order to know what to do after completing a core operation, it reads the result of the operation (if it was a read) on the $RESULT$ channel, running from process R. A few extra channels have been introduced in order to copy the cache command to inform the R process about it.*

$$
\begin{array}{l}
L \equiv \\
\ \ \mathbf{enum}\ \{\mathbf{false}, \mathbf{true}\}\ \mathbf{channel}\ RESULT; \\
*[\ CACHEOP?c, ADDR?addr; \ \ CC!c, CA!addr, RC!c, RA!addr; \\
\quad [\ c = \text{"}read\text{"} \ \longrightarrow \ RESULT?r; \\
\qquad [\ r \ \longrightarrow \ \mathbf{skip} \\
\qquad [\!]\ \neg r \ \longrightarrow \ CC!\text{"}refill\text{"}, CA!addr, CW!(MEM\_DATAL?) \\
\qquad]
\end{array}
$$

* In the future, we shall assume the existence of such channels without explicitly introducing them except where necessary to resolve ambiguities. Similarly, we shall make use of C-style declarations when this adds to the comprehensibility.

```
        [] c = "write"  ⟶  WRITE?x;  CW!x, MEMWRITE!x
        ]
    ]
```

In this process, the channels CC, CA, and CW refer to the cache core operation, address, and write data channels, respectively.

1.5.7. The R process

The remaining part of the sequential program for the $CACHE$ consists of reading the results from the cache core, computing the tags comparison result in_cache, and deciding on whether or not to refill. This is handled by the R process, which compares tags and informs process L of the result.

Finally, to complete the correspondence between the sequential cache and the decomposition, we are left with the channels to main memory for refilling and writing. There seems to be no reason to choose either one of the two processes L or R to handle these communications, but for reasons that will become apparent only after the cache has been pipelined, we send the commands to write to the memory through the R process. This is strictly necessary (even in the pipelined case) only for cache refills, but we control the memory writes from R as well to avoid undue channel proliferation.

```
R ≡
 enum {false, true} channel RESULT;
*[ RC?c, RADDR?addr;
    [ c = "read"  ⟶  hit := (RTAG? = tag(addr) );
      RESULT!hit,
      [ hit  ⟶  DATA!(CDATA?)
      [] ¬hit  ⟶  CDATA?_, MEM_ADDR!addr;
                  DATA!(MEM_DATAR?)
      ]
    [] c = "write"  ⟶  MEM_ADDR!addr
    ]
]
```

To verify the correctness of the given collection of processes, we first note that the system is trivially deterministic since there are no probes. Since there is also no pipelining, we may start, e.g., from process L and expand the communication actions to the $CORE$ and R processes as function calls. Doing this as an exercise, we see that the parallel composition $L \parallel CORE \parallel R$ is equivalent to the sequential specification from the previous section, and thus trivially implements its specification.

1.5.8. Introducing pipelining

In its simplest form, pipelining the cache corresponds to issuing two (or more) commands to the cache core from L before reading the result of the first command from R. In this way, the cycle of L–$CORE$–R is split so that two instructions can be in different stages of execution. Since the asynchronous datapath is already pipelined at the low level (to an extent far exceeding a two-token capacity), adding the second instruction carries nearly no cost (in contrast to the non-zero cost of adding pipelining to a synchronous system, expressed in the cost of additional latches and delay allowances for component variation, clock skew, data dependence, and temperature uncertainties) [7]. Since the pipeline stages operate as soon as data is available and become ready to operate on the next operands as soon as the data has been sent out, there is no requirement that the pipeline stages operate "in phase" with each other. This means that for an outside process connecting to L and R, the actual amount of pipelining present is not visible. Only the number of tokens injected into the pipeline and not yet read can affect the behavior of the environment. This leads to a separation of concerns, allowing low-level performance optimizations (physical pipelining) to be considered separately from algorithmic optimizations.

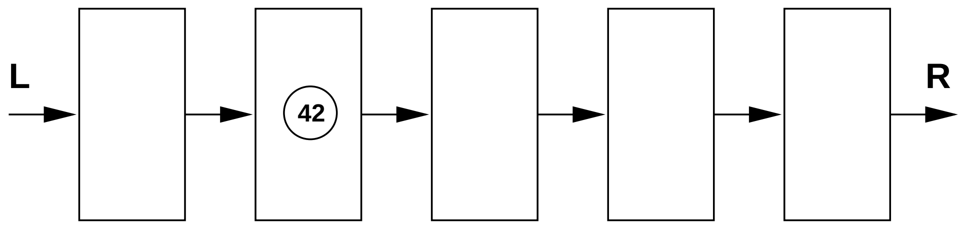

Figure 2. Pipelining in an asynchronous system. In the figure, an asynchronous pipeline is built out of five stages but is made to hold only one token. This system is physically pipelined but not logically pipelined.

In order to add successfully an additional token to the cache pipeline loop, we need to maintain the given sequential specification. The changes that have to be made can be deduced by noticing that the variables in the CHP programs seen so far no longer need to hold only a single value, but rather need to maintain as many values of state as there are tokens in the cache pipeline. In the L and R processes, this can be handled by introducing buffers that hold old values until it is certain they will no longer be needed. We shall add further buffers to the L process to avoid having to modify the $CORE$.

1.5.8.1. Initial modifications

In the unpipelined decomposition given above, the $RESULT$ channel is only used to communicate the result of the cache hit test on reads. This means that in a pipelined implementation, both the L and R processes would have to maintain enough state to deduce whether or not there was a cache read of which to read the $RESULT$. It simplifies every level of decomposition to communicate on the $RESULT$ channel for each cache core operation. We also move the $RESULT$ communication earlier. The transformed code for L becomes

$$
\begin{array}{l}
L \equiv \\
\quad \textbf{enum } \{\textbf{false}, \textbf{true}\} \textbf{ channel } RESULT; \\
*[\;\; CACHEOP?c, ADDR?addr, RESULT?r; \\
\quad\; CC!c, CA!addr, RC!c, RA!addr; \\
\quad\; [\;\; c = \text{"}read\text{"} \;\; \longrightarrow \\
\quad\quad\; [\;\; r \;\; \longrightarrow \;\; \textbf{skip} \\
\quad\quad\; [\!]\;\; \neg r \;\; \longrightarrow \\
\quad\quad\quad\quad CC!\text{"}refill\text{"}, CA!addr, CWRITE!(MEM\_DATAL?) \\
\quad\quad\;] \\
\quad\; [\!]\;\; c = \text{"}write\text{"} \;\; \longrightarrow \;\; WRITE?x; \;\; CWRITE!x, MEMWRITE!x \\
\quad\;] \\
\;\;]\;\; .
\end{array}
$$

Of course, we need to modify R as well (to "prime" $RESULT$), so that

$$
\begin{array}{l}
R \equiv \\
\quad \textbf{enum } \{\textbf{false}, \textbf{true}\} \textbf{ channel } RESULT; \\
RESULT!\textbf{true}; \\
*[\ RC?c, RADDR?addr; \\
\quad [\ c = \text{"}read\text{"} \longrightarrow hit := (RTAG? = tag(addr)); \\
\qquad RESULT!hit, \\
\qquad [\ hit \longrightarrow DATA!(COREDATA?) \\
\qquad [\!]\ \neg hit \longrightarrow CDATA?\_, MEM\_ADDR!addr; \\
\qquad\qquad\qquad DATA!(MEM\_DATAR?) \\
\qquad] \\
\quad [\!]\ c = \text{"}write\text{"} \longrightarrow RESULT!\textbf{true}, MEM\_ADDR!addr \\
\quad [\!]\ c = \text{"}refill\text{"} \longrightarrow \textbf{skip} \\
\quad] \\
]\ .
\end{array}
$$

With these modifications, the correspondence between the sequential specification and the composition $L \parallel CORE \parallel R$ is unaffected.

1.5.8.2. Adding pipelining

We now add logical pipelining (some physical pipelining has already been added by the process decomposition). We do this simply by communicating on the $RESULT$ channel before receiving anything in the R process.

The correctness problem alluded to in the previous section now becomes apparent. The pipelining transformation may add a write-after-write (WAW) hazard (with respect to the $CORE$ process). The main problem occurs when cache writes are supported. Assume that a load from a memory location is immediately followed by a store to the same location. If the load hits in the cache, the store will update the value in the cache and in the main memory, and all will be well. If, however, the load misses, the store will already have been issued to the cache core (since the cache is pipelined). Since the cache is write-through, the write will update the cache line immediately without checking any further conditions. Meanwhile, the refill of the line has begun, and the final state will be that the value that was written to the cache core will be overwritten by the old value from main memory! In the final state, an incorrect value will be found in the cache, but the value in memory will be—inconsistently—correct. Clearly, this violates the write-through cache invariant of Section 1.4.

We note that the value communicated on $RESULT$ no longer corresponds directly to whether or not there was a cache hit or miss—in fact, this channel has turned into a control channel for the L process, telling it exactly what to do; to finalize this part of the transformation, we change the L process so that the communcations on $RESULT$ match those on the cache core data channels. We solve the write-after-write problem by introducing the capability to repeat the cache instruction right after a cache miss. Now, a cache miss followed by a cache write to the same address will result in the following trace: Cache miss, refill from memory (and satisfaction of the register file's demand for the data), cache core write, refill overwrite (leading to the inconsistent state), and a second cache core write from the repeated store (cleaning up the inconsistency).

Compared to the sequential specification, the addition of pipelining introduces data hazards because it changes the $CACHE$ program such that operations from two consecutive passes through the program's loop are deterministically interleaved. As usual in cases like this, if the two passes refer to disjoint data, there can be no correctness problem. If, on the other hand, the operations refer to the same datum, a hazard is introduced. There are two basic ways to solve this problem—either we do not dispatch operations that may fall prey to data hazards, or else we dispatch the operations speculatively and check later whether or not a consistency problem occurred. We choose the latter alternative. ("We shoot first and ask questions later.")

The CHP corresponding to this is*

$$
\begin{array}{l}
L \equiv \\
\quad \textbf{enum}\ \{\text{"}\mathit{go}\text{"}, \text{"}\mathit{refill}\text{"}, \text{"}\mathit{repeat}\text{"}\}\ \textbf{channel}\ RESULT; \\
*[\ RESULT?r; \\
\quad [\ r = \text{"}\mathit{go}\text{"} \ \longrightarrow\ oldc := c,\ \ oldaddr := addr; \\
\quad\quad CACHEOP?c, ADDR?addr; \\
\quad [\!]\ r = \text{"}\mathit{repeat}\text{"}\ \wedge\ oldc = \text{"}\mathit{write}\text{"}\ \longrightarrow \\
\quad\quad\quad c := \text{"}\mathit{core\_write}\text{"},\ \ addr := oldaddr \\
\quad [\!]\ \textbf{else}\ \longrightarrow\ c :=\ oldc,\ \ addr := oldaddr \\
\quad]; \\
\quad [\ r = \text{"}\mathit{go}\text{"}\ \vee\ r = \text{"}\mathit{repeat}\text{"}\ \longrightarrow\ CA!addr, CC!c \\
\quad [\!]\ r = \text{"}\mathit{refill}\text{"}\ \longrightarrow\ CC!\text{"}\mathit{refill}\text{"} \\
\quad] \\
\ ]\ .
\end{array}
$$

* We have omitted the details of how the write value gets from the register file to the cache core. It needs to be repeated on a *repeat* operation.

(Note that refills are never repeated and that writes are never refilled.) The introduction of *oldc* and *oldaddr* is necessary to store the previous operation, so that it may be repeated. For instance, if a write is issued, then a read that misses, the $RESULT$ channel will carry a *repeat* token, and at that point L needs to redispatch the write.

On the R side, we could choose to cancel and repeat the instruction following each cache read miss. This would lead to correct behavior, but we can do better. We notice that a write instruction following a cache miss does not need to be repeated unless it refers to the same cache block (and will thus be overwritten on the next cycle). Conversely, why repeat a cache *read* following a miss? There is no possibility of a write hazard on a read following a read. On the other hand, there is an efficiency issue in this case. Consider what happens when two memory references to adjacent memory locations occur in sequence. If the first one misses, the second is highly likely to do so as well. However, if the two memory locations are in the same block, the second miss will already have been satisfied by the refill of the first and the second datum can be read out of the cache core after the first has been refilled rather than being refilled from main memory, a significant optimization.

The modularity of the L–$CORE$–R pipeline is shown by the relative ease with which we may change the behavior of the cache. Modifications to R that change caching policies, etc., are localized to this process.

The cases of non-conflicting writes and double refills may both be eliminated by adding a comparison between the block of the missed instruction and that of the next instruction. As a final optimization, we call writes that are repeated "core writes" so that they do not need to be repeated to main memory. (We take advantage of the inconsistency generated by the out-of-order writes to memory and cache.) The R process implements this as shown on the next page.

$R \equiv$

$\quad$ **enum** $\{\text{"}go\text{"}, \text{"}refill\text{"}, \text{"}repeat\text{"}\}$ **channel** $RESULT;$

$miss{\downarrow}; oldaddr := \_;$

$RESULT!\text{"}go\text{"}; RESULT!\text{"}go\text{"};$

$*[\ oldaddr := addr; RC?c, RADDR?addr;$

$\quad [\ c.op = \text{"}read\text{"} \longrightarrow h := (RTAG? = tag(addr))$

$\quad [\!]$ **else** $\longrightarrow$ **skip** $],$

$\quad [\ c.op = \text{"}write\text{"} \longrightarrow MEM\_ADDR!addr$

$\quad [\!]$ **else** $\longrightarrow$ **skip** $],$

$\quad [\ miss \longrightarrow ma := (block(addr) = block(oldaddr))$

$\quad [\!]$ **else** $\longrightarrow$ **skip** $];$

$\quad [\ \neg miss \wedge$

$\qquad (c.op = \text{"}write\text{"} \vee c.op = \text{"}core\_write\text{"}) \longrightarrow$

$\qquad\qquad RESULT!\text{"}go\text{"}, miss{\downarrow}$

$\quad [\!]\ (miss \wedge \neg ma) \wedge$

$\qquad (c.op = \text{"}write\text{"} \vee c.op = \text{"}core\_write\text{"}) \longrightarrow$

$\qquad\qquad RESULT!\text{"}go\text{"}, miss{\downarrow}$

$\quad [\!]\ miss \wedge c.op = \text{"}write\text{"} \wedge ma \longrightarrow$

$\qquad\qquad RESULT!\text{"}repeat\text{"}, miss{\downarrow}$

$\quad [\!]\ c.op = \text{"}refill\text{"} \longrightarrow RESULT!\text{"}go\text{"}, miss{\downarrow}$

$\quad [\!]\ c.op = \text{"}read\text{"} \wedge h \longrightarrow$ // hit

$\qquad\qquad RESULT!\text{"}go\text{"}, miss{\downarrow}, DATA!(MEM\_DATAR?)$

$\quad [\!]\ \neg miss \wedge c.op = \text{"}read\text{"} \wedge \neg h \longrightarrow$ // $miss$

$\qquad\qquad\quad RESULT!\text{"}refill\text{"},$

$\qquad\qquad\quad MEM\_ADDR!addr,$

$\qquad\qquad\quad miss{\uparrow}$

$\quad [\!]\ miss \wedge c.op = \text{"}read\text{"} \wedge \neg h \wedge ma \longrightarrow$

$\qquad\qquad\quad RESULT!\text{"}repeat\text{"},$ // $miss\ in\ delay\ slot$

$\qquad\qquad\quad miss{\downarrow}$

$\quad [\!]\ miss \wedge c.op = \text{"}read\text{"} \wedge \neg h \wedge \neg ma \longrightarrow$

$\qquad\qquad\quad RESULT!\text{"}refill\text{"}, DATA!(MEM\_DATAR?)$

$\qquad\qquad\quad MEM\_ADDR!addr, miss{\uparrow}$ // $pipelined\ miss$

$\quad]$

$]\ .$

1.5.9. Correctness argument

In our cache design, we have without comment sidestepped a progress pitfall. One might think that, given that the cache has the capability of reissuing requests internally, the best way to handle a cache miss would be to refill from memory into the cache core and then reissue the original lookup to the cache core, knowing that the miss has been satisfied, hence is guaranteed to generate a cache hit. This is not the case. Consider a program that consists of a single read from data memory. That would become, perhaps, a cache miss. The cache miss would lead to a memory refill that would complete, and the data would be available for the cache core to refill. The L process, however, would still be waiting for the nonexistent second memory reference to come through, since it is not supposed to handle the refill until *after* the next instruction has completed—in other words, the system would deadlock. We solved this problem by making cache refills also bypass the cache directly to the processor busses and then to the register file.

The other progress condition is that we avoid livelock given that we have introduced the *repeat* mechanism. Is it possible that the same instruction could be repeated over and over again? The answer to this is no. If we follow an instruction through the cache, we see that we may repeat both kinds of instructions: reads and writes. A read that is repeated will be processed after a *refill*, which means that it will either be refilled itself (if it misses), or that it will be read out correctly (if it hits). In either case, progress is made since it cannot turn into a *repeat*.

The reader will see by inspecting the program that the operations carried out will be properly matched as reads and writes at every process in the loop, and this is enough to demonstrate that the actions correspond to those of the sequential specification. This could be proved formally by introducing ghost variables into the programs. We have met the sequential specification by executing two consecutive passes partly in parallel and redispatching conflicting operations. While this does not maintain the write-through cache invariant strictly at the end of each cache operation, it does guarantee that data in the cache eventually mirrors the main memory and that states in which the invariant is violated are hidden from the outside interface.

1.6. Extending the pipelined cache scheme

The decomposition of the cache into the L and R processes introduces *speculation.* The amount of speculation is, naturally, dependent on how many tokens are injected into the L–$CORE$–R loop. In the simple case described so far, the speculation has been on whether or not a cache miss has happened. If the speculation fails, a "hardware exception handler" is invoked to handle the problem.

The speculation mechanism can be used in general to take advantage of "common case" operations. Implicitly, we may assume that cache reads that hit in the cache and plain memory writes form the majority of the operations. If we do this, then the cache architecture will be seen to be easily adapted to managing these cases at (almost) the same speed as before while we add other functionality. Taking hints from the MIPS Level 1 ISA, a few of the features that can be added in this way are:

- Multiple writeback units to increase tolerance to cache misses.
- Partial-word loads and stores.
- Virtual memory support through memory read and write exceptions and memory address translation via a translation lookaside buffer (TLB).

We shall show how each of these features may be added to the memory system by modifying the cache processes L and R without significantly affecting the complexity of handling the common cases of cache hit loads and word-sized stores. Such a design would efficiently handle the entire MIPS 1 ISA, and other features might also be contemplated.

1.6.1. Adding slack in the memory system

For a data cache (and for a system that can handle multiple threads of execution, perhaps speculatively, also for the instruction cache), it is not always necessary to satisfy loads in order. We have designed a pipelined cache system that would be capable of handling one load while simultaneously handling a cache miss. This is done by splitting the datapath associated with the R process in two. We alternate between the two output sections, 0 and 1. A cache miss on an access destined for output section 0 will allow the second token (in the pipelined cache) to exit via output section 1 and vice versa. This scheme is described in more detail in [16]. In order to add to this scheme the capability to satisfy more than one lookup after a cache miss, the amount of pipelining in the cache would have to be increased. This should be straightforward, but it has not yet been explored in any detail.

1.6.2. Adding partial-word operations

Partial-word operations are memory operations that do not operate on a full machine word. For instance, (ASCII) string operations require manipulation of byte-sized data (`char` data in the C programming language). Such operations are difficult to implement easily in a way that does not cause the more common word-sized operations to be slowed down; indeed, the presence of partial-word memory operations has been considered the Achilles' heel of the MIPS ISA.

In our asynchronous cache architecture, partial-word operations are implemented using a "merge buffer" that allows partial-word cache writes to be handled correctly without adding complexity to the cache core itself. Partial-word cache reads are implemented using an alignment shifter. In our discussion, we shall restrict ourselves to halfword reads and writes. Reads and writes can be given still finer granularity without qualitatively changing the pipelined cache mechanism.

We are going to illustrate how to add exactly the mechanism specified by the MIPS 1 ISA. Partial-word writes are specified to be write-through if they hit in the cache and write-around otherwise, avoiding the need to read in the cache line prior to writing the data. While this may have been a simplification for the pipelined synchronous MIPS cache, it does not lead to a substantial savings for an asynchronous cache, and it would be easy enough to implement a true write-through cache capable of partial-word operations. This ability to handle partial-word operations without a common-case penalty is a significant advantage of the asynchronous implementation.

The merge buffer is

$$
\begin{array}{l}
MERGEBUF \equiv \\
\texttt{*[}\;\; OFFSET?x, CACHEDATA?a, REGDATA?b; \\
\qquad \texttt{[}\;\; x = \text{"}lo\text{"} \;\;\longrightarrow\;\; c := bit\_or(mask\_hi(b), mask\_lo(a)) \\
\qquad \texttt{[]}\;\; x = \text{"}hi\text{"} \;\;\longrightarrow\;\; c := bit\_or(mask\_lo(b), mask\_hi(a)) \\
\qquad \texttt{]}; \\
\qquad R!c \\
\texttt{]}\;\; .
\end{array}
$$

For high performance, we wish to maintain cache pipelining during partial word writes. This gives rise to a write-after-write problem complementary to that experienced with pipelined refills. Basically, a cache read following a partial-word write to the same word needs to be repeated since the write will not yet have taken effect in the cache core. The data correctness problem with respect to the data in the cache core is the same as for the original pipelined cache mechanism, except that the coherence is now destroyed by explicit writes to the cache core, rather than by missed read operations. The following fragments illustrate the mechanism:

$$
\begin{array}{l}
L \equiv \\
\;\; \textbf{enum}\;\; \{\text{"}go\text{"}, \text{"}refill\text{"}, \text{"}repeat\text{"}, \text{"}hww\_hit\text{"}\}\;\; \textbf{channel}\;\; RESULT; \\
\texttt{*[}\;\; RESULT?r; \\
\qquad \ldots \\
\qquad \texttt{[}\;\; \ldots \\
\qquad //\;\; go\;\; covers\;\; write - around\;\; case\;\; as\;\; well \\
\qquad \texttt{[]}\;\; r = \text{"}go\text{"} \;\;\longrightarrow\;\; \ldots \\
\qquad \texttt{[]}\;\; r = \text{"}hww\_hit\text{"} \;\;\longrightarrow\;\; CC!\text{"}core\_write\text{"} \\
\qquad \ldots \\
\qquad \texttt{]} \\
\;\; \texttt{]}
\end{array}
$$

$R \equiv$

$hwop{\downarrow}, miss{\downarrow}, oldaddr := \_\_;$ $\;$ // *initialize*

$RESULT!"go"; RESULT!"go";$

```
*[ oldaddr := addr; RC?c, RADDR?addr;
   [ c.op = "read"  ∨  c.op = "hw_write"
          ⟶  h := (RTAG? = tag(addr))
   [] else ⟶ skip ],
   [ c.op = "write"  ∨  c.op = "hw_write"
          ⟶  MEM_ADDR!addr
   [] else ⟶ skip ],
   [ miss ∨ hwop ⟶ ma := (block(addr) = block(oldaddr))
   [] else ⟶ skip ];
   [
   ...
   [] c.op = "hw_write" ∧ h ⟶ hwop↑, RESULT!"hw_hit"
   [] c.op = "read" ∧ hwop ∧ ma ⟶ hwop↓, RESULT!"repeat"
   ...
   [] c.op = "hw_read" ∧ h ∧ !hwop ⟶
         DATA!(shift(MEM_DATAR?), LSB(addr)), RESULT!"go"
   ...
   ];
   [ c.op ≠ "hw_write" ⟶ hw↓ [] else ⟶ skip ]
 ]
```

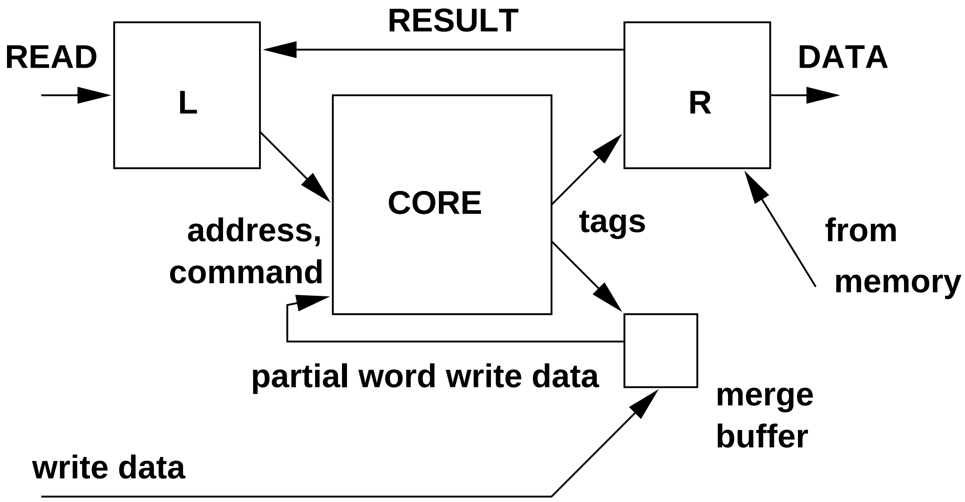

Figure 3. Schematic view of the cache loop modified to handle partial word operations.

1.6.3. Adding memory translation

In order to implement paged virtual memory, the current standard approach is to add a translation table from virtual memory addresses (each process's address space) to physical memory addresses. The contents of this translation table are usually maintained by an operating system (see, e.g., [18]).

Having each memory access by every user-level process processed by a software memory translation would be prohibitive. Commonly, part of the translation table is cached in a *translation lookaside buffer* (TLB) that maintains a small number (less than, say, 2^8) of virtual-to-physical mappings. Given a virtual address, the physical address is formed by combining the lower bits of the virtual address with the top bits of the virtual address translated via the TLB into the top bits of the physical address. The number of "lower bits" used by this algorithm determines the *page size* of the machine and the number of "upper bits" of the virtual address determines the size of the address space seen by a process (this need not be the same as the maximum size of physical memory).

Hardware memory translation is a substantial complication of the memory system. The main problem is how to handle the situation when a virtual address is presented to the memory system and that virtual address lacks a mapping in the TLB. This is usually* handled by an *exception,* i.e., the user's program state is saved and the operating system begins executing with the state of the machine being that which held exactly before the execution of the instruction that lacked a translation. This mechanism is called a precise exception. The "preciseness" of the exception is a necessity to be able to restart faulting instructions transparently without assistance from the user program.†

Adding exceptions to the memory system introduces a host of problems that can, if handled carelessly, have a substantial performance impact on the entire system. We are going to solve these problems by allowing the L process to speculate that the cache line looked up is the one containing the translated address. We allow the R process to check whether or not L's speculation was successful, just as in the case of the cache hit test.

The changes necessary to the pipelined cache system to implement virtual memory and exception support are very straightforward and do not affect the basic pipelined cache scheme. We add an exception information receive to L from the CPU writeback (so that the system can cancel writes that are not supposed to take place due to an already occurred exception):

---

* In some systems, primarily older CISC machines such as the Digital VAX, this mechanism is implemented in microcode within the CPU rather than by an operating system software routine. In such systems, the exception is raised only when the target physical page is not in memory and has to be "paged in" from magnetic storage. This does not qualitatively affect the nature of the problem.

† Although the MiniMIPS design does not support memory translation through a TLB, it should be clear that the main difficulty from the hardware design point of view is not the translation itself, but rather the implementation of the precise exceptions, due to the complex "sweeping up" of processor state that is required to be able to restart from exactly the same state. The reader may object that the implementation of interrupts in any processor has exactly the same implications. This is not the case. By their nature, interrupts may be deferred an arbitrary but finite time, and this may be used to stall the instruction fetch until all dirty state has been written back, while this is impossible with exceptions, where the effects of already issued (and partially executed) instructions have to be inhibited. The MiniMIPS *does* implement precise exceptions for other things, such as reserved instructions (to implement partial-word operations or other unimplemented instructions by emulating them in supervisor mode), and add overflows. The MiniMIPS exception mechanism is described in [11].

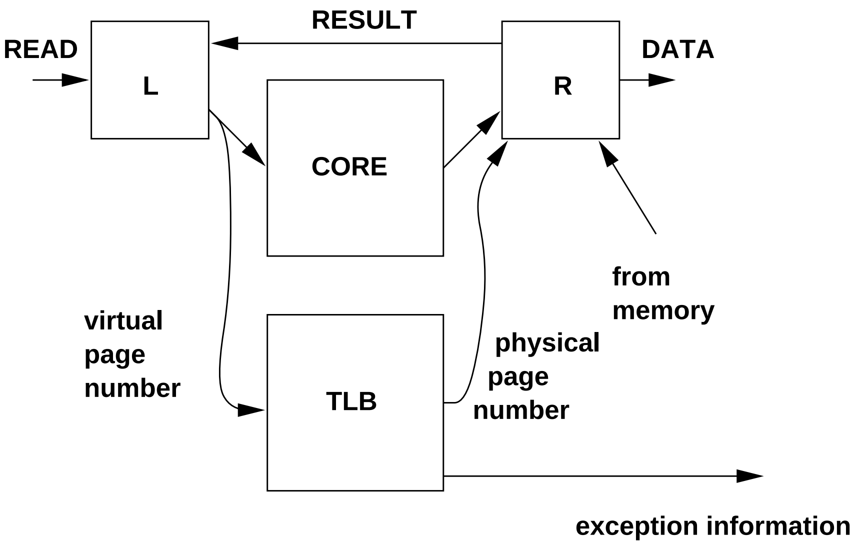

Figure 4. The pipelined cache architecture modified to support memory translation. The TLB lookup is done in parallel with the cache lookup. The read operation is only successful if both the physical page number hits in the TLB and the cache entry is valid. If no reference to the physical page number is present in the TLB, an exception is raised; if the cache line does not correspond to the referenced memory address, a cache miss is detected and refilled as in the untranslated cache.

$$
\begin{array}{l}
L \equiv \\
\quad \textbf{enum}\ \{\text{"}go\text{"}, \text{"}refill\text{"}, \text{"}repeat\text{"}\}\ \textbf{channel}\ RESULT; \\
*[\ RESULT?r; \\
\quad [\ r = \text{"}go\text{"} \longrightarrow oldc := c\ ;\ CACHEOP?c, ADDR?addr; \\
\quad [\!]\ r \neq \text{"}go\text{"} \wedge oldc = \text{"}write\text{"} \longrightarrow c := \text{"}core\_write\text{"} \\
\quad [\!]\ \textbf{else} \longrightarrow c := oldc \\
\quad]; \\
\quad [\ r = \text{"}go\text{"} \wedge c = \text{"}write\text{"} \longrightarrow WB?wb \\
\quad [\!]\ \textbf{else} \longrightarrow wb := \textbf{true}\ ]; \\
\quad [\ r = \text{"}go\text{"} \vee r = \text{"}repeat\text{"} \wedge wb \longrightarrow CA!addr, CC!c \\
\quad [\!]\ r = \text{"}refill\text{"} \longrightarrow COREC!\text{"}refill\text{"} \\
\quad [\!]\ \neg wb \longrightarrow \textbf{skip} \\
\quad] \\
]
\end{array}
$$

On the R side, we check for a missing TLB translation and forward this as exception information to the main CPU pipeline. We also must make certain not to issue writes to main memory if there is a TLB miss—the behavior on memory reads is immaterial since those instructions are going to be canceled in the CPU pipeline.

Further optimizations are possible; e.g., we could not repeat writes that will lead to TLB misses, or cancel reads after TLB misses. Due to the low frequency of TLB misses, such modifications are likely to be insignificant from the point of view of system performance.

$$
\begin{array}{l}
R \equiv \\
\;\;\mathbf{enum}\ \{\text{"}go\text{"}, \text{"}refill\text{"}, \text{"}repeat\text{"}\}\ \mathbf{channel}\ RESULT; \\
miss\!\downarrow; oldaddr := \_; \\
RESULT!\text{"}go\text{"}; RESULT!\text{"}go\text{"}; \\
*[\;\; oldaddr := addr; RC?c, RADDR?addr, TLBMISS?tlbm; \\
\quad\;\; [\;\; c.op = \text{"}read\text{"} \longrightarrow \;\; h := \;\; (RTAG? \;\; = \;\; tag(addr)) \\
\quad\;\; []\ \mathbf{else} \;\; \longrightarrow \;\; \mathbf{skip}\ ], \\
\quad\;\; [\;\; c.op = \text{"}write\text{"} \;\; \wedge \;\; \neg tlbm \;\; \longrightarrow MEM\_ADDR!addr \\
\quad\;\; []\ \mathbf{else} \;\; \longrightarrow \;\; \mathbf{skip}\ ], \\
\quad\;\; [\;\; miss \;\; \longrightarrow \;\; ma := (block(addr) \;\; = \;\; block(oldaddr)) \\
\quad\;\; []\ \mathbf{else} \;\; \longrightarrow \;\; \mathbf{skip}\ ]; \\
\quad\;\; [\;\; \neg miss \;\; \wedge \\
\qquad\;\; (c.op = \text{"}write\text{"} \;\; \vee \;\; c.op = \text{"}core\_write\text{"}) \longrightarrow \\
\qquad\qquad\qquad RESULT!\text{"}go\text{"}, miss\!\downarrow \\
\quad\;\; []\;\; (miss \;\; \wedge \;\; \neg ma) \;\; \wedge \\
\qquad\;\; (c.op = \text{"}write\text{"} \;\; \vee \;\; c.op = \text{"}core\_write\text{"}) \longrightarrow \\
\qquad\qquad\qquad RESULT!\text{"}go\text{"}, miss\!\downarrow \\
\quad\;\; []\;\; miss \;\; \wedge \;\; c.op = \text{"}write\text{"} \;\; \wedge \;\; ma \quad \longrightarrow RESULT!\text{"}go\text{"}, miss\!\downarrow \\
\quad\;\; \ldots \\
\quad\;\;],\;\; ERESULT!tlbm \\
\;\;]
\end{array}
$$

As a final note, the addition of a TLB in parallel with the cache lookup works well for a direct-mapped cache only if the cache size is smaller than or equal to the page size. If the cache is to be larger than this, a set-associative cache must be used if the operating system is to have complete freedom in the placement of virtual pages in physical memory. A less traditional alternative would be to allow the operating system to place virtual pages only in physical pages that agree in their least significant bits.

1.7. Summary

We have presented a general architecture for pipelined asynchronous caches and demonstrated how to extend it for virtual memory support, partial word operations, and deeper pipelining.* In Chapter Two, we shall show an application of this

* In [17], the arcane feature of "cache swapping" specified by the MIPS ISA is handled using

general architecture to the specific design of the Caltech MiniMIPS processor, a high-performance asynchronous MIPS-compatible processor designed at Caltech.

a "pipelined semaphore" structure.

Chapter Two.

The MiniMIPS Cache System

2.1. Introduction

The cache mechanism used in the Caltech MiniMIPS processor is based on the techniques developed in Chapter One of this Thesis. The MiniMIPS cache implementation is further intended to be close in behavior to the cache specified in detail by the MIPS Level 1 Instruction Set Architecture (ISA).

The MiniMIPS processor has separate instruction (I-cache) and data (D-cache) caches. The instruction cache was originally intended to be optimized entirely for reading, with no provisions for writing to the cache, and the data cache was intended to have two separate R sections, one for each execution bus (Z-bus) of the MiniMIPS processor. Such an architecture would have allowed the MiniMIPS D-cache to satisfy certain load requests out of order (in particular, this architecture could satisfy a load miss-load hit combination out of program order—i.e., with the load hit data appearing on the Z-buses before the load miss). This would have increased the tolerance of the MiniMIPS to memory latency.

The design of the cache system of the MiniMIPS processor was modified for layout reasons—it was felt that the extra area taken by the multiple writeback units of the D-cache would not have been justified by the relatively minor performance improvement. Also, the amount of actual layout labor involved was a significant factor in this decision.

In this Chapter, we shall present the complete design of the data cache first. We shall describe the design of the cache as as application of the techniques developed in Chapter One and as an exercise in process decomposition and explain some of the low-level microarchitecture used to achieve the expected performance of the system. Finally, the MiniMIPS instruction cache will be presented only in the ways that it differs from the data cache.

2.2. The MiniMIPS Memory System

The MiniMIPS processor has a slightly simplified memory system compared to the MIPS R3000 processor. While the extensions to the cache architecture described at the end of Chapter 1 were developed with an asynchronous MIPS R3000 in mind, most of them were not implemented in the MiniMIPS. Relevant things not implemented were:

- Partial word operations.
- Exceptions generated by the memory system (virtual memory support), omitted due to the lack of a TLB.
- Multiple out-of-order memory result writeback (not in MIPS R3000).

The basic pipelined cache structure was retained for performance (it is absolutely essential for the I-cache since the tags comparison delay would otherwise curtail the operating speed of the MiniMIPS processor), both for the I- and D-cache, which were made almost identical.

The MiniMIPS cache system was designed with the details of the MIPS instruction set in mind. The MiniMIPS caches use 48-bit cache lines: 32 bits of data and 16 bits of tags. The 16 bits of tags are generated from bits 12 through 27 of the address. Address bits 28–32 are hard-wired to zero in the cache; this technique is used to implement an "uncacheable" segment starting from address (hex) `0x10000000` to `0xffffffff`. Furthermore, this allows flushing the caches by reserving a four kilobyte block of zeros in the uncacheable segment. Reading this block as data or executing it* will flush the caches. Both caches are four kilobytes in size, and refills are 128 bits wide (i.e., the cache block size is four words or lines). Partly, the design was chosen to stay as close to the MIPS R3000 architecture as possible, and partly it was chosen for simplicity. Nothing fundamentally prevents the design from being extended to larger lines and/or blocks.

2.3. Design by Decomposition

Much of the design of the pipelined cache architecture presented here was carried out and tested in the language `mcc`.* A dual-writeback pipelined cache was designed by decomposition and reordering of the actions of a sequential specification, and then translated back to CHP. The simulation was a part of a larger simulation of the entire MiniMIPS processor.† This gave us a chance to experiment with different high-level architectures. The simulation was annotated with performance information and the ghost variables mentioned in Section 1.5.9, allowing performance measurements and run-time verification of invariants.

2.4. MiniMIPS-Specific Design Modifications

Although the vast majority of our design closely followed the generic cache architecture developed in Chapter 1, we made some refinements in order to make the best possible use of the leeway granted by the MiniMIPS specification. One of the performance-critical parts of the MiniMIPS processor is the "fetch loop" consisting of the *FETCH*, *ICACHE*, and *DECODE* units. This is the loop in which the program counter is maintained, and even though the MIPS ISA specifies a one-instruction branch delay slot (i.e., branches do not take effect immediately,

---

* For the I-cache—by happy coincidence, the MIPS instruction denoted by a word of zeros is defined to be the no-op instruction `NOP`.

* Multicomputer C, `mcc`, was developed by Dr. Marcel van der Goot.

† When this work was done, the scope of the asynchronous MIPS project had not yet been crystallized into the MiniMIPS architecture, so this simulator actually included all the detail of partial-word operations, memory translation, and dual Z-bus ports developed in Chapter One.

but rather with a one instruction delay), we found it difficult to meet its throughput target.

The fetch loop of the MiniMIPS processor has two items in it at any given time, effectively computing the program counters of two different instructions at a time. All MIPS branches involve register comparisons. Thus, when the processor encounters a branch instruction, it gets the register operand(s) involved from the register file, passes it (them) through a comparator, and makes its branch decision based on the result of the comparison. This means that in order to maintain full throughput (without modifications) during branches, the latency through a loop consisting of the fetch loop plus the register file and comparator must be less than three instruction fetch cycles. A back-of-the-envelope calculation convinced us early on that this would not be the case—indeed, this latency was found to be on the order of almost three instruction fetch cycles for normal, non-branching instructions and much more for branches. To avoid stalling the processor on branches, we developed two optimizations: pre-decoding and a simple branch prediction scheme.

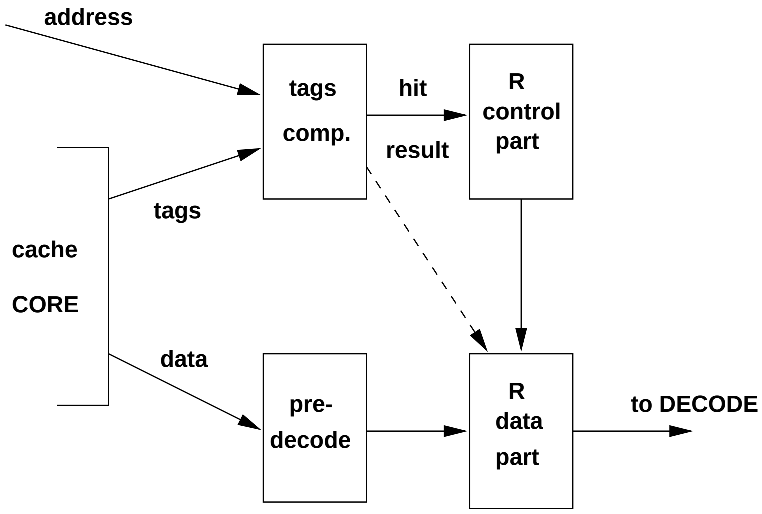

Figure 5. Placement of the pre-decode in parallel with the cache tags test. This arrangement allows the otherwise dead time of the tags test to be used to decode the instruction speculatively. As explained below, the result of the (current) comparison is actually sent directly to the datapath to minimize the latency, rather than via the control process.

2.4.1. Pre-decoding

A substantial part of the latency through the MiniMIPS cache subsystem (approximately 1 ns out of a total 3.5 ns cycle time) is taken by the tags comparison. The sequential "*retrieve line from cache core; compare tags; decode instruction*" exposes this latency on the critical fetch loop, on every cycle, whether or not a branch has been detected. In order to avoid this, we decided to decode speculatively the retrieved cache line in parallel with the cache hit test. This is done at essentially zero cost—the cost of managing a greater number of decoded bits in the R process as opposed to the 32 raw bits of the instruction plus the (insignificant) energy cost of sometimes pre-decoding the wrong instruction.

2.4.2. Branch prediction

We mentioned earlier that the fetch loop latency is exacerbated by branches, since they involve register value retrieval and comparison in the fetch loop. Instead of seeing this as a drawback, we can recognize that this time—during which the fetch loop is partially "drained"—is, in fact, an opportunity for the processor to do something useful!

We use the dead time after a branch instruction to fetch speculatively an additional instruction from the cache; this is a form of branch prediction. Following the recommendations of Hennessey and Patterson [2], we assume that backward "loop" branches are *taken* and forward "if" branches are *not taken.* (This is called "static branch prediction.") We use the dead time of the cache comparison to fetch and pre-decode the predicted instruction from the I-cache. Once the result of the branch comparison is available, the processor knows whether or not the prediction was correct. If it was correct, the prefetched instruction is executed; else, it is "tossed" and the correct program counter dispatched. We estimate a branch prediction rate of approximately 70% correct. In the correctly predicted cases, the mechanism hides basically all of the extra comparison latency, meaning that on average seven out of ten branches will execute at the same speed as normal instructions.

The branch prediction mechanism is implemented partly in the $FETCH$ process and partly as an addition to the R process of the I-cache. The only change is that the communications from R on the output channel $DATA$ are prefixed with a communication on $TOSS$, the channel from the $FETCH$ unit that informs the cache whether or not there has been a branch mispredict.

```
*[ ...
   [ ...
   [] c = "read" ∧ h ⟶ TOSS?t;
      [t ⟶ skip
      []¬t ⟶ DATA!(CACHE_DATAR?)
      ]
   [] ...
   ]
 ]
```

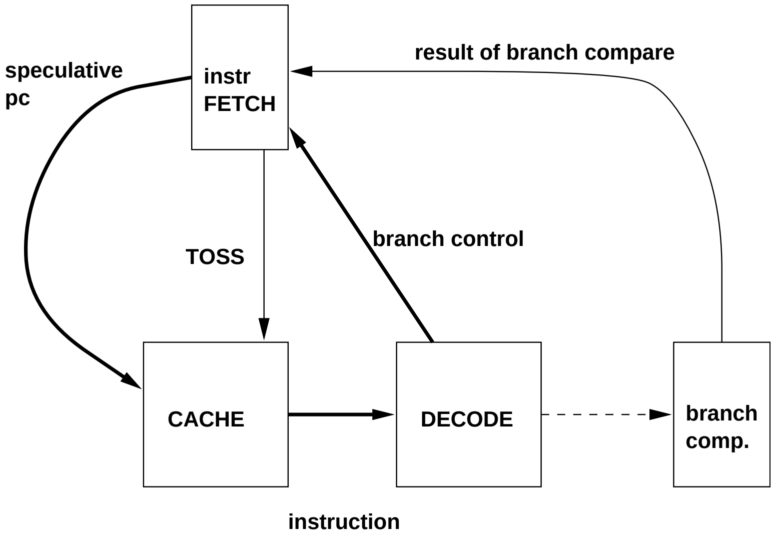

Figure 6. MiniMIPS branch prediction mechanism. The fetch loop is drawn with heavier arrows.

2.5. Decomposition of Cache Control

In this section, we shall describe the implementation of the MiniMIPS caches in some detail. We shall focus on the D-cache; the I-cache will be explained in terms of how it differs from the D-cache.

After the introduction of pipelining as described in Chapter 1, the remaining design steps carried out for the cache were quite straightforward decomposition operations. The basic structure of the L–$CORE$–R pipeline has been retained, although each of these processes has been decomposed into smaller parts.

2.5.1. Process L

Process L is decomposed into four separate parts:

- an address generating part, which takes the previously generated address plus the input address from the main pipeline and chooses between the two,
- a state loop, used to store the address in case it needs to be repeated or used for a refill—this goes from the address generator and loops back to it,
- a datapath process, which is used to repeat values from the register file used for storing in the memory system—this does not need to be a state loop since two consecutive stores never need to be repeated (we can get away with using the same value twice in succession rather than needing to interject another value between the two uses),
- control processes.

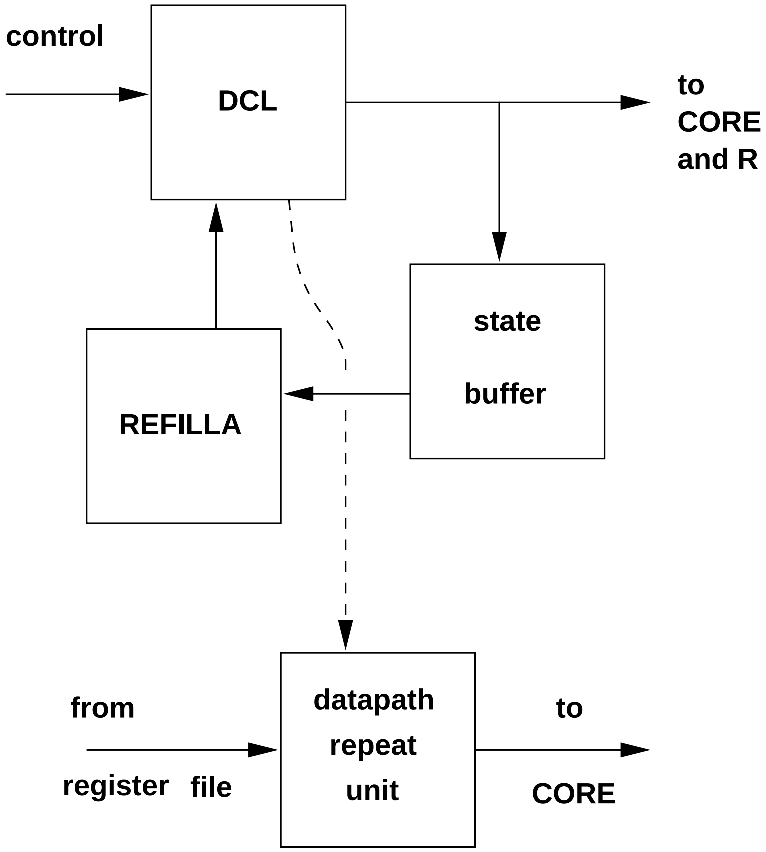

Figure 7. The L process in decomposed form.

2.5.1.1. Process DCL

As an example of the decomposition process, we follow the main control part of the L process to its final form. We start with the L process from Chapter One; after adjusting for the requirements of the MiniMIPS, we are left with

$$
\begin{array}{l}
DCL \equiv \\
*[\ RESULT?x; \\
\quad [\ x = \text{"}go\text{"} \longrightarrow A?a, C?c \\
\quad [\!]\ x = \text{"}refill\text{"} \longrightarrow c := \text{"}refill\text{"}, REFILLA?a \\
\quad [\!]\ x = \text{"}repeat\text{"} \longrightarrow XCOREA?a, XCOREC?c \\
\quad]; \\
\quad COREA!a, COREC!c \\
]\ ,
\end{array}
$$

where channels XRA and $XCOREC$ receive a buffered version of the previous command (sent on $COREA$ and $COREC$), so that it can be repeated. The final decomposition of DCL splits the process into two, one for the address, and one for the command, and this is what is implemented in the MiniMIPS.

2.5.2. Process R

The R process decomposition is more complex than the L process—this can be seen from the version in Chapter One. The R process can be split into several parts: A control part, implementing the if-statement seen in Chapter One, and many scattered pieces of datapath implementing the tags comparator, refill circuitry, etc.

The control part of R has been further decomposed. We implement the if-statement as a separate process with a one-of-five code output, one rail for each case of the if-statement so that

$$
\begin{array}{l}
CASEBOX \equiv \\
miss{\downarrow}; \\
*[\,C?c, MISS?miss, MATCH?ma, HIT?h; \\
\quad [\,c = \text{"}refill\text{"} \;\vee \\
\qquad (\;(c = \text{"}write\text{"} \;\vee\; c = \text{"}core\_write\text{"}) \;\wedge \\
\qquad\quad (\neg miss \;\vee\; \neg ma)\;) \;\longrightarrow\; R!\text{"}case0\text{"} \\
\quad [\!]\,(c = \text{"}write\text{"} \;\vee\; c = \text{"}core\_write\text{"}) \;\wedge\; miss \;\wedge\; ma \\
\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow\; R!\text{"}case1\text{"} \\
\quad [\!]\; c = \text{"}read\text{"} \;\wedge\; h \;\longrightarrow\; R!\text{"}case2\text{"} \\
\quad [\!]\; c = \text{"}read\text{"} \;\wedge\; \neg h \;\wedge\; (\neg miss \;\vee\; \neg ma) \\
\qquad\qquad\qquad\qquad\quad \longrightarrow\; R!\text{"}case3\text{"} \\
\quad [\!]\; c = \text{"}read\text{"} \;\wedge\; \neg h \;\wedge\; ma \;\wedge\; miss \\
\qquad\qquad\qquad\qquad\quad \longrightarrow\; R!\text{"}case4\text{"} \\
\quad] \\
]\;.
\end{array}
$$

In this program, the channels $MATCH$ and HIT carry information about whether the cache block of the current request matches that of the previous (to know whether or not to repeat the command as explained in Chapter One) and whether or not the current read is a cache hit. Since this information is not generated automatically by every command issued to the cache core, processes $MATCHBOX$ and $HBOX$ are used to generate dummy values on these channels for other commands—this is done to keep the decomposition as modular as possible.

The five "cases" correspond to:

case0. A refill or uneventful write command (no possibility of data hazards or other misbehavior)

case1. The repeat case for writes—a core write that follows a cache miss that will clobber the write (this should never happen for the case of *core_write* since that already constitutes a repeated command).

case2. A read hit—for most purposes, the same as *case0.*

case3. The refill case—a cache miss on a read either not following another cache miss or following a cache miss that was to a different block (the pipelined refill case; in this case, the R process may dispatch two refills to the memory system without even receiving back the results of the first one).

case4. The repeat case for reads—a cache miss whose refill may already be pending.

The $CASEBOX$ is where the control of the R process originates. The R channel from the $CASEBOX$ is copied to a number of processes that convert the cases to the appropriate communications to the datapath, the L process, and to memory.

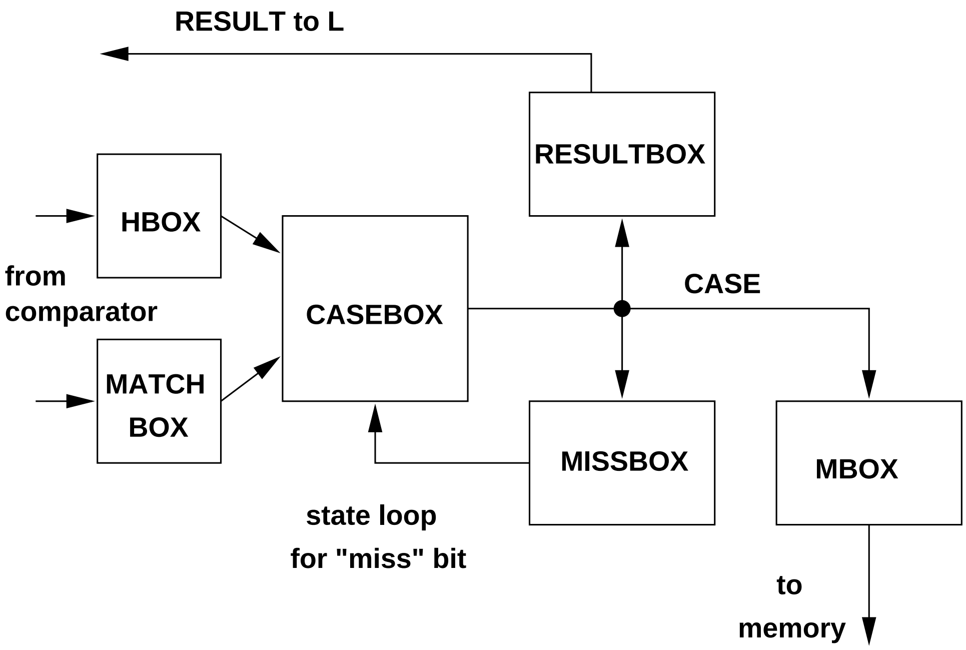

Figure 8. The R process in decomposed form.

2.5.3. 20-bit tags comparator

A 20-bit (ten one-of-four code) comparator was used to do the tags comparison in the MiniMIPS. This comparator is described in more detail in Section 2.6.

2.5.4. Interface to main memory

The interface to the main memory was the last part designed of the MiniMIPS. Original design plans called for the use of high-speed pipelined synchronous CMOS static RAM, which would have allowed an off-chip speed of up to 200 MHz, for a main memory bandwidth of 3.2 gigabytes per second. These plans were delayed in favor of a simpler system utilizing standard (but fast) asynchronous SRAM parts. The two caches (I- and D-cache) are connected to a memory controller, which arbitrates between requests from the two caches and satisfies them in any order. A write buffer is used to allow the cache system to proceed on writes without having to wait for the off-chip latency of the main memory.

2.6. Low-level Implementation of CHP

Almost all processes in the decomposition are implemented as precharge half-buffering pipeline stages as mentioned earlier. In other words, in terms of an L–R buffer, the handshake followed is

$$*[\ [f(L) \wedge \neg ri];\ R\Uparrow;\ [v(L)];\ lo\uparrow;\ [ri];\ R\Downarrow;\ [n(L)];\ lo\downarrow\ ]$$

where $f(L)$ denotes that L has reached the point where it is possible to infer a value for R (either this is the same as $v(L)$ or else $f(L)$ may denote some intermediate state), $v(L)$ that L is completely valid, and $n(L)$ that L is neutral.*

The implementation of the pipeline stages in terms of mask layout was done with the help of the University of California, Berkeley `magic` layout editor[8], modified at Caltech to incorporate a Scheme[1] interpreter allowing the user to construct useful higher-level functions.

2.6.1. Example: *TOSS2GEN*

In order to illustrate the type of design used throughout the MiniMIPS cache controller, we first need to mention a few details about the environment of the cache controller. As we have explained, the basic specification of the (D-)cache controller is

* The exceptions to the use of half-buffering logic were certain buffers used to maintain state, for which it was deemed worthwhile to use a reshuffling that decoupled the L and R operations more than the half-buffer reshuffling. Also, some buffering and logic were implemented using "weak-condition half-buffering logic"[7].

$$
\begin{array}{l}
*[\ LA?addr,\ \ LC?cmd; \\
\quad [\ LC\ =\ \text{"}read\text{"}\ \longrightarrow\ R!mem\,[addr] \\
\quad [\!]\ LC\ =\ \text{"}write\text{"}\ \longrightarrow\ LW?mem\,[addr] \\
\quad] \\
\ ]\ ,
\end{array}
$$

or in other words, exactly the same as the processor's memory. It is up to the cache controller to implement this program in an efficient way, and ideally, the interface to the cache controller would contain no more channels than this. Unfortunately, however, we made several optimizations in the MiniMIPS, aiming at reducing the total loop latency of the *FETCH–ICACHE–DECODE* loop and aiming at reducing the overall average latency of a load.

If the decoding to be done in the pre-decode were strictly a function D of the value in memory, all would be well, and the I-cache/pre-decode combination would be

$$
\begin{array}{l}
ICP \equiv \\
*[\ LA?addr;\ \ R!D(mem\,[addr])\ ]\ .
\end{array}
$$

This is unfortunately not the case. One bit of information, the exception mechanism's *valid-again* bit va, needs to cross the boundary imposed by the cache controller, so that

$$
\begin{array}{l}
ICP \equiv \\
*[\ VA?va, LA?addr;\ \ R!D(va, mem\,[addr])\ ]\ .
\end{array}
$$

The complication this represents is easily understood by considering what happens on a cache miss. In this situation, the operation will be retried, since in fact what was computed was not $D(va, mem\,[addr])$ but rather $D(va, X)$ where X is whatever data happened to reside at the unfortunate cache line that was implicated in the miss. In order to compute the correct decoded value, we clearly need va again, and this is not something that lets itself be done easily in the precharge half-buffering scheme[7] that we normally use.* We may recognize two distinctly different domains: one inside the cache control, where actions need to occur a variable number of times, dependent on *repeat* and *refill* operations, and another outside it, where actions need to match commands issued on *READ* and *DATA* in Figure 9. Any communications (Q in the figure) that cross this boundary need to be re-synchronized.

---

* Another bit, the *TOSS,* implicated in the single-instruction branch-prediction mechanism mentioned earlier, needs to be repeated using exactly the same mechanism; this is from whence the *TOSS2GEN* derives its name.

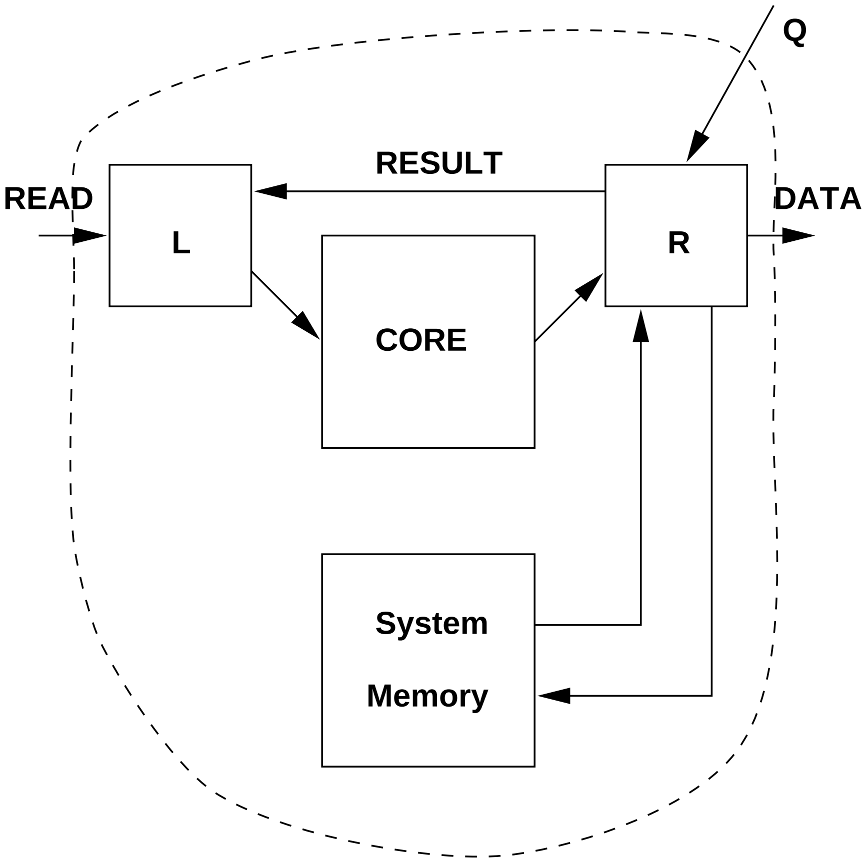

Figure 9. The channel Q crosses the cache control boundary and needs to be made to match events inside the cache control using the $TOSS2GEN$ process.

2.6.1.1. Specification of $TOSS2GEN$

Concretely, the program that needs to be executed is

$$
\begin{array}{l}
TOSS2GEN \equiv \\
*[\;\; TOSS?t \;\; ; \;\; TOSS2!t \;\; ; \;\; s := 0; \\
\quad *[\;\; s = 0 \;\; \longrightarrow \;\; CASE?c; \\
\qquad [\;\; c = 0 \;\; \vee \;\; c = 1 \;\; \longrightarrow \;\; \textbf{skip} \\
\qquad [\!]\;\; c = 2 \;\; \longrightarrow \;\; s := 1 \\
\qquad [\!]\;\; c = 3 \;\; \longrightarrow \;\; TOSS2!t; \;\; s := 1 \\
\qquad [\!]\;\; c = 4 \;\; \longrightarrow \;\; TOSS2!t \\
\qquad] \\
\quad] \\
]
\end{array}
$$

Informally, what this program does is that it reads the $TOSS$ from the main CPU pipeline (the $FETCH$, in fact) and sends it to the pre-decode on channel $TOSS2$. It repeats this until the cache operation has succeeded, which is indicated by the inner loop setting s to 1. At most, $TOSS$ will be sent three times, twice due to the case where c is 4, and once again for $c = 3$. (This is related to the progress argument in Section 1.5.9; in fact, livelock in that program would correspond to $c = 4$ repeatedly in this one.)

While the program for $TOSS2GEN$ may look innocuous enough, a seasoned designer will notice several potential problems: as many as six operations to be done in sequence (the precharge half-buffers that we use only do two operations in sequence, which is to say that they have a single synchronization point or state variable), a nested loop, a state variable, and plenty of conditional communications. It is highly unlikely that this process could be directly implemented in hardware with good results. Instead, process decomposition will be used to bring $TOSS2GEN$ to a manageable size.

2.6.1.2. Decomposing $TOSS2GEN$

The decomposition of $TOSS2GEN$ proceeds along the following path: We pick out the communications on $TOSS$ and $TOSS2$ and do these in a separate process. Since t is never inspected by the program, we could justly call this process a "datapath" process. The program for it is

$$
\begin{array}{l}
TOSS2GENLOGIC \equiv \\
*[\ \ TOSS?t;\ \ s := 0; \\
\quad *[\ \ s = 0 \ \longrightarrow\ SUCCESS?s;\ \ TOSS2!t\ \ ] \\
\ \ ]\ .
\end{array}
$$

This is to be composed with

$$
\begin{array}{l}
SUCCESSGEN \equiv \\
*[\ \ CASE?c; \\
\quad [\ \ c = 2 \ \longrightarrow\ SUCCESS!1 \\
\quad []\ c = 3 \ \longrightarrow\ SUCCESS!0;\ \ SUCCESS!1 \\
\quad []\ c = 4 \ \longrightarrow\ SUCCESS!0 \\
\quad []\ c = 1\ \vee\ c = 0 \ \longrightarrow\ \textbf{skip} \\
\quad] \\
\ \ ]
\end{array}
$$

so that

$$TOSS2GEN \ \equiv\ TOSS2GENLOGIC \ \ \|\ \ SUCCESSGEN\ .$$

2.6.1.3. Implementation of *TOSS2GENLOGIC*

In order to implement *TOSS2GENLOGIC* as a simple precharge half-buffer, we need to remove the nested loop. This is done by sending t on a channel back to the process itself, reducing the problem to one of conditionally acknowledging *TOSS* so that

$$
\begin{array}{l}
TOSS2GENLOGIC \equiv \\
*[\ S?s\ ; \\
\quad [\ s = 1 \ \longrightarrow\ TOSS2!(TOSSX?),\ \ XTOSS?\_ \\
\quad [\!]\ s = 0 \ \longrightarrow\ TOSS2!(XTOSS?) \\
\quad] \\
\ ] \\
\| \\
x := \_;\ \ *[\ TOSS!x, XTOSS!x;\ \ TOSSX?x] \\
\| \\
S!1;\ \ *[\ S!(SUCCESS?)\ ]\ \ ,
\end{array}
$$

where _ stands for an arbitrary value. Also, we rewrite *SUCCESSGEN* as a simple finite-state machine with two states, $x = 0$ and $x = 1$.

$$
\begin{array}{l}
SUCCESSGEN \equiv \\
x := 0; \\
*[[\ x = 0 \ \longrightarrow\ CASE?c\ \ [\!]\ x = 1 \ \longrightarrow\ \mathbf{skip}\ ]; \\
\quad [\ x = 1 \ \longrightarrow\ S!1;\ \ x := 0 \\
\quad [\!]\ x = 0\ \wedge\ c = 2 \ \longrightarrow\ S!1\ ;\ \ x := 0 \\
\quad [\!]\ x = 0\ \wedge\ c = 3 \ \longrightarrow\ S!0;\ \ x := 1 \\
\quad [\!]\ x = 0\ \wedge\ c = 1 \ \longrightarrow\ \mathbf{skip} \\
\quad [\!]\ x = 0\ \wedge\ c = 4 \ \longrightarrow\ S!0 \\
\ ]]
\end{array}
$$

and again note that this is non-trivial to implement as a precharge half-buffer due to the presence of the x state variable. The implementation turns x into a feedback loop as with *TOSS* above.

The final programs for $TOSS2GENLOGIC$ and $SUCCESSGEN$ may be converted into production rules using the methods described in [7].

2.6.1.4. Comparison to traditional QDI implementation

The approach taken in the MiniMIPS project of minimizing the number of reshufflings used marks a departure from tradition as exhibited in the Caltech Asynchronous Microprocessor[12]. The main reason for using the precharge half-buffer as the archetypal circuit in the MiniMIPS was that the performance of this class of circuits is well understood, and when generating production rules "by hand," it is much easier to use a single template. That this is not necessarily the best way to do things can be illustrated by considering the circuit for $SUCCESSGEN$ in Section 2.6.1.2. After the processor was designed, it was realized that this circuit may be implemented directly simply by sequencing the steps for the case $c = 3$, using approximately half the area of the circuit used in the MiniMIPS.

2.6.2. Circuit techniques in the 20-bit comparator

As an example of the circuit techniques used in the MiniMIPS cache subsystem, the tags comparator posed a particular challenge as a circuit design problem. The requirement was that this comparator be able to compare 20 bits of tags (actually ten one-of-four codes corresponding to the tags) and broadcast the result to the datapath (32 bits in the D-cache, about 50 bits in the I-cache due to the predecoding mechanism—the broadcast is necessary to inform the datapath whether to retain or discard the line read from the cache as valid and proper data or as a cache miss). The necessity to broadcast the result to the datapath in two stages follows from the pipelined completion mechanism described in [11]. This comparator was optimized as much as possible for the common case of a cache hit, and a broadcasting mechanism was folded into the design. The comparison and broadcast are done in two low-level pipeline stages (four stages of CMOS logic), at a maximum cycle rate of approximately 300 MHz with less than 1 ns latency in 0.6 micron scalable CMOS.

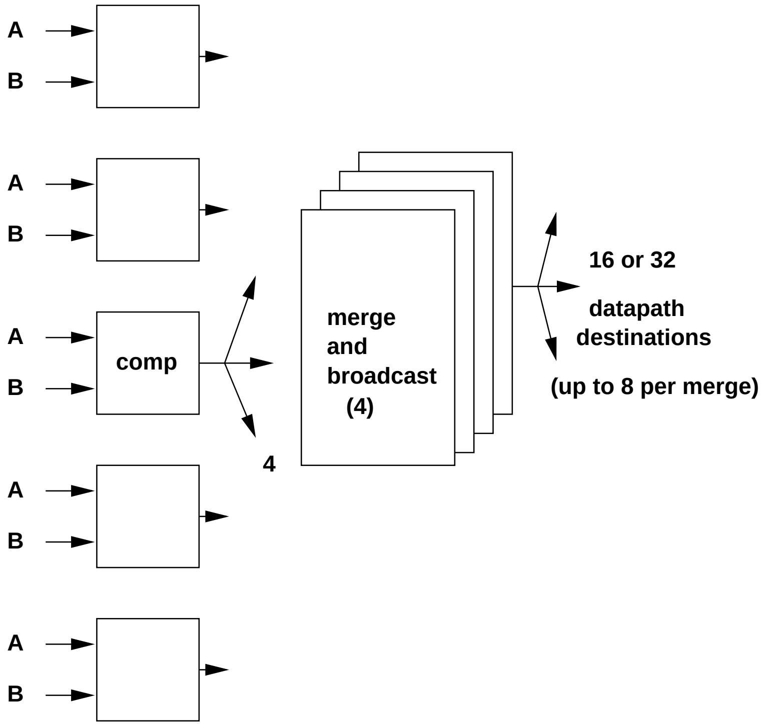

Figure 10. Simplified block diagram of tags comparator. The tags comparator combines the comparison and result broadcast operations in an effort to minimize the total latency of the cache system. The first stage compares two one-of-four codes in a single cell; this result is broadcast four ways to the four merge units, which merge the results from all the units in the first stage and broadcast the result to eight bits' worth of destination each, for a total of 32 bits of datapath (more than this in the I-cache).

The first stage of the comparator is a straightforward comparison of two quad-rail channels. The production rules for the logic computation are

$$
\begin{array}{l}
e \wedge ((a1\_0 \wedge b1\_0) \vee (a0\_0 \wedge b0\_0) \vee (a2\_0 \wedge b2\_0) \vee (a3\_0 \wedge b3\_0)) \wedge \\
\quad ((a1\_1 \wedge b1\_1) \vee \ldots \vee (a3\_1 \wedge b3\_1)) \quad \longrightarrow \quad t\_\!\downarrow \\
e \wedge ((a0\_0 \wedge (b1\_0 \vee b2\_0 \vee b3\_0)) \vee (a1\_0 \wedge (b0\_0 \vee b2\_0 \vee b3\_0))) \vee \\
\quad ((a0\_1 \wedge \ldots)) \quad \longrightarrow \quad f\_\!\downarrow
\end{array}
$$

The comparator is implemented in exactly this way, and this gives rise to some problems with charge sharing due to the capacitance of the large internal nodes in the pulldown network. These problems are handled by precharging the internal nodes. The first method we use to do this depends on the fact that a single enable signal is generated by the completion network of the cell. This signal is used to precharge internal nodes to `Vdd` through p-channel transistors at the same time as the outputs are precharged. The other, more aggressive approach used to reduce the occurrence of charge sharing is by precharging the largest internal nodes of the pull-down network via (again, fairly weak) n-channel transistors connected to `Vdd`. This method is also used in the true network of the tags comparator. HSPICE simulations indicate that these two approaches greatly reduce or eliminate the occurrence of unacceptable (static) charge sharing[9] in the comparator circuit.

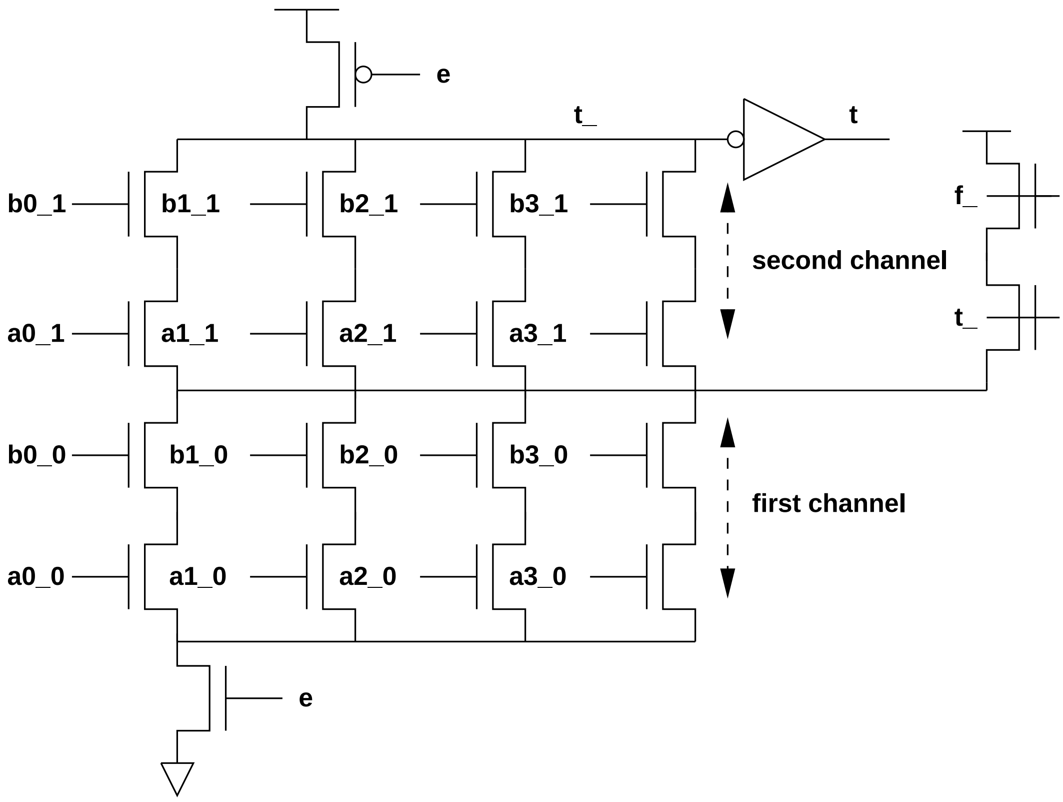

Figure 11. The logic to compute a true comparison result in the first stage of the tags comparator. The small precharge transistors for the internal nodes are shown (far right).

The design challenge in the second stage of the tags comparator was to improve the speed of the handshake with the first stage. In order to do this, careful attention was paid to the completion network by first pipelining the completion on a byte-wide basis and then by merging the output data completion of pairs of second comparator stages directly (i.e., instead of the usual two NAND gates and C-element). Unfortunately, this led to charge sharing problems due to the large internal capacitance of the dynamic NAND/C-element combination, which had to be handled by careful analog circuit design.

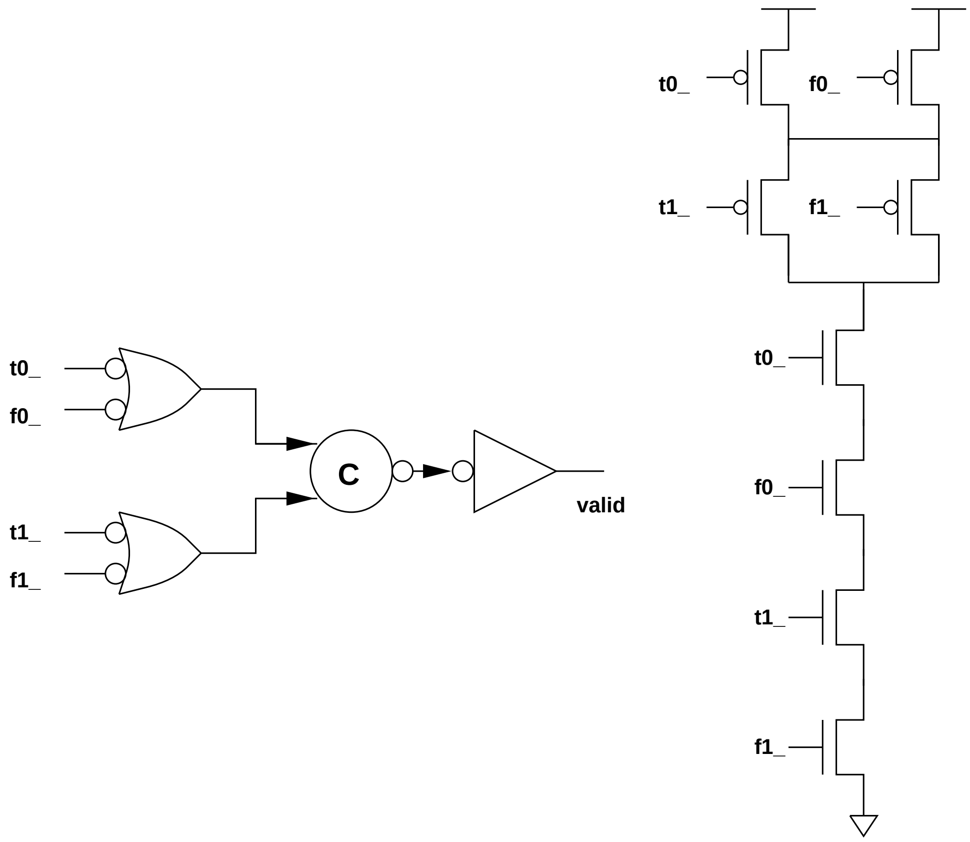

Figure 12. Merging of completion circuits used in the completion detection of the final stage of the comparator. The merged single-stage network (bottom) replaces the three stages of the original completion detection network (top).

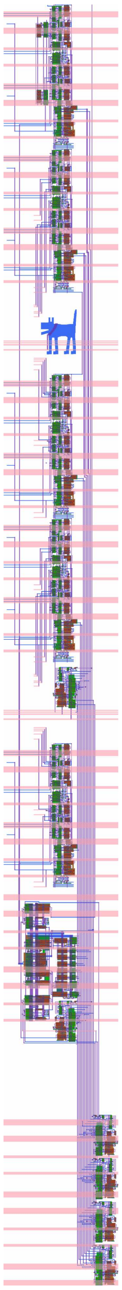

Figure 13. Layout of cache comparator in `magic`.

The main challenges faced in the design of the tags comparator were to improve the circuit in such a way that the throughput target could be met without compromising latency (which contributes to the throughput of the processor as a whole) or robustness.

2.6.3. Example: *DCACHEL*

To illustrate some of the concepts used in the MiniMIPS cache controllers with emphasis on the unorthodox communication actions introduced in section 1.5.2.1, we study part of the decomposition of the *DCACHEL* process. As mentioned in Section 1.5.8.2, the final CHP program presented in Chapter One for the *L* process leaves out the production and repetition of store data. Adding these back, *DCACHEL* becomes

```
DCACHEL ≡
*[ RESULT?r;
    [ r = "go"  ⟶  oldc := c,  oldaddr := addr;
        CACHEOP?c,  ADDR?addr
    [] r = "repeat"  ∧  oldc = "write"  ⟶
        c := "core_write",  addr := oldaddr
    [] else  ⟶  c :=  oldc,  addr := oldaddr
    ];
    [ c = "write"  ⟶
        [ r = "go"  ⟶  DATAL?d;  CWRITE!d
        [] r = "repeat"  ⟶  CWRITE!d
        ]
    [] else  ⟶  skip
    ];
    CA!addr,
    [ r = "go"  ∨  r = "repeat"  ⟶  CC!c
    [] r = "refill"  ⟶  CC!"refill"
    ]
 ] .
```

The addition of exceptions to the MiniMIPS makes it necessary to be able to cancel writes to the memory (just like writes to the register file are canceled to inhibit the effect of instructions that appear "after" the exception in the instruction stream). We introduce a channel *DO* for this functionality. Whenever a write ($rw = 1$) is dispatched to the cache control, it will be followed by a communication on *DO*. If a one is sent, the write is to be executed, else it is to be ignored. If the operation is a read rather than a write, there will be no communcation on *DO*. This complicates things since three separate channels need to be read in order to know what communication pattern to follow, and we may write

$$
\begin{array}{l}
DCACHEL \equiv \\
*[\ [\ skipresult \ \longrightarrow\ \mathbf{skip} \\
\quad\ \ [\!]\ \neg skipresult \ \longrightarrow\ RESULT?r \\
\quad\ \ ]; \\
\quad\ \ skipresult = 0; \\
\quad\ \ [\ r = \text{"}go\text{"} \ \longrightarrow \\
\quad\quad\quad savec := oldc,\ \ saveaddr := oldaddr; \\
\quad\quad\quad oldc := c,\ \ oldaddr := addr; \\
\quad\quad\quad CACHEOP?c,\ \ ADDR?addr; \\
\quad\quad\quad [\ c = \text{"}write\text{"} \ \longrightarrow\ \ DO?do;\ DATAL?d \\
\quad\quad\quad\quad [\ do \ \longrightarrow\ CWRITE!d \\
\quad\quad\quad\quad [\!]\ \neg do \ \longrightarrow\ skipresult = 1; \\
\quad\quad\quad\quad\quad oldc := savec,\ \ oldaddr := saveaddr \\
\quad\quad\quad\quad] \\
\quad\quad\quad [\!]\ c = \text{"}read\text{"} \ \longrightarrow\ \mathbf{skip} \\
\quad\quad\quad] \\
\quad\ \ [\!]\ r = \text{"}repeat\text{"} \ \longrightarrow\ addr := oldaddr, \\
\quad\quad\quad [\ oldc = \text{"}write\text{"} \ \longrightarrow\ c := \text{"}core\_write\text{"}, DATAL!d \\
\quad\quad\quad [\!]\ oldc = \text{"}read\text{"} \ \longrightarrow\ c := \text{"}read\text{"} \\
\quad\quad\quad] \\
\quad\ \ [\!]\ r = \text{"}refill\text{"} \ \longrightarrow\ addr := oldaddr,\ \ c := \text{"}refill\text{"} \\
\quad\ \ ]; \\
\quad\ \ CA!addr,\ \ CC!c \\
\ ]\ .
\end{array}
$$

The last program is unsatisfactory. When discussing ways to produce a circuit implementation of a CHP program such as this, it is important to argue in terms that are meaningful from the circuit level. The CHP constructs presented in Section 1.5.2.1 are helpful in this respect—although these constructs are unorthodox from the point of view of concurrent programming, they are quite natural from the point of view of circuit implementation. Using the value probe, the program may be simplified to

$$
\begin{array}{l}
DCACHEL \equiv \\
*[\ RESULT\text{¿}r; \\
\quad [\ \overline{DO, CACHEOP : \neg DO \ \wedge \ CACHEOP = \text{"}write\text{"}} \ \wedge \\
\qquad r = \text{"}go\text{"} \ \longrightarrow \\
\qquad DO?\_, CACHEOP?\_, ADDR?\_, DATAL?\_ \\
\quad [\!]\ r = \text{"}repeat\text{"} \ \longrightarrow \ addr := oldaddr, \\
\qquad [\ oldc = \text{"}write\text{"} \ \longrightarrow \ c := \text{"}core\_write\text{"}, CWRITE!d \\
\qquad [\!]\ oldc = \text{"}read\text{"} \ \longrightarrow \ c := \text{"}read\text{"} \\
\qquad]; \\
\qquad CA!addr,\ \ CC!c,\ \ RESULT?\_ \\
\quad [\!]\ r = \text{"}refill\text{"} \ \longrightarrow \\
\qquad CA!oldaddr,\ \ CC!\text{"}refill\text{"},\ \ RESULT?\_ \\
\quad [\!]\ r = \text{"}go\text{"} \ \wedge \\
\qquad (\overline{CACHEOP : CACHEOP = \text{"}read\text{"}} \ \vee \\
\qquad\ \ \overline{DO : DO}) \ \longrightarrow \\
\qquad oldc := c,\ \ oldaddr := addr; \\
\qquad CACHEOP?c,\ \ ADDR?addr,\ \ RESULT?\_; \\
\qquad [\ c = \text{"}write\text{"} \ \longrightarrow \ DATAL?d;\ \ CWRITE!d \\
\qquad [\!]\ \textbf{else} \ \longrightarrow \ \textbf{skip}\ ], \\
\qquad CA!addr,\ \ CC!c \\
\quad] \\
\ ].
\end{array}
$$

(Note that

$$
\begin{array}{l}
[\ \overline{DO, CACHEOP : \neg DO \ \wedge \ CACHEOP = \text{"}write\text{"}} \\
[\!]\ \overline{CACHEOP : CACHEOP = \text{"}read\text{"}} \vee \overline{DO : DO} \\
] \\
\equiv \\
[\ \overline{DO} \ \wedge \ \overline{CACHEOP}\ ]\ ,
\end{array}
$$

by our definition.) This program is much more satisfactory, being written as a single reactive process, especially elucidating the mechanism by which the exception-overriden writes are cancelled. In the final implementation, the datapath variable d is decomposed from this program, together with $addr$. The datapath variables as well as the *c/oldc* pair are implemented using feedback loops, avoiding the need for the direct implementation of state variables. This implementation may be derived by writing out the conditions for each statement in terms of the value probes and more or less directly implementing these probes as production rules according to the precharge half-buffer pipeline template.

2.7. Differences from Synchronous Implementations

Given our current design, we may ask what the application of asynchronous design methodologies to the cache pipelining problem in general and to the MiniMIPS cache system in particular has taught us about designing well-performing VLSI systems. The currently overwhelmingly most popular way of designing systems such as the ones studied in this Thesis is by the application of decades of experience with synchronous design. An analysis of a few of the differences seen between synchronous and asynchronous solutions in this case will highlight the most important lessons.

The major gain that the use of asynchronous design techniques has brought to the MiniMIPS caching system has been the modularity of the design. The timing requirements of the system components (the static "setup" and "hold" times of the traditional world-view) are here exposed where it is most convenient, i.e., in the subsystems' interfaces where they may be processed dynamically.

As an illustration of the modularity of the QDI design style, consider the case of the cache core. The cache core has a, comparatively speaking, long latency—it may be long enough (on average) that it would not fit in one processor "cycle." On a synchronous machine, this might have led to a system-wide reduction in the clock frequency. Furthermore, what has not been mentioned so far is that for ease of implementation, the minimum cycle time of the lowest level cache cell is actually on the order of *twice* the minimum cycle time of the rest of the chip, but by virtue of the asynchronous implementation, this is not readily seen outside the cache cell itself. The upshot of this is that given certain combinations of addresses issued to the cache, the cache throughput (and thus the latency) will suffer. This does not, however, affect the correctness of the cache.

In a synchronous implementation, an additional control structure would have to be used to achieve the asynchronous data-dependent latency behavior by holding back the dispatch of successive operations to the same cache bank, at substantial cost: the complexity of the control structure itself (including an address comparator) plus the more important additional latency of the address check *before* dispatching the commands to the cache core; also, this approach would have to be part of the interface of the synchronous cache to other parts of the processor—in effect, the synchronous implementation would include the same handshake signals as the asynchronous implementation, but it would use them only for flow control rather than also to maintain safe circuit timing, using the clock signal for this more "low-level" purpose.

The use of asynchronous design techniques allowed us to optimize the cache control for cache hits—cache miss transistors and circuitry were not designed to keep up with the core of the CPU, since misses were assumed to be less frequent than hits. On the other hand, a carefully designed synchronous comparator would almost certainly have been faster and simpler.

The strongest claim for the modularity of the asynchronous cache design probably comes from the (as yet unimplemented) partial-word circuitry described in Chapter One. By using speculative execution, it would be possible to allow a MiniMIPS cache to do partial-word operations without any significant performance penalty for the common case of full-word operations.

2.8. Directions for Future Work

In this Thesis, we have shown how to apply a pipelined asynchronous cache architecture to a simple sequential specification, namely, that of the Caltech MiniMIPS processor. This cache architecture could be extended to handle the infamous partial-word operations at little cost, and doing so would be an interesting test of asynchronous design solving a difficult real-world computer architecture problem. Furthermore, additional work in deeper pipelining, arbitrated cache architectures, and looser concurrency will be essential to future high-performance asynchronous computer systems.

2.9. Conclusion

We have demonstrated how to apply the techniques for the design of pipelined high-throughput asynchronous cache systems to a real-world example, namely the Caltech MiniMIPS processor. At this point, we might want to ask whether we have fulfilled the two promises made for asynchronous design methodologies at the beginning of this Thesis. We claimed that average-case performance could be achieved instead of worst-case performance. Nowhere is this shown more effectively than in the memory subsystem of the MiniMIPS processor. The cache is designed to make it appear to the rest of the processor as a straightforward memory—the parts that communicate with the cache dispatch an address to read from or to write to and then simply wait until the operation has completed. In the case of a cache hit, this takes a few nanoseconds; in the case of a miss, it may take a hundred times longer. This difference is logically invisible to the core of the CPU, but yet, average-case performance is obtained; on a cache miss, the processor simply stalls and waits until the result is available. Formally speaking, this behavior may be described by saying that the sequence of values on all input and output channels of the cache system is the same in both situations. As for the second promise, the modularity of the design style is shown in the same scenario. The amount of pipelining and other design details internal to the cache were private matters between the cache designer and his simulations; these decisions could all be changed without affecting other parts of the MiniMIPS processor.

References

1. H. Abelson and G. J. Sussman, with J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.
2. J. Hennessey and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.
3. H. P. Hofstee. Synchronizing Processes. PhD thesis, California Institute of Technology, 1995.
4. G. Kane and J. Heinrich. *MIPS RISC Architecture.* Prentice-Hall, 1992.
5. T. K. Lee. *Performance Analysis and Optimization of Data-Dependent and Inherently Disjunctive Asynchronous Circuits.* Ph.D. thesis. Caltech, 1995.
6. A. M. Lines. Private communication, 1997.
7. A. M. Lines. *Pipelined Asynchronous Circuits.* M.S. Thesis. Caltech, 1995.
8. R. N. Mayo, M. H. Arnold, W. S. Scott, D. Stark, and G. T. Hamachi. *1990 DECWRL/Livermore Magic Release.* WRL Research Report 90/7. Digital Equipment Corporation, 1990.
9. C. A. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.
10. R. V. Manohar. Ph.D. thesis, to be published. Caltech, 1997.
11. A. J. Martin, A. M. Lines, R. V. Manohar, M. Nyström, P. Penzés, R. G. Southworth, and U. V. Cummings. The Design of an Asynchronous MIPS R3000 Microprocessor. *17th Conferene on Advanced Research in VLSI,* ed. R. B. Brown and A. T. Ishii, IEEE Computer Society, 1997.
12. A. J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. *Sixth MIT Conference on Advanced Research in VLSI,* ed. W. J. Dally, MIT Press, 1990.
13. A. J. Martin. Synthesis of Asynchronous VLSI Circuits. *Formal Methods for VLSI Design,* J. Staunstrup, ed. North-Holland, 1990.
14. A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, P. J. Hazewindus. The Design of an Asynchronous Microprocessor. *Decennial Caltech Conference on VLSI,* ed. C. L. Seitz, MIT Press, 1989.
15. A. J. Martin. The Probe: An addition to communication primitives. *Information Processing Letters*, **20**:125–130, 1985.
16. M. Nyström. *MiniMIPS Cache Mechanism.* Unpublished internal document. Asynchronous Systems Architecture Project, California Institute of Technology, 1997.

17. M. Nyström. *MIPS Caching Schemes.* Unpublished internal document. Asynchronous Systems Architecture Project, California Institute of Technology, 1996.

18. E. I. Organick. *The Multics System.* MIT Press, 1972.

19. T. E. Williams. Performance of Iterative Computation in Self-Timed Rings, in *Journal of VLSI Signal Processing,* vol. 7, nos. 1-2. Kluwer, 1994.