Towards a memory-address translation representation scheme

R. Madhukar Yerraguntla
NXP Semiconductors, Noida
rathnakar.y@nxp.com

Ravi Shankar Gupta
NXP Semiconductors, Noida
ravishankar.gupta@nxp.com

***Abstract*—Verification with application executables is common phase in virtual development kits (VDKs), RTL simulations and emulation. It involves loading and dumping application hex images to/from memory abstractions, usually modelled as 2D arrays. This is usually achieved in 2 ways (i) frontdoor loading using design modules mimicking real silicon (ii) backdoor loading using external methods such as simulator API to initialize the design in an "image-loaded" state. The former is slow and inefficient since the design spends a lot of time in loading process along with additional design modules for support. The latter can be performed efficiently without additional design modules but requires a lot of platform-specific infrastructure with memory-dependent details (for ex: ECC, endianness, controller size). In this presentation we argue that a succinct representation of such details is possible for most memories. Such a representation is possible because of stereotypical operations on the memory abstractions. We show that tools processing such representations dramatically reduce the maintainable code size.**

I. Introduction and Background

A usual verification scenario at SoC and subsystem level involves running eventual application executable on the design. This task involves compiling the application code with GCC and converting it into a form amenable to be loaded directly into the design's memory arrays. The problem in such a conversion is that it varies from memory to memory and depends on various memory specific parameters (banks, controllers, interleaving) to data modifications while writing to the memory (ECC, encryption, parity etc.). The usual approach in tackling this problem is to maintain memory specific scripts which are brittle to changes and, hard to maintain and migrate. In this work we attempt to reduce this maintenance and bootstrap memory backdoor loading efficiently in a verifiable manner.

II. Related Work

The work [1] also attempts to solve a similar problem that our work addresses, but in a simulation verification setting. In [1], the authors develop a SystemVerilog driver to preload and manage memories during runtime. Their architecture requires a model of the memory to generate the data to backdoor load. In contrast, we aim to standardize the memory-address translation scheme in a concise format, that can independently verify the assumptions on the addressing scheme. Instances of such memory drivers can be auto-generated depending on the verification scenario. We illustrate one such example in the next sections.

III. Methodology illustration

Our approach involves representing the address translation logic (represented by physical parameters of the memory like controllers, banks, bank depth etc.) separate from the data modifications (ECC, parity, encryption). This approach drastically reduces the maintainable code to just that of data modifications and hence enables better re-use.

Consider a model of 256 KB memory to be preloaded with the application as in Figure 1. The memory is comprised of 8 controllers, each of which have a single bank with a depth of 4096 64-bit words. The input 64-bit data is appended with a 8-bit ECC and split into 4 partitions of 20 bits each (64 (data) + 8 (ecc)+ 4*2 (redundancy bits of 0) = 80 bits). After a controller is filled with 4096 units of data, the address assignment moves to the next controller and the process repeats.

This memory can be represented by the specification shown in Figure 2. The specification shows the start and end addresses of the memory along with the usual 8 number of controllers. ECC itself is modeled as a plugin that is executed once per address and data pair. The field *recurrence* with a value of 0 represents that there is no interleaving between the controllers. In other words, each controller is fully filled with data before moving to the next controller. The rest of the details in the specification determine where and how the redundancy bits are filled. The *hierarchy* field is the hook from the specification to the concrete location in the design array. For a given address, the framework computes the controller, bank, partition and the array index numbers which are filled in the hierarchy template to get the exact location(s) in the design. A single logical address could be made up of multiple entries from different partitions as the example illustrates.

A dry run with just the configuration yields the address arrangement that the framework computed and this can independently verify the assumed translation logic.

IV. Methodology's Datamodel

In this section we discuss the datamodel that drives methodology. In RTL the memories are modeled as 2D arrays. For example, the memory in Figure 1 would be 4 instances of `reg [4095:0] mem[19:0]` and the bus views the memory as a 64bit device. A tool implementing this methodology performs the task of the controller that processes and saves the data in the data arrays.

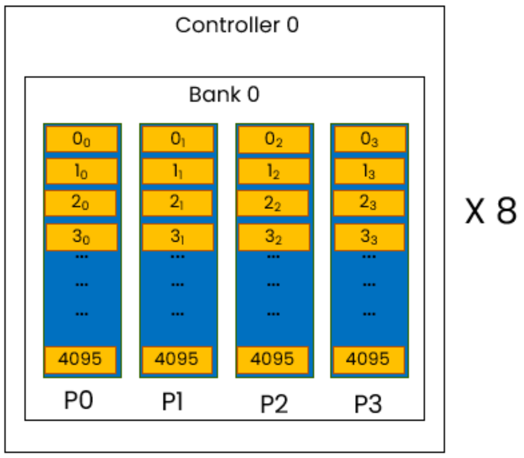
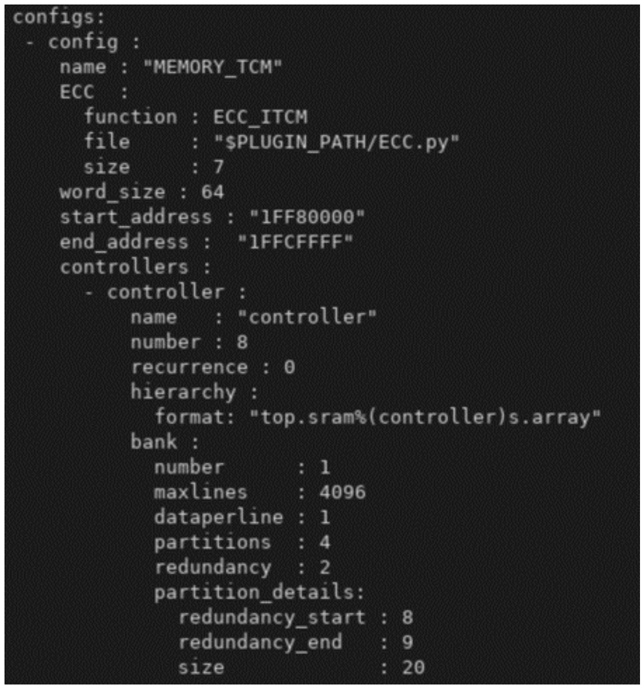


Fig. 1. A Simple 256KB memory



```
configs:
 - config :
    name : "MEMORY_TCM"
    ECC  :
      function : ECC_ITCM
      file     : "$PLUGIN_PATH/ECC.py"
      size     : 7
    word_size : 64
    start_address : "1FF80000"
    end_address :  "1FFCFFFF"
    controllers :
      - controller :
          name   : "controller"
          number : 8
          recurrence : 0
          hierarchy :
            format: "top.sram%(controller)s.array"
          bank :
            number      : 1
            maxlines    : 4096
            dataperline : 1
            partitions  : 4
            redundancy  : 2
            partition_details:
              redundancy_start : 8
              redundancy_end   : 9
              size             : 20
```

Fig. 2. Memory Specification

A. *Memory's Interface Details*

The memory interface section contains the name (a unique reference within the framework), address range, the bus size (`word_size`) along with ECC/encryption details. It is supposed to encapsulate all the tasks performed by the memory controller on an address-data pair. Multiple address ranges assigned to a memory, can be described using `alt_interfaces` keyword. Consistency checks are auto-run on the address spaces (same ranges assigned through each interface, for example). The `ECC` sub-section describes encryption and ecc details. The function is expected to take a data and address, and return the modified data to be written into the data array. To reverse this data modification, an optional function can be provided that reconstructs the original data but the default placeholder function removes the start `size` bits of the modified data.

B. *Physical Memory Information*

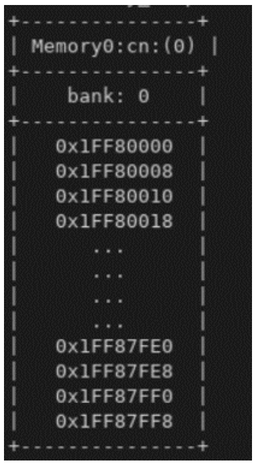
The physical layout of the memory arrays is described in the controllers section. Multiple controllers can be provided as a list in this section. Each controller is assumed to hold one or more memory arrays, and is assigned a contiguous range of addresses determined by the `recurrence` value and `dataperline` (of the underlying bank set). A bank is the smallest hierarchy where a logical unit of hex data can be constructed. It can consist of multiple arrays but each of the individual arrays, do not hold the complete data. In Figure 1, there is exactly one bank under each controller and each bank consists of 4 arrays (called partitions) which hold 20 bits of data in a single array entry (consists of input bus data, redundancies and ecc data). All 4 data components together need to processed to reconstruct the original 64 bit data at a particular address.

The controller `number` is a concise way to represent a list of controllers with the same underlying structure. One could have written out the controller structure 8 times in Figure 1 and Figure 4, and achieved a similar effect. The controllers in Figure 1 are contiguous address range of `0x8000 (4096*1*1*64/8 bytes)` addresses each. The special `recurrence` value of 0 fills all of the current controller's arrays before moving to the next controller. The configuration for Figure 4 would be similar to Figure 2 with the exception of controller `recurrence` being 4 (each of the current controller's arrays are filled for 4 lines before moving to the next in round-robin order).

Banks under a given controller also can have a recurrence but the framework performs consistency checks. For example, the total sum of recurrence in the banks must divide the controller recurrence. By default the banks are considered to have a recurrence of 1. In other words, the data is distributed `dataperline` units at a time to each bank.

Additional details regarding the data arrays within a bank include the positions of redundancy bits. Our claim is that this datamodel suffices to represent the address translation scheme of a wide range of memories including TCMs, BootROMs and SRAMs. The tool supporting this methodology generates mapping from each address in the memory range to memory hierarchies. For example, for address `0x0` in Figure 1, the mapping has 4 different hierarchies with controller value 0, bank value 0 and partition range $0-3$.

1) Linking the abstract model and Design: The hierarchy section uses the mapping and determines the de-



```
+--------------+
| Memory0:cn:(0) |
+--------------+
|   bank: 0    |
+--------------+
|  0x1FF80000  |
|  0x1FF80008  |
|  0x1FF80010  |
|  0x1FF80018  |
|     ...      |
|     ...      |
|     ...      |
|     ...      |
|  0x1FF87FE0  |
|  0x1FF87FE8  |
|  0x1FF87FF0  |
|  0x1FF87FF8  |
+--------------+
```

Fig. 3. Tool-generated address arrangement for Figure 2

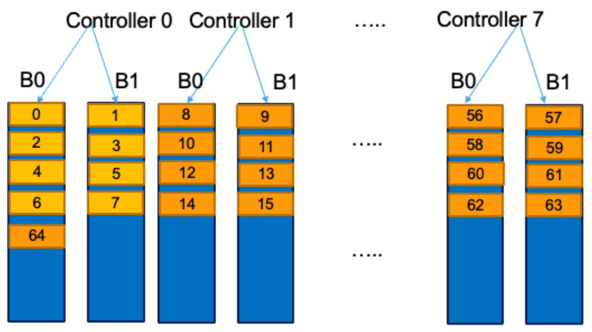


Fig. 4. Example SRAM Memory

sign hierarchy corresponding to each memory address. The computed values are substituted for in the corresponding keys, namely `%(controller)s`, `%(bank)s`, `%(partition)s`. This stratified design hierarchy generation is crucial step of the entire methodology. The hierarchy template is a major reason for the controller section accepting a list of controllers and not a single controller section. The template for the hierarchy might be different for controllers with otherwise same configuration.

Figure 3 shows the address arrangement for the configuration in Figure 2. One can verify the sanity of the configuration of memory.

V. Operational Modes

We discuss the operational modes of this methodology assuming a TCL interface as the target, which is usually the case with emulators and certain simulators. The principles outlined are applicable to any backend schema, including SV interface or a C/C++ based model. The extensions require an appropriate visitor-generator class over the datamodel. While the automation described here is considerably large, it is however a one-time effort to develop such tools. The users of the methodology are expected only to maintain the ECC

```
misc:
 load_cmd : "emu_load %(hier)s %(file)s"
 read_cmd : "memory -rd $hier $line"
 write_cmd: "memory -wr $hier $line $data"
```

Fig. 5. An example miscellaneous section

functions (if any, $\tilde{5}0$ lines) and the configurations ($\tilde{1}20$-150 lines) for memories.

A. Preload mode

This mode preprocesses a set of input hex files into output files that can be readily loaded into design memory arrays before the test execution. The pre-computed address-hierarchy mapping described in the previous section, allows a straightforward implementation for this phase. For each address, the ecc/encryption result is computed for the data-address pair and the corresponding redundancy additions described in the physical layout section (under bank) are performed. The data is then split up (or combined) according to the array width in the bank. These computations are rarely dependent on the previous addresses results and hence are highly parellelizeable. Dependency exists in case each bank array can hold more than one word, but even in such cases the dependent addresses can be determined upfront (hence also parellelizeable). These computations can be performed efficiently for a large data in the order of gigabytes, under minutes.

The miscellaneous section (`load_cmd` in Figure 5) in the configuration file takes a template for loading a set of values to a particular design hierarchy. The result from this mode of operation is a list of output files and a mapping command file (from hierarchy to files) that is to be executed before a test. One could load a sequence of tests without needing to restart the design, enabling a quick and easy scoreboarding methodology.

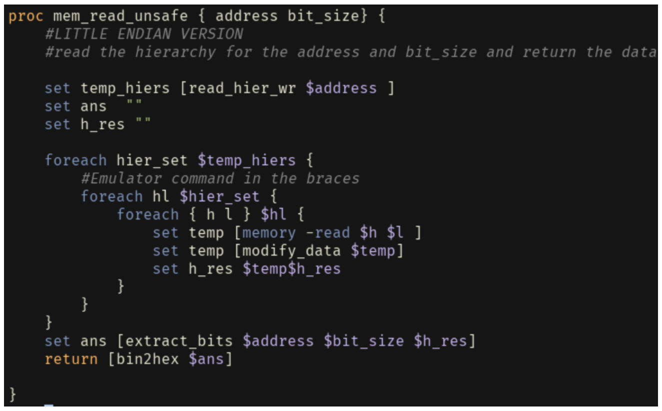
B. API mode

During the test execution on the design, verification engineers would like a debugger like access (using addresses to read and write) to certain memories (ROMs, caches for example). The usecase model would for such an operation is shown in Figure 6. A single read/write command might require more than one read/write to call to the emulator/simulator. The read operation is a reverse instance of operations in preload mode. The reconstruction of the original data happens by removing the added redundancies, combing/splitting the individual array entries in the bank and finally removing ECC bits. The default behaviors suffice for most memories (removing 0s in the middle of array data, no combine/split, remove first ecc bits); however for non-trivial cases the placeholders provided in the data are used. An instance of read call is shown in Figure 7. The `read_cmd` entry from the miscellaneous section is used in generating the `mem_read_unsafe` function and links the usemodel shown in Figure 6 to the actual emulator implementation.

```
// get 64bit data from 0x3400000
mem_read 0x3400000 64

// write 32bit data to 0x3407000
mem_write 0x3407000 DEADBEEF 32
```

Fig. 6. API usecase model



```
proc mem_read_unsafe { address bit_size} {
    #LITTLE ENDIAN VERSION
    #read the hierarchy for the address and bit_size and return the data

    set temp_hiers [read_hier_wr $address ]
    set ans  ""
    set h_res ""

    foreach hier_set $temp_hiers {
        #Emulator command in the braces
        foreach hl $hier_set {
            foreach { h l } $hl {
                set temp [memory -read $h $l ]
                set temp [modify_data $temp]
                set h_res $temp$h_res
            }
        }
    }
    set ans [extract_bits $address $bit_size $h_res]
    return [bin2hex $ans]

}
```

Fig. 7. Auto-Generated read command for Emulation

The write command is a read operation (to read contents of memory array) followed by a forward direction operation after modifying the data.

VI. Results

Using this framework, we were able to get rid of memory specific scripts for over 10 different SoCs spanning 100 different memories including TCMs, BootROMs, Flash memories and SRAMs. The maintainable code for each memory was reduced from about 1500 lines per memory to about 50-100 lines of configuration and $\tilde{3}0$ lines of data modification (ECC) functions. Figure 7 shows the auto-generated memory read command generated for emulation by this framework. The function `mem_read_unsafe` takes in an address and bit size, and returns the logical data (after removing ECC, redundancies etc.) to user, thus allowing a debugger-like transparent access to the memory. All of the functions that are not shown are parameterized with fields in the specification. For instance, the function `read_hier_wr` returns all relevant hierarchies for a given address following the explanation provided in the previous section.

As described earlier, an advantage of this framework is that changing 3 lines in configuration (read, write and load commands), one can generate the collaterals for a different emulator. A similar generator can produce a memory driver instance such as the one described in [1] for simulation!

VII. Conclusions and Future Work

We presented a framework that streamlines verification with memory abstractions. The core idea is to separate data modification operations (ECC, parity etc..) from the address translation logic which can be represented succinctly with a few parameters for a wide range of memories. The framework is not without its shortcomings. For instance, DDR memories cannot be represented in this framework since they have software-configurable address translation schemes which cannot be represented with one set of static parameters. We believe this representation scheme could eventually be standardized along the lines of IP-XACT to achieve uniformity across VDK, simulation and emulation verification.

References

[1] Geishauser, J., Chopra, A., Ruettiger, S., Rossi, L., Kakasaniya, S., and Zhang, L. uvm_mem – challenges of using uvm infrastructure in a hierarchical verification. In *2022 DVCON Europe proceedings* (2022), DVCON Europe.