AMP 2Ror1W Memory Core Integration Guide

This document provides information for the on chip integration of AMP ‘2 Read or 1 Write’ Memory core (2Ror1W) using eDRAM. 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 2Ror1W 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 datasheet document.

The 2Ror1W 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 core is as follows:

- *rtl* : Encrypted Verilog hierarchical model.
- *run_sample* : Testbench and test-related task files
- scripts: EDA_support files
- *doc* : Document (Release notes, Integration guide, etc.) directory

All the design files are encrypted with Cadence tools. Since the encryption algorithm is proprietary, the encrypted RTL models will NOT run on simulators from other CAD vendors. 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.

Before the integration process can begin the user must obtain models of the memory macros used in the core. These models should be procured directly from IBM. An Appendix at the end of this document, enumerates the different macro types used in the core. If however the user wishes to perform only behavioral simulation then he can proceed with the behavioral memory models provided in the kit (this is the default setting for model choice in the test environment. Please refer to the README file to use other model options).

The kit includes the test environment to check for the integrity of the delivery. 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 has been set to 1 ps /1 ps.

An example simulation can be run by following the instructions in Sec 2.1.1.

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. The user is urged to read the README file for details on the simulation environment.

2.1. Simulation Environment Features

The RTL environment incorporates features to facilitate ease of simulation and debugging. These features are described next. Other simulations features are described in the README file in the *run_sample* dir generated for every cut.

2.1.1. 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. The command should be issued from the *run_sample* directory.

./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.

2.1.2. 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.

2.1.3. ECC Error Injection

The testbench incorporates a task to allow injection of single/multi bank errors at any address location. In the implementation, the specified locations are simply tagged with an error marking. The actual user data is not altered.

The Verilog task provided to inject errors 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'. Single or multi-bank errors can be set by invoking the task single or multiple times. This marking is altered to 'multi' if the user invokes the task multiple times writing to the same row address across two or more banks. To inject errors in the spare-bank, the user specifies the *addr* as above but then sets the spare-bank field to ‘0x1’. This will tag the row-address (extracted from the *addr* specified) in the spare-bank with an error marking.

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.

Refer to the example provided in the README file.

2.2. RTL Connections

The table below provides suggestions on integrating the core's RTL model within the model hierarchy of the host chip. Please refer to the datasheet for additional information on the individual signals.

Table 2-1: Interface Pin Connections

Pin	Pin Type	Count	Description	Connections with the host:
addr_0,1	Input	w	Read Address	Address for the Read ports; parameter w represents number of words
read_0,1	Input	1	Read Command	Read command
dout_0,1	Output	b	Read Data	Read data output; parameter b represents bit width
read_vld_0,1	Output	1	Read Data valid	Indicates read data valid on dout
read_serr_0,1	Output	1	Single bank Read Error	Indicates Single-bank Read error (Refer Error handling section)
read_derr_0,1	Output	1	Multi bank Read Error	Indicates Double-bank Read error (Refer Error handling section)
read_paddr_0,1	Output	m	Physical Read Address	Read physical address (Refer Error handling section); parameter m represents number of physical words
addr_2	Input	w	Write Address	Write Address for the write data; parameter w represents number of words
write_2	Input	1	Write Command	Write command
din_2	Input	b	Write Data	Write data; parameter b represents bit-width
clk	Input	1	Clock	Master Clock for the core
refr	Input	1	Refresh	Refresh command
rst	Input	1	Reset	Reset input pin for the core
ready	Output	1	Ready	Core ready for functional access

2.3. Error Handling

No error checking is provided by the 2Ror1W 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 2Ror1W core is agnostic of any user error correction scheme as long as it's handled as part of the main data. Due to 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. The physical address is provided to help with address logging in case the user chooses to maintain statistics on ecc error events.

AMP 2Ror1W core uses buffer memory in addition to user base memory to implements its algorithm. Hence the physical address space of the total data-memory is bigger than the user's logical address space. The physical address location of the read data is provided on the Physical address output bus (*paddr*). This address in conjunction with individual memory Reference-designators can be used to identify the errant physical memory bank(s). The bit-makeup of the Physical address output (*paddr*) for the two cores is shown in Table 2-2.

Table 2-2: Physical Address Bit-makeup

Paddr	Physical Bank-Address	Physical Word-Address	Physical Row-Address
m = { x | y | z }	x-bits	y-bits	z-bits

The first x bits indicate the eDRAM/SRAM bank address. The remaining bits 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, applies only for eDRAM-based cores) and the address of the physical row of the eDRAM/SRAM bank.

It should be noted that the physical address returned can be different from the address provided originally by the user (for the read operation) due to address mapping schemes mentioned earlier.

The 2Ror1W 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 *paddr* point to the bank/row address of the read location where the error happened. For the multi-bank error case, the bank/row address information in this register 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 2Ror1W core blocks. A note on scan-insertion is also provided at the end.

3.1. MBIST Insertion

Two options are available to the user.

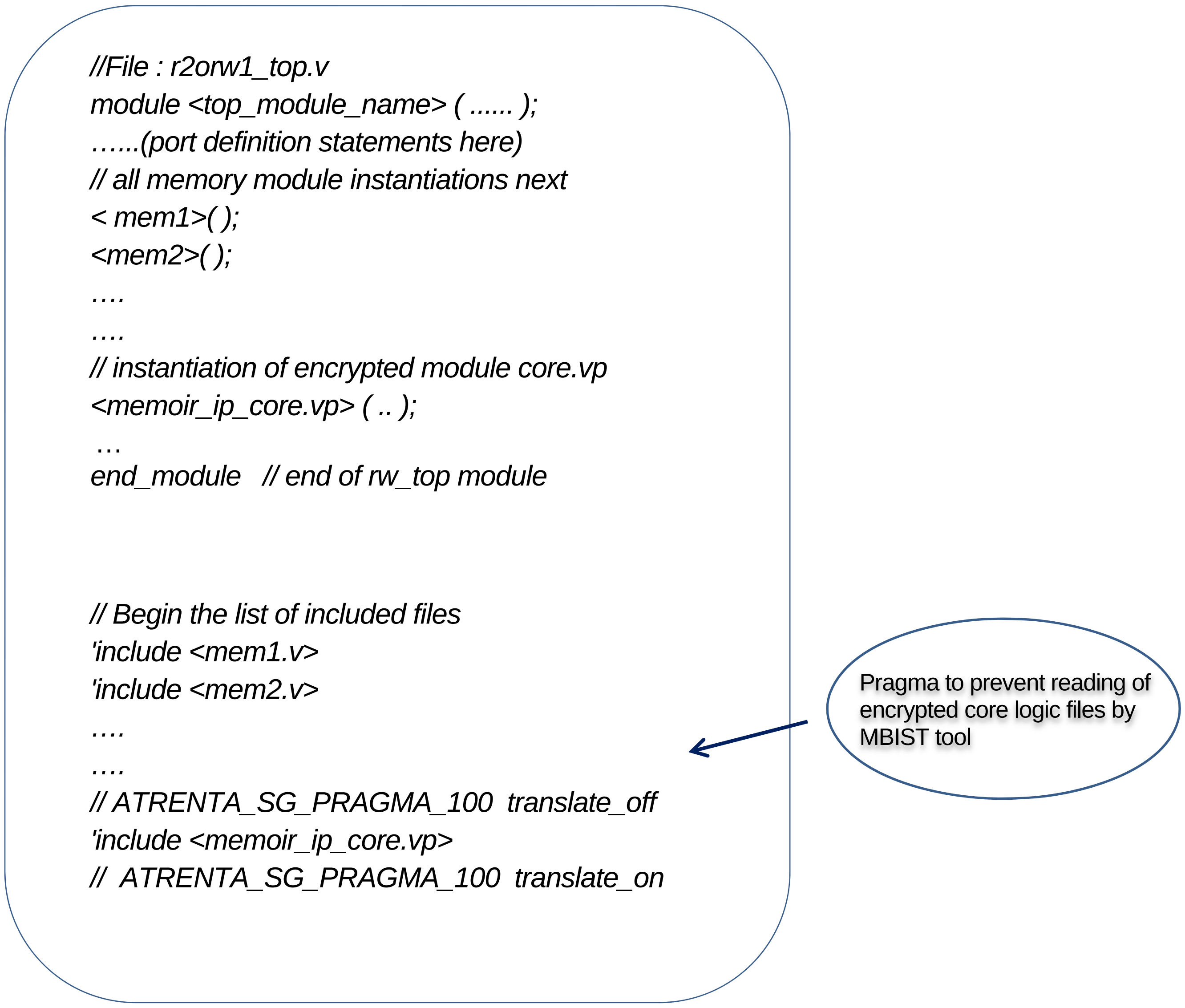
The first option is to use the Gate-level netlist (which has to be generated by the user). Guidelines from the IBM can be used directly to do MBIST insertion. The user is responsible for making sure that all timing constraints are met post-DFT integration.

The section explains the steps to perfrom RTL-level DFT integration. 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 2Ror1W core's top level file (2*ror1w_top.v*) is structured with the main memory modules instantiations separated from the core module instantiations. The core modules (*core_n.vp; where n=0,1,2,..*) contain all the logic of the 2Ror1W core and are 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 *include* statement of this file in an Atrenta-recognizable pragma. For the user's convenience this change is already incorporated in the top level RTL file provided in the delivery kit. The structure of this file is shown in Fig 3-1.



```
//File : r2orw1_top.v
module <top_module_name> ( ...... );
…...(port definition statements here)
// all memory module instantiations next
< mem1>( );
<mem2>( );
….
….
// instantiation of encrypted module core.vp
<memoir_ip_core.vp> ( .. );
 …
end_module   // end of rw_top module



// Begin the list of included files
'include <mem1.v>
'include <mem2.v>
….
….
// ATRENTA_SG_PRAGMA_100  translate_off
'include <memoir_ip_core.vp>
//  ATRENTA_SG_PRAGMA_100  translate_on
```

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

Figure 3-1

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 2Ror1W core's RTL file are prepared next. The TCL file should contain the following settings:

a) *set_option pragma ATRENTA_SG_PRAGMA_100*

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

b) *read_file -type waiver memoir_waiver.swl*

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

waive -rule "ErrorAnalyzeBBox" -msg "Design Unit.'.*ATRENTA_SG_PRAGMA_100.*'.*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.

A few other notes below:

a) This method will NOT work if the encrypted file is read using a filelist. The MBIST tool will issue a syntax error in that case.

b) The root path for the files included using *'include* option should be provided appropriately to the MBIST tool . E.g: If the full path to the encrypted rtl core.vp is " /<path1>/<path2>/<path3>/rtl/core.vp " and the top.v has " 'include <path3>/rtl/core.vp " in it, then the root path provided to the tool for the include directories should be (INCDIR) " /<path1>/<path2> ".

4. Physical Design Integration

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

4.1. Block Placement

The 2Ror1W 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.

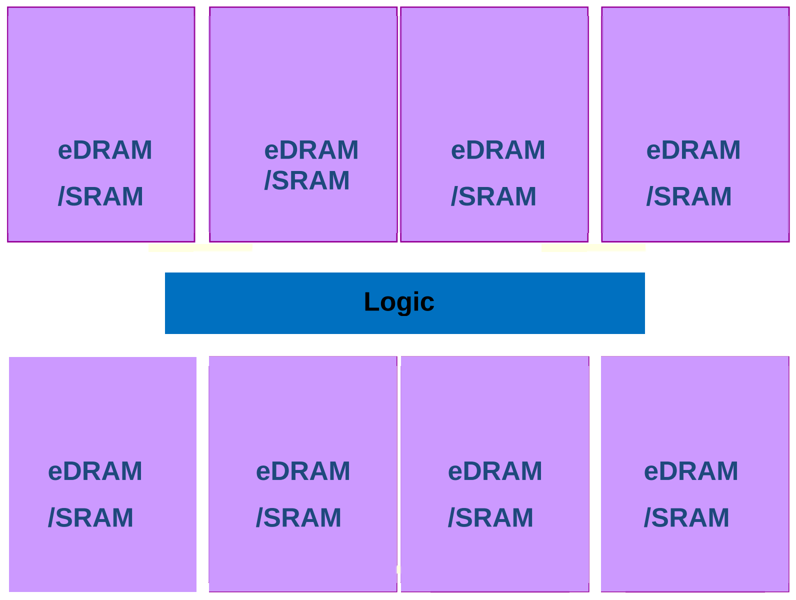


Figure 4-1: Suggested Block Placement

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

For every core, sample reference files are provided in the *scripts* directory to support synthesis, timing and formal verification of the generated core.

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

The ***lec.dofile.tcl*** file is a sample do-file used for matching cell-names between the RTL and user-synthesized gate-level netlist.

The ***syn.tcl*** file is a sample file to be used for post-layout static timing analysis of the generated core.

4.3. Core Specific Datasheet

A core-specific datasheet is provided in the *doc* directory. This document states the area, static 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.

To formula to calculate max. dynamic power is:

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

Where AFr and AFw are read and write activity factors.

Other core-specific timing parameters like Reset-to-Ready delay and Refresh scheme parameters (applicable only for eDRAM based cores). These are described in the generic Algorithmic Memory datasheet mentioned earlier.

5. Revision Change Log

Version	Date of Release	Notes
V0.1	Feb 14st 2012	Initial Release.