ARM926EJ-S Portable Functional Test Source Code - User Guide

IP Products - CPU

Document number:	CP023-PRDC-001757 4.0
Date of Issue:	12 June 2003
Author:	ARM
Authorized by:	

Abstract

This document provides a description and usage guide for the Portable Functional Test Source Code for ARM926EJ-S. The tests are intended to be used as part of verifying any ARM926EJ-S integration. A number of functional tests are provided in a form that allows them to be quickly and easily ported to any ARM926EJ-S based SoC.

Keywords

Integration, vector, portable

Reviewer list

Name	Function	Name	Function

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

This version accompanies the first full quality release of the following deliverable.

AT230-VE-70106 ARM926EJ-S Portable Functional Test Source Code

1.1.2 Change history

The change history for this document is managed within ARM by Domino.Doc.

1.2 References

This document refers to the following documents.

Ref	Doc No	Author(s)	Title
1	ARM-DII-0015	ARM	ARM926EJ-S Implementation Guide

1.3 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
AVS	Architecture Validation Suite. A series of tests that prove backwards compatibility with previous ARM architecture revisions
DVS	Device Validation Suite. These sets of tests target specific areas of the implementation rather than ARM architecture.
NCB	Non-Cacheable, Bufferable memory region
NCNB	Non-Cacheable, Non-Bufferable memory region
WB	Cacheable, Bufferable memory region (write-back)
WT	Cacheable, non-bufferable memory region (write-through)
MVA	Modified Virtual Address (VA modified by the FCSE Process ID)
VA	Virtual Address
FCSE	Fast Context Switch Extensions

2 INTRODUCTION

The Portable Functional Test Source Code should be used during ARM926EJ-S SoC integration and prototype silicon evaluation as part of a verification strategy. The tests may also be used during production test to supplement the ATPG based tests. A number of functional tests are provided in a form that allows them to be quickly and easily ported to any ARM926EJ-S based SoC. The tests may be run at full clock speed to gain confidence in at-speed operation.

3 SCOPE

This document describes how to use the Portable Functional Test Source Code.

Each of the 7 tests:

- Is compact in size, requiring only 16KB of external memory. This includes the space required for the test code itself, and any memory used for data manipulation. Access to the exception vector table is also required.
- Is independent of cache and TCM size.
- Is not reliant on the presence of trick-boxes or additional system logic such as coprocessors.
- Has a configurable base address for easy relocation in the memory map. This allows easy targeting to available system memory.
- Is self-checking.
- Reports the result in a system independent way.

The tests provide only limited functional coverage of the ARM926EJ-S macrocell itself. Coverage is constrained by the necessary assumptions regarding the target system, for example we have to assume that no coprocessor is implemented. A test template is provided to support the system designer in developing further tests.

The tests may be run at full system speed and will give a good indication of system performance. However, as the critical paths in any implementation will vary, the tests do not necessarily stimulate the worst-case timing path in every implementation.

Note:

1. Some usage is made by these tests of undocumented test registers. The use of such registers is not supported beyond their exact use by these tests.
2. The length of test patterns produced from these tests may be considered too long for use in production test. If this is the case then a subset of these tests can be chosen.

4 USER GUIDE

4.1 Deliverable Content

The Portable Functional Test Source Code deliverable comprises the following files.

Makefile – GNU makefile that compiles the test source files

bin2hex – utility for converting binary files to hexadecimal format

Seven test source files:

src/test_ie.s

src/test_ie_dc_ldst.s

src/test_ie_dhry.s

src/test_ie_bus.s

src/test_ie_java.s

src/test_ie_speed_cache.s

src/test_ie_thumb.s

A boot file – defining exception vectors:

src/test_boot.s

A template test source file:

src/test_ie_template.s

4.2 Requirements

In addition to the supplied files, access will be needed to:

- The ARM Developer Suite tools
- The GNU Make utility

These are required to compile the supplied ARM assembler source files.

4.3 Usage

The binary and hex image files can be generated with a command of the form:

```
gnumake CODEADDR=0x<code_address> BOOTADDR=0x<boot_address>
```

The code address can be any 16k aligned address other than zero. Supplying an invalid address will cause assembly to fail.

The boot address can be either zero (`0x0`) or hivecs (`0xffff0000`). Supplying an invalid address will cause assembly to fail.

This will generate binary and hex images for the test files within the ‘binaries’ subdirectory. To run each test case will require two files (the ‘boot’ file and the particular ‘test_ie_x’ file) to be converted to whatever format is required and loaded into memory on the test device. The test_boot file should be loaded in at the location of exception vectors on the target system, (either zero or high vectors), and the test_ie_x file at the location of available memory.

After a test has been run, completion is visible externally by monitoring the data bus for the following series of values:

```
0xffffffff

0x55555555

0xaaaaaaaa
```

These will be followed by a value indicating a pass or fail (1 for pass, 0 for fail) and the number of test parts (documented below) passed.

For systems in which the data values are not visible externally, this section of the tests should be replaced by the user with alternative code that will place the result indication values in a more appropriate location as required.

A template file test_ie_template.s is included for the purpose of developing new tests. This file contains a framework for the development of new tests, and includes setup code that will write four entries into the lockdown TLB, mapping 16k of memory at the test location as cacheable and bufferable. An entry for the exception vectors is also written. A finalization routine as detailed above completes the skeleton code.

4.4 Running the tests using the validation testbench

To run the tests in a standard ARM926EJ-S validation environment, the following steps are required:

- Copy the integration_vectors directory into your validation directory
- Add an entry for 'integration_vectors' in the `$ValSrcProject` section of `validation.cfg`, to add the integration_vectors directory to the validation search path – i.e.
  ```
  $ValSrcProject = ‘
  arm926ej
  arm9tdmi
  arm9e
  arm9ej
  ris_9es
  ris_926
  integration_vectors
  ';
  ```
- Create a ‘setup’ subdirectory of integration_vectors
- In this setup directory, create a setup file for each test you want to run, of the form:
  `test_boot <boot_address`; either 0 or ffff0000>
  `test_ie_<testname> <code_address` - whatever 16k aligned address you wish to run from>
  This file should be called `test_ie_<testname>.setup`
  - o Example: a setup file for the test_ie test, booting from address zero and with the code address `0x10000000`, would be called `test_ie.setup` and would be of the form:
  ```
  test_boot          0
  test_ie            10000000
  ```
- Then run the test as a standard validation test, using a command of the form:
  `validation test_ie_<testname> -PD ’BOOT_ADDR SETA 0x<boot_address`, same as above`>’ -PD ’CODE_ADDR SETA 0x<code_address`, as above`>’ –v`

- Example: to run the test_ie vector, having created the setup file as above, the command would be:

  ```
  validation test_ie –PD ‘BOOT_ADDR SETA 0x0’ –PD ‘CODE_ADDR SETA
  0x10000000’ -v
  ```

Full documentation on the use of validation can be found in Chapter 3 of the ARM926EJ-S Implementation Guide [1].

4.5 Running the tests using AXD

To run the tests on AXD:

- Start multi-ICE server
- start AXD
- File->Load to memory -> (binary file)
- Load `test_boot.bin` to the address specified by BOOTADDR in code compilation (do this by dialogue box that comes up after selected file to load)
  - Example: load `test_boot.bin` to address `0x0` (default)
- Load `test_ie.bin` (or whichever other test you want to load) to the address specified by CODEADDR in compilation.
  - Example: choose `test_ie.bin` and put it at `0x4000`
- In the cmd line interface of AXD, type
  - `let pc 0x4000`
  - `go`

4.6 Test Content

4.6.1 test_ie

This test is based on the AVS test T1-32 from the armv4 suite used as the instruction execution vector test for ARM926EJ-S. It is a short program which tests the operation of the ALU and condition code system in a thorough fashion, first setting the condition codes by hand using `MSR CPSR_flg,#<Imm>` and then conditionally executing branch instructions. Finally it performs comparisons of large and small positive and negative numbers.

There are four test parts.

Coverage report:

Reported Block Coverage	5274/8464=62%
Reported Arc Coverage	3543/6387=55%

These reported values are based on instance totals for the design as a whole, filtered to remove default cases etc.

4.6.2 test_ie_dc_ldst

This test is based on the DVS test data_cache_ldst, and had been extensively reworked in order to comply with the memory limitations imposed on the functional test vectors.

It is aimed at some basic checking of back-to-back load / store operations that access different regions. This test is really checking that changing the cache properties for adjacent memory accesses does not confuse the cache state machines.

The test does a block copy of some data using all combination of LDRs/STRs/LDMs/STMs, LDRs/STRs/LDRBs/STRBs and LDRs/STRs/LDRHs/STRHs, (16, 12 and 12 combinations respectively), and then checks the consistency of the copied data. A block of test data is setup in each of the 4 cache regions under test, NCNB, NCB, CNB & CB. The tests then copy data by reads/writes to alternate regions, e.g.:

```
          region 1                        region 2
r0->|-----------------|          r2->|----------------|
    |      reads      |              |     reads      |
    |-----------------|              |----------------|
    |                 |              |                |
    |                 |          r1->|----------------|
    |                 |              |     writes     |
    |                 |              |----------------|
r3->|-----------------|              |                |
    |     writes      |              |                |
    |-----------------|              |                |
```

With the same read data pattern in both regions, data is read / written alternately from both regions and written to the 2nd region write area. When complete the write data in the 2nd region will match the read data. The tests are performed for all combinations of cache hit/misses (for cacheable regions)

Combinations of LDRHs/STRHs with LDRB/STRBs or LDMs/STMs and of LDRBs/STRBs with LDMs/STMs are not tested.

There are 108 test parts.

Instance coverage report for the filtered design:

Reported Block Coverage	5509/8464=65%
Reported Arc Coverage	3703/6387=57%

4.6.3 test_ie_thumb

This test is based on the AVS test t_regress, and performs a quick check of all the Thumb instructions.

Tests:

- Branch Instructions:
  - Format 1: BX Rm (forward and backward addressing)
  - Format 2: B <target_address> (forward and backward addressing)
  - Format 3: BL <target_address> (forward and backward addressing)
  - Format 4: B<cond> <target_address> (forward and backward addressing)
- Data-processing Instructions
  - Format 1: ADD|SUB Rd, Rn, Rm
  - Format 2: ADD|SUB Rd, Rn, #3_bit_immed
  - Format 3: ADD|SUB|MOV|CMP Rd|Rn, #8_bit_immed

  - Format 4: LSL|LSR|ASR Rd, Rn, #5_bit_immed
  - Format 5: MVN|CMP|CMN|TST|ADC|SBC|NEG|MUL|LSL|LSR|ASR|ROR|AND|EOR|ORR|BIC Rd|Rn, Rm|Rs
  - Format 6: ADD Rd, PC|SP, #8_bit_immed
  - Format 7: ADD|SUB SP, SP, #7_bit_immed
- Special Data-processing (high register operations)
- Load and Store register Instructions
  - Format 1: LDR|LDRH|LDRB|STR|STRH|STRB Rd, [Rn, #5_bit_immed]
  - Format 2: LDR|LDRH|LDRSH|LDRB|LDRSB|STR|STRH|STRB Rd, [Rn, Rm]
  - Format 3: LDR Rd, [PC, #8_bit_immed]
  - Format 4: LDR|STR Rd, [SP, #8_bit_immed]
- Load and Store Multiple Instructions
  - Format 1: LDMIA|STMIA Rn!, <reg_list>
  - Format 2: POP <reg_list>, {PC}, PUSH <reg_list>, {LR}

There are 91 test parts.

Instance coverage report for the filtered design:

Reported Block Coverage	5907/8464=69%
Reported Arc Coverage	4215/6387=65%

4.6.4 test_ie_java

This test is based on the DVS test jdvs_stack_resource, and checks that each bytecode produced its expected result.

Bytecodes tested are:

j_nop, aconst_null, iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, lconst_0, lconst_1, fconst_0, fconst_1, fconst_2, bipush 0xf1, sipush 0xf123, iload 0, iload 4, iload_0, iload_1, iload_2, iload_3, iaload, aaload, faload, laload, daload, baload, caload, saload, aload 0, aload 5, aload_0, aload_1, aload_2, aload_3, lload, lload 6, lload_0, lload_1, lload_2, lload_3, pop, pop2, dup, dup_x1, dup_x2, dup2, dup2_x1, dup2_x2, swap, iadd, ladd , isub, lsub, imul, lmul, ineg, lneg, ishl, iand, land, ior, lor, ixor, lxor, i2l, l2i, i2b, i2c, i2s, lcmp, ifeq, ifne, iflt, ifge, ifgt, ifle, ifnull, ifnonull, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, if_acmpeq, if_acmpne, arraylength, goto, ret 9, istore 0, istore 4, istore_0, istore_1, istore_2, istore_3, fstore 0, fstore 4, fstore_0, fstore_1, fstore_2, fstore_3, lstore 0, lstore_0, lstore_1, lstore_2, lstore_3, dstore 0, dstore_0, dstore_1, dstore_2, dstore_3, iinc 0, 2, iinc 2, 1, iastore, fastore, lastore, dastore, bastore, castore, sastore

There are 122 test parts.

Instance coverage report for the filtered design:

Reported Block Coverage	5640/8464=66%
Reported Arc Coverage	4118/6387=64%

4.6.5 test_ie_speed_cache

This is a relocatable version of the test arm926ej_speed_cache that was included in the vector source for ARM926EJ-S as a test of expected critical speed paths.

The test is split into two main parts. The first exercises the expected speed path in the core, which is:

AUInvBAUEx -> Shifter -> Adder -> Vflag

The second exercises the cache chip select paths.

There are 82 checks in total.

Instance coverage report for the filtered design:

Reported Block Coverage	5414/8464=63%
Reported Arc Coverage	3642/6387=57%

4.6.6 test_ie_bus

This test aims to exercise various AHB conditions:

- Load and store multiples of all lengths are performed
- A writeback eviction from the data cache is forced
- A cache clean by MVA is performed
- There are a series of swap instructions to all four memory region types
- Finally load/store multiples across region type boundaries are checked.

There are 24 checks in total.

Instance coverage report for the filtered design:

Reported Block Coverage	4913/8464=58%
Reported Arc Coverage	3425/6387=53%

4.6.7 test_ie_dhry

This test is a piece of assembler code produced by the C compiler armcc using Dhrystone as the C source file.

Instance coverage report for the filtered design:

Reported Block Coverage	4715/8464=58%
Reported Arc Coverage	3335/6387=53%

4.7 Overall Coverage

Coverage of the ARM926EJ-S design from all 7 tests is summarised here.

Instance coverage report for the filtered design:

Reported Block Coverage	6631/8464=78%
Reported Arc Coverage	4929/6387=77%