Algorithm 1r4w\_a667 Microarchitecture Document

logo.png

Release Date: Apr 16, 2014

This document provides functional information for the 1r4w\_a667 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.

Algorithm 1r4w\_a667 Microarchitecture

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

Document Naming Convention

The naming strategy for the documentation is:

Microarchitecture\_module#\_function#\_alg#

The field that doesn’t apply stays empty.

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

For example:

Microarchitecture\_core\_nr1rw

Is a core documentation with nr1w capability

Or

Microarchitecture\_algo\_4r5ws\_a32

Is an algo documentation with 4r5ws capability, algo# a32

Or

Microarchitecture\_infra\_1r1w

|  |
| --- |
| 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 1r4w\_a667 IP provides 1R4W (5 port) functionality. It is built using 2 port memories (1R1W). Given the bulk of this IP is built on 2 port memories the area and power footprint of this IP is smaller than a traditional 5 port memory.

This IP is implemented to reduce the read latency (Reduced Latency or RL). The target latency for this IP is gated by the latency of reading the data out from the T1 memory. The state lookup and all the relevant decisions are made in parallel and no additional cycle of latency is added on the read data path.

# Interface description

The algo\_top’s main interfaces include the clking ports; the read and write ports, which talk to the primary interface; the t1/2/3 ports which talk to the (T1/2/3) memories respectively.

|  |  |  |  |
| --- | --- | --- | --- |
| Pin | Pin Type | Description | Notes |
| Clock and Initialization | | | |
| clk | Input | Clock | Clock input pin |
| rst | Input | Reset | Reset pin |
| refresh | Input | Refresh | Signal to the controller that eDRAM can be refreshed in this cycle. Only needed when IP uses eDRAM. (Optional) |
| 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 |
| Write Ports | | | |
| wr\_adr | Input | Write Address | Address bits (width N bits). Parameter N is the same as that used for Read address. |
| write | Input | Write Valid | Denotes that a write command is valid |
| din | Input | Write Data | Data (width W bits) associated with a write command; Parameter W is same that for Read data. |
| bp\_thr | Input | Threshold on writes | Back Pressure threshold; |
| wr\_bp\_1/2/3/4 | Output | Back Pressure | Asserts when #writes go beyond threshold. One signal for each write port. |
| **T1/2/3 Ports** | | | |
| t1/2/3\_writeA | Output | Write Valid | Denotes that the write command is valid for the respective memory. |
| t1/2/3\_addrA | Output | Write Address | Address bits for write command |
| t1/2\_bwA | Output | Bit write enable | Bit level masking of data |
| t1/2/3\_dinA | Output | Write data | Write Data input to memories |
| t1/2/3\_readB | Output | Read valid | Denotes that the read command is valid for the respective memory. |
| t12/3/\_addrB | Output | Read address | Address bits for read command |
| t1/2/3\_doutB | Input | Read data | Data associated with a read address |

# Algos.txt

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

# This algo should be 1r4Wt (temporal restriction on writes)

# built on 1R2W\_2p\_RL

1R4W\_A667(f, w, b) [RL]

vars : n \_\_ bank\_count(1)

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

b' \_\_ bits(b)

formulae : p = TAG\_BITS(n)

w' = ws.max()

macros:

type1(BASE) : n \* 1R1W(f, ws, b')

type2 : 2 \* 1R1W(f, w', b') 🡪 (NUMRDPT+1) =2 caches for storing the read data clash and 1 write cache. The write cache stores a clash to the same bank

type3 : 3 \* 1R1W(f, w', p) 🡪 (NUMRDPT+2) tags are needed to do an initial read. That is, for the write port, and all the read ports, we first need to read the tags to see where the data is residing for that address.

* *By just adding 1 write port, we make 1r2w memory. To behave like a 1r4w memory, some temporal restrictions are imposed. A fifo will rate limit the writes. For inseimi, the restriction is 2 in 6. That is, every 6 writes in a cycle, 2 will be drained.*

The above algorithmic description defines the characteristics (frequency, depth and width) of the memory macros used for 1r4w\_a667. There are three types of memories in the IP that memogen software has to pick – type 1 (T1), type2 (T2) and type3(T3). 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 1r4w\_a667 and is used for the primary data storage. The size of this memory is equal to total capacity requirement of the algorithmic memory. It is split into n banks. In 1r4w\_a667 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**: Reads are given higher priority. In case of a collision, write data is written to the T2 memory. The write data is referred to as the cache data. There are two T2 memories- a read cache and a write cache.
  + f – target frequency of T1 memory. This parameter is user defined.
  + 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 with of the IP. This parameter is user defined.
* **T3 Memory**: The state data is stored in T3 memory. The state data is referred to as the tag data. There are 3 tag memories. One for each port.
  + f – target frequency of T1 memory. This parameter is user defined.
  + 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.
  + p – is the width of the tag which contains state info, ECC protected.

# Block level description

An algo\_1r1w\_rl2 \_top instantiates:

1. a1\_loop (algo\_mrnwt\_base) : This module provides the fifos for rate limiting the writes, so that the 1r2w memory can be converted into 1r6w memory.
2. a2\_loop (algo\_nr2w\_1r1w) : This module consists of the bulk of the algorithm. It converts the 1r1w 2 port memories into nr2w memories. It translates the traffic into regular data and state reads/writes. If two writes were scheduled to the same T1 bank as a read, then one of the write gets sent to the T2 memory, along with the state information to the T3 memory, i.e. which bank the data belongs to.
3. The t2\_bus talks to 2(=M+1) t2 memories. Since we are using 2 port (1R1W) memories here, we need only one write cache to support the 2W functionality, other than the read cache.
4. The t3\_bus talks to 3(=M+2) t3 memories, where M is number of Rd ports (1 in this case). One for each port. On any transaction, a read will be dispatched first to the respective tag.



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

# Key Parameters

The 1r4w\_a667 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. |  |  |
| ECCBITS | Controls the ECC bits in the state memory |  |  |

# 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 1r4w\_a667 algo:

**Assumptions:**

***assert\_wrfifo\_overflow\_check****: assert property (@(posedge clk) disable iff (!ready) (wrfifo\_cnt\_nxt <= FNUMWRDS))*

* *Checks that the wrfifo\_cnt\_nxt never goes beyond the fifo size.*

**Leaf nodes/assertions to prove:**

***assert\_wr\_bp\_check****: assert property (@(posedge clk) disable iff (!ready) (wr\_bp[i] == ($past(wrfifo\_cnt,FCNTDEL) > bp\_thr)))*

* *Checks that the wr\_bp (breakpoint) output for each write port goes high, whenever the wrfifo fills up until the bp\_thr (a primary output: a threshold set by the user).*

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

*(rd\_vld && ($past(fakememinv,FLOPIN+DRAM\_DELAY+FLOPOUT) || (rd\_dout[select\_bit] == $past(fakemem,FLOPIN+DRAM\_DELAY+FLOPOUT)))));*

* *Checks that the data on rd\_dout bus is the same as what was stored on fakemem. The rd\_vld and data are checked FLOPIN+DRAM\_DELAY+FLOPOUT cycles after the read access.*

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

*($past(fakememinv,FLOPIN+DRAM\_DELAY+FLOPOUT) ||*

*$past(fakemem==mem\_wire,FLOPIN+DRAM\_DELAY+FLOPOUT) || rd\_fwrd) &&*

*(rd\_fwrd || !((rd\_padr[BITPADR-1:BITPADR-BITPBNK] == NUMVBNK) ?*

*(FLOPOUT ? $past(t2\_fwrdB\_1\_wire) : t2\_fwrdB\_1\_wire) :*

*(FLOPOUT ? $past(t1\_fwrdA\_sel\_wire) : t1\_fwrdA\_sel\_wire))));*

* *Checks that if data going out on mem\_wire doesn’t match with fakemem, then the rd\_fwrd signal should be high. And then it checks that the forwarding is happening from the cache or t1 memories.*

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

*((rd\_padr == {NUMVBNK,(FLOPOUT ? $past(t2\_padrB\_1\_wire) : t2\_padrB\_1\_wire)}) ||*

*(rd\_padr == {select\_bank,(FLOPOUT ? $past(t1\_padrA\_sel\_wire) : t1\_padrA\_sel\_wire)})));*

* *Checks that the value of padr matches the physical bank value; either t1 bank address or cache.*

***assert\_derr\_check****: assert property (@(posedge clk) disable iff (rst) (read && (rd\_adr == select\_addr)) |-> ##(FLOPIN+DRAM\_DELAY+FLOPOUT)*

*($past(fakememinv,FLOPIN+DRAM\_DELAY+FLOPOUT) ||*

*((rd\_padr[BITPADR-1:BITPADR-BITPBNK] == NUMVBNK) ?*

*(rd\_serr == (FLOPOUT ? $past(t2\_serrB\_1\_wire) : t2\_serrB\_1\_wire)) &&*

*(rd\_derr == (FLOPOUT ? $past(t2\_derrB\_1\_wire) : t2\_derrB\_1\_wire)) :*

*(rd\_serr == (FLOPOUT ? $past(t1\_serrA\_sel\_wire) : t1\_serrA\_sel\_wire)) &&*

*(rd\_derr == (FLOPOUT ? $past(t1\_derrA\_sel\_wire) : t1\_derrA\_sel\_wire)))));*

* *Checks all possible combinations of derr and serr.*

**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**