MIPS® BusBridge™ 2 Module Users Manual

Document Number: MD00429
Revision 02.06
October 5, 2011

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

MIPS™ BusBridge™ 2 module features an easily configurable, high-performance, low latency MIPS® core interface to the AMBA™ AHB. The module supports all members of the M4K™, 4KE™, 24K™, 34K™ family of high-performance RISC cores and allows for easy integration of the cores into any 32-bit AHB system. The interface is designed for semiconductor manufacturing companies, ASIC developers, and system OEMs wishing to rapidly integrate a MIPS core into an AHB-based system.

The MIPS BusBridge™ 2 module (MBB2) is configurable, allowing the user to either implement a reduced AHB master without arbitration (AHB Lite) or a full AHB master supporting the complete AMBA AHB specification. The reduced master option is designed for a single master AHB system where speed and simplicity are the main design issues. If a multi-master AHB system is required, then the full MIPS BusBridge™ 2 module configuration adds support for both arbitration and slaves using Split/Retry responses.

In order to support systems where the MIPS® core and the AHB bus system are running with the same frequency, as well as systems where the MIPS® core is running at a higher frequency than the AHB bus, the following clock ratios between core clock and AHB clock are supported: 1:1, 2:1, 3:1, 4:1, 5:1, and 6:1.

The following chapters make up the MIPS BusBridge™ 2 User's Manual:

Chapter 2, "Design Database" on page 11

Chapter 3, "Design Implementation" on page 21

Chapter 4, "Functional Simulation" on page 39

Chapter 5, "Synthesis" on page 71

NOTE: in the features and limitations lists that follow those marked with (MBB2) are differences between the MBB2 and the original MIPS™ BusBridge™ module.

1.1 Design features

- No delay in MBB2 module except when demanded by one of the bus protocols or timing considerations.
- CPU Core system interface (Note: Where a description applies to both the EC™ and OCP interfaces this document will use the term CoreIF.)
  - Supports all Core interface access modes (single and burst read/write).
  - CoreIF bursts converted to AHB bursts.
  - 2 outstanding read / write commands (one currently processed on the AHB bus and one stored inside the MBB2).

- AHB interface
    - All protocol modes supported including Retry, Split and Error handling.
    - Full bus arbitration supported.
    - 1 outstanding read / write command (required by protocol).
- Completely transparent operation as seen from the core (no address mapping).
- Supports 32-bit AHB only. (MBB2)
- Bus width conversion provided for the case where the CoreIF is 64 bits and the AHB is 32 bits.
- n:1 clock ratios supported where n may be in the range 1 .. 6.

1.2 Design limitations

- No endianness mixing supported. CoreIF and AHB systems are both either little or big endian.
- Only MIPS® cores with simple byte enable support can use the MBB2.
- This design does not support clock ratios where the AHB system is running at a higher clock than the core.
- Only systems where the CoreIF clock (cpu_clk) and AHB clock (hclk) are in phase (i.e. rising edge of hclk correspond with rising edge of cpu_clk) are supported.
- Selection of clock ratio is done at synthesis time (a single design may not support both clock ratios under software control).
- AHB locked cycles are never generated, hlock will always be 0. (MBB2)
- The AHB hprot signal will always be driven to 0.
- No support for extended AHB addressing where CoreIF = EC has a 36-bit address bus. (MBB2)
- For the M4K™ CPU Core the following are not supported: The external Lock/Unlock protocol from the MIPS32™ ll/sc instruction pair, cycle cancellation via Abort/AbortAck.

Chapter 2

Design Database

This chapter describes the directory structure and design database, as well as how to perform the installation and setup of the design database. The following four sections make up this chapter:

2.1 Overview of Delivery Directory Structure

The directory structure contains RTL code and testbench for functional simulation and synthesis. The structure effectively separates the RTL from configurations and results, making it possible to try different configurations and comparing results without making multiple copies of the entire database. This will also ease integration when new releases of the MIPS™ BusBridge™ 2 module (MBB2) are received.

When the MBB2 customer deliverable is unpacked, the top level directory will normally be called ‘‘mab’’ and will have the directory structure shown below. Note that as the installation proceeds and the user does functional simulation and/or synthesis additional directories will be created.

2.1.1 "external" Directory ($TECH_DIR & $TECH_GATE_DIR)

The "external" directory is used to hold links to the user's Synopsys synthesis technology libraries and the Verilog models for post-synthesis (gate-level simulation).

2.1.2 "bin" Directory

The "bin" directory contains executable scripts used globally in the design flow. In order to make the scripts reachable from subdirectories, the bin directory is added to the UNIX path by the bin/customer_source.me script.

The two major scripts are "run_test" which is used to execute the functional verification tests and "run_synth" for the synthesis flow.

After making a backup copy, the customer_source.me script should be modified and sourced before the design database can be used. The scripts set up various environment variables required by the design flow. Refer to Section 2.2 "Install and Setup the Design Database" for more information.

2.1.3 “integration” directory

This contains files needed to integrate the CPU core into the MBB2 verification structure. The major file here is mab_cpu_model.info.<m4k | m24k | m34k | mm4k> described in Table 2.2.

2.1.4 “design/rtl” Directory ($SUBSYS_RTL_DIR)

The “design/rtl” directory contains the Verilog description of the MIPS™ BusBridge™ 2 module design. There are three sub-directories holding different parts of the design.

2.1.4.1 “design/rtl”/mab

Contains the main MBB2 RTL files. See section 5 of Chapter 3, “Design Implementation” on page 21.

2.1.4.2 “design/rtl”/config

As delivered, this contains a template user configuration file mab_config.vh.default. See section 7 of Chapter 3, “Design Implementation” on page 21 for details.

2.1.4.3 “design/rtl”/shared

Contains a number of modules used for instantiated registers in the design, a header file for the MBB2 in general, and a header file containing a number of Verilog defines derived from the main MBB2 configuration file described above. See section 6 of Chapter 3, “Design Implementation” on page 21 for details.

2.1.5 “verif” Directory

The “verif” directory contains the scripts and test harness for running functional simulations on the MIPS™ Bus-Bridge™ 2 module. The functional simulation and structure of the tests is described in Section 4.1 “The Verilog Testbench environment”.

2.1.5.1 “verif”/tb

Top level testbench Verilog files and CPU ‘wrappers’ used to instantiate the various cores into the testbench. The top testbench file is named ‘testbench_mab.v’.

2.1.5.2 “verif”/shared

Contains the Verilog source files that constitute the AHB simulation sub-system.

2.1.5.3 “verif”/include

Verilog header files for the verification system.

2.1.5.4 “verif”/tb_config

Initially empty this directory will contain the output of the configure_mab_tb.pl script.

2.1.5.5 “verif”/custom_tb

Contains an example of an alternative, customer, testbench. See the README file for details.

2.1.6 “regression” Directory ($SUBSYS_SW_DIR)

The “regression” directory contains the test stimuli for the Verilog testbench. The stimuli are a collection of assembler and “C” programs executed individually, or as a total test regression suite, on the instantiated core. The structure of the tests and the tests performed are described in Section 4.3 “Using the Verilog Testbench” and Section 4.1 “The Verilog Testbench environment”.

2.1.6.1 “regression”/tests

A number of sub-directories hold the main “C” and assembler for the tests themselves.

2.1.6.2 “regression”/sys

Assembler code used to create wrappers in which to compile and run the tests. Includes start-up and exception handling routines.

2.1.6.3 “regression”/include

“C” header files needed to compile and build the functional tests.

2.1.7 “synth” Directory ($SUBSYS_SYNTH_DIR)

The “synth” directory provides support for synthesis using Synopsys’ DesignCompiler™, including general scripts and makefiles.

2.1.7.1 “synth”/config

Constraints and other configuration parameters that may be specific a given core or technology library are located here. Note that these files are not used directly in the synthesis flow but should be copied to the synthesis build directory and then modified to the user’s requirements. See Chapter 5, “Synthesis” on page 71 for details on the synthesis setup and flow.

2.1.8 “build” Directory ($SUBSYS_BUILD_DIR)

The “build” directory is the working directory, and all configurations, compiled testbenches and results are located in this directory. As delivered, the customer_source.me file in “bin” defines a default

```
$SUBSYS_BUILD_DIR = $SUBSYS_DIR/build/build1
```

and this directory is empty. As the user proceeds with functional simulation and synthesis new subdirectories will appear here:

- $SYNTH_OUT_DIR - synthesis configuration, constraints and results. The default value is ‘‘synth’’.

  Defines technology used, clock period, input/output constraints, etc. used for a particular synthesis run. Output files such as reports, logs, and netlists may also be found in this directory.

- tb_<core>_<model> - when the testbench or a software test is compiled, the compiled result is placed in one of these directories. <core> and <model> are defined dependent on the settings of $CORE and $MODEL in bin/customer_source.me. A VMC simulation of a 4K™ core will be put in a directory named “tb_4k_vmc”.

2.1.9 “doc” Directory

The “doc” directory contains

- The MIPS Bus Bridge 2 ™ Users manual in PDF format.
- A ReleaseNotes.txt file.
- The Common Design Interface specification in PDF format.
- A *Howto* subdirectory that hold a few useful hints and tips.

2.2 Install and Setup the Design Database

2.2.1 Creating Common Design Interface files

Before starting the installation process the user must create a Common Design Interface (CDI) file for the Core model. For reference the CDI specification document is included in the ‘‘doc’’ directory.

2.2.1.1 RTL Model

You should refer initially to the appropriate section of your Core Implementors Guide for how to do this. For simplicity, however, the basic commands needed are included here:

24K™ and 34K™Cores:

```
<core_install>/bin/tpz_syn_filelist -sim -top=tpz_top -c=<your_config_dir> > <your_CDI_file>
```

4K™/4KE™/M4K™ Cores:

```
<core_install>/bin/m4k_filelist -sim -top=m4k_top -c=<your_config_dir> > <your_CDI_file>
```

2.2.1.2 VMC Model

There is a template - vmc.cdi.skel - provided in the ‘‘integration’’ directory. Copy this to a file of you own choosing and edit following the instructions in the template file.

2.2.1.3 BFM model

For simulation runs that use a Bus Functional Model (BFM) this is not necessary since the BFM packages contain the necessary CDI files and the MBB2 verification system will search for the appropriate one.

2.2.2 Install Procedure

In addition the user is strongly advised to read the 2 files:

```
<mab_top_dir>/README

<mab_top_dir>/doc/ReleaseNotes.txt
```

1. Modify bin/customer_source.me reflect the core type, the link to technology libraries, the location of the build directory etc. as per the table below. It would normally be advisable to copy the original source.me to e.g. my_source.me and to do the edits in the copy.

   See Table 2.1 below for the customer_source.me environment variables that need to be set or modified.

   Note that $SUBSYS_DIR is the MBB2 installation directory.

Table 2.1 source.me variables

Variable	Effect	Default
CORETYPE	Sets the CPU core type. Values = MM4K | M4K | M4KE | M24KC | M24KF | M34KC | M34KF Note: The value of CORETYPE is passed to Verilog, where identifiers may NOT begin with digits. An “M” is prepended to the CORE names to work around this. This is why what is commonly called a 4KE™ is here called an M4KE. This is also why there is a CORE named “MM4K”, which is only to distinguish the “M4K” from the “4KE”	None
TECH_LIB	Link to technology library used for synthesis. Value = $SUBSYS_DIR/external/<any>	$SUBSYS_DIR/external/lib
SUBSYS_BUILD_DIR	Top level directory containing the testbench and synthesis sub-directories Value = $SUBSYS_DIR/build/<any>.	$SUBSYS_DIR/build/build1
SYNTH_OUT_DIR	Sub directory of $SUBSYS_BUILD_DIR used for synthesis.	synth
SYNTH_INTEGRATION	Points to the directory used to hold files needed to integrate the CPU model (RTL | VMC | BFM) into the MBB2 verification structure	$SUBSYS_DIR/integration
MAB_SIMTYPE	Select the type of simulator to use. Value = vcs | mti | nc in either upper or lower case	None
MAB_GCC	Selects the software tool chain type being used. Value = CODESOURCERY|SDE	
MAB_SDE_VERSION	If the MAB_GCC variable is set to ‘‘SDE’’ then Select the version of the SDE™ compilation tools being used. Values = <5 | 6>	None
MAB_CORE_HOME	Full path to the top of the processor core installation.	None
MAB_CORE_RELEASE	Version of the processor core used for simulation. Note that both RTL and VMC simulation must use the same revision. The value of this should be a string: xx_yy_zz The corresponds to the CPU RTL revision number.	None
MAB_BFM_HOME	Full path to the top of the BFM installation.	None

Variable	Effect	Default
MAB_BFM_RELEASE	Version of the BFM used when building the random testbench. The value of this should be a string: xx_yy_zz The corresponds to the CPU BFM revision number.	None
LMC_HOME	Root of the Core VMC installation. Typically this would be set to: $MAB_CORE_HOME/vmc_install	None

2. cd to the installation dir and do:

   source <file_created_in_step_1_above>

3. Create a link in the “external” directory to the synthesis technology libraries and post-synthesis Verilog models

   e.g.

   ln -s <my_tech_dir> $TECH_LIB

4. Create a file in $SUBSYS_INTEGRATION that points to the CDI files created in Section 2.2.1 “Creating Common Design Interface files”. The file created must have one of the following names:

   For 34K ™ Cores: mab_cpu_model.info.m34k

   For 24K ™ Cores: mab_cpu_model.info.m24k

   For 4K ™ and 4KE™Cores: mab_cpu_model.info.m4k

   For M4K ™ Cores: mab_cpu_model.info.mm4k

Table 2.2 mab_cpu_model.info entries

Name	Description	Options	Default
CPU_RTL_CDI	Path to CDI file for simulation with CPU RTL	Any file path	none
CPU_VMC_VCS_CDI	Path to CDI file for simulation with CPU VMC model. Simulator is VCS.	Any file path	none
CPU_VMC_VXL_CDI	Path to CDI file for simulation with CPU VMC model. Simulator is NC-Verilog or ModelSim	Any file path	none
CPU_NET_CDI	Path to CDI file for simulation with a post synthesis or post P&R gate level netlist	Any file path	none
CPU_BFM_CDI	Here for completeness only. The user should not set this unless instructed to do so by MIPS® Technologies technical support.	Any file path	none
CPU_BFM_LIB	The Core BFMs can be built with either the threads or pthreads libraries.	pthread | thread	pthread
CPU_BFM_TYPE	Leave unset unless requested.	new | old	new
CPU_VMC_LIB	The Core VMC models can be built with either the threads or pthreads libraries.	pthread | thread	pthread

Name	Description	Options	Default
CPU_VMC_TYPE	Leave unset unless requested.	new | old	new

Important Note: If an <any file path> value for one of the XXX_CDI entries does *not* start with a '/' it will be considered relative to the top level of the MBB2 installation. e.g.

```
<any file path> = integration/my.cdi
```

will be interpreted as:

```
${SUBSYS_DIR}/integration/my.cdi
```

5. Create the synthesis result output directory:

```
mkdir $SUBSYS_BUILD_DIR/$SYNTH_OUT_DIR
```

6. The MBB2 RTL must now be configured by creating the basic MBB2 configuration Verilog header file

```
$SUBSYS_RTL_DIR/config/mab_config.vh
```

by running the configure_mab.pl script in the ''bin'' directory. To get a list of possible options type:

```
configure_mab.pl -h
```

The Verilog 'defines in mab_config.vh are described in Table 3.4 of Chapter 3, "Design Implementation" on page 21

Note 1: configure_mab.pl will set one and only one of the MAB_CONFIG_BUS_XXX defines in mab_config.vh automatically based on the value of the $CORE environment variable.

Note 2: It is possible, but not recommended, to create this file 'by hand' by copying and editing the default file:

```
$SUBSYS_RTL_DIR/config/mab_config_default.vh
```

7. Use the configure_mab_tb.pl script to create a default testbench configuration file called ''default.cfg''. This will appear in the ''verif''/tb_config directory.

```
configure_mab_tb.pl -o default
```

See Section 4.2.2 "The "configure_mab_tb.pl" script"for more details. Note that users of pre-MR1 24K™ cores should also set the no24k-ocp-2-1-compliant flag when creating default.cfg.

```
configure_mab_tb.pl -o default --no24k-ocp-2-1-compliant
```

8. A tool path should be set up to the CAD tools used: The default simulator is VCS™ from Synopsys, though ModelSim™ from Mentor Graphics and NC-Verilog™ from Cadence are also supported. DesignCompiler™ from Synopsys is supported for synthesis.

In addition these tools from MIPS Technologies should be in the tools path:

- SDE-MIPS or SDE-Lite gcc based toolchain.
- The MIPSSim™ instruction simulator (if MIPSSim + BFM simulations are desired).

See Section 2.4 “Supported Tool Versions” for a complete list.

If the user wishes to do VMC simulation a few additional setup steps are necessary. It is assumed for what follows that the VMC installation process for the correct platform has been performed in the CPU core install directory and so the following environment variables have been set correctly. See “The CPU Model” section of Chapter 4.

- `LMC_HOME` is set. In general this will be to $MAB_CORE_HOME/`vmc_install`.
- `LD_LIBRARY_PATH` includes a path to `$LMC_HOME/lib/sun4Solaris.lib` or `$LMC_HOME/lib/x86_linux.lib`.
- `LM_LICENSE_FILE` includes a path to the user’s VMC license file.

9. Generate a VMC template file in the `$SUBSYS_DIR/vmc-models` directory. For VCS simulation this would be:

   cd $SUBSYS_DIR/vmc-models

   vcs -lmc-swift-template <vmc_model_name>

   where <vmc_model_name> is one of m34k_vmc_model, m24k_vmc_model or m4ke_vmc_model

For details on the creation of this template file for other simulators see the VMC section of your core Integrator’s Guide.

10. Now run the configure_mab_tb.pl script with the following options to generate a testbench configuration file for basic VMC simulation (see Section 4.2.2 “The “configure_mab_tb.pl” script” for more details on this script):

- --core-model=VMC
- --tb-type=DIRECTED
- -o <vmc_tb_config_name>

2.3 Getting Started

It is now possible to run functional simulation or synthesis using the scripts in the bin directory. For full details see the Verification and Synthesis chapters of this manual, but at this stage the user is recommended to try out the simple tests outlined below to ‘sanity check’’ the installation.

2.3.1 Simple Simulation Tests

When the core and the MIPS™ BusBridge™ 2 module installations have been set up, try to build the testbench and run a simple RTL simulation by invoking this command:

```
run_test --build --config=default samples/hello_world
```

The text similar to the one shown below should appear on the terminal window:

```
Chronologic VCS simulator copyright 1991-2001
Contains Synopsys proprietary information.
```

```
Compiler version 6.0; Runtime version 6.0;  Sep 25 17:29 2001

Info: Core is shiny 4K(TM) running MIPS32(TM)
Info: Verbose level is V0. Print enabled, VCD dump enabled
Info: Running in Little Endian
Info: Clock Ratio = 1:1
Info: Multi Master Priority:
        Round Robin
        Default = Master # 2

Message: Starts at time 13866
Message: Running in Little Endian
Message: Verbose level is V0
Message: Ends at Time 83590
Message:

***************
* Hello World *
***************

Info: Program terminated OK with exit code 00000000
Info: Simulation time:               114752

$finish at simulation time            114752000
           V C S   S i m u l a t i o n   R e p o r t
Time: 114752000 ps
CPU Time:  49.290 seconds; Data structure size: 2.7Mb
Tue Sep 25 17:30:31 2001
```

To test that a VMC installation has been set up correctly the following command should give similar outputs to the previous one:

```
run_test --build --config=<vmc_tb_config_name> samples/hello_world
```

where <vmc_tb_config_name> is that same as that used in step 10 of the installation.

For further information on more detailed simulation and regression refer to Chapter 4, “Functional Simulation” on page 39 and the README file found in the root of the design database.

2.3.2 Simple Synthesis Test

While a full description of the synthesis flow is delegated to Chapter 5, “Synthesis” on page 71 a simple check on the setup should be performed here by running the synthesis command to initialise the synthesis output directory:

```
run_synth -t init -b mab
```

This will produce a number of sub-directories in $SUBSYS_BUILD_DIR/$SYNTH_OUT_DIR: WORK, check, checkpoint, gate, report, log, mapped, unmapped.

2.4 Supported Tool Versions

VCS™: Version 2011.03.

NC-Verilog™: IUS Version 8.20.s8

ModelSim™: Questa Version 6.5c

DesignCompiler™: Version 2005.09.

The CodeSourcery CodeBench version for MIPS® microprocessors. Version 2011.03-92 or later. A freely downloadable ‘‘Lite’’ edition is available from this page (look for the MIPS download section):

http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/lite-edition

MIPSSim™ instruction set simulator: Version 4.9.6.

Chapter 3

Design Implementation

3.1 Introduction

This document describes the new MIPS^{TM} bus bridge product, the MIPS™ BusBridge™ 2 module (MBB2). MBB2 will interface a MIPS® processor core with either the EC^{TM} or OCP system interface to the AMBA^{TM} AHB. MBB2 will support the full EC, OCP, and AHB specifications as described in Figure 3-1, Figure 3-2, Figure 3-3.

This design will support clocking ratios where the Core interface is running at the same speed or faster than the AHB interface but not the case where the AHB is running faster than the Core.

Note:

- Where a description applies to both the EC™ and OCP interfaces this document will use the term CoreIF.
- Throughout this document clock ration 2:1 is used to mean all clock ratios n:1 with n>1.

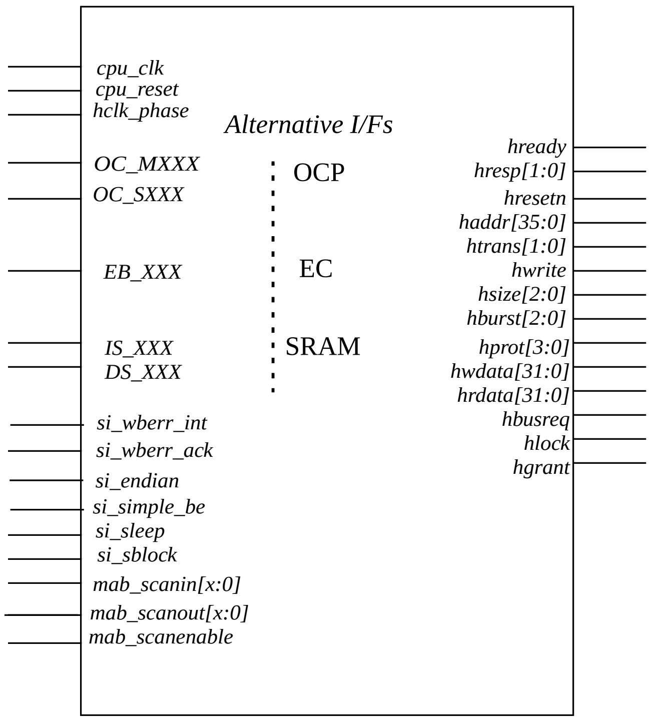
3.2 Design goals

- The design must be synthesized with clock frequency up to that of the 4K^{TM} , 24K^{TM}, 34K™ cores in same technology.
- The maximum latency degradation should be one clock on commands outbound to AHB and one clock on read data inbound from the AHB.
- Delays on signal originating from the AHB bus are to be kept at a minimum. In particular, the delay on hready and hresp[1:0] must be minimized since they are late signal in an AHB system.

3.3 User interface

Top level connection of the MBB2 is shown in Figure 3-1.

Figure 3-1 Top level connections of the BusBridge™ 2 module



The signals EB_XXX (EC™ bus) and OC_XXX (OCP bus) are alternatives selected at configuration time. For a description of these signals see Section 3.5.1 “External signal list”. Note that the bus signals names match those of the top level module of the CPU core for ease of integration.

3.3.1 Basic Operation.

There are some major differences between the CoreIF and the AHB that the MBB2 must allow for

Protocol differences -

- **Multiple masters:** The AHB bus accepts multiple masters and therefore requires arbitration of the bus, where the CoreIF only has one - the core.

- **Pending transfers:** The CoreIF accepts a number of outstanding transactions, whereas the AHB only allows one outstanding transaction before the master must stall and wait for the completion (hready) of the pending transfer.

- **Address/data phases:** For both the CoreIF and the AHB, each transaction has an address phase and a data phase. But there is a fundamental difference between them in that for AHB the data phase is tied to the previous valid address phase whereas the CoreIF’s read and write data phases are decoupled from the address phase.

  Note also that the AHB has no notion of sub-block ordering for bursts.

- **Early burst termination:** When the MBB2 e.g. has started a burst on a granted AHB, it is not certain that the MBB2 can keep the AHB granted until the burst has completed. E.g. another master with higher priority can take the bus from the MBB2.

- **Split/Retry:** Even after the MBB2 has been granted the bus and started an address phase on AHB the selected slave may respond with a SPLIT/RETRY on the hresp response bus and the MBB2's bus grant will be withdrawn. Under these circumstances the MBB2 must restart the arbitration process, reaquire the bus, and repeat the transfer attempt. From an AHB master's point of view slave SPLIT and RETRY responses are identical.

AHB transfer types generated -

- **Single transfer:** This transfer type consists of one address phase and one data phase. The data width can be 8, 16, 32 and 64 bit, and can change for each single transfer. The transfer data width of 64 bit is only supported by a 64 bit core.

  Because the CoreIF is a ‘‘byte enable’’ bus and the AHB is a ‘‘size/address’’ bus translations of complex byte enable patterns would take several AHB transfers. For this reason the CoreIF will operate in ‘‘simple byte enable’’ mode where only *natural* byte patterns appear.

- **Burst transfer:** This transfer type consists of multiple address and data phases. The data width in a burst is determined by the AHB bus width (32/64 with 64 bit not supported by 4K™cores). For a 64 bit CoreIF on a 32 bit AHB the number of beats in a burst will be 8, otherwise it will be 4.The core always transfers bursts in wrap-mode, and therefore only wrap-bursts on the AHB are supported.

For this reason we will assume that

1. CoreIF transactions are handled one at a time in the order they arrive at the MBB2.

2. At least one level of transaction pipelining will be supported - one command actually going out on the AHB while the next is being received from the CoreIF. In fact, in order to keep the AHB transfers streaming as much as possible, FIFOs will be provided between the CoreIF and the AHB that can store at least 4 addresses/write-data pairs.

3. If a burst is interrupted for any reason before completion, the remaining transfers will be transformed into single transfers. There are no timing differences on the AHB between a single transfer back-to-back and a burst transfer. The only difference is that a slave and the arbiter during a burst know how long the transfer is.

4. This specification requires that the core is running in simple byte enable mode (SimpleBE mode) and that sequential ordering protocol is selected (sequential addressing mode).

Note that in the above discussion bus arbitration is not relevant to AHB-Lite where there is only one master, the MBB2. This is a configuration option for the MBB2 which has the effect of hard-wiring the hgrant signal to ‘1’, and allows the hresp[1:0] bus to take on only the values ‘OK’ or ‘ERROR’.

3.3.2 Clocking and Reset

An external circuit must generate the required clocks (bus_clk for CoreIF and hclk for the AHB) with the proper phase relationship. An external clock phase (’hclk_phase’) signal is used to indicate the phase of hclk related to bus_clk See Figure 3-4. In the case where CoreIF = OCP the bus_clk may not be the same as the Core clock since the 24K™, 34K™core families allows the Core to run at a higher speed than the OCP interface. If this is the case then it is the responsibility of the user to generate the SI_OCPSync signal that relates bus_clk to the Core clock.

All flip flops in the design are effectively clocked on the positive edge of the internal gclk which is, effectively, a globally gated (by si_sleep) version of the input bus_clk. Note that this implies that the AHB clock, hclk, is not used in the design.

The hresetn from the AHB signal is synchronized to the cpu_clk clock domain and used as reset for the internal registers of the MBB2. The cpu_reset signal is, configurably, an input or an output.

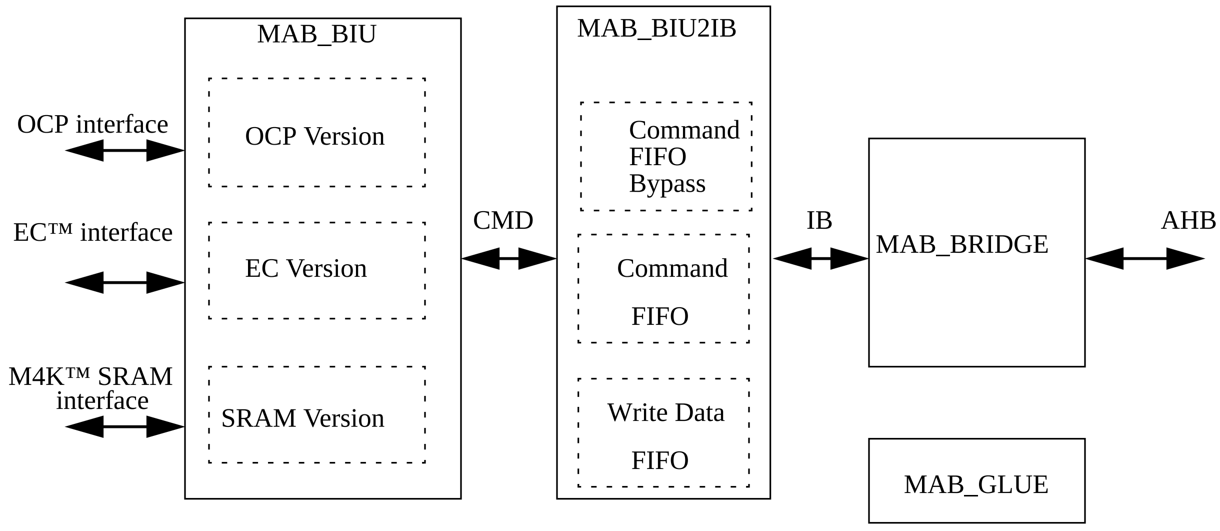
- If it is an output then it is driven from the synchronised hresetn described above.
- If it is an input then it is or'ed together with the synchronised hresetn to from the internal MBB2 reset.

3.4 Block Description

Figure 3-2 shows a top level picture of the MBB2 Bridge. As can be seen there is a Bus Interface block (MAB_BIU), build time selectable between OCP, EC, SRAM with a common interface via CMD to a block called MAB_BIU2IB. From there the IB bus connects to the main bridge module (MAB_BRIDGE). Also instantiated at this level is the clocking, reset, and fixed output block MAB_GLUE.

Note that the busses shown here are described in Section 3.5 "Main Signals".

Figure 3-2 MBB2 Top Level



3.4.1 MAB_BIU

The user can instantiate one of three modules depending on whether CoreIF is OCP, EC, or SRAM. The output is called the CMD bus which is very similar to the EC™ bus in that it has an address for each beat of a burst and write data is aligned with write address and control information. Therefore the EC™ BIU module does very little work in contrast to the OCP.

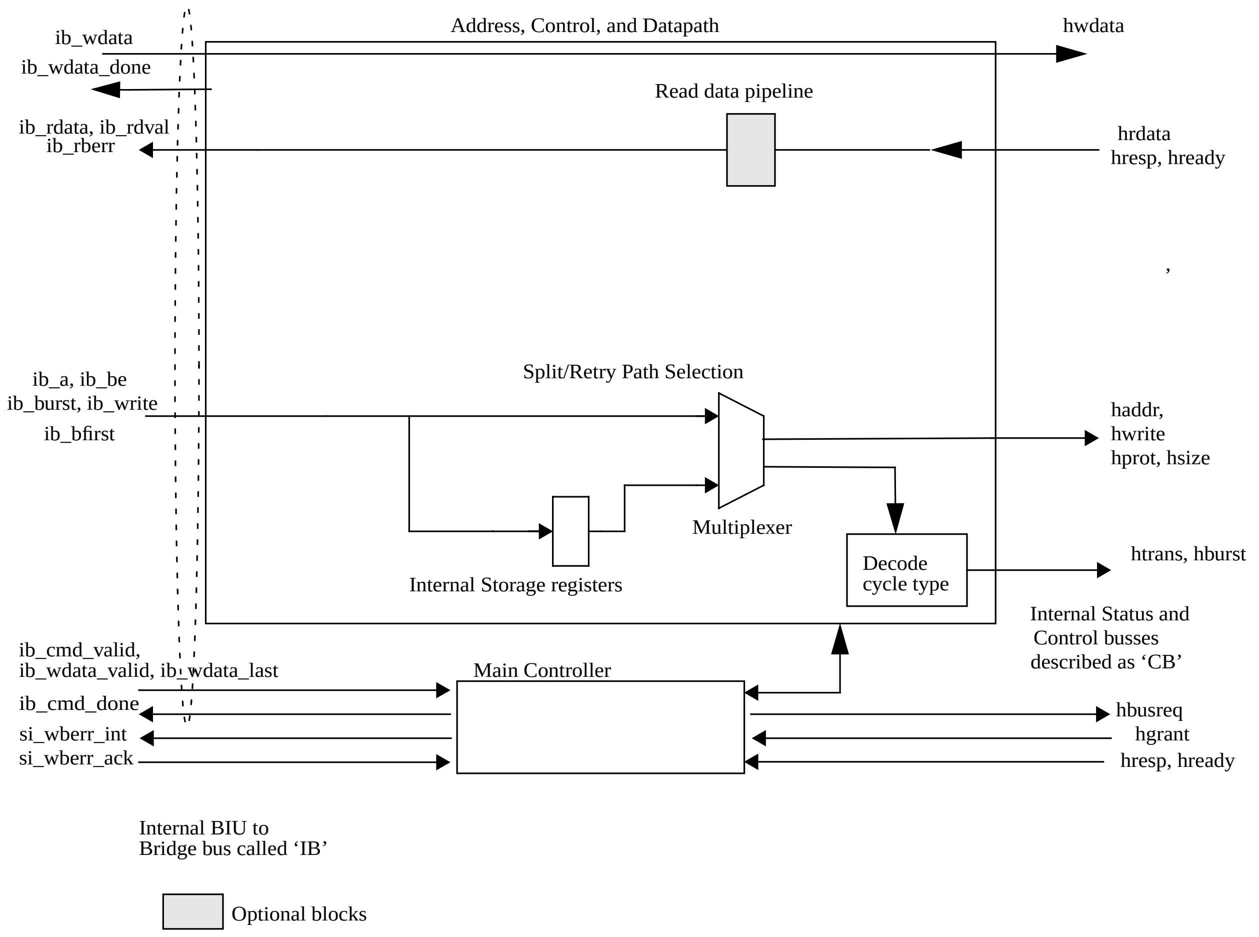
3.4.2 MAB_BIU2IB

This is basically a 4 deep FIFO that takes in commands on CMD and stores/forwards them onto IB. IB and CMD are in fact very little different. See Section 3.5 "Main Signals" for a description of the CMD and IB busses. Note that for fast start-up the address/command FIFO can be bypassed.

3.4.3 MAB_BRIDGE

Figure 3-3 shows a block diagram of the MBB2 BRIDGE module.

Figure 3-3 MBB2 Block Diagram



The design for the MAB_BRIDGE block is divided into two subblocks as shown in Figure 3-3.

3.4.3.1 The Address, Control Datapath unit:

The Address, Control and Datapath unit processes address and control informations passing from the IB to the AHB bus. For the most part IB values are passed straight to the AHB. The exceptions to this are the AHB htrans and hburst busses. The unit also has to store IB information in case an AHB slave responds with a SPLIT or RETRY.

Read data is passed directly from the AHB bus to the IB or, optionally, via a pipeline register. Write data goes straight from the IB to the AHB since it has already been registered in the MB_BIU2IB FIFO block.

3.4.3.2 The Main Controller unit:

The Main Controller unit provides control signals for the address/data path unit, primarily for AHB address phase pipelining and sends acknowledge signals back to the MAB_BIU2IB FIFOs when addresses and write data have been committed onto the AHB.

It also provides the appropriate handling for any error or SPLIT/RETRY responses from the AHB.

FInally, it handles the hbusreq/hgrant arbitration for the AHB. Note that in AHB-LITE mode the hgrant signal is hard-wired active at the top level of the MBB2.

3.5 Main Signals

3.5.1 External signal list

Table 3.1 External Interface Signals

Count	Name	Function	Direction
		CoreIF common signals	
1	cpu_clk	CoreIF bus clock.	I
1	cpu_reset	Reset signal related to CoreIF bus.	I
1	hphase	Phase indicator for hclk relative to cpu_clk	I
1	si_wberr_int	Asserted if write bus error occurred	O
1	si_sleep	Indicate if the MBB2 should enter sleep mode	I
1	si_wberr_ack	When asserted clears any write buffer error interrupts.	I
1	si_simple_be	Indicates to the core that simple byte enable should be used.	O
1	si_sblock	Indicates to the core that sequential addressing should be used.	O
1	si_endian	Indicate byte order of MBB2 ’0’ => Little endian ’1’ => Big endian	I
		EC™ - bus signals	
1	EB_ARdy	Ready to next address from core.	O
1	EB_AValid	Address valid indication from core.	I
1	EB_Instr	Indicate if a read operation is an instruction fetch	I
1	EB_Write	Asserted if an transaction is write direction	I
1	EB_Burst	Asserted if current transaction is part of a burst	I
1	EB_BFirst	Asserted if current transaction is first word in a burst	I
1	EB_BLast	Asserted if current transaction is last word in a burst	I
2	EB_Blen[1:0]	Indicates length of current burst (if any)	I
4 (8)	EB_BE[3:0] ([7:0])	Indicates active byte lanes in current transaction	I
35 (34)	EB_A[35:2] ([35:3])	Address information related to current transaction	I
32 (64)	EB_WData[31:0] ([63:0])	Write data bus	I
32 (64)	EB_RData[31:0] ([63:0])	Read data bus	O
1	EB_RdVal	Asserted if EB_RData holds valid data	O

Count	Name	Function	Direction
1	EB_WDRdy	Indicates if MBB2is ready to accept write data (always asserted)	O
1	EB_RBErr	Asserted together with EB_rdval if bus error occurred	O
1	EB_WBErr	Indicates write bus error (never asserted by current MBB2)	O
1	EB_EWBE	Indicates if write operation is stored inside MBB2	O
		OCP Bus Signals	
3	OC_MCmd[2:0]	Address phase command.	I
29/32	OC_MAddr[31:X]	Address bus. X = 3 for 24K™ cores prior to MR4 and = 0 otherwise.	I
4	OC_MTagID[3:0]	Tag for read command.	I
3	OC_MBurstLength[2:0]	Transaction burst length. Only values 1 and 4 will be generated by a MIPS™ 24K™ or 34K™core.	I
8	OC_MByteEn[7:0]	Read command byte enables.	I
3	OC_MBurstSeq[2:0]	Requested burst sequence. Will always be sequential.	I
4	OC_MReqInfo[3:0]	Additional information for the address phase of a cycle. Unused at present by the MBB2.	I
64	OC_MData[63:0]	Write data.	I
8	OC_MDatabyteEn[7:0]	Write Data byte enables.	I
1	OC_MDataValid	Write data and byte enable bus is valid.	I
1	OC_MDataLast	Last write data for current command.	I
4	OC_MDataTagID[3:0]	Unused by MBB2.	I
1	OC_MBurstSinglereq	A burst cycle has only a single address. Will always be ‘1’ from a MIPS™ 24K™ or 34K™ core.	I
1	OC_MBurstPprecise	A burst cycle must have exactly the number of data phases specified by OC_Mburstlength. Will always be ‘1’ from a MIPS™ 24K™ or 34K™ core.	I
1	OC_SCmdAccept	MBB2 has accepted the command currently on OC_Mcmd.	O
1	OC_SDataAccept	MBB2 has accepted the write data on OC_Mdata.	O
64	OC_SData[63:0]	Read data return to core.	O
4	OC_STagID[3:0]	Read data tag.	O
2	OC_Sresp[1:0]	Read response type.	O
1	OC_SrespLast	Last read data is on OC_Sdata.	O
8	oc_sdatainfo[7:0]	Unused by Core. Always driven to 0 by MBB2.	O
1	oc_serror	Unused by Core. Always driven to 0 by MBB2.	O
		SRAM signals	
1	IS_Read	Instruction side Read request.	O
1	IS_Write	Instruction side Write request.	O
1	IS_Sync	Instruction side Sync request.	O
1	IS_WbCtl	External write buffer flush (not used on current MBB2 version).	O
1	IS_Instr	Instruction side request is for an instruction fetch. Only relevant if the M4K™ is in unified SRAM mode.	O
30	IS_Addr	Instruction side address.	O
4	IS_BE	Instruction side byte enables for request,	O
1	IS_Abort	Instruction side request to abort transfer.	O

Count	Name	Function	Direction
32	IS_RData	Instruction side read response data.	I
1	IS_Error	Instruction side read error.	I
4	IS_RBE	Instruction side read response byte enables.	I
1	IS_Stall	Instruction side request to stall the transfer.	I
1	IS_AbortAck	Instruction side abort acknowledge	I
1	DS_Read	Data side Read request.	
1	DS_Write	Data side Write request.	
1	DS_Sync	Data side Sync request.	
1	DS_WbCtl	External write buffer flush (not used on current MBB2 version).	
30	DS_Addr	Dat side address.	O
4	DS_BE	Data side byte enables for request,	O
1	DS_Abort	Data side request to abort transfer.	O
32	DS_RData	Data side read response data.	O
1	DS_Error	Data side read error.	O
4	DS_RBE	Data side read response byte enables.	O
1	DS_Stall	Data side request to stall the transfer.	O
1	DS_AbortAck	Data side abort acknowledge	I
		AHB Bus signals	
1	hresetn	AHB bus reset.	I
32	haddr	Address bus.	O
2	htrans	Current transaction type.	O
1	hwrite	Current transaction is a write.	O
3	hsize	Size of current transaction - 8, 16, 32, 64 bits.	O
3	hburst	Burst length and type of current transaction.	O
4	hprot	Always driven to 0 by MBB2.	O
32	hwdata	Write data bus.	O
32	hrdata	Read data bus.	I
1	hready	Asserted when data phase ends on AHB bus.	I
2	hrest[1:0]	Asserted by slave to indicate response type.	I
1	hbusreq	Asserted if MBB2 requests the bus	O
1	hlock	Asserted if current transaction is locked. Always driven to 0 by the MBB2.	O
1	hgrant	Indicate if MBB2 is bus master	I
		Other signals	
		Scan test signals	
4	mab_scanin[x:0]	Scan chain input signals (number depend of scan length requirement)	I
4	mab_scanout[x:0]	Scan chain output signals (number depend of scan length requirement)	O
1	mab_scanenable	Enable scan mode of flip flops	I

3.5.2 Internal Busses

3.5.2.1 IB, CMD

This forms the connection between the mab_biu_<BUS>, the BIU FIFO module mab_biu2ib, and the main Bus-Bridge™ 2 RTL top level module mab_bridge. The purpose is to have a common set of signals, with an EC-like protocol, so that AHB portion of the RTL is identical in both the EC™ bus and OCP cases. Note that CMD and IB are very similar so that only one will be documented, ib_XXX , if the description of cmd_XXX is the same.

In the table below the Direction column indicates whether the signals is an input to (I) or and output from (O) the BIU block.

Table 3.2 IB signals

Name	Width	Description	Direction
ib_cmd	[1:0]	valid command on IB. Encoding is: bit 1: Command valid. bit 0: Pipelined command present.	O
ib_cmd_done	1	Command on IB has been sent out to the AHB. Asserting this signal shifts the next command from the FIFO in BIU2IB.	O
ib_a	[31:0]	Address bus	O
ib_bfirst	1	Address phase is first beat of a sequence. Note that this is asserted for a single 32-bit transfer request and the first beat of a dword request from the OCP.	O
ib_blast	1	Address phase is last beat of a sequence. Note similar to ib_bfirst so that ib_bfirst and ib_blast active at the same time imply a single transfer.	O
ib_write	1	Command is a write.	O
ib_burst	1	Command is part of a real burst request from the CPU. This is not set for dwords from and OCP CPU.	O
ib_size	[2:0]	Transfer size encoded as per AHB.	O
ib_blen	[1:0]	Burst length as per the EC™ specification. 0x1 => 4 words, 0x2 => 8 words.	O
ib_rdata	[31:0]	Read data.	I
ib_rdval	1	Read data is valid.	I
ib_rberr	1	Read error.	I
ib_threadid	[2:0]	OCP thread ID returned along with read data. N/A for EC™ cores.	I
ib_wdata_valid	1	Data on ib_wdata is valid.	O
ib_wdata_done	1	Write data in ib_wdata has been sent to AHB. Asserting this signal will shift the next entry out of the write data FIFO in BIU2IB.	I
ib_wdata	[31:0]	Write data.	O
cmd_active	1	From BIU to BIU2IB indicating that the CMD bus has a valid address phase beat,	I
cmd_stop	1	Indicates that the FIFO in BIU2IB is almost full. For EC™ cores this can allow for one overrun cycle.	I

Table 3.2 IB signals (Continued)

Name	Width	Description	Direction
cmd_a	[31:2]	CMD is a byte enable bus so doesn’t have the 2 lsbs.	
cmd_be	[3:0]	Byte enables for current CMD bus beat,	O

3.5.2.2 CB

This forms the connection between the BusBridge™ 2 module’s controller unit and the data path, address path, and bus request units. In the table below the Direction column indicates whether the signals is an input to (I) or and output from (O) the main controller module mab_cntrl.

Table 3.3 CB signals

Name	Description	Direction
ahb_aphase	The MBB2 should drive an address and command type on AHB.	O
ahb_dphase	The MBB2 is in the data phase of a transaction it has started.	O
ahb_retry_pending	A selected slave responded during an MBB2 data phase with RETRY or SPLIT.	O

3.5.2.3 Special signals

Most interface signals to the MBB2 module are made up of the CoreIF port and the AHB bus port. These signals conform to the specifications for these two interface standards. In addition to these pins, a number of extra pins exist, with the functions described below.

si_sleep

This is an input from the core. If configured for sleep mode the MBB2 will shutdown its internal clock in reponse to this when all pending commands have been completed all the way to the AHB.

si_subblock

An output that, when active, tells the core to use sub-block order for bursts. The MBB2 will always drive this to 0.

si_simplebe

An output that, when active, tells the core to use only ‘‘natural’’ byte enable patterns i.e those for single byes, half-words, words, and double words. The MBB2 will always drive this to 1.

si_wberr_int, si_wberr_ack

Signalling of write errors is performed by using of a general purpose interrupt pin on the core

If a write transaction on the AHB bus ends with an error response, then this is signalled to the core by asserting the si_wberr_int signal. This signal remains asserted until cleared by si_wberr_ack.

1. While si_wberr_ack is asserted, any new write error does not cause a new interrupt to the core.

2. While a write error is signalled by the si_wberr_int signal, the MBB2 module functions normally.

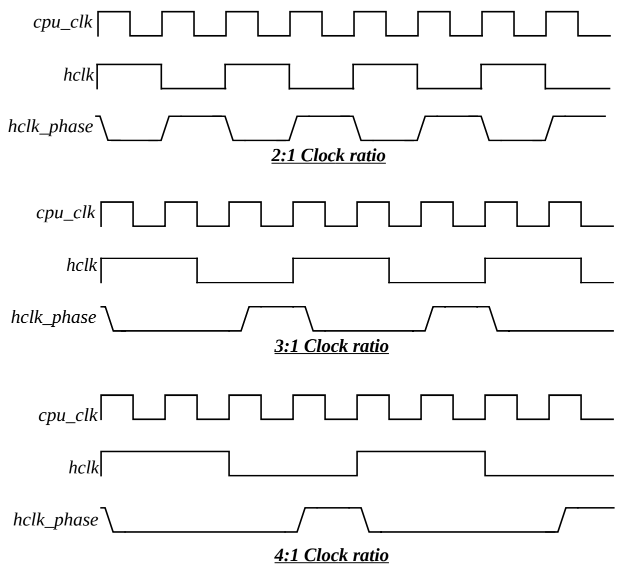
3. The si_wberr_ack signal may be driven by any general purpose pin in the system, so is treated as asynchronous to cpu_clk and is synchronized before use.

hclk_phase

This signal is used in systems where the CoreIF clock is running at 2, 3, 4, or 5 times the frequency of the AHB clock. The hclk_phase is used to indicate the rising cpu_clk edge that corresponds with a rising hclk edge, with the correct assertion of hclk_phase shown in Figure 3-4. In systems where the CoreIF clock and the AHB clock are running at the same frequency, the hclk_phase should be driven with a constant 1.

Note that hclk is not connected to the MBB2 module and is only shown here for reference.

Figure 3-4 Indication of hclk Phase



3.6 RTL Modules

The MBB2 implementation is organized in a number of Verilog modules, one file for each of the major blocks. These Verilog modules are held in files located in the design database under design/rtl/mab/.

Note that all modules take in gclk, gclk_en, greset, and mab_scanenable so these are omitted from the interface list except for the mab_glue module.

3.6.1 mab_top

Top level file. Includes the necessary sub block depending on configuration.

Functions:

- Instantiates one of the 2 BIU modules mab_biu_ocp or mab_biu_ec depending on the value of the configuration define MAB_CONFIG_OCP.
- Instantiates the actual top level bridge module mab_bridge and connects it to the selected BIU via the internal IB bus.
- Instantiates the clock and reset generator module

Interface:

- OCP or EC™ bus: All.
- AHB bus: All.
- External clock and reset signals.

3.6.2 mab_glue

Clock and reset control.

Functions:

- Generating the MBB2 internal clock gclk.
- Generating the MBB2 internal reset signal greset.
- Global clock gating.

Interface:

- Inputs: cpu_clk, cpu_reset, hresetn (from AHB), si_sleep (from core)hclk_phase.
- Outputs: gclk, greset.

3.6.3 mab_biu_BUS

Bus interface for CoreIF. BUS = ‘‘ocp’’,‘‘ec’’, ‘‘sram’’.

Functions:

- Receives commands from the core bus (OCP™, EC™, or the SRAM interface of the M4K™ CPU core) and translates them into an internal, IB/CMD, form to be passed to the mab_bridge module.

  Note that the IB address phase is very similar to the EC™ bus so that in the case where BUS = ec this module does very little work. For BUS = ocp, however, it must generate multiple address phase beats since the OCP interface of a 24K™ or 34K ™ core only provides a single address phase per transaction. For BUS = sram it, internally, converts the SRAM core interface to EC.

- Receives read responses from the mab_bridge module and converts them into the correct, OCP or EC, format. For CoreIF=OCP or EC64 the 32->64 width conversion is done via the mab_biu_read32to64 module instantiated here.

- Does write data splitting in the case where the CoreIF is 64-bit and the AHB is 32-bit. Also for CoreIF=OCP it aligns write data with the address information on CMD since for OCP write data may occur any time after a command phase has been accepted.

- Saves IB address phase information at the point where it is committed to the AHB (as indicated by ib_cmd_done). This is used by the mab_biu_read32to64 module to do the correct 32->64 bit packing for read data.

- Pipeline registers for OCP input signals if needed for high speed operation.

Interface:

- OCP or EC™ bus: All.

- CMD bus: All.

- IB bus: ib_rdata, ib_rdval, ib_rberr, ib_bfirst, ib_blast, ib_write, ib_a, ib_threadid, ib_cmd_done.

- si_endian.

3.6.4 mab_biu_read32to64

Read data path for 64 bit CPUs.

Functions:

- Takes in the 32-bit read data and read data response signals from the IB as well as the cycle type information (single, last, odd word) and outputs a 64 bit value on every other clock cycle for non-single word cycles. For single words the incoming IB read data is driven onto both halves of the 64 bit output bus.

- Also supplies read data valid information in OCP format.

Interface:

- IB: ib_rdata, ib_rdval, ib_rberr.

- Other: rdresp_ib_single, rdresp_ib_last, rdresp_ib_odd, cpu_rdata, cpu_rdresp, cpu_rdlast.

3.6.5 mab_biu2ib

Module that implements the MAB_BIU2IB FIFO function between the CMD and IB busses.

Functions:

- Address phase FIFO for values on the CMD bus.

- Write data FIFO for values on the CMD bus.

- Command FIFO bypass.

- Generates ib_cmd pipelining information from knowledge about the state of the FIFO and bypass logic.

Interface:

- Internal IB bus: All.
- Internal CMD bus: All.

3.6.6 mab_genfifo

This is ‘‘generic’’ RAM based FIFO configurable for width, depth (in powers of 2), and almost-full condition.

Functions:

- TBA.

Interface:

- TBA

3.6.7 mab_bridge

Top level wrapper for the main AHB protocol control and data paths

Functions:

- Instantiates and connects the mab_bridge_apath, mab_bridge_dpath, mab_bridge_cntrl modules.

Interface:

- Internal IB bus: All.
- AHB: All.

3.6.8 mab_bridge_adpath

AHB address phase path unit.

Functions:

- Connects the IB to the AHB address, size, burst mode, write, and transfer type (htrans) signals. This is done combinatorially every time the CB indicates that and AHB address is starting via the ahb_aphase signal.
- Stores IB address phase signals in the case where the AHB responds with SPLIT/RETRY.
- Connects the AHB read data and read responses to the IB read data path. This can be, configurably, pipelined if the MBB2 is running at a high core clock rate.
- Connects the IB write data path to the AHB write data bus. This is done directly.
- Records the occurrence of write errors on the AHB.

Interface:

- IB: ib_a, ib_write, ib_burst, ib_blen, ib_bfirst, ib_blast, ib_wdata, ib_rdata, ib_rdval, ib_rberr, ib_wdata_done.
- AHB: All.
- CB: All.
- Other: si_wberr_int, si_wberr_ack, si_endian.

3.6.9 mab_bridge_cntrl

Main controller unit.

Functions:

- AHB bus acquisition via hbusreq/hgrant.
- AHB protocol sequencing: A state machine handles AHB address phase start and address pipelining, passing address phase information to the data phase, SPLIT/RETRY responses.
- Flow control along the IB bus via ib_cmd_done, ib_wdata_done.

Interface:

- AHB: hresp, hready, hgrant, hbusreq.
- IB: ib_cmd, ib_cmd_done, ib_wdata_valid.
- CB: All.

3.7 Register Implementation

All instantiated registers in the design are clocked by gclk (derived from cpu_clk in the mab_glue module) and implemented through one of the generic flip-flop modules (listed in Section 3.7.1 "Generic modules" below) although it is possible for the user to replace them with customer specific versions. All conditional registers may be implemented with either a feedback multiplexer or with local clock gating, decided during configuration via MAB_CONFIG_GC.

For RTL simulation the supplied module mab_gcondclock.v will be used but for synthesis the user should replace this with a gate level netlist for the implementation technology. See step 5. of Section 5.1 "Synthesis Flow"

3.7.1 Generic modules

Generic modules for flip-flops and clock gating logic are stored under design/rtl/shared. The following generic modules are used in the design.

• mab_greg	Unconditional loaded register.
• mab_gcreg	Conditional loaded register. Implemented using either 'mab_gcreg_mux' or 'mab_gcreg_gclk' module depending on define settings.
• mab_gcreg_mux	Conditional loaded register implemented with a feedback multiplexer.

• mab_gcreg_gclk	Conditional loaded register implemented with gated clock. Clock gating will be made with ’mab_gcondclock’ or ’mab_gcondclock_n’ module depending on define settings.
• mab_gcondclock	Clock gating module with active high control input used for RTL level functional simulation.
• mab_gcondclock	Clock gating module with active high control input used for RTL level functional simulation.

- mab_clockandlatch_example An example synthesisable clock gating module with active high control input..

3.8 Design Configuration

The MBB2 design uses only 2 Verilog configuration header files.:

• mab_config.vh	User configuration. This is the only file where the user should make changes during configuration. Held in $SUBSYS_DIR/design/rtl/config.
• mab.vh	Contains derived defines for configuration as well as the AHB bus signal definitions. Held in $SUBSYS_DIR/design/rtl/shared..

Table 3.4 lists the possible configuration parameters in mab_config.vh

Table 3.4 RTL Configuration

Configuration control

Define name	Description
	Main Configuration
MAB_CONFIG_BUS_OCP	If set the CoreIF is OCP.
MAB_CONFIG_BUS_EC32	If set the CoreIF is the EC32™bus used by 4K™ and 4KE™ cores.
MAB_CONFIG_BUS_EC64	If set the CoreIF is the EC64™bus used by 5K™.
MAB_CONFIG_BUS_SRAM	If set the CoreIF is the SRAM bus used by M4K™ cores.
MAB_AHB_LOWLATENCY	If set the MBB will move a cycle appearing on the CoreIF directly to the AHB, assuming that the MBB2 is already the bus master.
MAB_CONFIG_OCP_PIPE_INPUTS	Only relevant for OCP. Set this to add a pipeline stage between the Core OCP bus and the logic in mab_biu_oc. Allows for operation with a very high speed core clock.
MAB_CONFIG_AHB_APHASE_REG	Disables the command FIFO bypass in mab_biu2ib so that AHB address phase signals are effectively registered.
MAB_CONFIG_AHB_RDATA_REG	Adds a pipeline register in the read data path from AHB -> IB.
MAB_CONFIG_CLOCKRATIO	Defines the CoreIF/AHB clock ratio. If not defined then the ratio is assumed 1:1.
MAB_CONFIG_SUPPORT_SLEEP	If defined clock enabling for support of sleep mode is included in code, otherwise asserting of si_sleep signal will have no effect on module.
AHB Configuration	
MAB_CONFIG_LITE	If a single master system is designed include this define. If not included an MBB2 module with support for multiple AHB masters is generated.
MAB_CONFIG_AHB64	If defined the AHB bus is assumed to have a 64-bit data path. (Not yet implemented).

Configuration control

Define name	Description
Clock gating control	
MAB_CONFIG_GATED_CLOCK	If defined conditional registers will be implemented with gated clock, otherwise conditional registers will be implemented with feedback multiplexers.
Scan width setting	
MAB_SCAN_WIDTH 4	With this define the width of the test scan ports of the design may be controlled.

3.9 References

[1] EC™ Interface Specification. MIPS Technologies. Document Number MD00052.

[2] Open Core Protocol Specification Release 2.1.

[3] AMBA AHB 2.0 Specification. ARM Limited. Document Number IHI 0011 A.

Chapter 4

Functional Simulation

The MIPS™ BusBridge™ 2 module (MBB2) is delivered with a Verilog testbench for functional simulation. In this testbench, MBB2 is instantiated along with a CPU core model and tested by running software on the core.

This chapter describes the functional simulation and software test cases, as well as briefly describes the testbench and provides a detailed list of the tests to be performed. References to file locations in the database are provided. The following sections make up this chapter:

4.1 The Verilog Testbench environment

This section describes the verilog testbenches used for verification. There are two types of testbenches, Directed and Random. The Directed testbench instantiates a model of core and uses directed tests written in C or assembly to create stimuli. The core model for the Directed tests can be RTL, VMC, or MIPSsim w/BFM. The Random testbench instantiates a Bus Functional Model (BFM) of a CPU core and uses a pre-generated stimuli of random bus cycles. Only the BFM is used in the Random tests.

4.1.1 Directed Testbench

This section briefly describes the Directed testbench used for MBB2 verification.

Figure 4-1 shows the Directed testbench used for MBB2 verification. The main components of the Directed testbench are:

1. Simple AHB Masters: The Simple AHB masters randomly assert their HBUSREQ signals to request AHB bus from Arbiter. This causes the AHB Arbiter to take away grants to other masters including the MIPS™ BusBridge™ 2 module.

2. Model of a MIPS® core. The models supported by the testbenches are the RTL, VMC and BFM with MIPSsim™.

3. Monitor Slave: Monitor Slave is a complex AHB slave used for various testbench features. The slave is described in detail later in this document.

4. Boot Rom AHB Slave: Boot Rom AHB Slave is a read only memory which is pre-loaded before start of simulation. The test case to be executed by the core is loaded into Boot Rom.

5. RAM: This is random access memory used for temporary storage and used to generate various AHB responses. The RAM is described in detail later in the document.

4.1.2 Random Testbench

This section briefly describes the Random testbench used for MBB2 verification.

Figure 4-2 shows the Random testbench used for MBB2 verification. The main components of the Random testbench are:

1. Simple AHB Masters: The Simple AHB masters randomly assert their HBUSREQ signals to request the AHB bus from the Arbiter. This causes the AHB Arbiter to take away grants to other masters including the MIPS™ BusBridge™ 2 module.

2. BFM of a MIPS® core: The BFM is used to generate transfers to the MIPS™ BusBridge™ 2 module. It is based on a script which randomly generates bus cycles for the BFM to execute. This has the effect of executing randomly generated bus cycles.

3. Random AHB Slave: This AHB slave generates the random AHB responses to AHB transfers. The slave can be configured to selectively not generate some of the AHB responses. The slave is described in detail later in the document.

Figure 4-1 Directed Testbench of the BusBridge™ 2 module

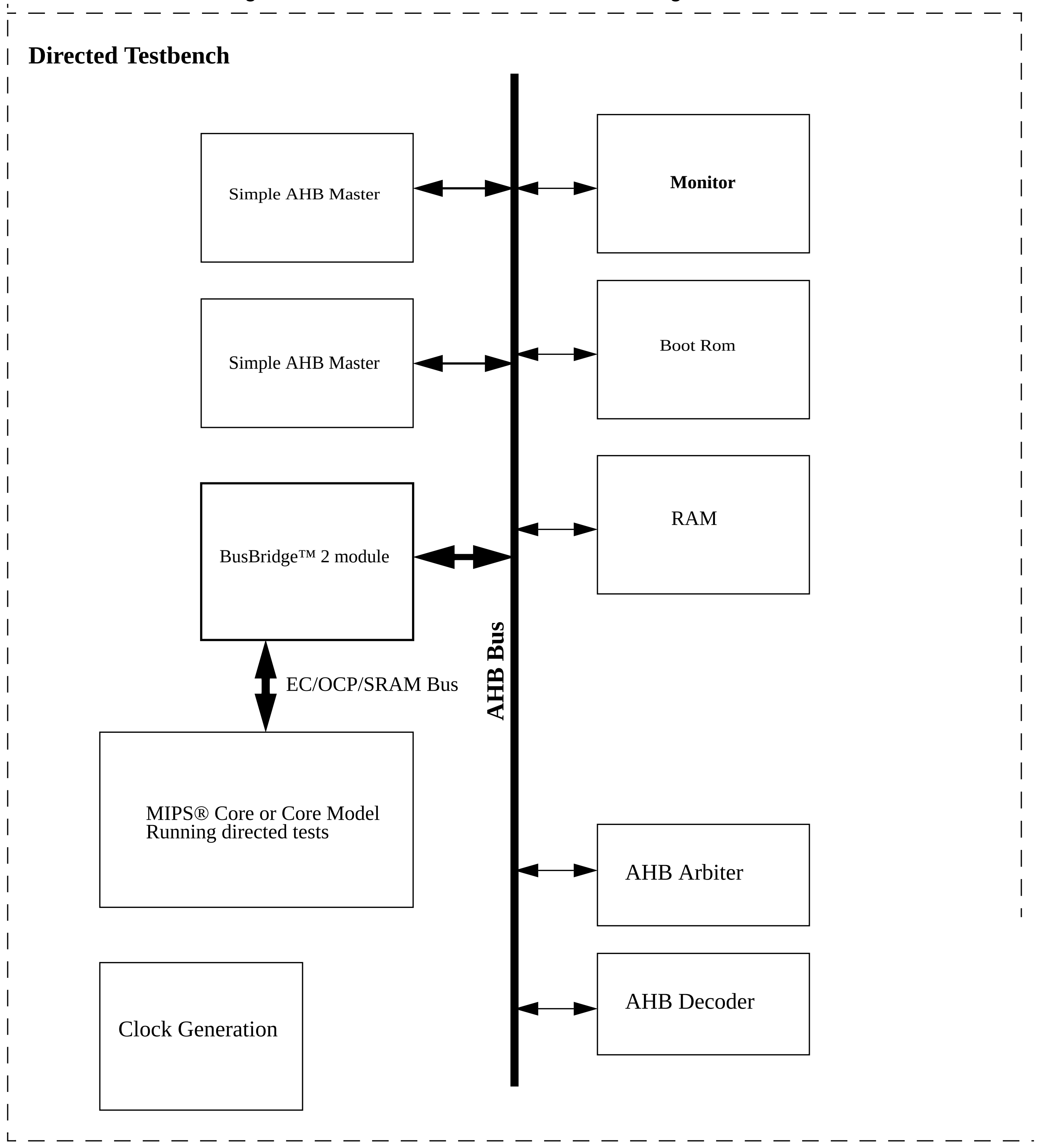
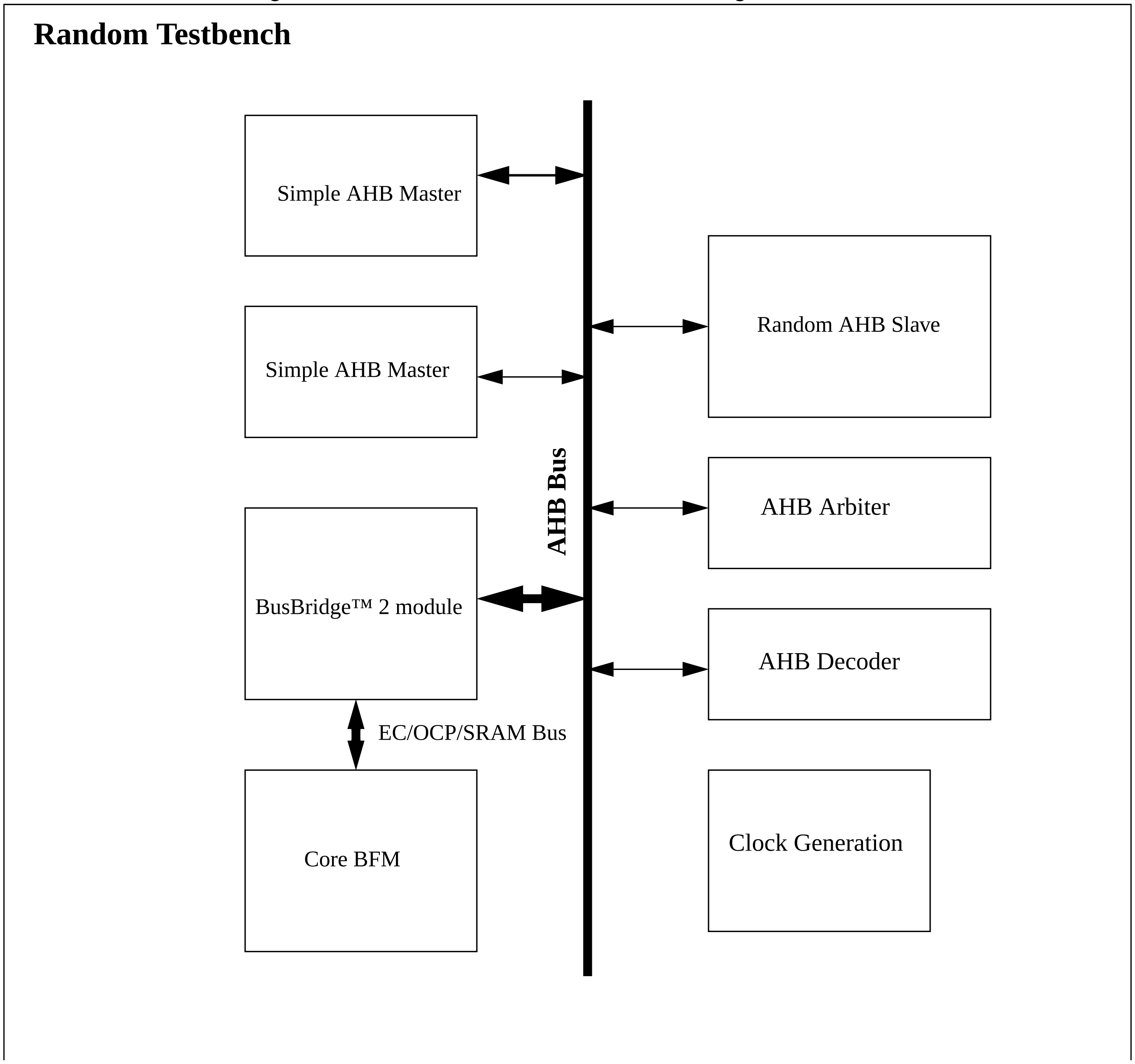


Figure 4-2 Random Testbench of the BusBridge™ 2 module



4.1.3 Testbench Components

This section describes the components of testbench in details. In the description that follows S1 means Slave1, S2 means Slave2 etc. Similarly M1 means Master1...

4.1.3.1 Clock/Reset Generation

This module is responsible for:

- Generation of the global MIPS™ BusBridge™ 2 module clock and hclk.

- Generation of the clock phase identifier.
- Generation of the MIPS™ BusBridge™ 2 module reset.

Files:

clkgen.v: This file contains all the Verilog code for the clock and reset generator.

4.1.3.2 Boot ROM (S1)

The boot ROM is read only memory containing the boot code. The boot code is pre-loaded into the memory from a hexadecimal file “testcase.asc” that is generated by the build_test script.

Files:

boot_slave.v: This file contains all the Verilog code for the boot ROM.

$SUBSYS_BUILD_DIR/tb/testcase.asc: This file contains the boot code.

4.1.3.3 Monitor (S2)

The monitor is a complex device that handles many tasks. At boot, a hexadecimal file “testcase_cfg.asc” is pre-loaded into the monitor controlling the endianness and clearing of caches. The “testcase_cfg.asc” file is then generated by the build_test script from specifications set up in the software Makefile.

These are some of the other features of the monitor:

- Returning OK or error message by writing to a register (MON_HAPPY, MON_SAD).
- Allowing signal tracing to be turned off or on by writing to a register (MON_VCDTRACE), if verbose level (MON_VERBOSE) is set to V0 or V2 (VCD dump enabled).
- Terminating character IO. Writing to a register (MON_PUTCH) basically echoes the character written to the workstation screen, if verbose level (MON_VERBOSE) is set to V0 or V1 (Print enabled). This allows usage of printf statements in “C” code.
- Setting RAM slave split or retry responses to none or random by writing to a register (MON_SET_SPLIT, MON_SET_RETRTY).
- Inserting none or a random number of wait states in the RAM slave data phase response by writing to a register (MON_SET_WS).
- Resetting a write error interrupt by writing to a register (MON_ACK).
- The monitor would also be modified to generate deterministic sequences of Split and retry responses.
- Changing verbose level by writing to a register (MON_VERBOSE). The verbose level is set to “V0” running a single test, and the verbose level is set to V1 running a full regression suite. The following verbose levels are defined:
  - V0: Print enabled, VCD dump enabled
  - V1: Print enabled, VCD dump disabled
  - V2: Print disabled, VCD dump enabled

- V3: Print disabled, VCD dump disabled

Files:

- monitor.v: This file contains all Verilog code for the monitor.
- $SUBSYS_SW_DIR/include/monitor.h: This file contains "C" defines for all the available registers in the monitor.
- $SUBSYS_BUILD_DIR/tb/testcase_cfg.asc: This file contains the boot configuration.

4.1.3.4 RAM (S3)

The RAM is a random access memory used for data (stack) and/or the application. By setting the variable COPY = d (copy data) in the software Makefile, the data (stack) is copied to the RAM; however, by setting the variable COPY = ad (copy application and data) in the software Makefile, both the boot code and data (stack) are copied to the RAM.

Files:

ram_slave.v: This file contains all the Verilog code for the RAM.

4.1.3.5 Address Decoder

The Address Decoder contains the address decoder logic, read data and response multiplexer, and the default slave.

For the directed testbench the address decoder logic maps the HADDR address as shown in Table 4.1.

Table 4.1 The Memory Map used for Directed testbench.

Address range	Size	Unit	Remarks
0x0000.0000 -> 0x0001.ffff	128 KByte	RAM (S3)	Random access memory used for data (stack) and/ or boot code.
0x0002.0000 -> 0x1f01.ffff		Default slave (S4)	Free memory space.
0x1f02.0000 -> 0x1f02.ffff	64 KByte	Monitor (S2)	The monitor is used to set up the test environment and display to screen from inside assembler and "C" programs.
0x1f03.0000 -> 0x1fbf.ffff		Default slave (S4)	Free memory space.
0x1fc0.0000 -> 0x1fc1.ffff	128 KByte	Boot ROM (S1)	Read only memory containing the boot code.
0x1fc2.0000 -> 0xffff.ffff		Default slave (S4)	Free memory space.

For the Random testbench the address decoder maps the full 32-bit address space to random AHB Slave.

The default slave is used to generate responses for transfers addressed outside any other slave (S1 to S3).

The default provides the following responses:

- ERROR response for NONSEQUENTIAL or SEQUENTIAL transfers
- OKAY response for IDLE or BUSY transfers.

Files:

decoder_top.v: This file contains all Verilog code for the address decoder logic, read data and response multiplexer, and the default slave.

4.1.3.6 Arbiter

The arbiter is controlling which master has access to the address and control bus as well as the write data bus. The address and control multiplexer, the write data multiplexer, and the dummy master is part of the arbiter.

The testbench has three masters - the MIPS™ BusBridge™ 2 module (M1) and two instantiations of the Simple Master (M2 and M3).

Each master has a request signal, which is asserted when the master wants the bus granted. The arbiter grants the master access to the bus by asserting a grant signal to the master.

It is possible to set up the arbiter to run in three different arbitration scheme modes:

- priority - high priority master is granting access before a low priority master
- round robin - the next (requesting) master is granting access, M1, M2, M3, M1, etc.
- random - the grant to MIPS™ BusBridge™ 2 module is taken away at random. In order to use it the simulator runtime option **"+randomGrants"** must be specified.

It is also possible to set up which master is the default master.

Both setups are done in the software makefile by setting the PRIORITY and DEFAULT variables described in Table 4.9.

The arbiter also controls split and locked transfers:

- If the arbiter receives a split response, then the master attempting the transfer is not granted access to the bus until the slave indicates it is ready to complete the transfer. The dummy master must be granted the bus if all other masters are waiting for split transfers to complete.

Files:

arbiter_top.v: This file contains all Verilog code for the arbiter logic, write data multiplexer, address & control multiplexer, and the dummy master.

4.1.3.7 Random AHB Slave

Random AHB Slave is a simple AHB slave which drives various types of random responses in response to AHB transactions. The slave generates all AHB responses and adds random waits before a response is issued. The slave can turn off RETRY, ERROR and SPLIT responses when simulations are run with appropriate runtime simulator options.

For read transfers slave drives the address of read transfer as the read data. For write transfer slave expects the data which is same as address of transfer.

Run time options:

Following run time simulator options can be used with random AHB slave:

1. **+disableSplit**: If specified, the slave does not drive SPLIT response. By default slave generates SPLIT responses.

2. **+disableError**: If specified the slave does not drive ERROR response. By default slave generates ERROR responses.

3. **+disableRetry**: If specified the slave does not drive RETRY response.By default slave generates RETRY responses.

The slave also implements a timeout counter to prevent runaway simulation.

4.1.3.8 Testbench

This is the top-level of the testbench, and instantiates the MIPS™ BusBridge™ 2 module and all surrounding modules, including the core. Also, all unused signals to the core are terminated with a constant high or low.

Files:

- testbench_mab.v

4.1.3.9 The CPU Model

The CPU core executes the test programs. To speed up simulation, it is recommended that the generic configuration of the CPU listed in Table 4.2. is used.

Table 4.2 Generic CPU Configuration in Top-Level Testbench

Option	Configuration
Gated clock.	Turn gated clocks off.
MMU	No TLB. Use fixed map translations. None of the tests uses the MMU, so unused functionality to slow down simulation is not wanted.
Register file	Use register-based.
EJTAG	Disable as much as possible to reduce overhead in simulations
Integrated BIST.	Disable.
Cache RAMs	Set to two-way associative. 4K per way. Same for both I and D.

The generic configuration has the advantage that all logic not directly related to the MIPS™ BusBridge™ 2 module is disabled. Notice that the sample software does not support the use of TLB.

4.1.3.10 Cache Clearance

With the generic configuration of all the supported core, the caches are cleared by the testbench to speed up the simulations.

4.2 Testbench Scripts

This section describes the scripts used to configure the testbenches, build tests and run tests.

4.2.1 The "configure_mab.pl" script

The MIPS™ BusBridge™ 2 needs to be configured before use. The Verilog include file ${SUBSYS_RTL_DIR}/config/mab_config.vh specifies this configuration. While one could edit this file by hand, it is recommended that the script “configure_mab.pl” is used instead. This script accepts options on the command line, reads a default configuration file, and generates the actual mab_config.vh used. Each time a change is desired in the MBB2, the configure_mab.pl should be used. A list of the command line options are shown in the table below.

The file ${SUBSYS_RTL_DIR}/config/mab_config.vh is described in detail in Section 3.8 “Design Configuration”

Table 4.3 Command line options for configure_mab.pl

Command Line Option	Description	Default Value
--clock_ratio=<yes | no>	Specifies the whether clock ratio support is enabled.	yes
--reg_ahb_outputs=<yes | no>	Configures the RTL to register the AHB outputs. The possible options are yes and no.	no
--reg_ahb_inputs=<yes | no>	Configures the RTL to register the AHB inputs. The possible options are yes and no.	no
--enable_clock_gating=<yes | no>	Configures the RTL to use clock gating. The possible options are yes and no.	no
--pipe_ocp_inputs=<yes | no>	Configures the RTL to pipe the OCP bus inputs into the BusBridge™ 2 module. The possible options are yes and no.	no
--support_sleep=<yes | no>	Configures the RTL such that the BusBridge™ 2 module supports core initiated sleep. The possible options are yes and no.	yes
--ahb_lowlat=<yes| no>	In the case where MBB2 already owns the AHB bus this allows a cycle starting in the CPU system interface to appear on the AHB bus with no delay.	no

4.2.2 The “configure_mab_tb.pl” script

The script “run_test” is used to build and run tests. There currently 18 command line options to run_test, and these specify fully the simulator, endianness, testbench type, etc. In order to save the user from having to remember and specify such a large set of options each time, testbench configuration files can be used. Like the configure_mab.pl script described in the last section, configure_mab_tb.pl script can be used to create configuration files for testbench from defaults and command line options. But unlike the single configuration for MBB2, there can be many testbench configuration files. These configuration files contain subset of command line options of the run_test script.

configure_mab_tb.pl puts the config files created in the directory ${SUBSYS_VERIF_DIR}/tb_config. These are given the suffix.cfg. The command line options for this script are described in the table below. If an option is specified in configuration and on the command line option of run_test script, the command line option overrides the value in configuration.

Table 4.4 Command line options for configure_mab_tb.pl script

Command Line Option	Description	Default Values
--tb-config | --o=<config_name>	Name of configuration file to create. All the configuration files are written to ${SUBSYS_VERIF_DIR}/tb_config directory. A suffix of .cfg is added to the name of configuration specified with this option.	mab_tb_config
--simulator | --sim=<VCS|MTI|NC|XL>	Specifies the simulator to be used for simulations. Possible values are: VCS, NC, MTI and XL.	VCS unless specified via the environment variable MAB_SIMTYPE
--core-model=<RTL|NET|VMC|BFM>	Specifies the model of the core to use. Possible values are RTL, NET (gate level netlist), BFM or VMC.	RTL
--tb-type=<DIRECTED|RANDOM>	Specifies the type of verilog testbench to use. Possible values are Directed or Random.	DIRECTED
--tb-top=<filename>	Specifies the top level testbench Verilog file.	verif/tb/testbench_mab.v
--ahb-type | --ahb=<FULL|LITE>	Specifies the AHB bus to instantiate in the testbench. The possible options FULL or LITE. At present the testbench only support full AHB bus.	FULL
--use-vcs-vm | --vcs-cm	Collect code coverage statistics if VCS is used as simulator.	Code coverage is not collected by default.
--mab-model=<RTL|NET>	If set to NET the simulation will use a gate level Verilog netlist for MBB2. The source of this netlist is specified by the --gate-sim-filelist option.	RTL
--gate-sim-filelist=<vc_file_name>	Specifies a verilog command(vc) file for testbench compilation. This vc file specify the netlist file and libraries used by netlist. This vc file is in addition to vc file used by run_test.	None.
--[no]24k-ocp-2-1-complaint	Specifies that the 24K™core is OCP 2.1 compliant. This option cannot be specified on the command line for run_test and therefore configurations must be used.	24K™core is assumed to be OCP2.1 compliant.
--[no]4k-top-is-wrapper	Specifies that the 4KE™core is revision 3_2_0 or later. This option cannot be specified on the command line for run_test and therefore configurations must be used.	4KE™core is assumed to be revision 3_2_0 or later.

Table 4.4 Command line options for configure_mab_tb.pl script

Command Line Option	Description	Default Values
--[no]cache-support-init	Specifies that caches instantiated in the testbench supports hardware init. This option cannot be specified on the command line for run_test and therefore configurations must be used.	No Cache init support is assumed.

4.2.3 The "run_test" script

run_test is the primary script which is used to build the testbenches, the tests, and then run the tests. The invocation is:

run_test <options> <test_name>

Internally, run_test first runs the two scripts build_tb and build_test. To build the testbench itself, run_test passes a --build flag to “build_tb”. By default the testbench is built in the directory

$SUBSYS_BUILD_DIR/tb_<m34k | m24k | m4k | mm4k>

If --build-name=<othername> is specified on the run_test command line this becomes:

$SUBSYS_BUILD_DIR/tb_<m34k | m24k | m4k | mm4k>_othername

Note: The output of the testbench compile phase is saved in the testbench directory as ‘‘compile.log’’.

To compile and build the test, <test_name> is passed to “build_test”. This will compile the <test_name> software and copy the results to the testbench directory

“run_test” will then run the simulation.

The Table Table 4.5 shows the command line options for run_test.

Table 4.5 Command line options for run_test

Command Line Option	Description	Default Value
`--simopts=<simulator_options>`	Pass <simulator_options> to simulator as run time options.	No additional runtime options are passed to simulator.
`--vcs_cm`	Build with VCS code coverage switched on. Running testbench built with this option would collect code coverage statistics.	Testbench is not build with VCS code coverage switch on.
`--log=<log_file_name>`	Create a log file with name log_file_name.	Log file with name <test_name>__<act__><model__><vcs_cm__><simopts>.log is created.
`--build`	If --clean is *not* on the command line then passing this option builds the testbench. If --clean is on the command line then passing --build will result in the contents of the testbench dir. being removed.	By default testbench is not built.

Table 4.5 Command line options for run_test (Continued)

Command Line Option	Description	Default Value
`--build-name=<build_name>`	Specifies the suffix for the build directory.	-

`--config-file-name |`
`--config=<config_name>`
	Specifies the testbench configuration file. The actual file used will be config_name.cfg
	in
	$SUBSYS_VERIF_DIR/tb_config
		-
		Note: that if no config file is specified then run_test uses an internal default testbench setup.

`--interactive | --I`	Build testbench and run it interactively.	By default the testbench is not run interactively.
`--simulator | --sim=<VCS|MTI|XL|NC>`	Specifies the simulator to use for running simulation. The possible options are VCS, NC, MTI or XL.	VCS unless specified by the environment variable MAB_SIMTYPE
`--core-model=<RTL|NET|VMC|BFM>`	Specifies the model of core to use. Possible options are rtl, net (gate level netlist),bfm or vmc.	rtl is the default model used.
`--tb-type=<DIRECTED|RANDOM>`	Specifies the testbench to use. Possible options are rtl or random.	By default RTL testbench is used.
`--tb-top=<filename>`	Specifies the top level testbench Verilog file.	verif/tb/testbench_mab.v
`--ahb-type=<FULL|LITE>`	Specifies the configuration of AHB bus. The possible options are lite or full.	By default AHB Bus is configured as Full AHB.
`--endianness | --endian=<LITTLE|BIG>`	Specifies the endianness to be used by testbench and tests. Possible options are little or big.	By default endianness is set to little endian.
`--master_priority=<LOW|HIGH|DEFAULT>`	Specifies the priority of the Bus-Bridge™ 2 module as a master on AHB bus. This option is ignored when bus is configured as AHB lite bus. Possible options are LOW, HIGH or DEFAULT.	By default master is configured as a HIGH priority master. This generates maximum coverage from testing point of view.
`--clean`	Cleans the testbench and/or the test directories. The testbench directory is cleaned if --build is on the command line. If a test is specified on the command line then the corresponding test directory is cleaned.	
`--q`	Run quietly	
`--n`	Build the testbench, build the test, but stop short of actually invoking the test.	
`--mab-model=<NET|RTL>`	If set to NET then a gate level netlist of MBB2 will be used for simulation. This netlist is specified by the --gate-sim-filelist option.	RTL

Table 4.5 Command line options for run_test (Continued)

Command Line Option	Description	Default Value
`--gate-sim-filelist=<vc_file>`	Specifies a verilog command(vc) file for testbench compilation. This vc file specify the netlist file and libraries used by netlist.	-
`--clock-ratio=<RATIO>`	Specifies the clock ratio to be used for simulation. Supported values for RATIO are: 1:1, 2:1, 3:1, 4:1, 5:1 and 6:1	1:1
`<test_name>`	The test name to run should be the last command line option. If no test name is present only the testbench is built if asked for.	none.

4.2.4 The “run_regression” script

The run_regression script can be used to run a regression using directed tests in verilog testbench.Table 4.6 below describes the command line options for run_regression script.

Table 4.6 Command line options for run_regression

Command Line Options	Description	Default Value
--config=<config_name>	Specifies the testbench configuration file. The actual file used will be config_name.cfg in $SUBSYS_VERIF_DIR/tb_conf	default.
--dirname=<dirname>	The sub-directory of $SUBSYS_BUILD_DIR/log used to hold the log files from the regression tests. This name is also passed to “run_test” as the build-name so that the testbench build/run directory is consistent with the directory in $SUBSYS_BUILD_DIR/log	defaultname
--clock-ratio=<RATIO>	Specifies the clock ratio to use for running regression. run_regression supports running tests in ratios 1:1, 2:1, 3:1, 4:1, 5:1 or 6:1.	1:1
--clean	When a test is run that has run before (by name), the associated log directory is always cleaned. If a test fails, the testbench build/run directory is left alone. If a test passes, the testbench build/run directory is cleaned and only the associated log directory remains. Setting the --clean will force a clean of testbench build/run directory even it the test failed.	not on by default
--q	Run quietly	not on by default

Table 4.6 Command line options for run_regression (Continued)

Command Line Options	Description	Default Value
<test>	The test to run. If this argument is not specified then full regression for the specified clock ratio is run.	none.

4.3 Using the Verilog Testbench

This section describes the rules and methods for building a test and how to execute the test. All input stimuli for a given test resides in a single directory called a test directory. The directory structure for tests is organized as shown in Table 4.7.

Table 4.7 Test Directory Structure

Test root	block_name	test_name
$SUBSYS_SW_DIR/tests	/samples	/hello_world
		/walking_one
	/mab	/cache_01.01
		/interrupt_01.01
		...
	/transactions	/read_01.01
		/read_01.02
		...
		/write_01.01
		/write_01.02
		...

In the samples directory are some tests that can be used as a starting point for new tests. For example, in the directory “regression/tests/samples/hello_world”, all files for the hello_world program are found.

The source files that reside in a test directory like hello_world are listed in Table 4.8.

Table 4.8 Source Files in a Test Directory

Name	Description
makefile	This file controls various properties for the test.
<name>.c	.C files are C source files. At least one source file must be present with a main function.
<name>.S	.S files are assembler source files. Parts of a test can be written in assembler if required. If no main function is declared in a C source file, then a label called main must be found in an assembler source file.

The makefile in the test directory controls the various attributes under which the test is performed. Table 4.9 lists all the options. These are all dynamic options, thus the same testbench build can run with all the combinations.

Table 4.9 Makefile Controllable Options

Option	Variable name	Values	Description
Endianness	ENDIAN	EL or EB	This variable controls the endianness under which the CPU will execute the test program. When the testbench starts, the SI_Endian pin of the CPU core is set according to the variable. The variable is also used by the compiler to compile the test program for the right endianness. EL: Little endian. EB: Big Endian.
Copy application/data	COPY	<blank>, ad or d	If required, then it is possible to copy data or application + data to the RAM during the boot. This can be used to simulate the situation where boot is done from a slow device or a read-only device. <blank>: No copy takes place (invalid value except if the customer replaces the boot ROM with a read/writable boot slave). ad: Both application and data is copied to the RAM d: Only data are copied to the RAM.
Cached or uncached operation	CACHE	0 or 1	This variable controls how the CPU executes the code (cached or uncached). There are pros and cons. Cached operations will generate bursts. Uncached operations on the other hand will create more activity. 0: Run program uncached. 1: Run program cached.
Location of exception vector at boot	EXCEP_VECTOR_AT_BOOT	0 or 1	This variable determines where the exception vector is placed after boot. The option is only used by the utility function excep_install_exc_in_ram which can install an exception handler specified by the test program. If the boot ROM is used, then the value must be 1 in order to install a new exception handler (the boot ROM slave is read-only). 0: Exception vector placed at 0x80000000 (i.e. RAM) 1: Exception vector placed at 0xbfc00200 (i.e. in boot area)

Table 4.9 Makefile Controllable Options (Continued)

Option	Variable name	Values	Description
Clock configuration	CLKCFG	NNN (each letter is either 1 or 0)	This variable controls the MBB2 to HCLK ratio. The following values are valid: 001: Clock ratio 1:1 010: Clock ratio 2:1 011: Clock ratio 3:1 100: Clock ratio 4:1 101: Clock ratio 5:1 110: Clock ratio 6:1
Files shared with other tests	SHARED_FILES		Example: Assume test.c from some other test is reused for this test. This option is useful if the same test must be run (e.g. cached/uncached, little endian/big endian in a regression test). This option ensures that the source code is exactly the same in all the tests. SHARED_FILES = ../../some_other_dir/test.c
Configure if the testbench will clear the core caches or not	CLEARCACHE	0 or 1	CLEARCACHE = 1 (or not present) causes the testbench to fill the caches with 0's before simulation begins (this option only applies to the 5K™ core). The caches can then be used without software, saving a lot of simulation cycles. If software is run uncached, then the caches do not need to be cleared by software. Set CLEARCACHE to 1 in order to save simulation cycles. If CLEARCACHE = 0, then the caches are full of X's at simulation start. The boot code automatically clears all present caches. CLEARCACHE must always be 0 if a 4K™ core is instantiated and the software runs cached.
Multi master priority	PRIORITY	xxx_yyy_zzz (each letter is either 1 or 0)	This variable controls the multi master arbiter priority scheme. The first 3 bits defines the high priority master, the next 3 bits defines the middle priority master, and the last 3 bits defines the low priority master. PRIORITY = 001_011_010 means: Master M1 is high priority, master M3 is middle priority, and master M2 is low priority. If PRIORITY is set to 000_000_000, then a round-robin priority scheme is selected

Table 4.9 Makefile Controllable Options (Continued)

Option	Variable name	Values	Description
Default master	DEFAULT	NNN (each letter is either 1 or 0)	This variable controls which master is handled by the arbiter as the default master. DEFAULT = 010 means that default master is set to M2.

4.3.1 Building a Test

The test is built automatically by run_test when --build is specified on the command line option of run_test. The following run_test command can be used:

```
run_test <other_run_test_options> <block_name>/<test_name>
```

See Table 4.10 for a definition of <block_name> and <test_name>. Check warnings and errors from the build_test script. The script compiles all files, links them, and converts the result to input files for the testbench. The testbench files are finally copied to the location where the testbench is executed ($SUBSYS_BUILD_DIR/tb). The linker will then link with a number of initialization routines which perform the following functions:

- Sets up the core registers.
- Defines a simple exception handler.
- Clears the caches if applicable.
- Changes the code to run cached if applicable.
- If required, copies data and application to the RAM slave.
- Sets up the stack.
- Calls the sub-function main.

After building the test, the script run_test runs it.

4.3.2 Rules for Tests on the Verilog Testbench

This section describes the conventions and rules for the tests that are performed on the top-level testbench. There are conventions for the valid block_name(s) and test_name(s) shown in Table 4.10. These conventions are listed in Table 4.10.

Table 4.10 Valid Block_Name and Test_Name

Valid block_name	Valid test_name	Description
samples	Any text string	All tests in the samples directory are intended as a starting point for new tests. The test_name will indicate what the test is doing.

Table 4.10 Valid Block_Name and Test_Name (Continued)

Valid block_name	Valid test_name	Description
mab	<any text string>_xx.yy	All tests must have a text name followed by numbers in the format xx.yy. The xx numbers may be used to indicate tests that belong to the same group of tests, while the yy numbers are used to label a specific test within the group. The text string will indicate what the test is doing. This document lists all the tests that will be performed on the MIPS™ BusBridge™ 2 module.
transactions	<any text string>_xx.yy	All tests must have a text name followed by numbers in the format xx.yy. The xx numbers may be used to indicate tests that belong to the same group of tests, while the yy numbers are used to label a specific test within the group. The text string will indicate what the test is doing. This document lists all the tests that will be performed on the MIPS™ BusBridge™ 2 module.

Any user is free to create additional tests in their local copy of the database with other names, but the rules in Table 4.10 must be followed for any test in the repository.

In addition the following rules apply to all tests:

- All tests *must* terminate by writing to the MON_HAPPY register in the monitor.
- A test that passes must write the value 0 using the “C” command: REG32(MON_HAPPY) = 0x0.
- A test that fails can write any value different than 0 to the register (e.g. REG32(MON_HAPPY) = 0x33); but the value in case of an error can be freely selected to give additional information about the error.

4.3.3 Running the All Tests

The run_regression script described in Section 4.2.4 “The “run_regression” script” can be used to run all the tests.

4.4 Tests Performed with the Verilog Testbench

This section describes all tests performed using the top-level testbench to simulate the MIPS™ BusBridge™ 2 module. All tests comply with the rules defined in the previous section.

4.4.1 Test Description Format

For each group of tests the following information is given:

- The purpose of the tests.
- The options set in the Makefile in the test directory (i.e. if the test must run in big or little endian).
- A description of the test.

The information can cover more than one test if the only difference between the tests are differences in the Makefile. The test description is shown for the blocks samples, mab and transactions. The “samples” Block

4.4.1.1 hello_world (The Famous Hello World Example)

Purpose: To have a simple program to get started.

Makefile options

Makefile Options

	hello_world
ENDIAN	EL
COPY	d
CACHE	0
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init
SHARED_FILES	<empty>
PRIORITY	000_000_000
DEFAULT	010

Description:

The program is written in C and gives some examples of useful monitor commands. This program also uses printf to show information gathered from the monitor and puts (print string) to print “Message: Hello World”. The program ends with the MON_HAPPY command to stop simulation.

4.4.1.2 walking_one (A Light Version of cache_01.01)

Purpose: To have a little program that makes cached and uncached accesses. The uncached accesses are different sizes.

Makefile options:

Makefile Options

	walking_one
ENDIAN	EL
COPY	d
CACHE	1
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init
SHARED_FILES	<empty>
PRIORITY	001_011_010
DEFAULT	001

Description:

The program is written in C and uses a number of tables defined in order to generate 8, 16, 32, and if applicable, 64-bit accesses. The bit pattern of the data is created in such a way that only one bit is set or not set (walking 1 or walking 0). All single bits in an access size are covered (e.g. 8-bit accesses are covered with the data numbers 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40 and 0x80).

The tests create all supported access types on the bus (i.e. single fetch, burst fetch, burst read, burst write, single read and single write).

4.4.2 The “mab” Block

4.4.2.1 cache_01.01 and cache_01.02 (Comprehensive Test)

Purpose: To verify that a number of different transactions can take place in both little and big endian.

Makefile options:

Makefile Options

	cache_01.01	cache_01.02
ENDIAN	EL	EB
COPY	d	
CACHE	1	
EXCEP_VECTOR_AT_BOOT	0	
CLKCFG	001	
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init	
SHARED_FILES	<empty>	./../cache_01.01/test.c
PRIORITY	001_011_010	
DEFAULT	001	

Description:

The program is written in C and shared between the tests. The test consists of the following two parts, each repeated 100 times:

1. The first part of the test is writing 255 “random” words to an array, which are then placed on the stack. The data cache is then invalidated and the words are read back in order to compare them with what was written. If an error is detected, then the simulation is stopped by the monitor slave. In case of no errors, the other part of the test is executed.

2. The other test uses a number of tables defined in order to generate 8, 16, 32, and if applicable, 64-bit accesses. The bit pattern of the data is created in such a way that only one bit is set or not set (walking 1 or walking 0). All single bits in an access size are covered (e.g. 8-bit accesses are covered with the data numbers 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40 and 0x80).

The tests create all supported access types on the bus (i.e. single fetch, burst fetch, burst read, burst write, single read and single write).

4.4.2.2 endian_01.01 and endian_01.02 (Software Test of Endianness)

Purpose: To indicate that endianness is handled correctly throughout the MIPS™ BusBridge™ 2 module in both little and big endian.

Makefile options:

Makefile Options

	endian_01.01	endian_01.02
ENDIAN	EL	EB
COPY	d	
CACHE	0	
EXCEP_VECTOR_AT_BOOT	0	
CLKCFG	001	
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init	
SHARED_FILES	<empty>	./../endian_01.01/test.c
PRIORITY	001_011_010	
DEFAULT	001	

Description:

The program is written in C and shared between tests. The test can decide the endianness of the system, and is performed by writing a 32-bit word to the RAM slave. The test then accesses a single byte within the word. Depending on the contents of the byte, the software selects either little or big endianness.

This test has some limitations, as it only tests that a 32-bit write and a 8-bit read fulfill the endianness requirements.

4.4.2.3 interrupt_01.01 and interrupt_01.02 (The Use of Interrupt)

Purpose: To verify that a write error initiates the MIPS™ BusBridge™ 2 module to generate an interrupt in both little and big endian.

Makefile options:

Makefile Options

	interrupt_01.01	interrupt_01.02
ENDIAN	EL	EB
COPY	d	
CACHE	1	
EXCEP_VECTOR_AT_BOOT	0	
CLKCFG	001	
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init	
SHARED_FILES	<empty>	./../interrupt_01.01/test.c
PRIORITY	001_011_010	
DEFAULT	001	

Description:

The program is written in C and shared between tests. A new exception handler is installed and a write error is generated by a write access to the read-only boot ROM slave. When the interrupt occurs, the interrupt handler makes an acknowledge signal to the MIPS™ BusBridge™ 2 module by writing to the monitor. At the same time, a message is printed on the screen displaying that an interrupt has been received.

4.4.2.4 sleep_01.01 and sleep_01.02

Purpose: To verify that the BusBridge™ 2 module will enter sleep mode when the core executes a *wait* instruction and leaves it when an interrupt is received.

Makefile options:

Makefile Options

	sleep_01.01	sleep_01.02
ENDIAN	EL	EB
COPY	ad	
CACHE	1	
EXCEP_VECTOR_AT_BOOT	0	
CLKCFG	001	
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init	
SHARED_FILES	<empty>	./../sleep_01.01/test.c ./../sleep_01.01/sleep.S
PRIORITY	001_011_010	
DEFAULT	001	

Description:

The main part of the test program is written in C. This program calls two small assemble routines that execute a *wait* instruction. One of the routines execute a *sw* before and the other a *lw* instruction just before the wait instruction. The flow of a single loop of the test is.

1. The program starts a hardware timer (in monitor.v) that will generate a interrupt when expired.

2. One of the assemble routines are called and the core enters sleep mode.

3. When the hardware timer expires, an interrupt is generated and the core will wake up.

4. The interrupt handler checks that an interrupt is expected.

This flow is run many times and the number of interrupts is compared with the expected.

4.4.3 The “transactions” Block

4.4.3.1 read_01.01 (2 Single Word Read Transactions)

Purpose: To make an example of simple transactions in which simulation results are easily inspected in a graphical viewer (e.g. VirSim™ from Synopsys).

Makefile options:

Makefile Options

	read_01.01
ENDIAN	EL
COPY	d
CACHE	1
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init
SHARED_FILES	<empty>
PRIORITY	001_011_010
DEFAULT	001

Description:

The program is written in assembler. The data accesses are uncached in order to make immediate activity on the ECi, executing two load word-instructions from consecutive, uncached word-memory addresses.

4.4.3.2 read_01.02 (2 Single Halfword Read Transactions)

Purpose: To make an example of simple transactions in which simulation results are easily inspected in a graphical viewer (e.g.VirSims).

Makefile options:

Makefile Options

	read_01.02
ENDIAN	EL
COPY	d
CACHE	1
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init
SHARED_FILES	<empty>
PRIORITY	001_011_010
DEFAULT	001

Description:

The program is written in assembler. The data accesses are uncached in order to make immediate activity on the ECi, executing two load halfword-instructions from consecutive, uncached word-memory addresses.

4.4.3.3 read_01.03 (2 Single Byte Read Transactions)

Purpose: To make an example of simple transactions in which simulation results are easily inspected in a graphical viewer (e.g.VirSims).

Makefile options:

Makefile Options

	read_01.03
ENDIAN	EL
COPY	d
CACHE	1
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init
SHARED_FILES	<empty>
PRIORITY	001_011_010
DEFAULT	001

Description:

The program is written in assembler. The data accesses are uncached in order to make immediate activity on the ECi, executing two load byte-instructions from consecutive, uncached word-memory addresses.

4.4.3.4 read_01.04 (mixed-sized single read transactions)

Purpose: To make an example of simple transactions in which simulation results are easily inspected in a graphical viewer (e.g.VirSims).

Makefile options:

Makefile Options

	read_01.04
ENDIAN	EL
COPY	d
CACHE	1
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init
SHARED_FILES	<empty>
PRIORITY	001_011_010
DEFAULT	001

Description:

The program is written in assembler. The data accesses are uncached in order to make immediate activity on the ECi, executing six load instructions of different bit-width from consecutive, uncached word-memory addresses.

4.4.3.5 read_02.01 (2 Bursted Read Transactions)

Purpose: To make an example of simple transactions in which simulation results are easily inspected in a graphical viewer (e.g.VirSims).

Makefile options:

Makefile Options

	read_02.01
ENDIAN	EL
COPY	d
CACHE	1
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init
SHARED_FILES	<empty>
PRIORITY	001_011_010
DEFAULT	001

Description:

The program is written in assembler. The data accesses are cached in order to make bursts on the ECi, executing two load instructions from two different cache lines.

4.4.3.6 write_01.01 (2 single word write transactions)

Purpose: To make an example of simple transactions in which simulation results are easily inspected in a graphical viewer (e.g.VirSims).

Makefile options:

Makefile Options

	write_01.01
ENDIAN	EL
COPY	d
CACHE	1
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init

Makefile Options

	write_01.01
ENDIAN	EL
SHARED_FILES	<empty>
PRIORITY	001_011_010
DEFAULT	001

Description:

The program is written in assembler. The data accesses are uncached in order to make immediate activity on the ECi, executing two save word-instructions from consecutive, uncached word-memory addresses.

4.4.3.7 write_01.02 (2 Single Halfword Write Transactions)

Purpose: To make an example of simple transactions in which simulation results are easily inspected in a graphical viewer (e.g.VirSims).

Makefile options:

Makefile Options

	write_01.02
ENDIAN	EL
COPY	d
CACHE	1
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init
SHARED_FILES	<empty>
PRIORITY	001_011_010
DEFAULT	001

Description:

The program is written in assembler. The data accesses are uncached in order to make immediate activity on the ECi, executing two load halfword-instructions from consecutive, uncached word-memory addresses.

4.4.3.8 write_01.03 (2 Single Byte Write Transactions)

Purpose: To make an example of simple transactions in which simulation results are easily inspected in a graphical viewer (e.g.VirSims).

Makefile options:

Makefile Options

	write_01.03
ENDIAN	EL
COPY	d
CACHE	1
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init
SHARED_FILES	<empty>
PRIORITY	001_011_010
DEFAULT	001

Description:

The program is written in assembler. The data accesses are uncached in order to make immediate activity on the ECi, executing two load byte-instructions from consecutive, uncached word-memory addresses.

4.4.3.9 write_01.04 (Mixed-sized Single Write Transactions)

Purpose: To make an example of simple transactions in which simulation results are easily inspected in a graphical viewer (e.g.VirSims).

Makefile options:

Makefile Options

	write_01.04
ENDIAN	EL
COPY	d
CACHE	1
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init
SHARED_FILES	<empty>
PRIORITY	001_011_010
DEFAULT	001

Description:

The program is written in assembler. The data accesses are uncached in order to make immediate activity on the ECi, thereby executing six load instructions of different bit-width from consecutive, uncached word-memory addresses.

4.4.3.10 write_02.01 (2 bursted write transactions)

Purpose: To make an example of simple transactions in which simulation results are easily inspected in a graphical viewer (e.g.VirSims).

Makefile options:

Makefile Options

	write_02.01
ENDIAN	EL
COPY	d
CACHE	1
EXCEP_VECTOR_AT_BOOT	0
CLKCFG	001
CLEARCACHE	1 if cache supports hw init 0 if cache does not support hw init
SHARED_FILES	<empty>
PRIORITY	001_011_010
DEFAULT	001

Description:

The program is written in assembler. The data accesses are cached in order to make bursts on the ECi, and the program fills two different cache lines in order to make write bursts.

4.5 Test performed in Random testbench

The random testbench uses randomly generated tests which are played to MIPS™ BusBridge™ 2 module using a Bus Functional model of a MIPS® core. The tests can be run using the run_test script.

There are separate tests for 4K™, 24K™, and 34K™ cores and are present in the directories specified in the table below.

Table 4.11 Directories for 4K™, M4K™, 24K™ and 34K™ BFM tests

Core Name	BFM test directories
4K™/4KE™	${SUSSYS_SW_DIR}/bfm_tests/M4K
M4K™	${SUSSYS_SW_DIR}/bfm_tests/MM4K
24K™	${SUSSYS_SW_DIR}/bfm_tests/M24K
34K™	${SUSSYS_SW_DIR}/bfm_tests/M34K

The names of tests is the names of subdirectories present under the directories listed in Table 4.11.

4.5.1 Setup to run a simulation using random testbench

In order to run simulations using random testbench, a BFM model of the core should be setup correctly. Following are the steps to integrate BFM models with BusBridge™ 2 module's testbench.

1. Create a soft link to the top level BFM distribution directory and name it as <core_name>_bfm. <core_name> should be m4k for 4K™ cores, m24k for 24K™ cores, and m34k for 34K™ cores. The top level BFM directory contains the directories scripts, SunOS etc.

2. The $LD_LIBRARAY_PATH environment variable should have been setup during the BFM install process to include a path to the BFM shared objects library. If not then do this:

   setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:$SUBSYS_DIR/<core_name>_bfm/{Linux | SunOS}/lib

4.5.2 Example to run random testbench

Following is a simple example of a script to run bfm test in random testbench. It assumes that BFM is setup and working correctly.

```
# Execute configure_mab_tb.pl. The following options must be specified:
# --core-model=BFM
# --tb-type=RANDOM
configure_mab_tb.pl --core-model=BFM --tb-type=RANDOM -o=core_random_tb

# Execute run_test script to run test. The config file created using
# configure_mab_tb.pl must be specified to run_test.
run_test --config=core_random_tb --build example
```

4.6 Post Synthesis Simulation

4.6.1 Gate Level Netlist for MBB2

Any of the tests in $SUBSYS_DIR/regression/tests can be done using a post synthesis netlist. The way to do this is to use the run_test script like this:

run_test --mab-model=net --gate-sim-filelist=<your_filelist> <other options> <test name>

There is an example of the ‘‘filelist’’ in the $SUBSYS_VERIF_DIR/gate_level directory.

To run gate level regressions a testbench configuration file needs to be created:

configure_mab_tb.pl -o <your_gate_config_name> --mab-model=net --gate-sim-filelist=<your_filelist>

Then the regressions can be run with the gate level netlist:

run_regression [--clock-ratio=XX] --config=<your_gate_config_name>

4.6.2 Using a Gate Level Netlist of the CPU Core

It is possible to simulate with a post synthesis gate level netlist of the CPU core itself. To do this the following procedure is required:

Step 1: Run the netlist2top.pl scripts in the ‘‘bin’’ directory as follows:

```
bin/netlist2top.pl --netlist=<N> --topmodule=<M> --techlib=<T> --ramdir=<R>
```

where

N = full path name of your netlist Verilog file.

M = the top module name. For 24K™ and 34|K™ cores this is *tpz_top*. For 4K™/4KE™/M4K™ cores it is *m4k_top*

T = full path name of the technology simulation Verilog library.

R = full directory path to the directory containing simulation models for any RAMs used in the Core. The Verilog source files in this directory must have a ‘.v’ extension.

The result of running this script will be:

1. The top level module definition in the netlist file will be extracted to:

   integration/<M>.v

2. A CDI file will be created:

   integration/core-gate.cdi

Step 2: Link the core configuration file used for core synthesis to the integration directory

```
ln -s <your_core_synthesis_config_file> ${SUBSYS_DIR}/integration
```

Step 3: Set the CPU_NET_CDI entry in your mab_cpu_model.info.<m4k | m24k | m34k | mm4k> file to point the CDI file in 2 above. i.e.

```
CPU_NET_CDI integration/core-gate.cdi.
```

You are now ready to simulate. To select Core netlist simulation use the ‘‘NET’’ value for the --core-model command line option to run_test and configure_mab_tb.pl.

4.7 Creating and using a Custom Testbench

It is possible to replace the standard testbenches that come as part of the MBB2 distribution with a user defined one. To create one follow these steps:

1. Make a directory to hold the custom testbench. e.g.

   ```
   mkdir ${SUBSYS_DIR}/my_test_dir
   ```

2. Put the top level Verilog file for the testbench into this file. e.g.

   ${SUBSYS_DIR}/my_test_dir/my_testbench_top.v

3. Create a filelist for your testbench in the same directory (see below for the format of entries to this file). This file must have the same root as the testbench top level Verilog file followed by a ‘‘.filelist’’ extension. e.g.

   ${SUBSYS_DIR}/my_test_dir/my_testbench_top.filelist

The entries to the filelist file are of the following type:

> +incdir <directory name> : For include file search directories.
>
> -y <directory name>: For Verilog source file search directories.
>
> -v <file name>: For Verilog files containing multiple module definitions. e.g. technology simulation libraries.
>
> <file name>: For individual Verilog files.

To use the custom defined testbench all is required is to add the option to the run_test and/or configure_mab_tb.pl scripts:

> --tb-top=<file defined in step 2. above>

There is a sample custom testbench in the $SUBSYS_VERIF_DIR/custom_tb directory. See the README file in thast directory for details.

4.8 Interactive Simulation with NC-Verilog™

Readers of this section should be familiar with the NC-Verilog™ notion of access types.

When simulating non-interactively the access permissions to simulator objects are the default ones (no read, no write, no connectivity) whereas when simulating interactively (--I on the run_test command line) this is added to the NC-Verilog elaborator command line:

> -ACCESS +r

While this is sufficient for most purposes such as viewing simulation waveforms in Simvision™ or generating VCD files it might not be enough for some types of debug. If this is the case the user should create an ‘‘access file’’ containing the required access types. The file should be:

> $SUBSYS_VERIF_DIR/config/ncelab_access.txt

If this file exists then

> -AFile SUBSYS_VERIF_DIR/config/ncelab_access.txt

is added to the elaborator command line. This applies to both interactive and non-interactive simulations.

4.9 Simulation with MIPSSim™+BFM

This is very straightforward assuming that the MIPSSim executable is in the serach path. The user has merely to add

--core-model=bfm --tb-type=directed

to the run_test or configure_mab_tb.pl command lines.

Chapter 5

Synthesis

The design database is supplied to enable synthesis using DesignCompiler™ from Synopsys. The subsystem delivery includes Makefiles and constraint files. For the purposes of this chapter it is assumed that the installation procedure in Section 2.2 "Install and Setup the Design Database" of Chapter 2, "Design Database" on page 11 has been followed.

5.1 Synthesis Flow

The synthesis flow separates technology and timing-related information from generic setup i.e. all timing and library configuration (constraints) are kept in $SUBSYS_SYNTH_DIR/config, while general Makefiles and DesignCompiler™ TCL command scripts reside in $SUBSYS_SYNTH_DIR.

The synthesis flow requires files from three locations in the design database:

- $SUBSYS_SYNTH_DIR contains a Makefile and TCL command files for running synthesis
- $SUBSYS_SYNTH_DIR/config contains constraints and configuration files in TCL format.
- $SUBSYS_RTL_DIR contains the RTL.

In addition two environment variables should have been set up during installation:

- $TECH_DIR/ points to technology libraries located in the $SUBSYS_DIR/external/ dir. This should already have been setup during the MBB2 installation process.
- $SYNTH_OUT_DIR points to the users synthesis results sub-directory of $SUBSYS_BUILD_DIR.

To complete the preparation for synthesis -

1. Create a subdirectory $SUBSYS_BUILD_DIR/$SYNTH_OUT_DIR/config.
2. Copy the files lib_info.tcl, global_constraints.tcl, mab_constraints-<BUS>.tcl from $SUBSYS_SYNTH_DIR/config to this dir. <BUS> = OCP if the MBB2 is to be synthesised for use with a 24K™ or 34K™ core, <BUS> = SRAM if the MBB2 is to be synthesised with an M4K™, otherwise <BUS> = EC.
3. Edit the copied Tcl files. See section 5.3 below for more details.

   Note: $SUBSYS_SYNTH_DIR/config/examples has working samples of the lib_info.tcl and global_constraints.tcl files.
4. Link (or copy) the mab_config.vh file (created during the installation process) from $SUBSYS_RTL_DIR/config to $SUBSYS_BUILD_DIR/$SYNTH_OUT_DIR/config.

5. If gated clocks are enabled in mab_config.vh then a clock gating module needs to be created in $SUBSYS_RTL_DIR/shared. The module name must be “mab_clockandlatch’ and the file name “mab_clockandlatch.v”.

   There is an example file called mab_clockandlatch_example.v in this directory.

Once this setup has been completed a synthesis can be started by directly invoking the

```
 $SUBSYS_SYNTH_DIR/Makefile
```

with a target and the name of the block ($SUBSYS_BLOCK) that is to be synthesized. However it is most easily done by using the run_synth script (see next section for details).

5.2 Starting the Synthesis

After final setup of the synthesis output and configuration directories as per the previous section use the run_synth script in $SUBSYS_DIR/bin to do a complete synthesis simply by invoking it thus:

```
run_synth -t synth -b mab
```

This will, in effect, run 3 commands:

- run_synth -t init -b mab: to initialise the synthesis output directory.
- run_synth -t read-b mab: to read in the design files, parse them, and create and unmapped database.
- run_synth -t compile -b mab: reads in the unmapped database from the previous step together with the synthesis constraints files and produces a final mapped and optimised database and netlist.

For a full list of run_synth options type

```
run_synth -h
```

5.2.1 Checkpointing the synthesis run

For long synthesis runs the user may wish to protect themselves against crashes by generating checkpoint files. To do this add this command line option to the run_synth command line:

```
-CP <checkpoint_period>
```

If the checkpoint period is 0 no checkpointing is performed.

The checkpoint files will be placed in the $SUBSYS_BUILD_DIR/$SYNTH_OUT_DIR/check

5.2.2 Re-running synthesis

If there is an error in either the ‘‘read’’ or ‘‘compile’’ phases then fixing the problem will, in general,allow

```
run_synth -t synth -b mab
```

to run again and recreate the unmapped and/or the mapped databases.

However there are some circumstances under which this won’t happen., e.g. DesignCompiler™ cannot find one of its libraries. If this is the case then the best procedure is:

1. Save whatever information is wanted from the current $SUBYS_BUILD_DIR/$SYNTH_OUT_DIR and its sub-directories.

2. Run

   run_synth -t clean.

This will remove all the subdirectories of $SUBYS_BUILD_DIR/$SYNTH_OUT_DIR except for ‘‘config’’, which will be left unchanged.

3. Re-initialise with

   run_synth -t init.

The synthesis process can now be restarted.

5.3 Constraints and Library Configuration

The constraints and library configuration are found in $SUBSYS_SYNTH_DIR/config. These 2 files will require editing after being copied into the user’s own synthesis configuration directory:

- lib_info.tcl

- global_constraints.tcl

For the variables and values that need to be modified open these files in your favourite text editor and search for the string ‘‘__USER_DEFINED__’’.

Note that the

- mab_contsraints-<BUS>.tcl

file will, in general, not require editing and can be used as supplied. However since the timings for the AHB side of the MBB2 are in the users’ domain some changes may be needed. If so search for the string ‘‘__MAB_AHB_TIMINGS__’’.

5.4 Collecting the Synthesis Results

The synthesis results, reports, log files etc. are held in these sub-directories of $SUBSYS_BUILD_DIR/$SYNTH_OUT_DIR:

gate	Post synthesis gate level netlists
log	Log files from the read and compile phases.
check	Outputs from the ‘check_design’ command.
report	Timing and other reports.

checkpoint	Checkpoint files produced if checkpointing is enabled
unmapped	Unmapped design database produced after the read phase.
mapped	Mapped database produced after the compile phase.
WORK	Synthesiser working directory.

5.5 Detailed Analysis

If other types of reports are needed, then start dc_shell manually and read a design block (mapped db file). The easiest way to accomplish this is to use the $SUBSYS_SYNTH_DIR/Makefile, which has a target for this purpose (read_db). The syntax is:

```
cd $SUBSYS_SYNTH_DIR

make read_db SUBSYS_BLOCK=<blockname> [DB=mapped|unmapped]
```

blockname = mab

DB = mapped or unmapped. If nothing is specified, then “mapped” is assumed.

Issuing this command reads in the design and constraint files and gives a dc_shell prompt. From this prompt, the user can run a new compile or generate various reports.

It is also possible to use PrimeTime for generating additional reports. The syntax is then:

```
make pt SUBSYS_BLOCK=<blockname>
```

Appendix A

Revision History

Revision	Date	Description
00.09	November 8, 2004	Final internal release revision.
01.00	November 9, 2004	First external release revision.
01.01	November 10, 2004	• Misspelling of date on title page corrected • Added new section on re-running synthesis after an error..
01.03	November 10, 2004	Fixed some ™ issues
01.04	November 17, 2004	• NC-Verilog™ support now added • Supported tools section updated.
01.06	December 8, 2004	• 4KE™ support noted • Improved description of --clean flag to run_test..
01.08	Feb. 21, 2005	• Documented all changes for MR. • ModelSim™ support now added.
01.10	Mar 11,2005	• Added new section describing CDI generation. • Added table of entries for the mab_cpu_model.info files. • Added section describing simulation with a CPU core gate level netlist. • Added section describing the creation and use of a custom testbench. • General clean up.
01.11	Mar 17th, 2005	• Spell check
02.00	Mar 18th,2005	• MR1 release
02.01	Mar 25th, 2005	• Documented fix for NC-Verilog interactive simulation
02.03	April 28, 2006	• Changes for MR2 release
02.04	May 3rd, 2006	• Small changes • Added explicit how to for MIPSim+BFM simulation
02.05	May 16, 2006	• Small changes.
02.06	October 5, 2011	• Updated to work with the latest 24K™ and 34K™ releasees. • Supported tools versions updated. • Support for the CoreSourcery toolchain.