Sadram

A New Memory Addressing Paradigm

Robert Trout
Sadram, Inc.
Lititz PA USA
hrg.trout@gmail.com

David Lynch
Sadram, Inc.
Lititz PA USA
davidLynch@dlasys.net

ABSTRACT

The purpose of the Sadram Architecture (Self Addressing DRAM) is to minimize CPU ↔ memory traffic used merely for address computations. These savings are achieved by providing a symbolic addressing mode alongside the conventional linear mode. Linear addresses, such as 0,1,2,3,…etc. are dense, numeric, and inextricably bound to the structure of memory; symbolic addresses, such as 'hi', 'bye', or 'gone', are not linear, not dense, and closer to real world addressing. Sadram provides direct benefits to the user in addition to improving access efficiency.

Sadram CONCEPTS

- Memory rows in sorted order.
- Parallel processing of a row buffer.
- Pipeline data movement.
- Configuration of memory cells.

KEYWORDS

- Symbolic memory addressing.
- Sequencer-array, Sequencer-group, Sequencer-cell.
- Pipeline data movement.

1 Introduction

The traditional separation of memory from the CPU, known today as 'von Neumann architecture' actually goes back to the pioneering work of Charles Babbage. [1] His motivation was economy of logic; the price of this economy was moving data between memory and the CPU ('mill' in his terminology). However, in modern systems data movement has become *the* bottleneck and constitutes a significant fraction of the total power budget.[2,5]To lower the cost of data movement the concepts of PIM (process in memory), PNM (process near memory) and PUM (process using memory)[5] arose in the 1980's when memory transitioned from core memory to semiconductor memory. The challenge then, as now, is what functionality should be embedded in the memory – too much encroaches on tasks more suited for the CPU, too little yields no gains. Sadram takes a middle ground which can be described as 'address management'. Addresses are abstracted as symbolic values and mapped inside the memory. This saves multiple CPU accesses used merely for addressing tasks.

The importance of memory to computing was recognized as early as the 1850's by Charles Babbage.[1] It has been recognized for decades that memory is a significant, if not the principal, constraint on system performance[2,5]. Memory constitutes the largest number of transistors in any system and is usually the largest cost element. Seeking performance improvements in memory that do not rely upon performing the same elementary steps more rapidly is an obvious path to improving system performance.

Memory transistors are not well suited for processing tasks; over the decades they have been optimized to retain data. CPU transistors, by contrast, have been optimized to move data; the two roles are incompatible. Accordingly, Sadram uses a logic layer bonded to the memory and communicating using Thru Circuit Vias (TSV's). This hybrid architecture can be expected to improve performance and save power (the distance between the logic and the memory is much smaller than the distance between the memory and the CPU). Samsung's Acquabolt architecture, which implements A/I primitives in a logic layer uses exactly this concept. Samsung reports 2.5 * speed improvement and 70% reduction in power.[4]

Most memories, including DRAM, are block devices, requiring an entire row (block) of data be read to/written from a row buffer to access a single bit. This is antagonistic to the very purpose of DRAM which is to provide *random* access. Sadram exploits the blockiness of DRAM to perform parallel row-wide computations.

This parallelism is expressed as multiple cells spread across the (DRAM) row (Figure 2). Because the cells are numerous they must be simple. Sadram cells perform only three functions: compare a target byte against a memory byte, register a single byte, and move a byte to a neighboring cell. A target bus running *across* the row broadcasts the target (key); fully decoded control-lines are broadcast across the row to each cell.

The function of the cells is to maintain the row in sequential order – hence the name *sequencer-cell*. The row of sequencer-cells is called a *sequencer-array*. The term *sequencer-group* denotes a smaller set of cells encapsulating a single 64-bit word. Comparison of memory versus target values within the group is a sequential process, however, all other sequencer operations are highly parallel.

The sequencer-groups all operate in parallel. The move function of each cell/group exploits a pipeline move – again an inherently parallel process.

A simple CPU (called samPU) is implemented in the logic layer to control these sequencers; samPU incorporates specific opcodes for row scan, move, read, write, and opcodes to access fields within the data structures erected over the sorted rows.

The functions of the sequencer-array are tailored to the size of the user key field and supporting binary fields using a configuration vector. Configuration is used to setup each cell/group for subsequent read/write/scan/move operations. Configuration is the magic by which fixed sized groups accommodate variable keys.

On this simple foundation an immense variety of tasks can be performed. Obvious examples include sort, database index management, and row-wide searches. Less obvious examples include variable length records (using the record number as the key), strings, and garbage collection in support of dynamically allocated objects (using Sadram's sort capability). Surprising applications include improving numerical accuracy (summation errors can be minimized using sorted data), high density data compression, and encryption.

Sadram is currently implemented in an FPGA from which some performance measurements have been obtained. Because Sadram is implemented in logic separate from the memory, only minimal changes are required to the memory itself. Sadram may, therefore, be applied to any block addressed memory technology.

2 Components of Sadram

Sadram creates indexes into user data as that data is written to or read from memory. Ideally this indexing function is completely hidden behind the read/write of the user data. The sequencer is the critical piece of hardware underlying indexing.

An array of sequencer-cells (C0.0, C0.1, etc.) is shown in Figure 2. The cells *simultaneously* compare a single target (T[0], T[1],…) against a vector of values stored in a DRAM row. Based on the results of these comparisons the position of the target within the row is determined. The sequencer then 'right shifts' some part of the row and inserts the target item. The row is thereby maintained in sorted order.

The parallelism inherent in the sequencer is the key advantage of the Sadram architecture. There is no intrinsic limit to the width of the sequencer (C0.0, C0.1, …).

3 The Sequencer-Cell

The cell is the lowest level component of a sequencer. Five cells are shown in Figures 1: cells 0.0, 0.1, 0.2, 1.0, and 1.1. Each cell is connected to 8 bits of the DRAM row and 8 bits of the target. The compare results (cmp-I and rcmp-O) propagate diagonally within a group: cell 0.0 ↔ cell 0.1 ↔ cell 0.2, and cell 1.0 ↔ cell 1.1, etc. As directed by signals on the control bus (generated by the sequencer-group described below), each cell may:

- Load its konfig register from the DRAM byte.
- Compare the target byte with the DRAM byte and reports the results on cmp-O.
- Store the DRAM byte in its internal register.
- Send its register to the next group on the prev-I/next-O bus (shift operation).

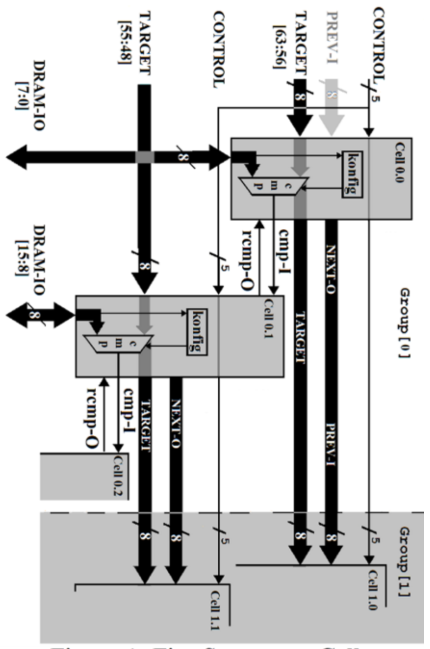

Figure 1: Five Sequencer Cells

4 The Sequencer-Group

Sequencer-cells are aggregated into sequencer-groups. Each group handles multiple bytes constituting a full data word; each byte is assigned to a sequencer-cell. A group is typically 8 cells (64 bits) wide.

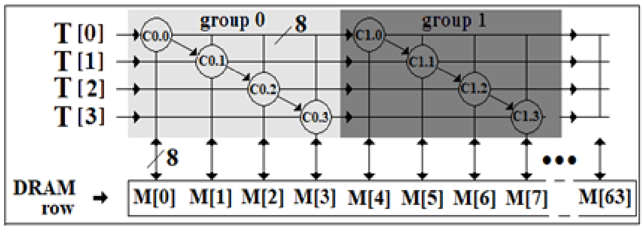

Figure 2: Two Sequencer Groups

The number of cells in a group (called the 'target_bus_size') is a fundamental determinant of sequencer size and performance. The group provides control codes to each of its cells; the action of each cell is determined by these control codes and the results of the cell's compare operation.

The result of each cell's comparison is encoded on a compare-bus which is transmitted from each cell to its neighbor on the right (see Figure 2, cmp signals). The result of the last cell is daisy chained back on a bus called the rcompare bus (Figure 1, rcmp signals). cmp and rcmp are each two bits wide and encode the conditions >, <, or == from the comparison

Sequencer-cells within the group are physically arranged on a diagonal (see Figure 2) so that each cell has direct access to its part of the target bus (T[i] in Figure 2) and direct access to its part of the DRAM bus (M[i] in Figure 2). The compare bus rides this diagonal; the group reports out the result from the last cell in the diagonal. Transmission between cells (ie., down the diagonal) is intrinsically sequential, however, the groups themselves all operate in parallel.

5 The Sequencer-Array

The groups (group 0, group 1, in Figure 2) are aggregated into a sequencer-array. The array is spread across the DRAM row. Each group takes its opcodes from the sequencer-array and generates the control codes for its subordinate sequencer-cells. Each sequencer-array manages a pair of rows.

The primary purpose of the sequencer-array is to maintain these row pairs in sorted order. The array performs this function by orchestrating the compare and shift operations of its groups & cells. The sequencer-array takes its instructions from a small CPU called the SamPU (described below).

The sequencer-array, groups, and cells are capable of maintaining a row in sorted order. They perform these functions by comparing in parallel an incoming target (key) against each datum in their row buffers, right shifting some elements of these rows, and inserting the target in the space thus created.

Overflow of the row pairs is handled by a process called mitosis (described below). Maintaining a pair of rows in sorted order is the most elementary part of the Sadram architecture. Mitosis and higher level tasks such as sort are more complex and not well suited to direct hardware implementation. Accordingly, Sadram splits into two components – a hardware component and a software component: hardware for speed and software for complex functions. The software is supported by a custom designed CPU called a SamPU (described below).

The sequencer-array adds entries to the DRAM row-pair under its control. When these rows becomes full mitosis switches the partners of each row-pair with empty rows (see Figures 4,5&6). Thus two half-full row-pairs are created. These are integrated into the indexbase. Mitosis involves no data movement - it is all done with pointer shifting. Mitosis may provoke a secondary mitosis of the parent page or a tertiary mitosis of the parent book (Figure 3).

6 SamPU

SamPU is a conventional register machine with a 16-bit opcode and data width equal to the target bus size (typically 64 bits). SamPU incorporates specialized opcodes to control the sequencers and opcodes to read/write fields of the indexbase. These opcodes are *speed critical* and are implemented directly in the logic layer. Other opcodes such as conditional jumps and arithmetics are less demanding of speed. SamPU executes the opcodes from a program store**.** Both SamPU and the program store are implemented in the logic layer.

7 The Program Store

The program store is a combination of read only memory (ROM) and read-write memory (R/W). The ROM memory is required to provide basic services at boot-up and the R/W memory is loaded from DRAM after the operating system is loaded. Both the ROM and the R/W memory are part of the same 16-bit address space. Overlays allow programs to exceed the SamPU 16-bit (65536) address range.

R/W memory is intrinsically more costly than ROM but has several advantages:

- R/W mem allows changes to be implemented in the field.
- R/W mem can be overlayed from DRAM if required.
  This makes the size of a program essentially unlimited.
    - Code can be loaded for manufacturing or diagnostics.
    - Code can be loaded for special tasks like encryption.

SamPU opcodes are designed to support multiple indexes simultaneously; this could be implemented using multiple sequencers or by the (slower) technique of multiplexing target keys through a single sequencer.

8 The Indexbase

A database of indexes, called an indexbase, is created in SRAM implemented in the logic layer. The indexbase allows access to acquire some 'understanding' of the data it is storing rather than operate as a blind clerk storing and retrieving data at absolute addresses. This understanding is encapsulated in the indexbase. The indexbase allows the user to access the data in sorted order, access by key, and perform other functions. The logical structure of the indexbase is illustrated in Figure 3.

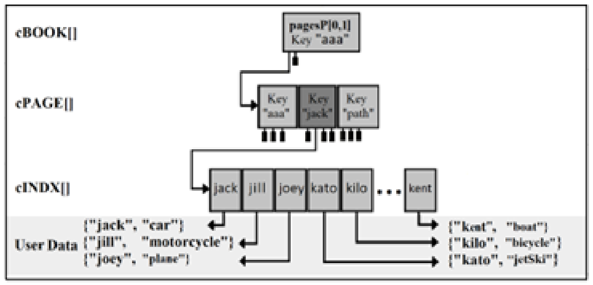

Figure 3: Structure of Indexbase

The lowest level in Figure 3 contains user data. This is not part of the indexbase. The data is naturally organized into records, such as {“jack”, “car”}; one of these fields (“jack”) is designated as the key field and the remainder are non-key fields such as “car”. The three levels of index erected over this data are:

- cINDXs - containing the key and the address of the user-data record.
- cPAGEs - containing the addresses of two cINDX[]s and a copy of cINDX[0].key.
- cBOOKs - containing the addresses of two cPAGE[] and a copy of cPAGE[0].key.

All three structures are stored in SRAM but they are paged to DRAM if necessary; the priority of retention in SRAM is:

cBOOK > cPAGE > cINDX

cBOOK and cPAGE have identical structures. The physical layout of the sequencers is dictated by the hardware but they are configured differently by software to support cINDX[], cPAGE[], or cBOOK[]s, and keys of different sizes.

9 Mitosis

The number of cINDXs, cPAGEs, or cBOOKs that can be stored in a single DRAM row is limited by the size of that row. The addition of a new key to the indexbase will cause a new cINDX entry to be added to a cINDX[] array. If that row is already full a process called Mitosis is invoked.

Mitosis expands the space available to the cINDXs. Mitosis at the cINDX level may provoke mitosis at the cPAGE level, which may in turn provoke mitosis at the cBOOK level. cPAGE comprises a pair of cINDX[] arrays called loP and hiP (ABCJ and KLXY in Figure 4). (A cBOOK comprises a pair of cPAGE[] arrays rows called pageL and page H; the cBOOK is not shown in Figure 4 but mitoses the same way as a cPAGE). The paired arrays loP and hiP are inter-connected to operate like a single array for most purposes. However, they operate as two separate arrays when mitosis is invoked. Mitosis takes loP & hiP (or pageL & pageH) and pairs them with empty rows, then adjusts the pointers in the parent structure to reflect this new reality. Mitosis is merely pointer shuffling; it neither moves, adds, or deletes user data.

The steps involved in mitosis are illustrated in Figures 4, 5, & 6.

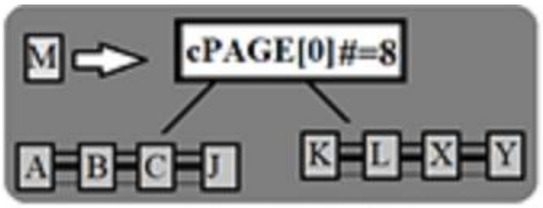

Figure 4: Before Mitosis

The indexbase comprises a single cPAGE pointing to a pair of cINDX[]s (ie., ABCJ and KLXY).

- cPAGE[0] contains a total of 8 cINDXs.
- Both cINDX[]s are full.
- Addition of ‘record M’ triggers mitosis.

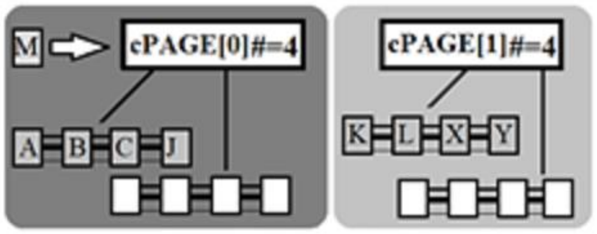

Figure 5: After Mitosis

After Mitosis the indexbase comprises two cPAGEs.

- Each cPAGE points to two cINDX[]s.
- Half of the cINDX[]s are full and half are empty.

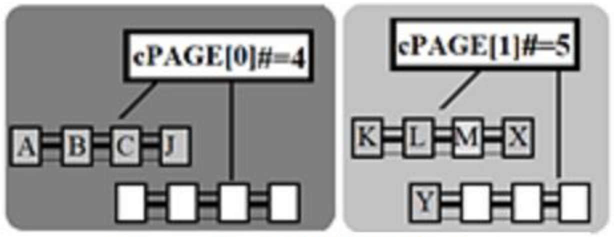

Figure 6: Insert Following Mitosis

After mitosis is complete the indexbase can accept the new entry:

- M is inserted in cPAGE[1]
- ‘Y’ moves from cPAGE[1].loP[3] to cPAGE[1].hiP[0]
- cPAGE[0] now has 4 cINDXs
  cPAGE[1] now has 5 cINDXs

As Figures 4 and 5 illustrate mitosis leaves rows partially filled. Subsequent writes may fill up these partially filled rows (Figure 6), but in the worst case they will remain half full indefinitely.

10 SamCompile

The software for SamPU is written in a high-level 'register transfer language'. The language includes constructs such as if … then … else, while statements, do statements, and for statements. Raw opcodes are also part of the language including call and return. Register transfer languages use the syntax $0 = $1; and $0 = $4 + $5 etc. (where $number refers to one of the SamPU registers). Programs are written in a plain ASCII file and translated to object code using a compiler called SamCompile.

SamCompile incorporates a macro expander, using an extended version of the C++ #define syntax. The compiler outputs binary or hex code directly and performs the function of a linker. The output code can either be incorporated into ROM, loaded into SamPU program storage, used by software emulation, or loaded onto an FPGA emulation board.

The code may incorporate overlays. The code space for a SamPU program is limited by the 16-bit address, ie., 2^{16} bytes; this is divided into a root area of 49152 bytes and an overlay area of 16384 bytes. A maximum of 255 overlays may be loaded into the overlay area. The maximum size of a SamPU program is therefore 49152 + 255 * 16384 = 4,227,072 (0x408000) bytes. The root module may be implemented in ROM, however, overlays must be implemented in R/W memory. Both types of memory are implemented in the logic layer.

11 SamPU.exe, SamPU, and Complicit Coding

SamPU.exe is a GUI debugger which runs on a PC host; SamPu.exe communicates with the CPU using a number of different mechanisms depending upon the $platform:

- Direct software access for the software emulation.
- USB for the FPGA emulation.
- Direct port access for hardware implementation.

The print and $assert opcodes are executed on the host. The two processors (host and SamPU) are complicit in these opcodes – a technique called complicit coding. Complicit opcodes pass binary packages of information to the host for interpretation. *These opcodes are for debugging purposes only*; no text data, symbol table, or line maps are stored in the SamPU processor.

Test programs loaded into the overlay area are heavy users of the print and $assert opcodes. These programs are used to verify the operation of the microcode, for manufacturing tests, for development, and are available for custom user routines.

12 Applications of Sadram

There are many applications of Sadram. Obvious applications include sort, database functions, and row lookup. Less obvious applications include variable length records (the record number is used as a key and the records are 'sorted' based on this key), sparse array storage, and file directory management.

Some surprising applications include numerical accuracy control; sort is free, so a vector of values can be automatically stored in absolute magnitude order. Summation from lowest to highest avoids rounding losses.

String processing (using the garbage collection paradigm) are another opportunity for improvement under Sadram.

Even more surprising is that sequencers can be used for certain arithmetic functions such as integer division and record packing.

12.1 Sort

Sort is a fundamental algorithm that consumes a very large percentage of computer resources. In the 1960's Knuth estimated that 25% of all computer power was used for sort and in some installations it was as much as 50% [3]. Many algorithms (such as databases) do not use sort overtly but still organize their data in sorted structures.

Sort is a straightforward application of Sadram. The indexing is done as the data is written to memory, ie., indexing is overlapped with writing. At the end of the write-process, the data can then be accessed in sorted order. Sort is nothing more than writing with a direct addressing protocol and reading back with a different (ie., symbolic) protocol.

12.2 Database Functions

Databases are innate users of sort. A database is organized so that any datum can be quickly retrieved without resorting to a breadth-first search. This implies that the data is ordered according to *some* principle (key hashing, time stamp, quick key, etc.); but sorted order is the most common principle. A database is typically a block of user data with indexes (often in separate files) that point to the user data. Databases permit record deletion and record replacement; these operations require the indexes be adjusted. Such operations differentiate databases from sort. But the central advantage of the Sadram architecture remains – the indexing is done at the hardware level and does not require external CPU 'address computations'.

12.3 Variable-Length Data

Conventional DRAM addressing can only access an array of data by organizing that data in fixed length records. The address of an individual record is computed with a simple linear equation. Hidden behind this simplicity is virtual addressing which remaps the linear address to the physical address. Sadram condenses these two operations into a single symbolic address and performs the address resolution in the memory itself. Sadram uses a table-lookup paradigm so records can be variable in length – an architecture which dramatically reduces memory requirements.

12.4 String Processing

String processing is uniquely difficult for fixed length processors. CPU architecture does not naturally support variable length items (such as strings). In addition, string processing often involves creating multiple substrings each of which points to a part of a longer string. There are a number of techniques for handling variable length strings. One of the oldest, and most efficient, relies upon a technique called garbage-collection. Strings are allocated, subdivided, and combined in a large area of memory called a string pool. The pool eventually fills with active and dead entries – dead in the sense that there are no active references to the string or part thereof. At this point a process called garbage collection is invoked. Garbage-collection involves compacting the string pool and removing unreferenced strings. The process is non-trivial because references may be overlapping as a consequence of the substring operation noted above.

Garbage-collection requires a *sort* of valid references. This is an obvious candidate for Sadram Sort. However, Sadram can do much better than merely improving the performance of sort. As described in earlier sections the concept of Sadram is that the references to data (keys) are extracted as the data is handled for other purposes. In the Sadram architecture these references are maintained in a sorted structure, so that when garbage-collection becomes necessary the sort is already done; garbage-collection can focus on the compaction phase. The compaction phase itself exploits the shifting operations of the sequencer.

13 Compiler Integration

It is not sufficient to create a new memory addressing paradigm and expect the public to embrace the paradigm without also providing the tools to exploit this technology. Sadram is intended to be implemented in hardware and anticipates layers of software to take advantage of Sadram functions. This can be provided at two levels:

- By way of C++ library calls.
- Extending one of the standard languages to incorporate syntax to exploit the Sadram architecture. Strings are a case in point and notably absent in the standard C/C++ language.

Integrating Sadram style strings in a standard C++ or Perl compiler is under active development.

REFERENCES

[1] Doron Swade, 2001. *The Cogwheel Brain*, ISBN 978-0-349-11239-8, Abacus, London.

[2] R.L. Sites, 1996. "It's the Memory Stupid!" *Microprocessor Rep*. 10, 10 (Aug. 1996).

[3] Donald E. Knuth, 1998. *The Art of Computer Programing, Volume 3*, 3, ISBN 0-201201-89685-0, Addison-Wesley, Boston, MA.

[4] Samuel K. Moore, 2021. "Memory Chips that Compute will Accelerate AI", *https://spectrum.ieee.org/processing-in-dram-accelerates-ai*

[5] Onur Mutlu, Saugata Ghose, Juan Gomez-Luna, and Rachata Ausavarungnirun, "A Modern Primer on Processing in Memory", *https://people.inf.ethz.ch/omutlu/pub/ModernPrimerOnPIM_springer-emerging-computing-bookchapter21.pdf*