

FSRAM: Flexible Sequential and Random Access Memory for Embedded Systems

Ying Chen, Karthik Ranganathan, Amit K. Puthenveetil, Kia Bazargan and David Lilja

Department of Electrical and Computer Engineering
University of Minnesota
200 Union St. S.E., Minneapolis, MN 55455, USA
{wildfire, kar, amit82, kia, lilja}@ece.umn.edu

Abstract

We propose a novel RAM architecture for embedded systems that allows both random-access and sequential access for reads and writes. Using small “links” in each row that points to the next row to be prefetched, our design significantly improves memory access time, while reducing power consumption at the expense of negligible area overhead.

1. Introduction

Rapid advances in high performance computing as well as semiconductor technology have drawn considerable interest in high performance memories. The increases in hardware capability have not only led to a great strain on the power supply but also on the access time required by the memory to read and write data. On-chip memory performance directly affects designers’ decisions on how to allocate expensive on-chip silicon area. Off-chip memory power consumption has become the energy consumption bottleneck as embedded applications are gradually becoming more data-centric and require large storage capacities.

Most of the recent research has tended to focus on improving performance and power consumption of on-chip memory structures [1, 2, 3] rather than off-chip memory. Moon *et. al.* [4] investigate a low-power sequential access only for on-chip memory designed to exploit the numerous sequential access patterns in DSP applications. Prefetching techniques from traditional computer architecture are also used to enhance on-chip memory performance for embedded systems [5, 6, 7]. There are some other studies that investigate energy efficient off-chip memory for embedded systems, such as automatic data migration for multi-bank memory systems [8].

However, none of these studies particularly investigate adaptive random and sequential access support or hardware-supported prefetching schemes on off-chip memory structures. Our study demonstrates a novel low-power off-chip memory structure with flexible access patterns, which is called flexible sequential and random access memory (FSRAM). Besides normal random access, FSRAM

uses an extra “link” structure for sequential access that skips the row decoder and thus saves power consumption, and decreases delay access times considerably. Moreover, the link structure aggressively prefetch data into on-chip memory. In order to eliminate potential data cache pollution caused by prefetching, a small fully-associative prefetch buffer is used in parallel with data cache. VHDL and HSPICE models of FSRAM have been developed to evaluate the effectiveness at circuit level. In order to show system level performance improvement, architecture level simulations of embedded and multimedia applications are used. Our results show that the performance of the memory can be significantly improved with little extra area used by the link structures.

The rest of the paper is organized as follows. Section 2 discusses the related works. In Section 3, the idea of flexible sequential access memory is introduced and explained. The experimental setup and timing and system level performance evaluations are described in Section 4. Finally, Section 5 summarizes and concludes.

2. Related work

The research related to this work can be classified into three categories: on-chip memory optimizations, off-chip memory optimizations, and hardware-supported prefetching techniques.

In their papers, Panda *et. al.* [1, 2] address data cache size and number of processor cycles as performance metrics for on-chip memory optimization. Shiue *et. al.* [3] extend the previous work to include energy consumption and show that it is not enough to consider only memory size increase and miss rate reduction for performance optimization of on-chip memory because the power consumption actually increases. In order to reduce power consumption, an on-chip sequential access only memory is specifically designed for DSP applications [4] and it demonstrates the low-power potential of sequential access.

A few papers have addressed the issue of off-chip memory optimization, especially power optimization,

address decoder is shared for both memory and the “link” structure, but used as write decoder for the link structure, while the read write decoder is only required for the memory. As can be seen, each memory word cell is associated with a link structure, an OR gate, a MUX, and a sequencer.

The link structure indicates which successor memory word to access in sequential access mode. With the size of 2 bits, the link can link to four successor memory word lines (e.g., in our case N+1, N+2, N+4, and N+8). This is similar to the “next” pointer in a linked-list data structure. Note that Moon *et. al.* [4] hardwired the sequencer cell of each row to the row below it (i.e., only provided access to memory word line N+1). By allowing more flexibility, and the ability to dynamically modify the link destination, we can provide great potentials for speedup, as the row address decoder can be bypassed in many memory accesses.

The OR block is used to generate the sequential address. If any of the 4 inputs to the OR block is high the sequential access address (SA_WL) will be high (Figure 2.a). Depending on the access mode signal (SeqAcc) the MUXes choose between row address decoder and the sequential cells. The role of the sequencer is to determine the next sequential address depending on the value of the link (Figure 2.b). If WL is high then one of the 4 outputs is high as described in the following. However if reset is high then all 4 outputs go low irrespective of WL.

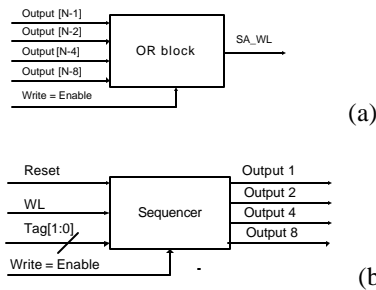


Figure 2.(a) Block diagram of the OR block, (b) block diagram of sequencer.

As shown in the timing diagram (Figure 3), at the start of a new cycle the OR gate generates the sequential address while the random address is generated by the address decoder. The MUX chooses one of the two addresses. The value of the word line

important to note that our FSAM does not *require* the memory to have two ports. The reason we chose two ports was that most modern memory architectures have multi ports to improve memory latency.

is stored in the sequencer while write is still high. Once write goes low, the word line is pulled down. (All word lines are automatically pulled low when write bar is asserted by means of a pull down transistor). During the period while write is low, the sequencer uses the stored word line value and the link value to determine whether any of its outputs should be asserted. This output is now fed to the OR gates. At the start of the next cycle, the process is repeated once again.

The figure shows an example in which a data value of 0 is stored in row 0, and a value of 1 is stored in row 2. The steps involved are:

1. First, row 0 is to be addressed, hence RA/SABar is to be kept high after address is given to the decoder.
2. This signal selects the output of the MUX and activates the memory line.
3. The data bit is written and WRITE goes low, hence WL0 follows.
4. During WRITE low, Pre1Pre0 is made 01 to select the next alternate location, which is number 2.
5. When WRITE goes high next time, the OR gate output is enabled and out02 is sent out into input of MUX at row 2. Since WRITE is kept high, MUX gives out WL2.
6. This combined with high WRITE writes the bit into the location.
7. Steps 3 through 7 repeat for continued access.

DELAY of the random write is : $A + B + E$

DELAY of the sequential write is : $C + D + E$

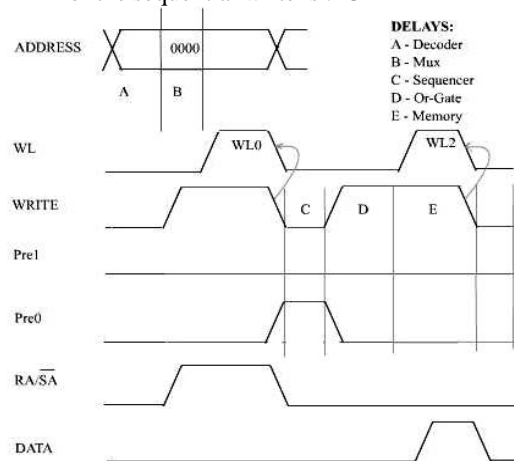


Figure 3. Timing diagram of a memory word cell

Read access can be done similarly. To summarize, a block of data can be read by first accessing the first

element using the row address decoder, and then following the links in the sequential mode. The next subsection discusses how and when the link values are actually written.

3.2. Access Mechanism in FSRAM

We should note that read and write operations to the memory data elements and the link RAM cells can be done independently. The word lines can be used to activate both the links and the data RAM cells for read or write (not all control signals are shown in Figure 1).

There are a number of options for when to write the link values:

1. The links can be computed at compile time and loaded to the data memory while instructions are being loaded to the instruction memory. This is the least flexible approach, but could avoid some control circuitry that support runtime updating of the links.
2. The link of a row could be written while the data from another row is being read.
3. The link can be updated while the data of the same row is being read / written. In our experiments, we consider this case. The dynamic configuration of the links will help in later prefetches as discussed below.

The link associated with each off-chip memory word line is updated according to data cache miss trace dynamically through run-time reconfiguration of the sequential access target. In this manner, the sequential accessed data blocks are linked at compulsory misses. Since the read decoder for memory is write decoder for the link structure, link structure is updated in parallel with memory access. The default link value is 0, which actually means the next line.

In our experiments, the on-chip data cache and the prefetch buffer are accessed in parallel. When it is a miss in both data cache and prefetch buffer, the required block is fetched into the data cache from off-chip memory. Furthermore, a data block pointed to by the link of the data being currently read, is pushed into the prefetch buffer if it is not already in the on-chip memory (*i.e.*, the link is followed and the result is put in the prefetch buffer). When it is a miss in data cache while a hit in prefetch buffer, the required block and victim block in data cache are swapped and a data block linked to the required data block, but not already in on-chip memory, is pushed into the prefetch buffer.

4. Experimental Setup and Performance Evaluations

We implemented the FSRAM architecture in VHDL to verify its functional correctness at RTL level. We successfully tested various read/write combinations of row data *vs.* links. Depending on application requirements, one or two decoders can be provided so that the FSAM structure can be used as a dual-port or single-port memory structure. In all our experiments, we assumed dual-port memories since modern memory structures have multiple ports to decrease memory latency.

In addition to the RTL level design, we implemented a small 8x8 (8 rows, 8 bits per row) FSRAM in HSPICE using 0.18 μ m technology to test timing correctness and evaluate the delay of sequencer blocks. Note that unlike the decoder, sequencer block's delay is independent of the size of the memory structure: it only depends on how many rows it links to (in our case: 4).

4.1. Timing and area

By adding sequencer cells, we will be adding to the area of the memory structure. However, in this section we show that the area overhead is not large, especially considering the fact that in today's RAMs, large number of memory bits are arranged in a row. An estimate of the percentage increase in area was calculated using the formula

$$\text{Increase in area} = \left(\frac{A1}{A1 - A2} - 1 \right) \times 100\%$$

Where $A1$ = Total Area, $A2$ = area occupied by MUX and sequencer. Table 1 shows the results of this comparison. The sequencer has two SRAM bits that are not that many compared to the number of bits packed in a row of the memory. It can be seen that the sequencer cell logic does not occupy a significant area either.

No. of bits per row of memory	Increase in area
8 (1 byte)	216%
16 (2 Bytes)	119%
64 (8 bytes)	23.0%
256 (32 bytes)	7.12%
512 (64 bytes)	3.10%

Table 1. Area overhead of FSRAM with various memory word line sizes.

The above results indicate that the percentage increase in area drops substantially as the number of bits in each word line increases. Hence the area

overhead is almost negligible for large memory blocks.

Using the HSPICE model, we compared the delay of the sequencer cell to the delay of a decoder. Furthermore, by scaling the capacity of the bit lines, we estimated the read/write delay and hence, calculated the overall speedup of 15% of sequential access compared to random access.

4.2. System level performance

To evaluate the system level performance of FSRAM, we used SimpleScalar 3.0 [10] to run Mediabench [11] benchmarks using our memory structure. The processor configurations are based on Xscale [12]: 32KB data and instruction L1 caches with 32 byte lines, 2-way associativity and 1 cycles latency, no L2 cache, and 50 cycle main memory access latency. According to the HSPICE model simulation, sequential access delay is 85% of random access delay. The default prefetch buffer size is 8-entry. The machine is 2-issue, with a 32-entry load/store queue, one of each following units: integer unit, floating point unit and multiplication/division unit, all with 1 cycle latency. The branch predictor is bimodal and has 128 entries. The instruction and data TLBs are fully associative and have 32 entries. The link structure in off-chip memory is simulated by using a big enough table to hold the miss address and their link values. We used four benchmarks from Mediabench [11] for the performance simulations, *adpcm*, *epic*, *g721* and *mesa*.

We are interested in the performance of FSRAM with different on-chip data cache sizes. Therefore, three cache sizes are tested: 2KB, 8KB, 32KB. Due to the small memory footprint of the benchmarks the specific cache size addressed in this study will perform similar to larger caches in real systems, where memory footprint is much larger. In order to evaluate prefetching effects, FSRAM is compared with tagged next line (TNLP) with the same prefetch buffer size.

As shown in Figure 4 (next page), TNLP with a small size data cache outperforms the baseline processor with a bigger size data cache, while FSRAM outperforms both the baseline and the TNLP method. For most of the benchmarks and on average, FSRAM with an 8KB data cache outperforms TNLP with a 32KB data cache. The average performance speedup gained by FSRAM over the baseline processor is 8.6% with a 2KB data cache, 9.3% with an 8KB data cache, and 9.5% with a 32KB data cache. So we propose FSRAM as an effective approach of using off-chip memory to improve performance without expanding expensive on-chip memory.

A number of factors play roles in the improvement in the performance. While the baseline processor does not perform any prefetching, the tagged next line prefetching prefetches only the next word line. The fact that our method can prefetch with a stride, is one contributing factor in the smaller memory access time. Furthermore, prefetching is realized using sequential access, which is faster than prefetching using random access. Another benefit that our method offers compared to traditional prefetching techniques is that in our method, prefetching with different strides does not require an extra large table to store the next address to be accessed.

5. Conclusions

The proposed FSRAM mechanism allows us to eliminate the use of address decoders during sequential accesses and also random accesses to certain extent. The link structure/configuration explored in this paper is not the only one possible. It is very much possible to have a multitude of configurations. Depending upon the requirement of an embedded application, a customized scheme can be adopted whose level of flexibility during accesses best suits the application. For this, apriori knowledge of access patterns within the application is needed. In the future, it would be useful to explore power-speed trade-offs which may bring about a net optimization in the architecture.

Acknowledgement

This work is supported by the Minnesota Supercomputing Institute.

References

- [1] P. R. Panda, N. D. Dutt, and A. Nicolau. "Data cache sizing for embedded processor applications." Technical Report TCS-TR-97-31, University of California, Irvine, June 1997.
- [2] P. R. Panda, N. D. Dutt, and A. Nicolau. "Architectural exploration and optimization of local memory in embedded systems." International Symposium on System Synthesis (ISSS 97), Antwerp, Sept. 1997.
- [3] W. Shiue, C. Chakrabati, "Memory Exploration for Low Power Embedded Systems." IEEE/ACM Proc. of 36th. Design Automation Conference (DAC'99), June 1999.
- [4] J. Moon, W. C. Athas, P. A. Beerel, J. T. Draper, "Low-Power Sequential Access Memory Design."

IEEE 2002 Custom Integrated Circuits Conference, pp.741-744, Jun 2002.

[5] H. Sbeyti, S. Niar, L. Eeckhout, "Adaptive Prefetching for Multimedia Applications in Embedded Systems." DATE'04, EDA IEEE, 16-18 february 2004, Paris, France

[6] A. D. Pimentel, L. O. Hertzberger, P. Struik, P. Wolf, "Hardware versus Hybrid Data Prefetching in Multimedia Processors: A Case Study." in the Proc. of the IEEE Int. Performance, Computing and Communications Conference (IPCCC 2000), pp. 525-531, Phoenix, USA, Feb. 2000

[7] D. F. Zucker, M. J. Flynn, R. B. Lee, "A Comparison of Hardware Prefetching Techniques For Multimedia Benchmarks." In Proceedings of the International Conferences on Multimedia Computing and Systems, Himshima, Japan, June 1996

[8] V. De La Luz, M. Kandemir, I. Kolcu, "Automatic Data Migration for Reducing Energy Consumption in Multi-Bank Memory Systems." DAC, pp 213-218, 2002

[9] C. Yang, A. R. Lebeck, "Push vs. Pull: Dataq Movement for Linked Data Structures." In International Conference on Supercomputing 2000 (ICS'00), pages 176--186, May 2000.

[10] Doug Burger and Todd M. Austin. "The simplescalar tool set version 2.0." Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems." In Proc. of the 30th Annual International Symposium on Microarchitecture (Micro 30), December 1997

[12] Intel corporatin, "The intel XScale Microarchitecture technical summary", Technical report, 2001

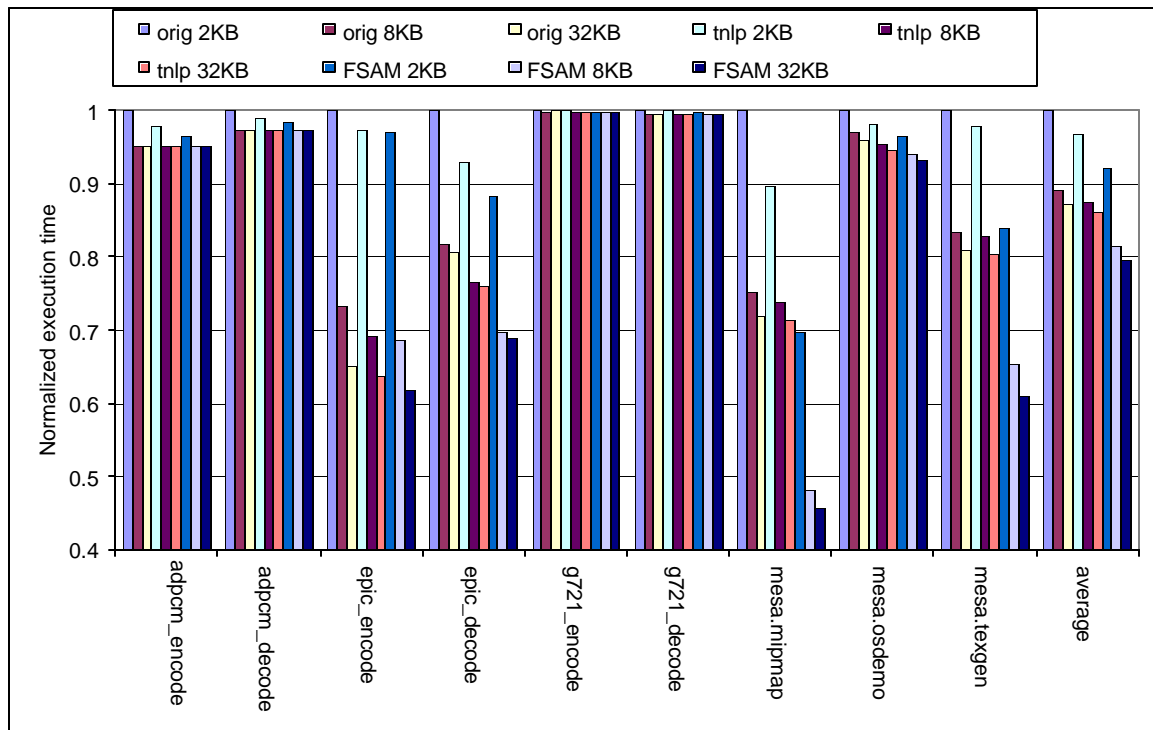


Figure 4. Performance results of FSRAM, next line prefetching (TNLP) and the baseline processor (orig) shown as normalized execution time with various data cache sizes (2KB, 8KB, 32KB). The baseline is the processor (orig) with no prefetching schemes and a 2KB data cache.