Memory Access Scheduler

Matt Cohen and Alvin Lin
6.884 Final Project Report

May 10, 2005

1 Introduction

We propose to implement a non-blocking memory system based on the memory access scheduler described in [1]. This system provides enhanced performance over conventional memory systems by taking advantage of the timing characteristics of a DRAM. Specifically, operations on different DRAM banks can often be done in parallel rather than in the serialized fashion that is conventionally used. Consecutive operations to the same row can be done with much lower latency. The data is returned out of order, allowing advanced processors to capitalize on their own out-of-order capabilities.

Out-of-order memory scheduling is analogous to out-of-order execution in a processor. If a specific functional unit needed for an instruction is unavailable, out-of-order processors can execute a later instruction if it is independent of any instructions that are already in flight. Similarly, a memory access scheduler can divide use of its resources (banks and rows

as opposed to the functional units of a processor) to increase throughput. A typical DRAM is composed of several banks, each of which is made up of many rows, each of which contains several words (columns). A DRAM access requires a strict sequence of operations on each of these resources, each of which takes several clock cycles. Rixner et al. [1] give a simple example of how this sort of scheduling can take a series of memory accesses that would normally take 56 cycles to complete, and finish it in 19 cycles. This is a dramatic example of what an intelligent scheduler can do.

In order to work with the processor projects, we need to implement not just the scheduler, but also a cache and main memory, as shown in Figure 1.

The dashed lines represent well-defined interfaces that the project must conform to. Memory requests will be sent out by the processor, and data will be returned to it, along with an identifying tag to allow out-of-order completion.

In order to make the project manageable, each block will be done in a simple manner and gradually built up in complexity until the memory system is in its final version.

1.1 Cache

Initially, work on the cache will focus simply on getting it to function properly. This implies starting with the simplest possible cache, i.e. a blocking, direct-mapped cache. A direct-mapped

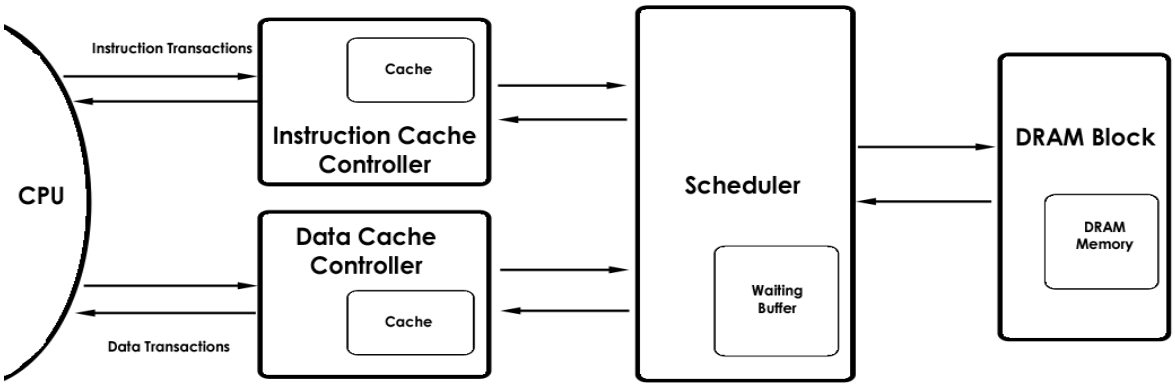

Figure 1: Block diagram of memory system

cache makes it easy to generate hits and misses and see the effects of both. For these tests a simplified 'magic memory' will be used, and no scheduling algorithm will be used. Once we are fully convinced of the cache's basic functionality, we will progress to a non-blocking cache, which is a requirement of the project. Later attempts may investigate effects of associativity, word size, etc. The overall cache size will always remain small in order to better illustrate the effects of memory scheduling.

Note that instruction and data caches are separate, but share a scheduler and main memory.

1.2 Scheduler

Based on results of the cache tag check, data is either sent to the processor, or the memory request is forwarded to the scheduler. The simplest scheduling algorithm is a FIFO policy, with instruction requests going ahead of simultaneously arriving data requests. The next level is to give priority to instruction requests over any outstanding data requests. Prioritizing data requests would reduce one of the major advantages of out-of-order execution, ruining the efforts of our classmates who have designed such processors. Beyond that, Rixner et al.[1] suggest several different policies that can be used to decide which memory operation should be performed at any time. Each policy has its advantages, typically dependent upon memory access patterns. Our plan is to implement several of these policies and compare their results both for patterns for which they are optimized and for patterns for which they are not recommended. Given enough time, we would like to then implement some kind of adaptive algorithm which can choose an appropriate policy based on recent history. For example, if recent memory accesses show large amounts of row locality, the scheduler may switch to an 'open' policy, while if there is little row locailty, a 'closed' policy may be used. These and other policies are defined in [1].

1.3 Main Memory

Main memory will be implemented using verilog models provided by Micron Technology, Inc. The DRAM can be tested on its own, without a cache, just using memory requests from the processor. A FIFO scheduling algorithm can be used for this level of testing. This block requires the least amount of design work, as it has already been implemented by Micron. The only task is to properly interface with it.

2 High-Level UTL Design

This section breaks down the UTL design starting with the entire memory system and then examines the UTL description of the major components of the system.

2.1 Memory Unit Transactions

The memory unit has two input queues and two output queues to the CPU, as shown in Figure 2. There is one I/O pair of queues for the instruction memory messages and another pair for the data messages. Both port pairs are capable of receiving the transaction request `Load<tag,address>` from the CPU, which indicates that the CPU is requesting a data word from the memory. This data word is placed on the corresponding output queue to the CPU with the form `Reply<tag,data>`. The memory unit is also capable of receiving an additional message of the form `Store<address,data>` from the CPU. This message results in the internal architectural state of the memory unit being modified to store the data in the transaction payload.

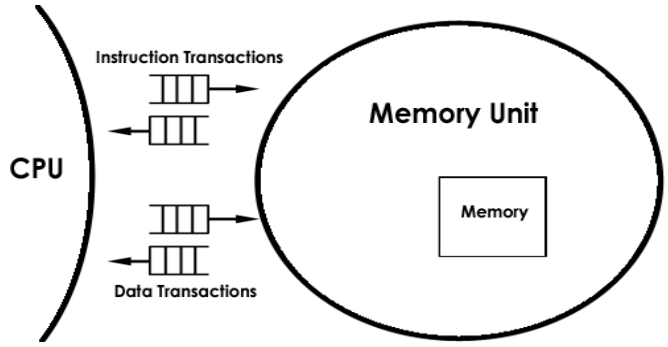

Figure 2: UTL Block Diagram for Memory System as a Whole

2.2 UTL Description of Components

The entire memory system can be fully described as four separate units: the instruction cache controller, the data cache controller, a scheduler, and the DRAM unit. The following sections explain the UTL design of each of these components. Figure 3 shows a diagram of these components and will serve as a useful reference over this entire section.

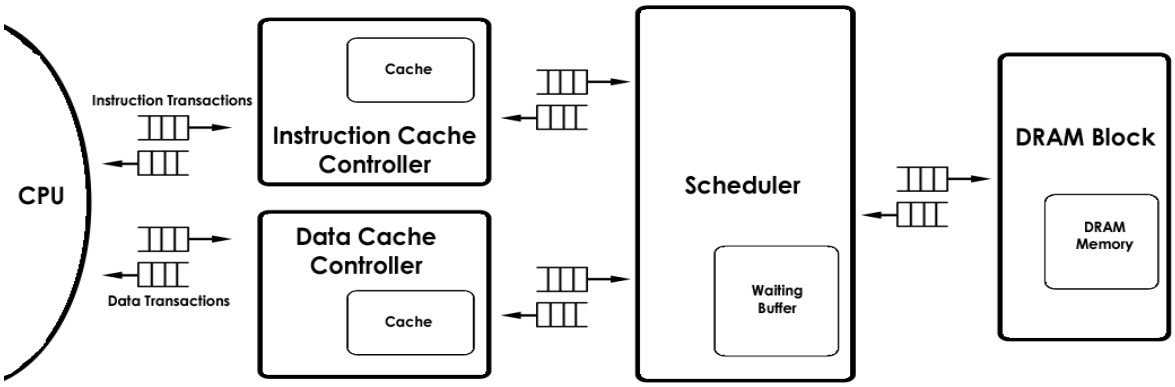

Figure 3: UTL Block Diagram for Subunits of Memory System

2.2.1 Cache Controller Transactions

Incoming memory requests from the CPU can be modeled as messages on input queues into two separate cache controllers: the instruction cache controller and the data cache controller. The UTL design for each these two units is identical.

The cache controllers accept messages of the form `Load<tag,address>`, which indicate that the CPU is requesting data from the memory. The internal architectural state of the cache controllers consists of a cache filled with the data from recent memory accesses. If the `Load` request references an address contained within the cache, the response `Reply<tag,data>` is formed using the data from the cache. This message is then placed on the queue going back to the CPU. If the `Load` request references an address that is not found in the cache, the cache controller must forward the `Load<addr,tag>` request on to the scheduler input queue.

The cache controllers are also able to accept requests of the form `Store<address,data>`. Upon receiving such a transaction request, the cache controller looks to see if the address is contained within the cache. If it is, the controller updates its internal state with the new data. Whether the data is found in the cache or not, the cache controller forwards a `Store<address,data>` message onto the scheduler block input queue. Note that in the event of a cache miss on a `Store` transaction, the data is not written into the cache; it is simply passed on to the scheduler. This is a description of a write-through, no-write allocate cache controller policy.

From the scheduler, the cache controllers will receive `ReturnReply<tag,data>` messages. Each of these return replies contains the data that was read out of the DRAM. Upon receipt of a `ReturnReply` message, the data is written into the cache and then a `Reply<tag,data>` is sent to the CPU.

2.2.2 Scheduler Transactions

The scheduler block is responsible for interfacing between the cache controllers and the DRAM unit. Its internal architectural state consists of a waiting buffer for requests that have not yet been exported to the DRAM block. The scheduler receives `Store` and `Load` messages from both the instruction cache controller and the data cache controller. Using a scheduling algorithm, it is then responsible for making requests to the DRAM block. The details of these DRAM messages are discussed further in Section 2.2.3.

When the scheduler sends a `Read<col>` request to the DRAM block, it will receive a `Reply<data>` response. This response must be packaged with the address and tag (which were temporarily stored in the scheduler's architectural state) and sent to the corresponding cache controller as`ReturnReply<tag,data>`.

Two UTL level optimizations can be implemented on the scheduler block to save on transactions between blocks. First, if a `Load` message arrives to an address that already has a pending `Load` transaction in the waiting buffer, only one request to the DRAM block needs to be made. Second, if a `Store` message arrives to an address that has a pending `Store` transaction in the waiting buffer[1], the original `Store` transaction in the waiting buffer can be deleted without ever exporting it to the DRAM. These two optimizations will not be implemented in the original design but will eventually be implemented to improve the performance of the system.

2.2.3 DRAM Transactions

The DRAM block consists of the actual DRAM device as well as a control wrapper to interface with the scheduler block. The architectural state is the actual stored data in the DRAM as well as the currently activated row (or lack thereof). There are four possible requests that the scheduler block can send to the DRAM block:

1. `Precharge<bank>`: The activated row in the selected bank is cleared so that a new row can be activated.
2. `Active<row>`: The selected row is activated and ready to be read or written to.
3. `Read<col>`: The selected column is read. The DRAM block sends a `Reply<data>` to the scheduler block.
4. `Write<col>`: The selected column is written.

3 Test Strategy

The test strategy for the memory access controller consists of two phases. First, verification tests must be performed on the system to ensure that the reads and writes are accurate. Second, performance tests must be performed to experiment with the effect of changing heuristics on the

[1]The waiting buffer cannot have any pending `Load` transactions to this address if this optimization is to be used.

latency, jitter, and bandwidth of memory accesses. As the heuristics are changed, the verification tests must be periodically run to ensure that the system is still providing the correct outputs.

3.1 Verification Tests

Given an ordered set of input memory commands, the memory access controller must provide identical tagged read data as a reference model. The memory access controller can return these tags out of order, but the data corresponding to each tag must match those in an in-order reference model. This reference model is not quite a "golden model" because it lacks the heuristics that the final memory access controller will use. A simple PERL script can negotiate between the output of the reference model and the output of the implementation to ensure that they contain the same data.

In order to run tests, an input file must be provided to a testbench, which will then probe the model with the data in the input file. The data in the input file provide a listing of RAM accesses that simulate those provided by a CPU. The format of the input file is shown in Table 1. The 2-bit "Command" indicates whether the incoming command is a NOP, a READ, or a WRITE. The encoding for these commands is shown in Table 2. The 8-bit "Tag" corresponds to the tag generated by the CPU. The 32-bit "Addr" refers to the address of the memory access. The 32-bit "Data" corresponds to the data for a write.

ICommand(2)	ITag(8)	IAddr(32)	DCommand(2)	DTag(8)	DAddr(32)	DData(32)

Table 1: Data Format for Testbench Input Data File

Encoding	Command
0X	NOP
10	READ
11	WRITE

Table 2: Encoding For Testbench Input Commands

The testbench will receive tag/data pairs from the implementation it is probing as a result of read requests. As the testbench receives these pairs, it will log them into a file in the order it receives them. The format of this file is shown in Table 3. The SimTime is optional and only used if the implementation is being evaluated for performance. Otherwise, it can be omitted or ignored. To check an implementation for correctness, a SUT (System Under Test) is probed with the input data and its output is logged. Then, the reference implementation is probed with the same input data. If and only if the outputs are identical (disregarding the order of lines and the SimTimes), then the SUT provided the correct output. The SUT must pass all possible input stimuli in order to be verified as functionally correct. Various types of input stimuli are discussed in Section 3.3.

Tag(8)	Data(32)	Instr(1)	(SimTime)

Table 3: Data Format for Testbench Output Data File

3.2 Performance Tests

The actual implementation of the memory controller will be tested for performance while tweaking the heuristics involved with the memory access scheduling. In order to gauge a measure of performance, the SimTime in Table 3 must be logged. The memory controller with heuristics activated will be compared to a memory controller with simple in-order execution of its memory operations. Comparing the timestamps of the results on these two models will provide meaningful estimates of latency, bandwidth and jitter.

3.3 Input Stimuli

Various input stimuli will be used to verify correctness and test performance. A very simple test input will be one that writes to several sequential memory addresses and then reads out of those memory addresses in sequence. To gauge the range of possible performance increases due to memory access scheduling, the two extreme cases of memory accesses will be tested. First, the worst case input stimulus will be accesses to different rows within a single bank. This will force the DRAM to precharge after every column access, preventing memory access scheduling from having any benefit. The opposite case scenario is repeated accesses to a single row in a single bank. In this case, the scheduler should realize that it doesn't need to precharge and begin returning data with full bandwidth and the minimum CAS latency. Another useful test case is repeated accesses in different rows while striping across all banks. Here, memory access scheduling will be beneficial because one bank can be precharging while another is being activated or accessed.

Another input stimuli on these implementations will be random accesses. While not necessarily useful from a performance analysis standpoint, this test could discover any verification errors that may not be caught by the hand-crafted test cases. Final tests to run are stimuli using real life memory traces of memory intensive operations, such as graphics processing, stream processing, FFT, and other publicly available memory traces on the Internet. These will provide significant insight into the potential benefit of using memory access scheduling in a real PC.

4 Micro-architectural RTL Design

4.1 Cache Controller RTL Design

An initial design of a direct-mapped blocking cache with single-word cache lines has been completed. The next step is to convert to a non-blocking cache with single-word cache lines; since this can be done without modifying any lower levels at all. When this is completed, the design will be changed to four words per cache line.

4.1.1 Blocking Cache

The cache is implemented as a regfile. Each cache line consists of a valid bit, tag bits, and data bits. All are fully parameterizable. No dirty bit is necessary since the cache is write-through, no-write-allocate.

On a read, the cache controller examines the line with the given index, and on a tag hit, returns the found data. On a miss, the `waiting` flag is set, indicating that no new requests should be taken (the cache is blocking). A read request is then sent to the scheduler. When the scheduler returns the data, the cache is updated, the data is returned to the processor, and the `waiting` flag is cleared.

On a write, the cache controller examines the appropriate cache line. If there is a tag match, or the line is invalid, the data is written to the cache. Whether or not there is a cache hit, the data is also sent to the scheduler to be written.

With only a few simple changes, the original cache was converted to one multiple words per line. Specifically, different Bluespec data types are used to package data and some control statements are added to choose the appropriate word. While the implementation in Bluespec is obviously, the synthesized hardware should not be very different all.

Conversion to multi-word cache lines required scheduler changes as well. The scheduler controls all DRAM accesses, including initialization. Therefore, the DRAM burst mode must be changed to a larger burst length. Additionally, the scheduler must be prepared to return four words at once. Parameterizing this proved difficult, and since arbitrary burst lengths are not possible with the DRAM, the burst length is fixed at 4. Fixing the cache line size makes this rather easy to implement. A burst length of four is useful to maximize throughput on our DRAM data lines. Similarly, this provides some prefetching, which will improve processor performance by minimizing memory accesses.

4.1.2 Non-blocking Cache

The non-blocking cache must be able to service cache hits while still processing cache misses. For a non-blocking cache, it is necessary to store the addresses of pending reads in order to write them back to the proper cache language. On a cache miss, an entry is added to the pending request buffer (PRB). The PRB is a small direct-mapped memory that stores the requested address along with four valid bits and four processor tags. Also present is a buffer tag to determine whether or not a PRB access is a hit. When a read miss occurs, first the buffer line corresponding to that address is checked to see if it's a PRB hit. If it's a miss and there are valid bits set, then there are outstanding requests to another address, and so the new request cannot be sent. If none of the valid bits are set, the processor tag is added to the correct field (depending on which word in the block has been requested) of the PRB, the corresponding valid bit is set, and a read for that block is sent to the scheduler, with the tag being the PRB index. If another miss occurs to the same block, another valid bit is set and tag entered, but no read request issued. When the scheduler returns the data, the tag is used as an index to the PRB, and words are unloaded to the processor one at a time, according to which valid bits are sent.

If a store miss occurs, the PRB is checked. If a read has been issued to that block already, the

store is blocked. This prevents stale data from being stored to the cache (WAR hazards). This could have been solved with dirty bits, but that became overly complex. Blocking also occurs if there are two consecutive reads to the same address since there will be no room in the PRB for the second entry.

4.2 Scheduler RTL Design

Two designs were implemented for the scheduler: an in-order scheduler and the true memory access scheduler. As discussed in Section 3, these two implementations allow for a comparison of scheduling policies with the traditional in-order scheduling used by memory controllers. The micro-architectural RTL design for these two schedulers is discussed in the following two sections.

4.2.1 In Order Scheduler

The in order scheduler takes instruction and data requests from the cache controller and feeds them sequentially into the DRAM. Instruction requests are given priority over data requests. This is done because instruction request bandwidth and latency are often the bottleneck in CPU performance.

The in order scheduler must keep track of the state of the DRAM. Unlike the full memory access scheduler, though, the in order scheduler only needs to track whether the DRAM is busy or idle. If the DRAM is idle, the in order scheduler can proceed with the next pending memory access requested by the cache controller.

Once a memory access is selected, the in order scheduler must process that request and send it to the DRAM. The first step is to activate the row that the request references. The next step is to either write the data or read the data to this row. If a read is chosen, the actual data must be captured several clock cycles later when the DRAM puts the data on the lines. Finally, the bank is automatically precharged so that it is ready for the next command. This scheduler does not employ any heuristics to optimize bandwidth. When either an instruction read or a data read is chosen, the tag is temporarily saved so that it can be returned to the cache controller with the corresponding data.

4.2.2 Memory Access Scheduler

The memory access scheduler employs a more advanced scheduling policy to optimize both latency and bandwidth. Like the in order scheduler, it receives both instruction requests and data requests from the cache controller. The memory access scheduler first sorts the requests into data structures corresponding to each bank of the RAM. For the four-bank DRAM, there are four such data structures, and each one contains the pending requests to the corresponding bank.

During each clock cycle, a bank scheduler for each bank looks at the bank data structure and determines which operation it recommends for that bank. The recommendation will depend on the policies set for the memory access scheduler. The bank scheduler will recommend either a precharge, a column access, or a row activation, regardless of whether or not that operation can actually occur on the given clock cycle. The bank scheduler does have access to the current state of the DRAM. This is important in many scheduling decisions. For example, the bank scheduler

needs to know which row is open to pick the appropriate command to issue. In another example, it would not make sense to issue a precharge command if there is no currently active row. The bank scheduler is also responsible for issuing refresh commands to the DRAM. These occur periodically and interrupt whatever commands are pending for a given bank.

The recommendations of the four bank schedulers are directly connected to a channel arbiter which is responsible for maintaining the legal timing to the DRAM. When the DRAM is available for a command, the channel arbiter is responsible for picking which of the four recommendations to honor. This decision is based on the scheduling policy. A reasonable policy would be to prioritize column accesses over row accesses and precharges. However, many other policies are reasonable and can be experimented with in the architectural exploration.

4.2.3 Bluespec Scheduler Implementation

This section describes how the scheduler is implemented using Bluespec. The inputs to the scheduler are FIFOs from the cache controllers. Eight separate rules sort the input requests into the four bank waiting buffers. Each of these sorting rules corresponds to a bank and an either the data or the instruction input request queue. There are eight rules (as opposed to 4) to minimize potential write conflicts between the rules. These sorting rules are predicated upon the head of the input queue being targeted towards the corresponding bank of the sorting rule. This prevents multiple rules from firing simultaneously and conflicting.

One other rule performs the function of the channel arbiter. This rule calls a function on each waiting buffer to determine the best request to honor. This decision is made based on the scheduling policy. For example, a request will likely be chosen if it goes to the currently active row. If no row is active in the bank, a request can be chosen that goes to a row with many other requests to that same row. This will allow the row to be accessed more before there are no pending requests and the row must be closed.

The channel arbiter then looks at the suggested requests and decides which one to execute. It will pick a read or write request to an open row if that request can be honored in the current clock cycle. Otherwise, it will try to activate a row corresponding to a bank request. If no rows are able to be activated, the arbiter will perform a precharge to close a currently active row.

The channel arbiter must indicate to the waiting buffers which lines have been processed. There are data vectors corresponding to the same indices as the waiting buffers that have a processed bit indicating if the request has been processed. These are separate data structures so that the channel arbiter can update which lines have been processed in the same clock cycle as the sorting rules sort the input requests into the bank waiting buffers. There is a cleanup rule for each bank that removes all processed entries when the waiting buffer gets full. This is done simply by invalidating all the waiting buffer entries that have been processed. These cleanup rules are predicated on the waiting buffer being full and at least one entry in the processed buffer being processed.

The channel arbiter must keep a schedule of current, future, and former channel state in order to ensure that the timing requirements are met. It does this by making a string of shift registers that track the state of each bank. Every clock cycle, the registers shift so that the future state can be tracked. This allows the channel arbiter to capture the read data coming back from the DRAM. The DRAM was configured with a burst length of 4 to try and keep the data lines busy.

The previous state must be preserved to ensure that DRAM requests are issued too soon after a previous DRAM request.

5 Design Exploration

Because of the problems with compiling the optimized scheduler, design exploration was limited changing the size of the cache and the PRB and determining the effects on performance. Both streaming and random memory access patterns were generated. Streaming accesses did not benefit at all from caching. This was in fact the major impetus for aggressive scheduling.

Results from this exploration are below. 6000 random accesses were generated, 3000 instruction loads and 3000 data accesses. The total time in cycles is plotted in Figure 4 against the PRB size. PRB size directly relates to number of outstanding memory accesses allowed. Note that for larger cache sizes, PRB size becomes less relevant. In particular, for the 512-word cache, PRB size has practically no effect. The random memory accesses cover a span of 512 words, so the entire memory is essentially cached. For smaller caches, there is a clear trend in that more PRB entries speeds up performance.

6 Conclusion

The simplest form of memory access scheduling processes the memory requests sequentially and waits for a row to precharge before moving on to the next request. This scheduler is relatively easy to design and implement in hardware but is not efficient and wastes the DRAM resources as well as slows down the CPU. The true memory access scheduler interleaves DRAM requests and chooses the next request to be honored based on a scheduling policy.

The problem with creating a full-blown memory access scheduler is that the complexity of the system explodes. The channel arbiter needs to look at every memory location in the waiting buffers and thus a massive web of combinational logic. Due to this complexity, it was impossible to compile the full memory access scheduler written in Bluespec. Instead, a simpler system using FIFOs as waiting buffers and limited interleaving functionality was synthesized, shown in Figure 5.

As memory systems increasingly become the major bottleneck for computer systems, there will be a greater push for memory systems like the one described in this paper. Despite their complexity and sheer amount of combinational logic, their superior performance will eventually necessitate their use.

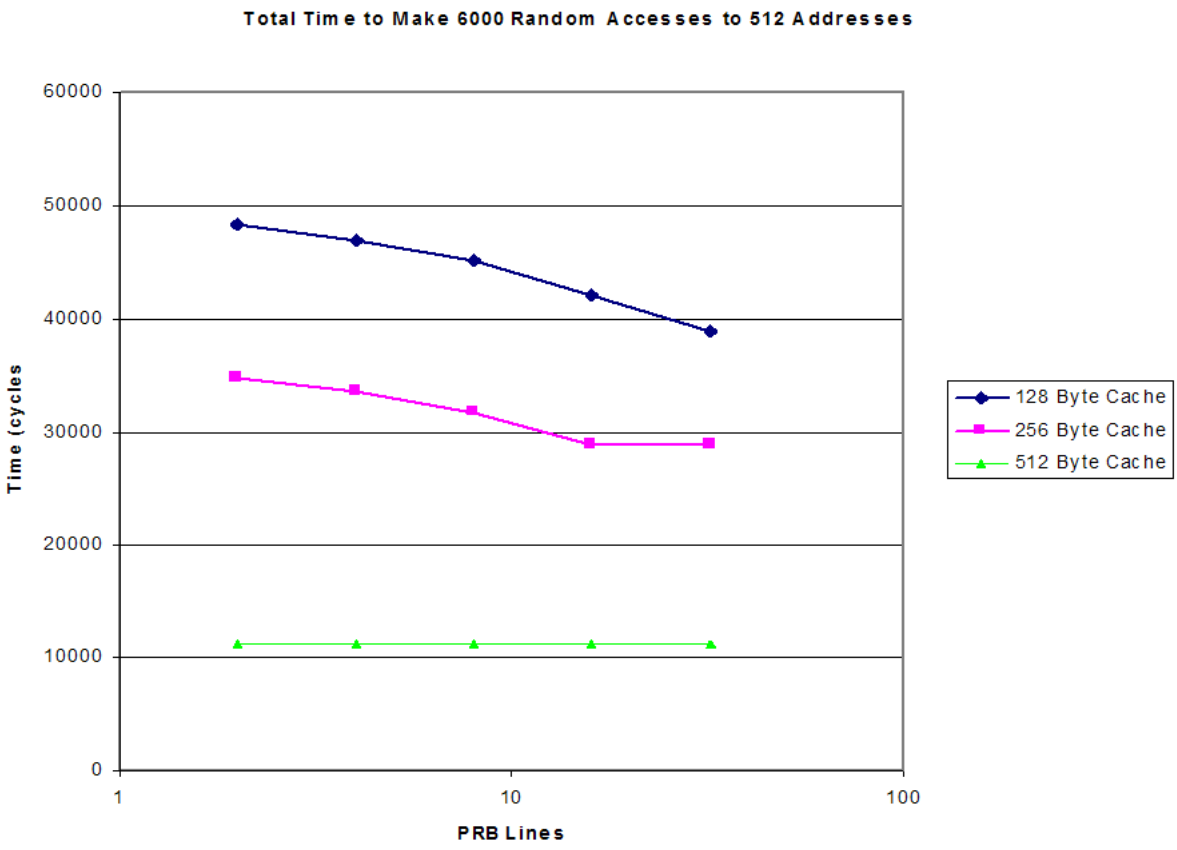

Figure 4: The Effect of Pending Request Buffer Size on Memory Bandwidth

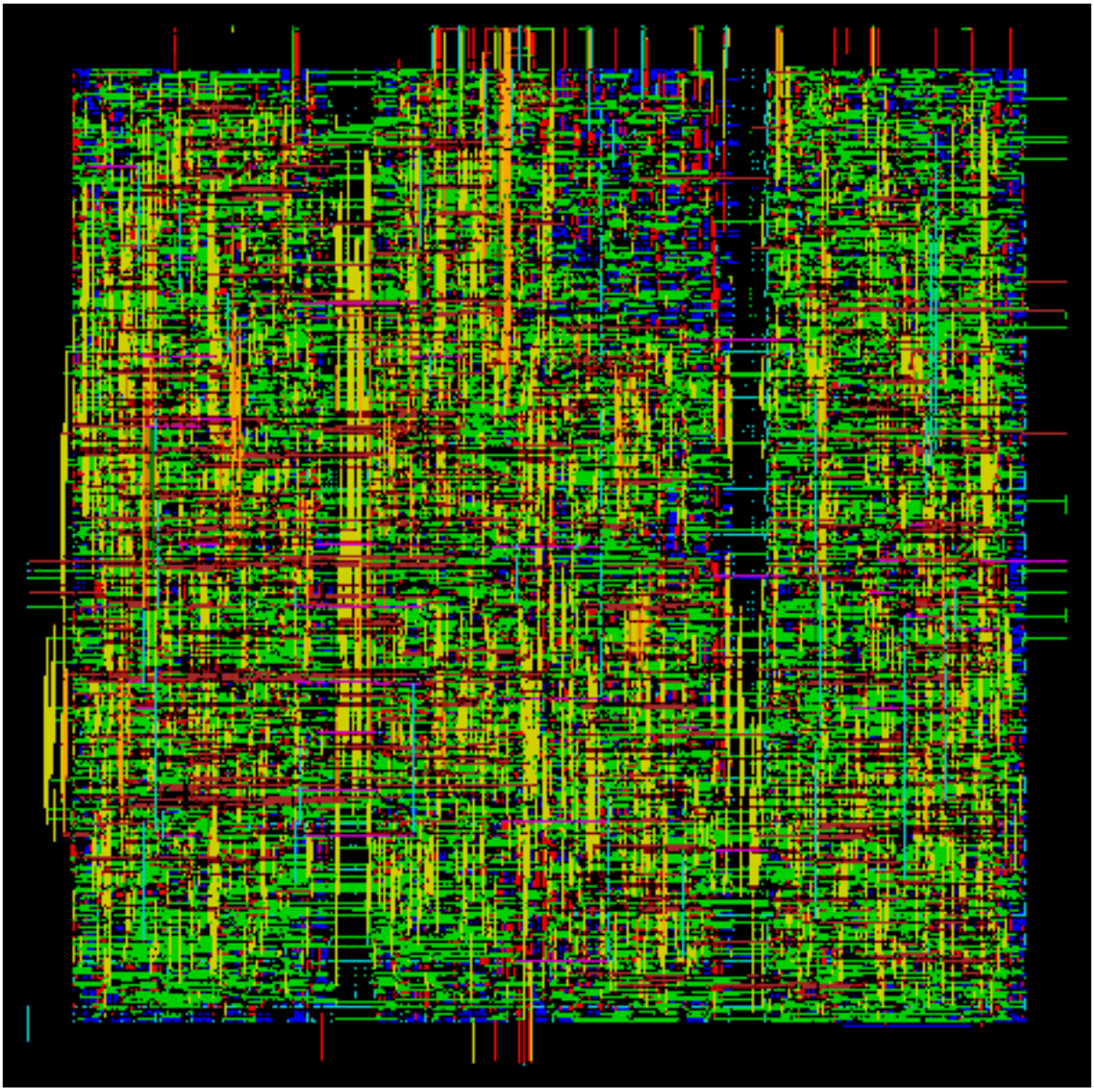

Figure 5: Image of Synthesized Design

References

[1] S. Rixner, W. Dally, U. Kapasi, P. Mattson, J. Owens,
"Memory access scheduling,"
in *Int'l Symp. on Computer Architecture*, Jun 2000, pp 128–138.