AMP Memory Core User Guide

This document provides information for the on chip integration of AMP memory core using eDRAM or SRAM. This document is confidential and is provided under a mutual non-disclosure agreement. It should not be distributed outside of the receiving company.

1. Introduction

AMP memory core product is generated as a 'soft-IP' core by the AMP compiler. This document describes the contents of the delivery kit and the process to integrate them onto the host chip.

Engineers involved in the RTL, DFT and Physical (P&R, Timing, Power) integration of the host chip as well as the Manufacturing/Diagnostics software folks are the target audience for this document. It is also recommended that the reader be familiar with the functionality of the core by reading the Multiport_mRnW datasheet document.

The AMP core is delivered as a kit containing the various views needed for the RTL, DFT and Physical integration of this core.

1.1. Delivery Kit

The core is delivered as a tarball containing all the requisite directories and files..

The directory structure for each memory core is as shown below:

- *rtl* : Verilog model of the design with encrypted core-logic
- *src :* Source code files (encrypted) for the core-logic of memory core
- *run_sample* : Testbench and test-related task files
- scripts : Physical design integration script files
- *doc* : Document (core-specific datasheet, User guide, etc.) directory

2. RTL Integration

This section provides information on running simulations with the Verilog model files and on integration of Verilog model in the user's chip hierarchy.

2.1. RTL model

The Verilog model for the memory core (*AMP_(algo-type)_(base-mem)_(size).v*)[1] is present in the *rtl* directory. **The user should use this file for integration in the host-chip**. It references IBM-provided models for the physical memory instantiations. The user must procure these models directly from IBM. The memory-core specific datasheet, included in the doc sub-directory listed above, enumerates the different physical memories used in the design.

The RTL model also references an encrypted parameterized logic files which implement the RTL-IP of the memory core. These files, specific to each memory core type, are present under the *'src'* sub-directory. The user HAS to use Cadence RTL compiler for the synthesis process since encryption is done using Cadence tools.

A Verilog model (*AMP_(algo-type)_(base-mem)_(size)_beh.v*) containing behavioral models of the underlying physical memories is also provided in this directory. It is provided strictly to allow faster simulations.

2.2. Simulation Environment Features

The *run_sample* directory contains the simulation environment for the generated memory core. The Verilog models and the testbench have been verified using NC-Verilog version 10.20-s076 and is the minimum version needed by the user to perform simulations on the core. The user is encouraged to run a sample simulation to ensure the compatibility of the user's setup with the configuration settings within the *env* files. The timescale for the testbench and the core Verilog file(s) has been set to 1 ps /1 ps. The user should run a quick simulation, as outlined in the section below, to check the integrity of the delivered kit.

The simulation features are described next. Other details related to simulation are described in the README file in *run_sample* directory.

2.2.1. Running the simulation

The user can run a simulation by issuing the following command, where name implies the name of the core's RTL file.

./run <name> : To simulate the RTL model

OR

./run_beh <*name*> : To simulate the behavioral model

A successful simulation is indicated by 'Test PASSED' message on the screen.

[1] Algo-type = 1rw, 2ror1w, etc. ; Base-mem = Type of base physical memory used to generate the memory core, e.g. DRAMA, SRAMA, RF1D, etc; ; size = size of memory core specified as <rows>x<bits>

2.2.2. Backdoor Access

The Backdoor Memory access debug feature allows the user to access the IP's memory locations, bypassing the front end Read/Write command interface. The user can write or read memory locations directly using this mechanism.

The two Verilog tasks provided to do this are:

write (backdoor_addr, backdoor_data);

read (backdoor_addr, backdoor_read_data);

The user should set the appropriate path to these tasks when running from his environment. These tasks are zero cycle tasks hence their effect is immediate.

2.2.3. ECC Error Injection

Note: This feature is provided only for those AMP memory cores which use the Read Error functionality. Read error events happen only for certain memory cores, depending on the underlying algorithm. The core-specific datasheet states if the Read error outputs are used for that core.

This task allows injection of single/multi bank errors at any address location. Upon execution, the specified locations are tagged with an error marking - the actual user data is not altered.

The syntax for this task is: *put_error (addr, spare-bank);*

The usage of this task is as follows: To inject errors in the main memory space, the user specifies the *addr* and sets the spare-bank field to '0x0'. A single bit error at the specified address can be created by invoking the task once. The error is changed to type 'multi' if the user invokes the task second time writing to the same address but with spare-bank flag set to '0x1'.

Error-manifestation: When reads are performed subsequently from the error-injected locations, read error outputs (either single or double) are asserted appropriately allowing the user to simulate his error handling logic. It should be noted that any read to a row-address which was injected with error, irrespective of the read's bank-address, will result in a read error.

The user can clear injected errors by writing to the address where the error was injected. Note that this clears the error only for the bank and corresponding row of the write-address. Any errors marked in the corresponding row of the spare-bank will be cleared as well.

2.3. Error Handling

No error checking is provided by the AMP memory core for the user-data memory. The user can choose to add ECC/CRC or any other schemes as part of the data itself. The AMP memory core is agnostic of user's error correction scheme. The physical address is provided to help with address logging in case the user chooses to maintain statistics on ecc error events. Due to buffering and address mapping schemes deployed within the core, the physical location of data for a given user read address may not be the same as the location implied by the user provided read address. Also due to buffering, the physical address space of the total memory may be bigger than the user's logical address space. The physical address location of the read data is provided on the physical address output bus (*read_padr*). The bit-makeup of the Physical address output (*read_padr*) is shown in Table 2-1.

Table 2-1: Physical Address Bit-makeup

Read_padr	NME	Physical Bank-Address	Physical Word-Address	Physical Row-Address
m = {1 | x | y | z }	1-bit	x-bits	y-bits	z-bits

The 1-bit msb of the physical address represents the presence (or absence) of non-main memory error (NME) in case an error condition occurs (assertion of either *read_serr* or *read_derr* – explained later in this section). Under normal conditions this bit is always '0'.

The values of x, y, and z are memory core specific. The first x bits indicate the eDRAM/SRAM bank address. For instance, if the memory core consists of 16 banks in total (including the spare bank) then x = 4. This information is made available in the core specific datasheet provided in the *doc* sub-directory. The remaining bits (y, z) pertain to the row address of the physical location. The row address is comprised of y-bits of word-address (to reflect sub-packed words within the same physical row) and z-bits representing the address of the physical row within the bank. The user can deduce the value of y (for eDRAM based cores) by observing the bit-width of eDRAM bank macro and dividing it by the bit-width specified for the memory core. For instance, if the width of eDRAM macro is 432 and the memory core bit-width is 72, then y= floor (432/72=) 6. Once x and y are determined then z is simply the remaining lsbs of the physical address.

The AMP memory core does monitor data integrity of its internal metadata structures. If data corruption is detected during a Read transaction the user is notified with an assertion of the *read_serr* (or *read_derr*) outputs. In case of read error, the bits of *read_padr* point to the bank/row address of the read location where the error happened. Then, if the NME bit is '0' it implies that the error happened in a memory location (indicated by the remaining bits of physical address). If this bit is '1', it implies that the error occurred in some non-memory location (like fifos or registers implemented in the core's logic). For the multi-bank error case, the bank/row address information points only to the first bank where the error was detected. The information for other erring banks is not provided.

2.3.1. Error Recovery Options

Performing the reset sequence on the core is a suggested first step. Assuming the error was 'soft', normal operation should ensue next time around. The user can choose to maintain statistics of this occurrence by logging the physical addresses. If the number of occurrences for a given bank (or row) is above a threshold then that memory location has probably developed a physical defect. Direct access to the memories is possible only thru the IBM's Test-Interface. The user could choose to run 'memory BIST' on the affected memory. Details of running BIST and other possible courses of action are made available by IBM directly.

3. DFT ('Design for Test') Integration

DFT integration pertains to insertion of Memory BIST functionality to the memory elements of the AMP memory core blocks.

3.1. MBIST Insertion

Two options are available to the user. The first option is to use the Gate-level netlist (which has to be generated by the user). Guidelines from IBM can be used directly to do MBIST insertion. The user is responsible for making sure that all timing constraints are met post-DFT integration.

The second option is to perfrom DFT integration at RTL-level. One suggested approach using Atrenta's RTL MBIST (SpyGlass-MBIST) tool is discussed next.

Step-1: Prep the core's top level RTL file

The AMP core's top level file (*AMP_(algo-type)_(base-mem)_<cut.size>.v*) is structured with the main memory modules instantiations separated from the core module instantiation. The core module (*algo_1rw_b1_top_wrap* in the example below*)* contains all the logic of the AMP core and is encrypted.

Since MBIST is inserted only on main memory modules the core module inclusion needs to be masked out when MBIST tool reads in the source files. This is done by wrapping the instantiation statement for this file in a SpyglassMBIST-recognizable pragma ('ifndef…'endif structure). For the user's convenience this change is already incorporated in the top level RTL file. The structure of this file for the 1RW eDRAM based core is shown in Fig 3-1. The string '1rw_b1' in the module name refers to the algorithm type.

```
//File : AMP_1rw_DRAMA_(size).v
module <top_module_name> ( ..... );
…...(port definition statements here)

// all physical memory instantiations next
< mem1>( );
<mem2>( );
….
'ifndef ATRENTA_SG_HIDE_ENCRYPTED_RTL
// instantiation of encrypted module
 algo_1rw_b1_top_wrap # ( param-list)
 des (….);
'endif

end_module   // end of 1rw_top module
```

Pragma to prevent reading of encrypted core logic files by MBIST tool

Figure 3-1: Structure of Top-level Verilog file

Step-2: Prepare script files to run MBIST tool:

The TCL file (script file) and an associated file (*.swl*) used for running the MBIST tool on AMP memory core's RTL file are prepared next. The TCL file should contain the following settings:

a) ***set_option pragma ATRENTA_SG_HIDE_ENCRYPTED_RTL***

This enables the pragma option (described earlier) which prevents the MBIST tool from reading the encrypted core files.

b) ***read_file -type waiver amp_waiver.swl***

The *memoir_waiver.swl* file (prepared by the user) should contain the rule below:

waive -rule "ErrorAnalyzeBBox" -msg "Design Unit.'.*ATRENTA_SG_HIDE_ENCRYPTED_RTL*'.*has no definition; black-box behavior assumed and module interface inferred" -regexp*

This prevents error messages being generated by the MBIST tool when it blackboxes the encrypted modules for the core logic.

Implementing the two steps described above should result in an error free run of the MBIST tool.

4. Physical Design Integration

Suggestions for physical placement and meeting timing constraint are presented in this section.

4.1. Block Placement

The AMP memory core cuts contain macros for SRAM/eDRAM along with the core logic. For best timing performance, it is recommended that all the base memory blocks be placed close to each with the core logic placed in between. One compact way to pack the blocks is shown below.

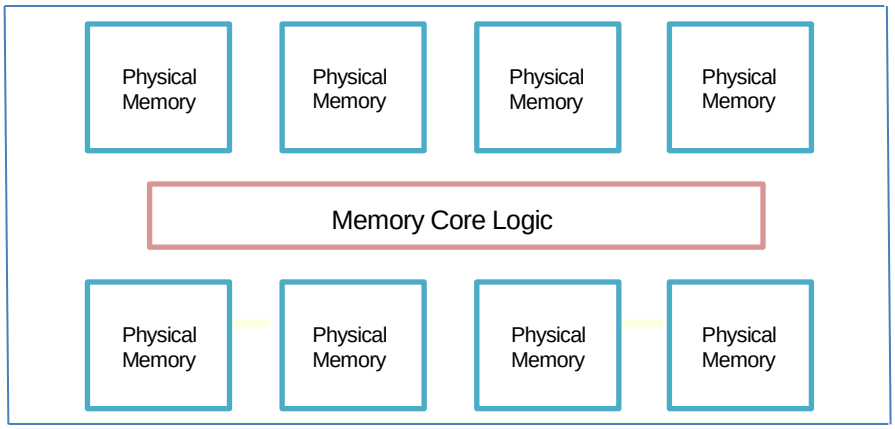

Figure 4-1

Notes:

a) This layout is for representation only. The actual numbers and sizes of the macros may vary depending on the size of the functional cut.

b) The core logic block is shown sprinkled around in the area between the base memories. The area of core logic is relatively much smaller than the memory macros.

4.2. Physical Design Closure

Sample reference files are provided in the scripts directory to support synthesis and equivalence verification of the generated core.

The ***synthesis.tcl*** file is a sample synthesis script for synthesizing the generated RTL using Cadence

The ***equivalence.dofile*** is a sample do-file used for formal verification between the RTL and user-synthesized gate-level netlist.

The ***timing.tcl*** file is a sample script file for performing post-layout static timing analysis of the generated memory core.

Note: These files are for reference only. The users have to modify them to fit their environments as appropriate.

4.3. Core Specific Datasheet

A core-specific datasheet is provided in the *doc* directory. This document states the area, frequency of operation, static power and the maximum read and write dynamic power numbers for the core. In addition it lists the components of the base memory library used to generate the core.

The formula to calculate maximum dynamic power is:

Maximum Dynamic Power = [(Max.read power) * AFr] + [(Max.write power) * AFw]

Where AFr and AFw are read and write activity factors.

5. Revision Change Log

Version	Date of Release	Notes
V1	May 8, 2012	Initial Release.