Algorithm 4r5ws\_a32 Microarchitecture Document

logo.png

Release Date: Apr 16, 2014

This document provides functional information for the 4r5ws\_a32 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 4r5ws\_a32 Microarchitecture

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

|  |
| --- |
| Contents |

[1. Functional Overview 1](#_Toc385371829)

[2. Interface description 2](#_Toc385371830)

[3. Algos.txt 5](#_Toc385371831)

[4. Block level description 8](#_Toc385371832)

[5. Key Parameters 10](#_Toc385371833)

[6. Formal 11](#_Toc385371834)

[7. Revision Change Log 15](#_Toc385371835)

# Functional Overview

The 4r5ws\_a32 IP provides 4R5Ws (4 Rd ports and 5 write port, sequential) functionality. It is built using two 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 mulit-port memory. The two port memory is mainly used to store the data and the internal state needed by the IP logic.

This IP assumes 80 words and that the consecutive reads and writes will be to the Sequential memory addresses. The memories are divided into slow and fast memories. The read and write ports assume some constraints on them which are described in the Renaissance mRnW sequential Functional Datasheet v1\_3.

# 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 2 port (T1) memories; the t2 ports which talk to the 2 port (T2) memories, ; the t3 ports which talk to the 2 port (T3) memories and ; the t4 ports which talk to the 2 port (T4) memories.

Table ‑: Interface Pins

|  |  |  |  |
| --- | --- | --- | --- |
| Pin | Pin Type | Description | Notes |
| Clock and Initialization | | | |
| clk | Input | Clock | Clock input pin |
| lclk | Input | Slow Clock | Slow 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 |
| read\_vld | Output | Read Data Valid | Denotes valid read data |
| read\_serr, read\_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. |
| **T1/2/3/4 Ports** | | | |
| t[1/2/3/4]\_writeA | Output | Write Valid | Denotes that the write command is valid for the respective memory. |
| t[1/2/3/4]\_addrA | Output | Write Address | Address bits for write command |
| t[1/2/3/4]\_bwA | Output | Bit write | Bit –wise write enable signal for the write data |
| t[1/2/3/4]\_dinA | Output | Write data | Write Data input to memories |
| t[1/2/3/4]\_readB | Output | Read valid | Denotes that the read command is valid for the respective memory. |
| t[1/2/3/4]\_addrB | Output | Read address | Address bits for read command |
| t[1/2/3/4]\_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.

4R5Ws\_A32(f, w, b) [RL]

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

b' \_\_ bits(b)

m \_\_ bank\_count(4, 4)

ws \_\_ words\_array(n, ceil(w,20\*4))

ws' \_\_ words\_array(m, ceil(19\*w,20))

formulae : p = TAG\_BITS(n)

w'' = ceil(w, 20\*4)

w' = ws.max()

f' = ceil(f \* 2, 3)

macros:

type1(BASE) : 4 \* n \* 1R1W(f, ws, b')

type2 : 4 \* 2 \* 1R1W(f, w', b')

type3 : 4 \* 3 \* 1R1W(f, w', p)

type4 : 4 \* 1R1W(f', ws', b')

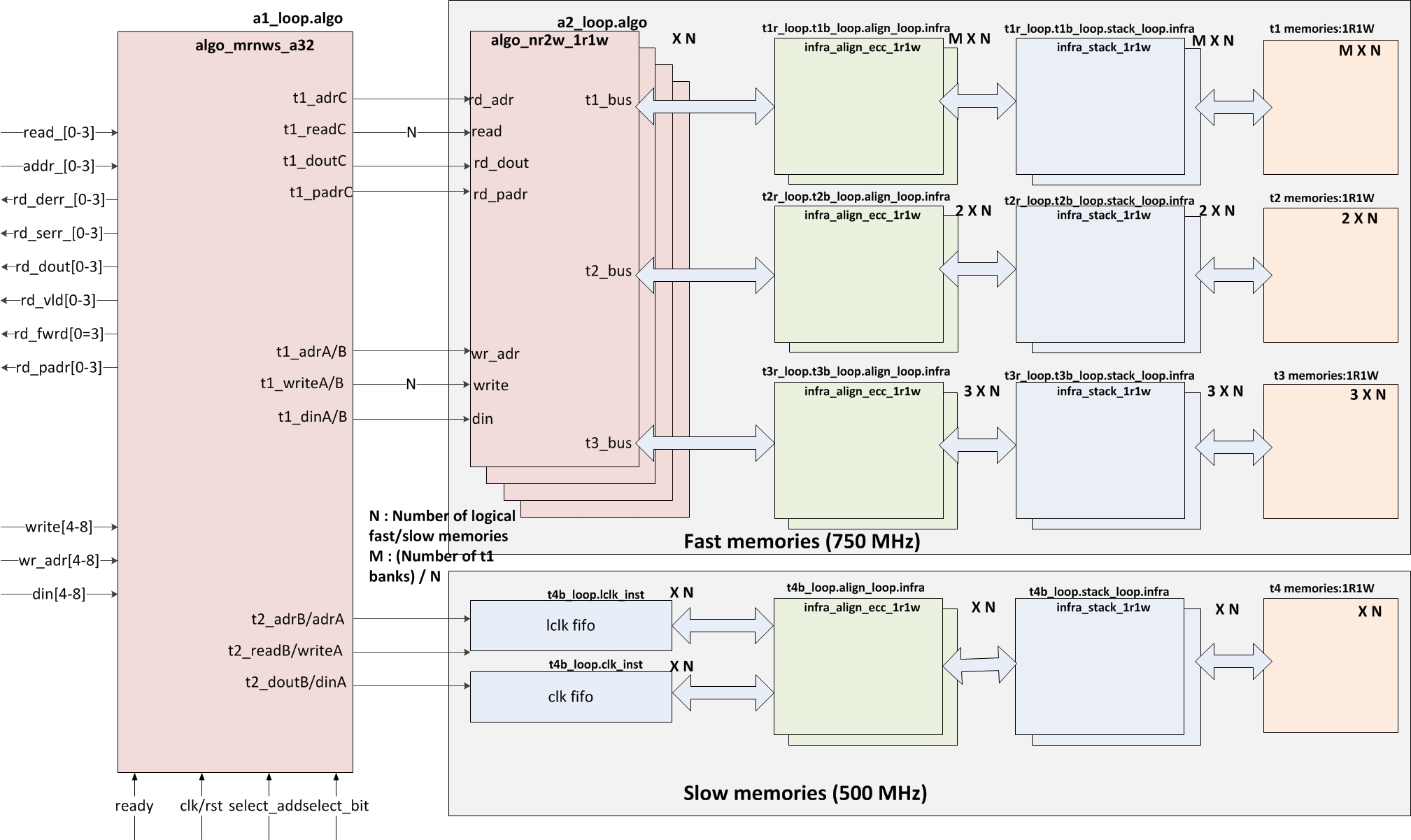
The above algorithmic description defines the characteristics (frequency, depth and width) of the memory macros used for 4r5ws\_a32. The math has been done assuming there are 80 words in the memory; 4 words in fast memory and 76 words in slow memory. There are four types of memories in the IP that memogen software has to pick – type 1 (T1), type2 (T2), type3(T3) and type4(T4). T4 is the slow memory and T1,T2 and T3 together provide the fast memory functionality.

* **T1 Memory:** This memory is the bulk of the memory used in the 4r5ws\_a32 for fast memory. The size of this memory is equal to total capacity requirement of the algorithmic fast memory. In 4r5ws\_a32 1R1W (two 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 on T1 reads. In case of a collision, write data is written to the T2 memory which serves as a cache. The 4r5ws\_a32 has two caches for the 3 ports. One read cache and one write cache. One write cache has been optimized out.
  + 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:** This memory is used to store the state data referred to as the tag data. Reads are given higher priority on T1 reads. In case of a collision, state data is written to the T3 memory. The tag for 4r5ws\_a32 consists of the bank number. Note that this structure is protected by ECC (SECDEC). The 4r5ws\_a32 provides three memories for tag. One for each port (1R+2W).
  + 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, the write enable and state info, ECC protected.
* **T4 Memory:** This memory is used for storing the slow memory data.
  + f – 2/3 of target frequency for T4 memory.
  + ws’ – The maximum depth of the T4 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 t4 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. a1\_loop (algo) : This module consists of the module algo\_mrnws\_a32. It splits the read/write traffic into fast traffic and slow traffic. The T1 port interfaces with the fast memory. The T2 output port of the a1\_loop interfaces with the slow memory.
2. a2\_loop (algo) : The module algo\_nr\_2w\_1r1w takes the T1 output of a1\_loop and implements a 1r2w functionality. N, the number of logical fast/slow memories is assumed 4 in this case. Also M, the number of T1 banks is 4. T1 memories store the bulk of fast memory writes. In case of rd-wr clash, the data and tag will be stored T2/T3 memories.
3. lclk/clk fifo: The interface between slow memories and fast logic is synchronized using fifos.
4. align\_loop and stack loop: Used to interface with each memory.



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

# Key Parameters

The 4r5ws\_a32 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 |  |  |
| ENAPAR | Enable Parity |  |  |
| ENAECC | Enable ECC |  |  |
| ENAHEC | Enable Half ECC |  |  |
| ENAQEC | Enable Quad ECC |  |  |

# 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/fv/<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 4r5ws\_a32 algo:

Primary design constraints:

Leaf nodes/assertions to prove:

*<Need to add constraints>*

**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

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)));***

dout check assertion

for logic cone 1

***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/assertions 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. To report more information, use –ll option with the fchkr.sh script.

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**