Algorithm 1r1rw\_a100 Microarchitecture Document

logo.png

Release Date: Apr 16, 2014

This document provides functional information for the 1r1rw\_a100 Algorithmic Memory® core.

Copyright 2012 (c) Memoir Systems. All rights reserved. No part of this document may be reproduced in any form or by any means, including electronic storage and retrieval, or translation into another language, without prior agreement and written consent from Memoir Systems, as governed by United States and international copyright laws.

algo\_1R1RW\_A100 Microarchitecture

|  |  |  |
| --- | --- | --- |
| Date | Reviewer | Comments |
| May 12 2014 | \_\_ | Initial Release. |

Document Naming Convention

The naming strategy for the documentation is:

Microarchitecture\_module#\_function#\_algo#\_rev#

The field that doesn’t apply stays empty.

Microarchitecture\_[core,algo,infra]\_[1r2w,4r5ws, etc]\_[a121,a32,etc]\_[00,01,02,etc]

For example:

Microarchitecture\_core\_nr1rw\_\_00

Is a core documentation with nr1w capability, revision 00.

Or

Microarchitecture\_algo\_4r5ws\_a32\_02

Is an algo documentation with 4r5ws capability, algo# a32, revision 02.

Or

Microarchitecture\_infra\_1r1w\_\_00

|  |
| --- |
| Contents |

[1. Functional Overview 1](#_Toc385371829)

[2. Interface description 2](#_Toc385371830)

[3. Algos.txt 3](#_Toc385371831)

[4. Block level description 4](#_Toc385371832)

[5. Key Parameters 5](#_Toc385371833)

[6. Formal 6](#_Toc385371834)

[7. Revision Change Log 7](#_Toc385371835)

# Functional Overview

The 1r1rw\_a100 IP provides 1R1RW (2 port) functionality. It is built using two port memories (1R1W).

This IP is implemented as XOR algorithm. A spare bank of memory stores the XOR data for each row. When we get two reads accessing the same bank, the second read data can be retrieved by XORing all the data for that row.

For more information on this algo, please refer to the core documentation core\_nr1rw\_1r1w, and infra documentation infra\_align\_ecc\_1r1w and infra\_stack\_1r1w.

# Interface description

The algo\_top’s main interfaces include the clking ports; the read and write ports, which talk to the primary interface; the t1 ports which talk to the single port (T1) memories and the t2 ports which talk to the spare (T2) memory.

|  |  |  |  |
| --- | --- | --- | --- |
| Pin | Pin Type | Description | Notes |
| Clock and Initialization | | | |
| clk | Input | Clock | Clock input pin |
| rst | Input | Reset | Reset pin |
| ready | Output | Ready | Assertion implies memory core ready for functional operation |
| Read Ports | | | |
| rd\_adr | Input | Read Address | Address bits (width N bits) to denote up to 2N addresses;  N is parameter. Same value for all ports |
| read | Input | Read Valid | Denotes that a Read command is valid |
| rd\_dout | Output | Read Data | Data (width W) associated with a read address;  W is parameter. Same value for all ports |
| rd\_vld | Output | Read Data Valid | Denotes valid read data |
| rd\_serr rd\_derr | Output | Read Error | Denotes read data error. Data on *dout* pins is invalid |
| rd\_padr | Output | Physical Address | Physical Address (width P bits) associated with the read. Parameter P depends on the physical address space of the memory core |
| Read/Write Ports | | | |
| rw\_read | Input | Read Valid | Denotes that a Read command is valid |
| rw\_write | Input | Write Valid | Denotes that a write command is valid |
| rw\_addr | Input | Read Address | Address bits (width N bits) to denote up to 2N addresses;  N is parameter. Same value for all ports |
| rw\_din | Input | Write Data | Data (width W bits) associated with a write command; Parameter W is same that for Read data. |
| rw\_dout | Output | Read Data | Data (width W) associated with a read address;  W is parameter. Same value for all ports |
| rw\_vld | Output | Read Data Valid | Denotes valid read data |
| rw\_serr rw\_derr | Output | Read Error | Denotes read data error. Data on *dout* pins is invalid |
| rw\_padr | Output | Physical Address | Physical Address (width P bits) associated with the read. Parameter P depends on the physical address space of the memory core |
| **T1 Ports** | | | |
| t1\_writeA | Output | Write Valid | Denotes that the write command is valid for the respective T2 memory. |
| t1\_addrA | Output | Write Address | Address bits for write command |
| t1\_dinA | Output | Write data | Write Data input to T2 memories |
| t1\_bwA | Output | Bit write enable | Bit wise enable for write data |
| t1\_readB | Output | Read Valid | Denotes that the read command is valid for the respective t3 memory |
| t1\_addrB | Output | Address | Address bits for read command |
| t1\_doutB | Input | Read data | Data associated with the read address |
| **T2 Ports** | | | |
| t2\_writeA | Output | Write Valid | Denotes that the write command is valid for the respective T2 memory. |
| t2\_addrA | Output | Write Address | Address bits for write command |
| t2\_dinA | Output | Write data | Write Data input to T2 memories |
| t2\_bwA | Output | Bit write enable | Bit wise enable for write data |
| t2\_readB | Output | Read Valid | Denotes that the read command is valid for the respective t3 memory |
| t2\_addrB | Output | Address | Address bits for read command |
| t2\_doutB | Input | Read data | Data associated with the read address |

# Algos.txt

The memogen software uses this file to evaluate and estimate the best-fit memories for the given algo.

*# 1R1RW\_2p\_XR*

*1R1RW\_A100 (f, w, b) [XR]*

*vars : n \_\_ bank\_count(1) 🡪 n>=1*

*ws \_\_ words\_array(n,w)*

*b' \_\_ bits(b)*

*formulae : w' = ws.max()*

*macros:*

*type1(BASE) : n \* 1R1W(f, ws, b') 🡪 n copies of ws size of T1 mems.*

*type2(BASE) : 1 \* 1R1W(f, w', b') 🡪 One spare copy of max bank size.*

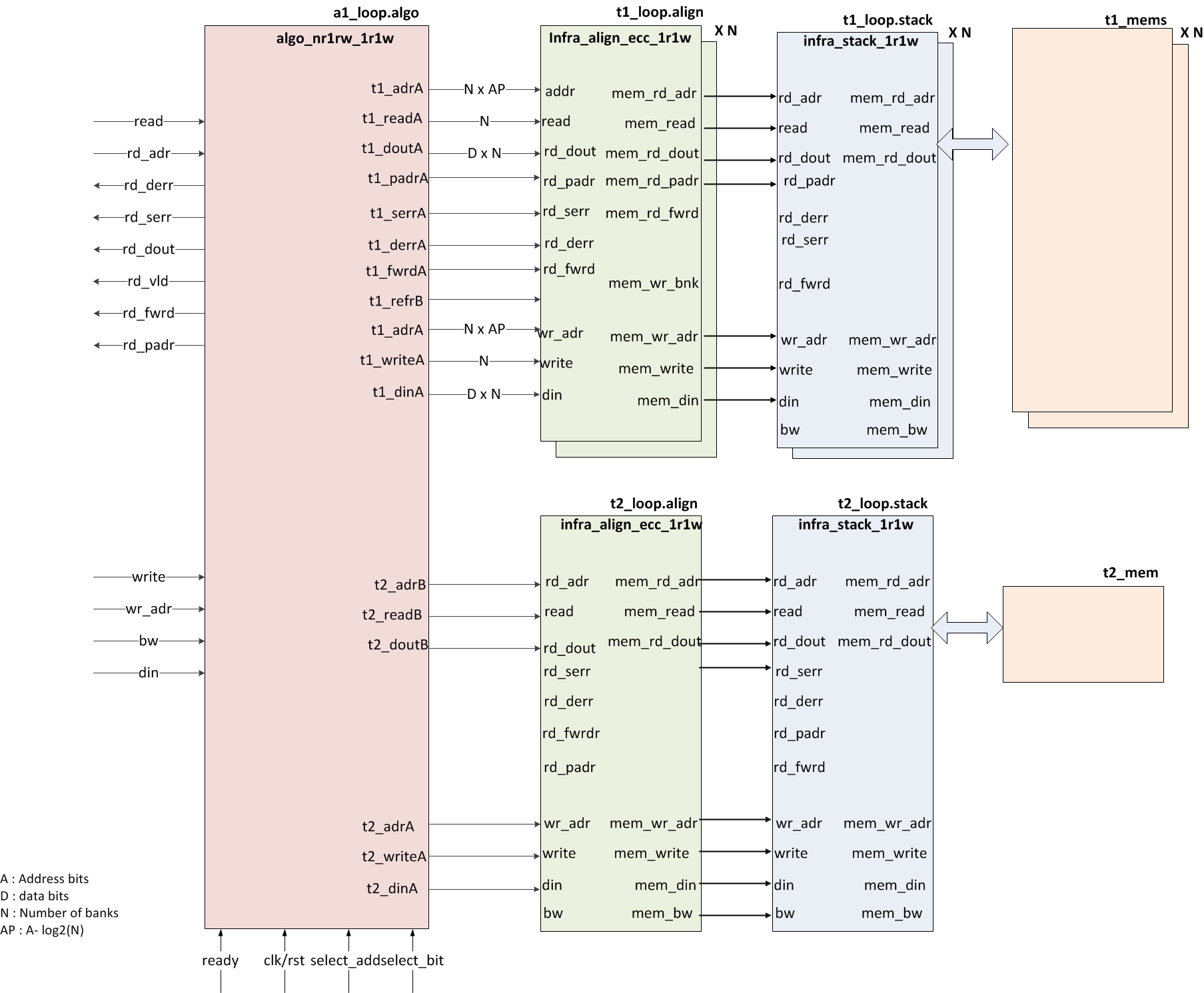
The above algorithmic description defines the characteristics (frequency, depth and width) of the memory macros used for 1r1rw\_a100. There are two types of memories in the IP that memogen software has to pick – type 1 (T1) and type2 (T2). The software uses the process information, user-defined parameters and the algorithmic description to decide which memories will best suit the requirements.

* **T1 Memory:** This memory is the bulk of the memory used in the 1r1rw\_a100 and is used for the primary data storage. The size of this memory is equal to total capacity requirement of the algorithmic memory (that is approximately, ws x n x b). In 1r1rw\_a100 1R1W (2 port memory) is used as physical memory for T1 memory.
  + f – target frequency of T1 memory. This parameter is user defined.
  + n – is the optimal number of T1 macros or number of banks, in which the complete memory depth/capacity will be split. This parameter is optimized based on the physical library.
  + ws – The optimal depth of the T1 memory macros. For IP sizes which are non-power-of-two the tool can optimize the area using different size T1 macros. This parameter is optimized based on the physical library.
  + b’ – data width of the IP. This parameter is user defined.
* **T2 Memory:** This is the spare memory, used to store XORed data. Whenever there is a clash of two reads going to the same bank, then the second read is performed on all the banks except the target bank. Then the data is XORed to retrieve the actual data for that read. The size of this memory is equal to maximum bank size. The width is equal to the width of the IP. In 1r1rw\_a100 1R1W (two port memory) is used as spare memory.
  + f – target frequency of T1 memory. This parameter is user defined.
  + n – is the optimal number of T1 macros or number of banks, in which the complete memory depth/capacity will be split. This parameter is optimized based on the physical library.
  + w’ – The maximum depth of the T1 memory macros. For IP sizes which are non-power-of-two the tool can optimize the area using different size T1 macros. w’ would be the depth of the deepest T1 macro used. This parameter is optimized based on the physical library.
  + b’ – data width of the IP. This parameter is user defined.

# Block level description

An algo\_1r1we\_rl2\_pseudo\_top instantiates:

1. For a1\_loop (algo) : This loop consists of the module algo\_nr1rw\_1r1w – core\_nr1rw\_1r1w. The core consists of the bulk of the algorithm; it translates the virtual reas/write bus (denoted by v) into physical bus (denoted by p) and xor bus (denoted by x). The core flops the primary inputs depending on the FLOPIN parameter and dispatches read commands to T1 and T2 memories. If two reads are trying to access the same bank, then the second read data is retrieved by sending read commands to all other banks except the target bank, followed by XORing these data .
2. For t1\_loop (t1 memories): This module consists of n instances each of infra\_align\_ecc\_1r1w and infra\_stack\_1r1w. These modules talk to the T1 memories.
3. For t2\_loop (t2 memories): This module consists of one instance each of infra\_align\_ecc\_1r1w and infra\_stack\_1r1w. These modules talk to the T2 memories.



**Figure1: Block diagram for Algo\_top**

# Key Parameters

The 1r1rw\_a100 algo uses the following key parameters:

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Description** | **Software Controlled** | **Hadware controlled** |
| WIDTH | Data width of the IP; It defines width of the data bus | <To be filled> | <To be filled> |
| NUMADDR | Depth of memory IP; Total address space for the IP |  |  |
| NUMVROW | Number of Virtual Rows; Number of rows in the T1 bank |  |  |
| NUMVBNK | Number of Virtual Banks; Number of T1 banks |  |  |
| BITPADR | Number of bits sufficient for all the banks in the design (T1 and T2 banks), used for physical address interface |  |  |
| NUMWRDS | Align Parameter |  |  |
| NUMSROW | Number of rows for select address; Select address is used for formal verification convergence |  |  |
| NUMWBNK | T2 packing parameter. Internally: T1 Number of banks per stack |  |  |
| SRAM\_DELAY | Delay to access T1 banks |  |  |
| DRAM\_DELAY | Delay to access T2 banks |  |  |
| FLOPIN | Controls the flopping option on input command signals. |  |  |
| FLOPOUT | Controls the flopping option on output signals. |  |  |
| FLOPCMD | Controls the option to flop the command to memory. |  |  |
| FLOPMEM | Controls the flopping option of data from memory. |  |  |
| ENAPAR | Enable Parity |  |  |
| ENAECC | Enable ECC |  |  |
| ENAHEC | Enable Half ECC |  |  |
| ENAQEC | Enable Quad ECC |  |  |
| ECCWDTH | ECC width |  |  |

# Formal

The Cadence’s formal tool (ifv) is run using the frun.sh script which takes run\_ifv as it’s input. It is run under run\_sample/ifv/<cut> directory. It pulls tcl scripts from the tree and runs all the formal proofs. To run a specific proof, frun.sh can be run with –r <check> option. And to run in gui mode, a –gui option may be used.

Following are the primary design constraints and the assertions for the 1r1rw\_a100 algo:

Primary design constraints:

Leaf nodes/assertions to prove:

*assume\_xmem\_nerr\_check: assert property (@(posedge clk) disable iff (rst)*

*(a1\_loop.algo.ip\_top\_sva.xmem\_nerr[0]== t2\_loop[0].align\_loop.infra.ip\_top\_sva.mem\_nerr));*

* Assume that align\_loop.mem\_nerr will propagate to a1\_loop.xmem\_nerr

*assume\_xmem\_serr\_check: assert property (@(posedge clk) disable iff (rst)*

*(a1\_loop.algo.ip\_top\_sva.xmem\_serr[0]== t2\_loop[0].align\_loop.infra.ip\_top\_sva.mem\_serr));*

* Assume that align\_loop.mem\_serr will propagate to a1\_loop.mem\_serr

*assert\_wr\_range\_check: assume property (@(posedge clk) disable iff (rst) write |-> (wr\_adr < NUMADDR));*

Leaf nodes/assertions to prove:

*assume\_mem\_nerr\_check: assert property (@(posedge clk) disable iff (rst) mem\_nerr |-> ##(SRAM\_DELAY+FLOPMEM) !core.dout\_serr\_mask);*

* *If no error is set, then after the delay, core’s dout\_serr\_mask is not set.*

*assume\_mem\_serr\_check: assert property (@(posedge clk) disable iff (rst) mem\_serr |-> ##(SRAM\_DELAY+FLOPMEM) core.dout\_serr\_mask);*

* *If serr is set, then core’s dout\_serr\_mask should be set.*

*assert\_dout\_nerr\_check: assert property (@(posedge clk) disable iff (rst) (read && (rd\_adr == select\_addr) && mem\_nerr) |-> ##(SRAM\_DELAY+FLOPMEM)*

*(!rd\_serr && (rd\_dout == $past(fakemem[WIDTH-1:0],SRAM\_DELAY+FLOPMEM))));*

* *On a valid read transaction, with no errors; after SRAM\_DELAY+FLOPMEM cycles, rd\_dout should equal what was stored on fakemem; if serr is zero.*

*assert\_dout\_serr\_check: assert property (@(posedge clk) disable iff (rst) (read && (rd\_adr == select\_addr) && mem\_serr) |-> ##(SRAM\_DELAY+FLOPMEM)*

*((PARITY==0) || (rd\_serr && (!rd\_fwrd || (rd\_dout == $past(fakemem[WIDTH-1:0],SRAM\_DELAY+FLOPMEM))))));*

* *On a valid read transaction, with single bit ecc error; after SRAM\_DELAY+FLOPMEM cycles, rd\_dout should equal what was stored on fakemem if it is not a forwarded data;and serr should be set.*

*assert\_fwrd\_check: assert property (@(posedge clk) disable iff (rst) (read && (rd\_adr==select\_addr)) |-> ##(SRAM\_DELAY+FLOPMEM)*

*(rd\_fwrd == $past(core.forward\_read,SRAM\_DELAY+FLOPMEM)));*

* *On a valid read transaction; after SRAM\_DELAY+FLOPMEM cycles, core\_forward\_read should be propagated to the rd\_fwrd.*

*assert\_padr\_check: assert property (@(posedge clk) disable iff (rst) (read && (rd\_adr==select\_addr)) |-> ##(SRAM\_DELAY+FLOPMEM)*

*(rd\_padr == {select\_word,select\_srow}));*

* *On a valid read transaction; after SRAM\_DELAY+FLOPMEM cycles, select\_word\_select\_row should be propagated to rd\_padr.*

**Memoir ip**

FAKEMEM

FAKEMEM

FAKEMEM

FAKEMEM

FAKEMEM

FAKEMEM

FAKEMEM

FAKEMEM

1RW SRAM

1R1W SRAM

Logic Cone

1R1W SRAM

Logic Cone

Logic Cone

Read Data Bus (dout)

dout check

FAKEMEM

Golden Reference Model

Assertion

Write path

Read path

Write Data Bus

The diagram above shows the logical view of how the IP is verified formally. The complete IP is sliced into several logic cones. Each cone has a corresponding golden memory model (fakemem). When a write request is launched at the input of any logic cone, it gets captured in the golden memory. On a read, we compare the read data returned by the logic cone with golden memory. The read check happens on the interface of the cone of logic. This check is captured in the form of an SVA assertion or a property. Formal verification when run, will try to prove or break these assertions. The assertions cover the complete code, including the logic inside each cone and all the interfaces between the cones.

The diagram below zooms into one of the logical cones. It shows a simplified golden memory model and a simplified code for a read dout assertion. The formal tool exhaustively toggles all the (primary) input pins, complying with the constraints, to try and break the assertions.

rd\_dout

rd\_addr, read

wr\_adr, bw, din

Logic cone 1

FAKEMEM

dout check assertion

for logic cone 1

Golden Memory for

Logic cone 1

Logic cone 2

FAKEMEM

Golden Memory for

Logic cone 2

***assert rd\_dout\_check: assert property (@(posedge clk) disable iff (rst) (read && (rd\_adr == select\_addr)) |-> ##(MEM\_DEL) (fakememinv || (rd\_dout[select\_bit] == fakemem)));***

***reg fakememinv;  
reg fakemem;  
always @(posedge clk)  
  if (rst)  
    fakememinv <= 1'b1;  
  else if (write && (wr\_adr == select\_addr) ) begin  
    fakememinv <= 1'b0;  
    fakemem <= din[select\_bit];  
  end***

To evaluate the formal coverage of the design, we have a post processing flow developed inhouse. The fchkr.sh script retrieves all the checks in the design. Then it parses all the logfiles to make sure that all the assertions are (formally) proven to give us a 100% coverage of our design. It prints out the constraints, asssertions and proven assertion for each run.

Several design for verification techniques have been used to guarantee that formal tool is able to converge proofs completely on the IPs. The simple semantic of memory IPs (i.e. read data to an address should be the last data written to that address) have also been a key in the formal results we achieve for the IP. The goal of the techniques mentioned above is to reduce the state space both temporally and spacially. We use these to help the formal tool converge the properties exhaustively.

# Dynamic Simulation

The dynamic simulations are run for the following scenarios:

1. Reset checks
2. Boundary and capacity checks
3. Checking that the assumptions of formal.

A dynamic simulation testbench gets automatically generated with the cut. A dynamic simulation can be run by simply going to run\_sample/ directory and running the ./run script.

# Revision Change Log

|  |  |  |
| --- | --- | --- |
| Version | Date of Release | Notes |
| V1.0 | Apr 16 2014 | Initial Release. |

**Memoir Systems Inc.**

**2350 Mission College Blvd. # 1275, Santa Clara, CA 95054**

**Phone: 1-408-550-2382**

**Email: support@memoir-systems.com**