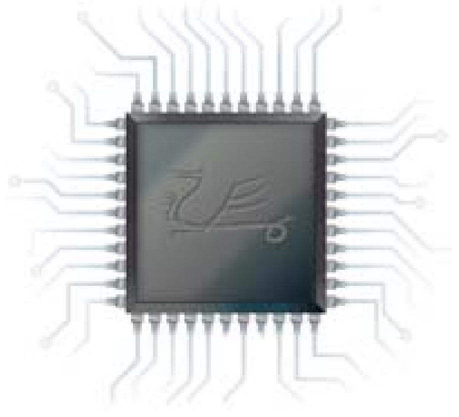
Reference Manual for the Verilog Memory Model Simulation Component

(version 0.2 draft)



Simon Southwell

August 2021

Copyright

Disclaimers

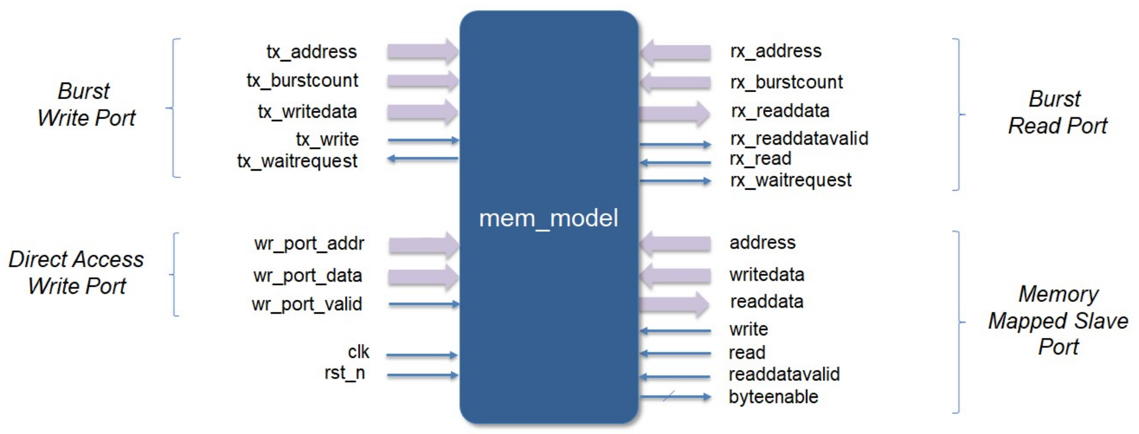
Simon Southwell (simon@anita-simulators.org.uk)

Cambridge, UK, August 2021

Introduction

The `mem_model` component is a Verilog simulation test component that allows for a very large memory address space without reserving large amounts of memory, defining large Verilog arrays, or building a truncated memory map into a test bench which could be subject to change in the design. The model uses the Verilog PLI to access a C model, pushing the majority of the functionality away from the simulator, make the test bench lightweight, and the memory accesses very fast in simulation compute time.

The component is a lightweight behavioural Verilog module, and uses the PLI interface to communicate with a set of C/C++ software to implement the actual memory model. The Verilog components itself looks like the following:



The component has a clock and reset and four data transfer interfaces.

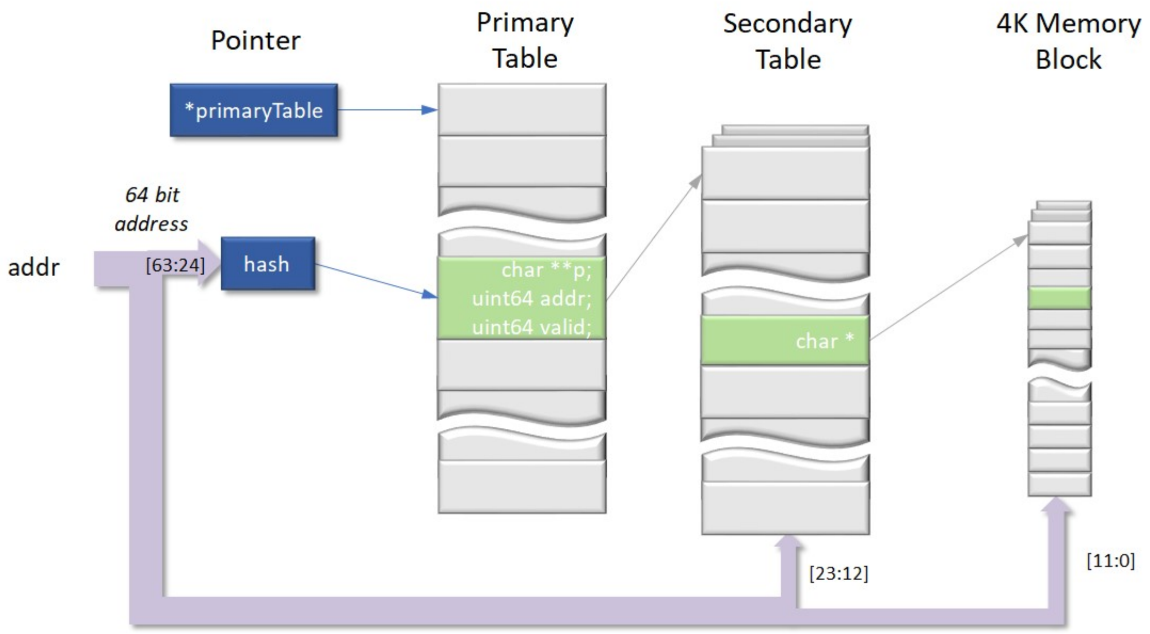
- A memory mapped slave port, for connection to, say, a CPU bus for straight forward word reads and writes with byte enables.
- A burst read port for DMA like read transfers. Note, 4K byte boundaries should not be crossed in a single burst.
- A burst write port for DMA like write transfers. Note, 4K byte boundaries should not be crossed in a single burst.
- A simple SRAM like direct access write port, useful in streaming test data directly to memory.

Not all the interfaces need to be connected, but unused interfaces should have their strobe inputs tied off to 0. If multiple interfaces are active, thought, the model will service them in parallel. If two interfaces are accessing the same location in the same cycle, though, behavior is undefined.

Internal Memory Structure

The memory model component has access to a full 32 bit address space via internal C/C++ memory model software, originally used on a PCIe verification component [1], which actually has capabilities for a full 64 bit address space. It does this with routines defined in `mem.c`, which initialise with no actual memory space allocated. As shown on the diagram, normally only `ProcessInput()` has access to the memory, and `mem.c` provides two functions for writes and reads—`WriteRamByteBlock()` and `ReadRamByteBlock()`. The user code has access to these functions, as well as some byte and word access hybrid versions.

The full 64 bit space capability relies on the fact that a simulation run cannot possibly write to all 2^{64} locations. Instead, the space is divided into 4K byte chunks which get dynamically allocated as required, and are accessed via references in a series of tables which further divided the address space. The starting point for a lookup is the `PrimaryTable`. This table has 4K entries, but maps all the top 40 bits of the address space into this space, using a simple hash, XORing the bits in a certain way and then bit reversing the 12 bit result. The `PrimaryTable` entry structure (`PrimaryTable_t`) has a valid field and an address for storing the top 40 bits of the address that hits on the location. If another address upper 40 bits hashes to the same location, then the index pointing to the table entry is simply incremented until an empty entry is found, or we searched the whole table (an error condition).



The primary table entry also contains a pointer to a pointer, which references a secondary table, dynamically allocated when first written to. The secondary tables sub-divide the address space of the lower 24 bits of the address into the 4K byte blocks required. The upper 12 bits of the lower address index into the secondary table, whose entry points to a 4K byte block of memory, dynamically allocated on first access.

Reading from a location simply involves traversing the table. The top 40 bits of the read address are hashed, and index into the primary table. The Primary table entry address is compared with the read address 40 bits and, if different, the index is incremented until a match, an invalid address is encountered or the whole table is searched. The last two cases are an error condition. The secondary table is then accessed with the next 12 bits and (if pointing to a valid byte block), the lower 12 bits used to retrieved the data. At any point in the traverse, an unallocated table entry of byte block is considered a fatal error—it is not legal to access locations that have not be written.

Compiling PLI code

The model's source code is a set of C files that must be compiled into a shared object, and entries added to a PLI table in order to add the necessary functions in the Verilog domain. The exact procedure varies somewhat between simulators, but examples for ModelSim are given here. In a file, `veriuser.c`, the following code must be present

```
#include "veriuser.h"
#include "vpi_user.h"

s_tfcell veriusertfs[] =
{
    {usertask, 0, NULL, 0, MemRead,   NULL,  "$memread",   1},
    {usertask, 0, NULL, 0, MemWrite,  NULL,  "$memwrite",  1},
    {0}
};

p_tfcell bootstrap ()
{
    return veriusertfs;
}
```

This table can only be defined once, so if there are other entries required from other PLI code, then these must be combined into a single table. For convenience, the table entries are defined in `mem.h` as a `#define` of `MEM_MODEL_TF_TBL`.

Various flags are required when compiling the code to generate a shared object, load the simulator's PLI library and ensure no (as yet) unreferenced entries are removed at link time. The verification environment (see next section) has example compilation code in a `makefile`, using the virtual processor (VProc) component [2], and calling it's `makefile`. The VProc repository has several examples of compiling a shared object, suitable for loading to various simulators, and the reader should reference these for more details.

For ModelSim, once the shared object is correctly compiled it can be loaded when running `vsim` by using the `-pli <mysharedobj>.so` command line option.

Test Environment

A Verilog test environment for the model is provided. It is dependent on VProc [2], which should be checked out into the same directory as the location of the memory model's repository. Currently it only supports ModelSim.

The test bench folder (`test`) contains the following files:

- `tb.v`: the top level test bench Verilog
- `cpu.v`: a VProc wrapper block that instantiates VProc, and drives the memory mapped master bus
- `files_vlog.tcl`: a list of all the Verilog files required by `vlog`.
- `makefile`: the compilation and execution make file
- `cleanvlib.do`, `compile.do`: ModelSim files for cleaning the Verilog work library and compiling the code, respectively.
- `sim.do`, `simg.do`, `simlog.do`: ModelSim files for running the simulation in various modes (batch, GUI, with logging).

The source code for the Virtual Processor is in a sub-folder `src`. As well as the top level VUserMain0 code (the virtual processor is set as node 0), it has some driver code for generating the access traffic on the master bus. This driver code is in `mem_vproc_api.cpp` and `mem_vproc_api.h`. The code provides some simple functions to read and write byte, half-word (16 bits) and word (32 bit) values. It is from these functions that calls to the VProc API are made, hiding the details of these, and also managing the byte enable settings with delta time updates.

Direct Access to Memory from C/C++

The test environment example discussed above is all that is needed to access memory with bus transactions—in this case from the VProc based component. Another advantage of having the memory model in C, running alongside the test code running in the virtual processor, all of which is compiled into a single package is that the VProc program can have access to the memory model C code without the need to issue a simulation transaction. This could be advantageous for, say, loading a large set of test data into memory, or for extracting and processing captured data sent to memory from the simulation. This consumes no simulations cycles, and runs at the normal speed of an executable program running on the host PC or workstation.

By including the `mem.h` header in the VProc test code, an API is presented to read and write data to the underlying C model. The prototypes for the API functions are sown below:

```
void     WriteRamByte       (const uint64_t addr, const uint32_t data,
                             const uint32_t node);

void     WriteRamHWord      (const uint64_t addr, const uint32_t data, const int le,
                             const uint32_t node);

void     WriteRamWord       (const uint64_t addr, const uint32_t data, const int le,
                             const uint32_t node);

void     WriteRamDWord      (const uint64_t addr, const uint64_t data, const int le,
                             const uint32_t node);

uint32_t ReadRamByte        (const uint64_t addr, const uint32_t node);
uint32_t ReadRamHWord       (const uint64_t addr, const int le, const uint32_t node);
uint32_t ReadRamWord        (const uint64_t addr, const int le, const uint32_t node);
uint64_t ReadRamDWord       (const uint64_t addr, const int le, const uint32_t node);

void     WriteRamByteBlock  (const uint64_t addr, const PktData_t* const data,
                             const int fbe, const int lbe, const int length,
                             const uint32_t node);

int      ReadRamByteBlock   (const uint64_t addr, PktData_t* const data,
                             const int length, const uint32_t node);
```

The first four functions allow the writing of byte, half-word (16), word (32) and double word (64) values directly to memory. An address and data arguments are given and, for the access greater than a byte, and 'little endian' flag (`le`) to indicate whether to access little endian (when non-zero) or big-endian (when zero).

The node argument (for this API) is more like a handle for multiple instantiations of the model and need not necessarily be tied to a VProc node. The expected usage would be to either have a one-to-one connection to a VProc if, say, modelling a multiprocessor system, with each CPU having its own memory. A more likely scenario, though, is that the memory is shared between processors. In this case a test bench would instantiate multiple memory components, one for each processor, but the calling PLI C code uses a common node number. This is, in fact, the default for the `mem_model.[ch]` code. The Verilog components then act as separate ports into the same memory sub-system. The direct access API, then, becomes a means for shared memory communication between code running on multiple VProc models.

The next four functions are the equivalent for reading from memory, the difference being that the read data is returned by the function. Note that for all the functions the address is a 64 bit data type. As mentioned before, this model, taken from a PCIe root complex model [1], can simulate a full 64 bit address space, even if the Verilog component is limited to 32 bit addresses.

The two block access functions would not normally be used directly (the previously mentioned functions call these), but they can be used if necessary. These take and address, as before, and then a pointer to an array of `PktData_t`—essentially byte values—where data is taken from, for writes, or data placed into, for reads. The `fbe` and `lbe` arguments are byte enables for the first and last 32 bit words, being 4 bit bitmaps. The byte enables are expected to be contiguous for the whole block, so `fbe` can only be 0xf, `0xe`, `0xc` and `0x8`, whilst the lbe can only be `0x1`, `0x3`, 0x7 and `0xf`, though `lbe` can be 0 of there is only 4 bytes or less. The byte enables between the first and last word's is implied as `0xf`. Following the byte enables is a length argument, specifying the number of bytes to be transferred.

References

[1] *PCIe Virtual Host Model Test Component*, https://github.com/wyvernSemi/pcievhost/blob/master/doc/pcieVHost.pdf, Simon Southwell, March 2017.

[2] *Virtual Processor (VProc),* https://github.com/wyvernSemi/vproc , Simon Southwell, June 2010