Application Note 108

ARM1026EJ-S Coprocessor Reference Design

Document number: ARM DAI 0108A
Issued: Jan 2003

ARM

Application Note 108
ARM1026EJ-S Coprocessor Reference Design

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
January 2003	A	First release

Proprietary Notice

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

http://www.arm.com

Table of Contents

1 Introduction

1.1 About the ARM1026EJ-S Pipeline follower

The ARM1026EJ-S coprocessor interface enables multiple external coprocessors to be attached to the processor. To enable maximum performance from the coprocessors, the ARM10 processor issues coprocessor instructions as early in the pipeline as possible. Instructions are issued speculatively, and they can be canceled later in the pipeline if, for example, an exception or a branch misprediction occurs. As a result, a coprocessor must be able to cancel issued instructions until they are retired in the ARM10 pipeline. The coprocessor can stall the ARM10 pipeline if it requires more time to process an instruction, or it can initiate an undefined instruction exception if the issued instruction is not supported. Also, the data transfer between the ARM1026EJ-S and coprocessor pipelines must be synchronized. All these aspects require coprocessors to have pipeline follower logic, which can closely track the progress of instructions in the ARM10 pipeline.

The ARM10 pipeline follower hides the details of the ARM1026EJ-S coprocessor interface from the rest of the coprocessor core. It enables coprocessor designers to concentrate on designing the coprocessor rather than dealing with the complexity of the ARM1026EJ-S coprocessor interface. They can use a simple interface provided by the pipeline follower to build the coprocessor core. The pipeline follower and the coprocessor core can be hooked up to form the complete coprocessor.

1.2 What does this application note cover?

- Overview of the ARM10 pipeline follower
- RTL code
- Reference design.

2 ARM10 pipeline follower

2.1 Interfaces

The pipeline follower interfaces with the ARM10 on one side and the coprocessor core logic on the other. The interfaces are shown in Figure 2.1. The pipeline follower and the coprocessor core together form an ARM10 coprocessor. The signals in italics are for hand shaking with the other coprocessors in a multi-coprocessor configuration.

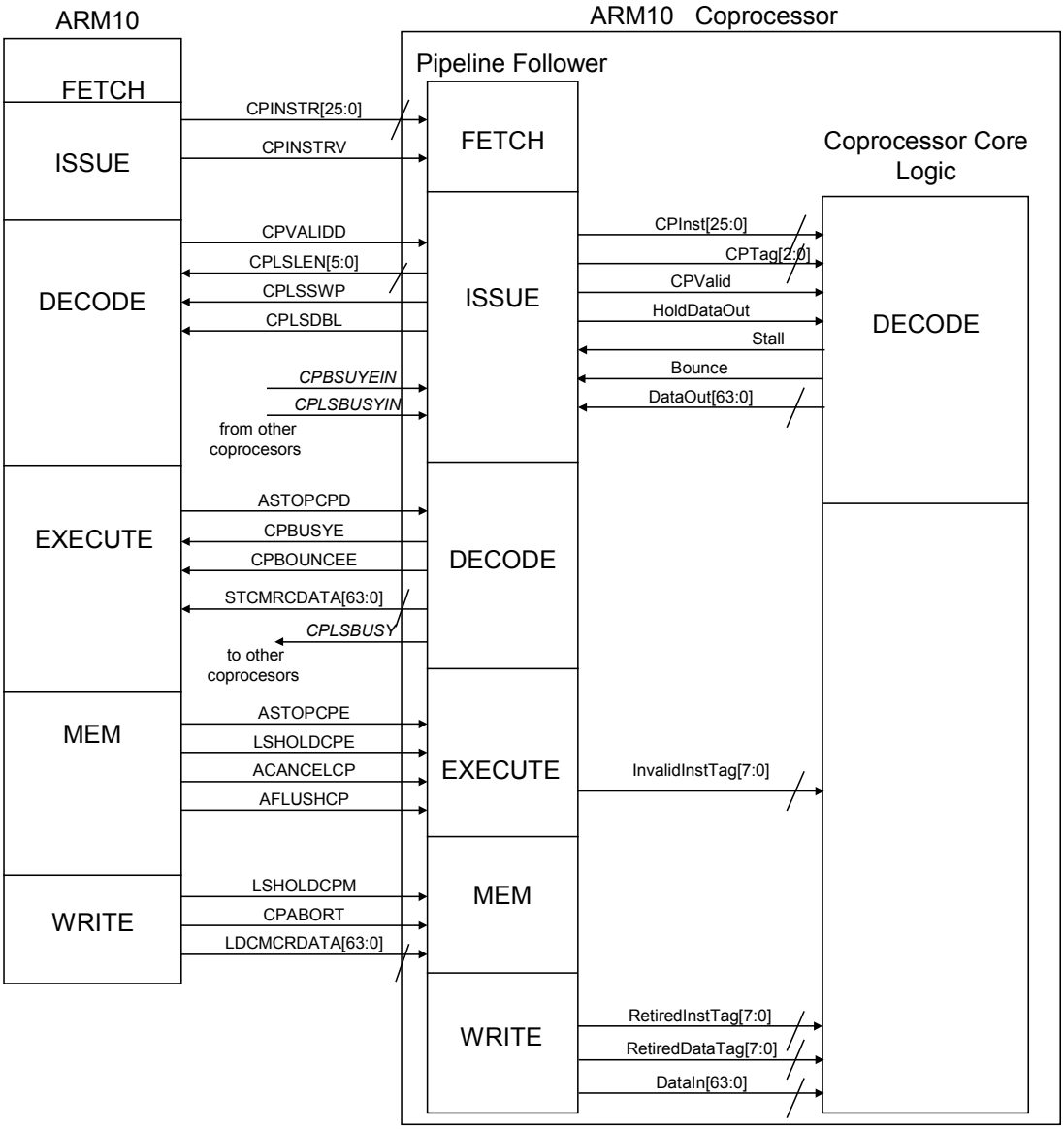

Figure 2.1 ARM10 pipeline follower

2.2 Signal description

Signal Name	Group	Description
ARM10 Coprocessor Interface		
CPINSTR[25:0]	Input	Instruction from ARM10 Issue stage
CPINSTRV	Input	Valid CP instruction in ARM10 Issue stage
CPVALIDD	Input	Valid coprocessor instruction in ARM10 Decode stage
ASTOPCPD	Input	ARM10 stalled in Decode stage in previous cycle
ASTOPCPE	Input	ARM10 stalled in Execute stage in previous cycle
LSHOLDCPE	Input	ARM10 LSU stalled in Execute stage in previous cycle
LSHOLDCPM	Input	ARM10 LSU stalled in Memory stage in previous cycle
ACANCELCP	Input	ARM10 instruction in Execute stage in the previous cycle cancelled
AFLUSHCP	Input	ARM10 instructions until the Execute stage in the previous cycle cancelled
CPABORT	Input	ARM10 instruction in the Memory stage in the previous cycle cancelled
CPRST	Input	Coprocessor reset
LDCMCRDATA[63:0]	Input	Data bus from the ARM10 to the coprocessor
CPLSLEN[5:0]	Output	Length of a Load/Store multiple instruction
CPLSSWP	Output	Upper and lower words on the data buses must be swapped by ARM10
CPLSDBL	Output	Load/Store involves a double word transfer
CPBUSYE	Output	Stall the ARM10 Execute stage
CPBOUNCEE	Output	Undefined instruction in the ARM10 Execute stage
STCMRCDATA[63:0]	Output	Data bus from the coprocessor to the ARM10
Other Coprocessors		
CPBUSYEIN	Input	CPBUSYE signal from the other coprocessors OR-ed together
CPLSBUSYIN	Input	CPLSBUSY signals from the other coprocessors OR-ed together
CPLSBUSY	Output	coprocessor involved in a load/store multiple data transfer
Coprocessor Core Interface		
CPInst[25:0]	Input	Coprocessor instruction
CPValid	Input	coprocessor instruction is valid
CPTag[2:0]	Input	Tag associated with the coprocessor instruction
HoldDataOut	Input	Hold the data on DataOut bus
InvalidInstTag[7:0]	Input	Tags of cancelled instructions in a one-hot encoded format
RetiredInstTag[7:0]	Input	Tag of the retired instruction in one-hot encoded format
RetiredDataTag[7:0]	Input	Tag of instruction owning the data on the DataIn bus in one-hot encoded format
DataIn[63:0]	Input	Data to the ARM10
LSLEN[5:0]	Input	Length of load/store multiple data transfer (hardwired)
LSSWP	Input	Upper and lower words on the data buses must be swapped by ARM10 (hardwired)
LSDBL	Input	Load/Store involves a double word transfer (hardwired)
Stall	Output	CPInst is not accepted by the coprocessor Core due to pipeline hazards
Bounce	Output	CPInst is not supported by the coprocessor Core
DataOut[63:0]	Output	Data from the ARM10
Other signals		
CPCLK	Input	Coprocessor clock
CPEN	Input	Coprocessor enable

2.3 Coprocessor core interface

The pipeline follower can be hooked up directly to the ARM10 coprocessor interface without any extra logic. A designer does not have to understand the ARM10 coprocessor interface, and can use the simple interface provided by the pipeline follower to build the coprocessor core. Therefore, this application note only covers the details of the coprocessor core interface of the pipeline follower. The interactions of the pipeline follower with the coprocessor core consist of three stages:

1. Instruction issue.
2. Instruction cancellation.
3. Instruction retirement.

2.3.1 Instruction issue

The pipeline follower asserts the **CPValid** signal whenever there is a valid coprocessor instruction on the **CPInst** bus. Each issued coprocessor instruction has a distinct tag associated with it. The tag ranges from 0-7, and is available on the **CPTag** bus. The pipeline follower uses the tag to communicate the status of the issued instruction in the later stages.

The **CPInst** and **CPTag** signals are stable at the beginning of the issue cycle. But, the **CPValid** signal is stable only towards the middle of the clock cycle. Therefore, the coprocessor core must decode **CPInst** every cycle without qualifying it with **CPValid**. If the **CPValid** signal is HIGH at the end of the clock cycle, then the instruction is valid and can be clocked in. The responses from the coprocessor core in the issue stage are considered valid by the pipeline follower only if **CPValid** is HIGH in that clock cycle.

During the instruction decode, the coprocessor core might find that it cannot accept the new incoming instruction because of hazards. The coprocessor core drives the **Stall** signal, irrespective of **CPValid**, until the hazard is resolved and it can accept the incoming instruction. If the instruction is valid, then the pipeline follower holds **CPInst**, **CPTag**, and **CPValid** signals as long as the **Stall** signal is HIGH. The coprocessor core is expected to clock in the incoming instruction in the cycle in which **CPValid** is asserted and **Stall** is deasserted. That is, instruction issue happens in the cycle in which **CPValid** is HIGH and **Stall** is LOW.

During the instruction decode, the coprocessor core might find that the incoming instruction is not supported. The coprocessor can bounce the instruction to a software handler by asserting the **Bounce** signal. This causes ARM10 to execute the undefined instruction handler to take care of the bounced instruction. As before, the **Bounce** signal is considered valid by the pipeline follower only in the cycle in which **CPValid** is HIGH. **Bounce** can be asserted immediately or after a number of stall cycles. The **Stall** signal overrides the **Bounce** signal in cycles where both are asserted.

MRC, MRRC, and STC instructions involve data transfer from the coprocessor core to the pipeline follower. The pipeline follower expects the data in the issue cycle, that is, cycle in which **CPValid** is HIGH and **Stall** is LOW. Therefore, the coprocessor core has to drive the required data on the **DataOut** bus during the cycle in which it registers the instruction. If there is a delay in reading the register file, or if there is a data hazard, the coprocessor core must assert the **Stall** signal until the data is ready.

In the case of an STCL instruction (STC multiple), a doubleword has to be sent out every cycle starting from the issue cycle of the instruction. The number of words in the store multiple is dictated by the **LSLEN** signal hardwired by the CP core. The pipeline follower might not be

able to clock in data in some cycles of a burst transfer due to stall conditions in the ARM10 pipeline. The **DataOutHold** signal is asserted in such cycles. The coprocessor core is expected to hold the data on the **DataOut** bus if the hold signal is HIGH. The **DataOutHold** is only asserted in the case of an STCL instruction. The pipeline follower does not issue any new instruction until the STCL data transfer is over, that is, the **CPValid** signal is deasserted until the STCL data transfer is over.

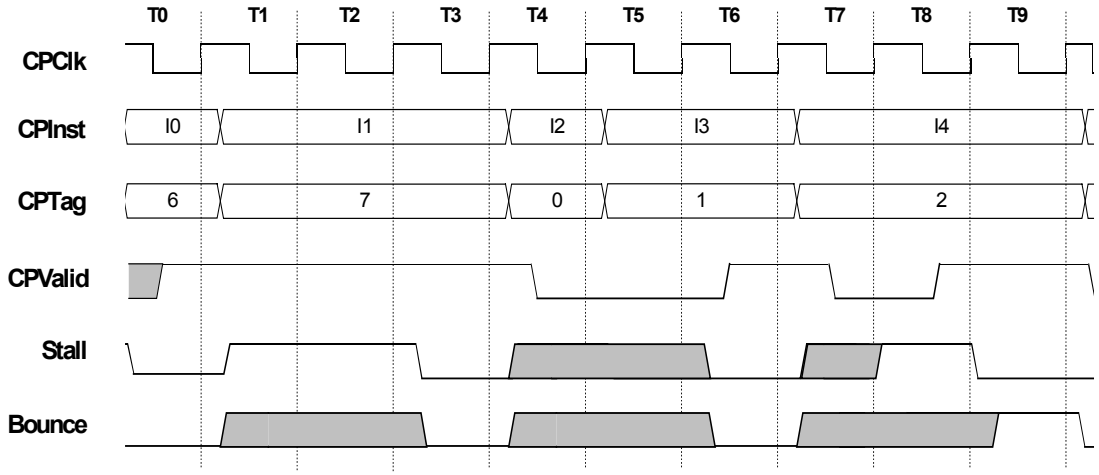

Figure 2.2 Instruction issue

Figure 2.2 shows the handshaking between the pipeline follower and the coprocessor core during instruction issue. The coprocessor core decodes instruction I0 in cycle T0. Because there is no hazard (**Stall** is LOW) and because **CPValid** is HIGH towards the end of T0, the coprocessor core clocks in I0 and its tag at the T0/T1 clock edge. The pipeline follower issues instruction I1 in cycle T1. The coprocessor core decodes I1 in cycle T1and asserts the Stall signal due to a hazard. The pipeline follower holds **CPInst**, **CPTag**, and **CPValid** because **Stall** is HIGH and the instruction is valid. The same hazard exists in cycle T2. In cycle T3 the hazard is resolved (**Stall** is LOW). The coprocessor core clocks in instruction I1 in T3/T4 clock edge. The coprocessor core decodes the next instruction I2 in cycle T4.The core might or might not assert the **Stall**/**Bounce** signal during the instruction decode. This has no effect on the pipeline follower because **CPValid** is LOW. The coprocessor core does not clock in instruction I2 because the **CPValid** was not asserted for the instruction. Immediately in the next cycle, the pipeline follower issues a different instruction, I3. Again, the core might or might not assert the **Stall**/**Bounce** signal during decode of I3. At the end of cycle T6, the coprocessor core clocks in instruction I3 and its tag because **CPValid** is HIGH and **Stall** is LOW. Instruction I4 is decoded by core in cycle T7. **Stall**/**Bounce** signal might or might not be asserted. The instruction is held in cycle T9 because **Stall** is asserted and the instruction is valid in cycle T8. In cycle T9, the coprocessor core asserts the **Bounce** signal indicating that it does not support the incoming instruction. Asserting **Bounce** when **CPValid** is HIGH and **Stall** is LOW leads to an undefined instruction exception in the ARM10 core. The coprocessor core does not clock in instruction I4 and its tag on T9/T10 clock edge because the instruction is not supported.

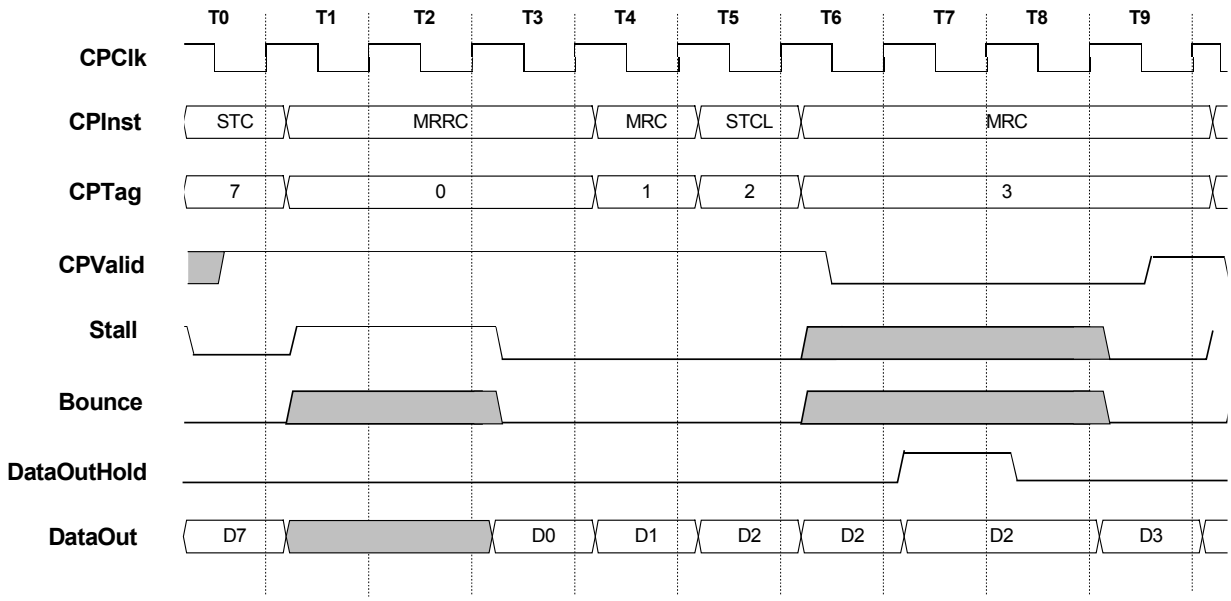

Figure 2.3 Data Transfer to the pipeline follower during instruction issue

Figure 2.3 shows the handshaking for data transfer from the coprocessor core to the pipeline follower. For MRC, MRRC, and STC instructions the coprocessor core must drive data on the **DataOut** bus in the same cycle in which the instruction is issued[1]. If there is a delay in reading the value from the register file or if there is a data hazard, then the coprocessor core must assert **Stall** signal until the data is ready. In the example, for the MRRC instruction, the coprocessor core takes two cycles to drive the data on to the **DataOut** bus. In cycle T3 the **Stall** signal is deasserted when the data is ready on the **DataOut** bus. Because the instruction is valid and there is no stall, the pipeline follower clocks in the **DataOut** bus on the T3/T4 clock edge.

An STCL instruction is issued in the same way as an STC instruction. The difference is that the coprocessor core has to drive a new doubleword in every successive cycle, until the end of the burst transfer. Sometimes the pipeline follower is not ready to accept data in a certain cycle. The **DataOutHold** signal is asserted by the pipeline follower to indicate this. In the given example, the pipeline follower is not ready to clock in the data in cycle T7. It asserts the **DataOutHold** for a cycle. The coprocessor core holds the data on the **DataOut** bus until the hold signal is deasserted. Also note that the **CPValid** signal is deasserted until the STCL data transfer is over, that is, no new instruction is issued to the coprocessor core until the STCL data transfer to the pipeline follower is over.

[1] For STC, MRC, and MRRC instructions the issue stage in the coprocessor core consists of instruction decode and register read. This can potentially be a speed path depending on the implementation of the coprocessor core. But the logic can be split into two or more cycles by using the **Stall** signal. Initially, during instruction decode the core can assert the **Stall** signal, and in one of the subsequent cycles when data is ready the **Stall** signal can be deasserted.

2.3.2 Instruction cancellation

Instructions issued to the coprocessor core might be cancelled at any time before they are retired, due to events such as branch mispredictions or exceptions in the ARM10 pipeline. Every cycle the pipeline follower sends out tags of cancelled instructions to the coprocessor core using the **InvalidInstTag** signal. The signal is one-hot encoded and it is valid for a single cycle. For example, if coprocessor instructions with tags 7, 0, and 1 are cancelled as a result of a pipeline flush in the ARM10 processor, then a value of 8'b10000011 is driven on the **InvalidInstTag** signal for a single cycle. The coprocessor core must cancel instructions pointed by the **InvalidInstTag** signal in the same cycle.

2.3.3 Instruction retirement

The pipeline follower sends out the tags of coprocessor instructions that retired in the ARM10 pipeline using the **RetiredInstTag** signal. The pipeline follower also sends out tags of instructions owning data on the **DataIn** bus using the **RetiredDataTag** signal. Both the signals carry the tag information in one-hot encoded format.

For instructions that do not involve data transfer from the pipeline follower to the coprocessor core (CDP, MRC, MRRC, STC, and STCL), only the **RetiredInstTag** signal is valid. For MCR, MCRR, and LDC instructions the **RetiredDataTag** is valid in the same cycle as the **RetiredInstTag**. For LDCL (Load multiple) instructions, **RetiredDataTag** is valid in cycles in which data is available on the **DataIn** bus. The **RetiredInstTag** is valid only during the last cycle of burst transfer.

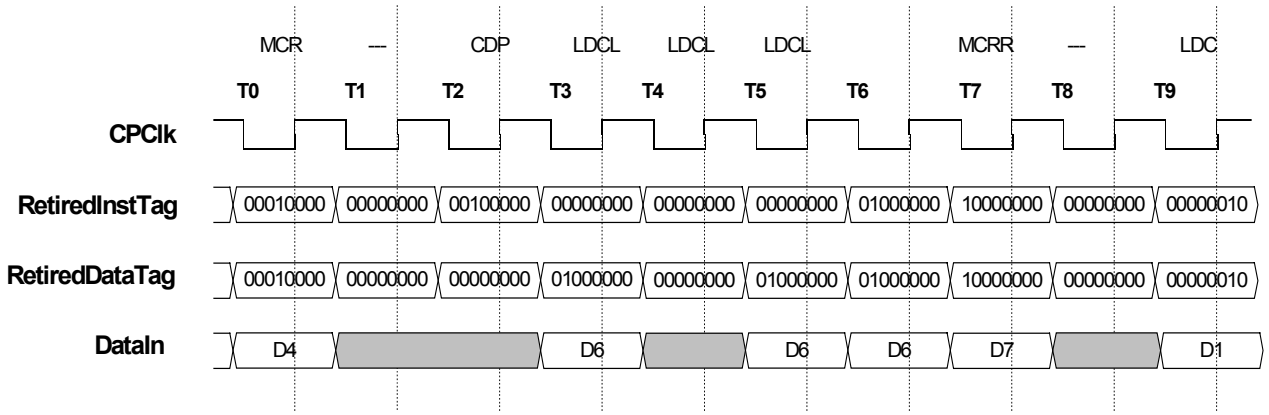

Figure 2.4 Instruction retirement

Figure 2.4 shows an example of instruction retirement. In clock cycle T0, an MCR instruction with a tag of 4 is retired by the pipeline follower. The pipeline follower sends out the instruction's tag in one-hot encoded format (00010000) on **RetiredInstTag**. The tag is sent out also on the **RetiredDataTag** indicating that the data associated with the MCR instruction is available on the **DataIn** bus. In clock cycle T2, a CDP instruction with a tag of 5 is retired by the pipeline follower. The tag is sent out only on **RetiredInstTag** because CDP instruction does not involve any data transfer from the pipeline follower to the coprocessor core. In clock cycle T3, the pipeline follower has the first two words in an LDCL instruction ready. The

pipeline follower sends out the tag of the LDCL instruction on **RetiredDataTag** along with data on the **DataIn** bus. The pipeline follower places the tag of the instruction on **RetiredDataTag**, whenever the data associated with the load multiple is available on the bus. In clock cycle T5, which is the final cycle of the transfer (dictated by the **LSLEN** signal hardwired by the coprocessor core), the pipeline follower places the tag on both **RetiredInstTag** and **RetiredDataTag** indicating the end of transfer. For a normal LDC, the **RetiredDataTag** and **RetiredInstTag** are valid on the same clock cycle as shown in cycle T9.

No instruction in the coprocessor core can be cancelled after it has been retired by the ARM10 pipeline follower. Also, the cancel and retire signals are never asserted simultaneously for any instruction. But, there is a case in which the retire signal is asserted after the cancel signal. This happens during Data Abort on an LDCL (load multiple) instruction. During Data Abort the LDCL instruction's tag is sent out on the **InvalidInstTag** bus. And in the following cycle the tag is also sent out on **RetiredInstTag** and **RetiredDataTag** buses, although the data on the **DataIn** bus is invalid. This feature simplifies the design of the counter that tracks the LDCL instruction in the coprocessor core. The counter is incremented whenever a tag is valid only on the **RetiredDataTag** bus. And, the counter is reset whenever the same tag is valid simultaneously on both the **RetiredInstTag** and **RetiredDataTag** buses (irrespective of whether the instruction was cancelled or not). The coprocessor designers do not have to use this feature and, alternatively, the LDCL counter can be controlled by using only the **InvalidInstTag** and the **RetiredDataTag** buses.

2.4 Attaching coprocessors to the ARM10 core

To connect multiple coprocessors to the ARM10 core the control signals from the coprocessors must be ANDed or ORed together as appropriate. The ARM10 core provides internally the ANDing and ORing required to directly attach the control signals of up to two coprocessors. There are two inputs to the ARM10 core for all of the coprocessor control signals and these are internally combined. When attaching more than two coprocessors external ANDing or ORing of the control signals is required before they reach the ARM10 core. The type of gate that must be used for each signal depends on whether it is an active HIGH or active LOW signal. More details are available in the ARM1026EJ-S Technical Reference Manual. Only a single coprocessor data bus input port is provided, so external gates are required to combine the data buses from multiple coprocessors.

When attaching a single coprocessor, the unused coprocessor inputs to ARM10 must all be carefully tied off. Note that some inputs to the ARM10 core must be tied HIGH and some must be tied LOW depending on whether the input is active HIGH or active LOW. More details are available in the ARM1026EJ-S Technical Reference Manual.

2.5 Supported configurations

The ARM10 pipeline follower can be used in both single and multi-coprocessor configurations. The pipeline follower has three extra signals (**CPBUSYEIN**, **CPLSBUSY**, and **CPLSBUSYIN**) when compared to the ARM10 coprocessor interface, and they are used for handshaking between the coprocessors in a multi-coprocessor configuration. The method of hooking up these signals is shown in Figure 2-5 to Figure 2-7.

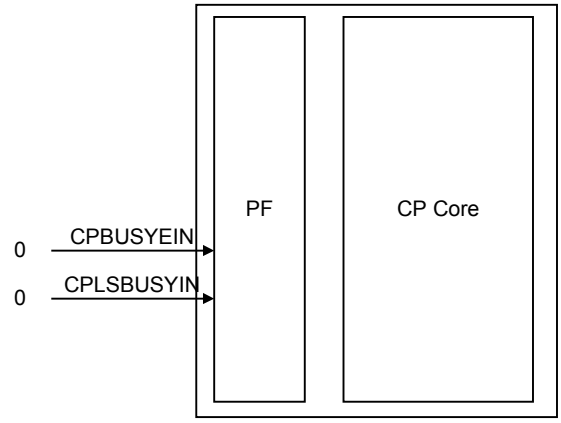

Figure 2.5 Single coprocessor

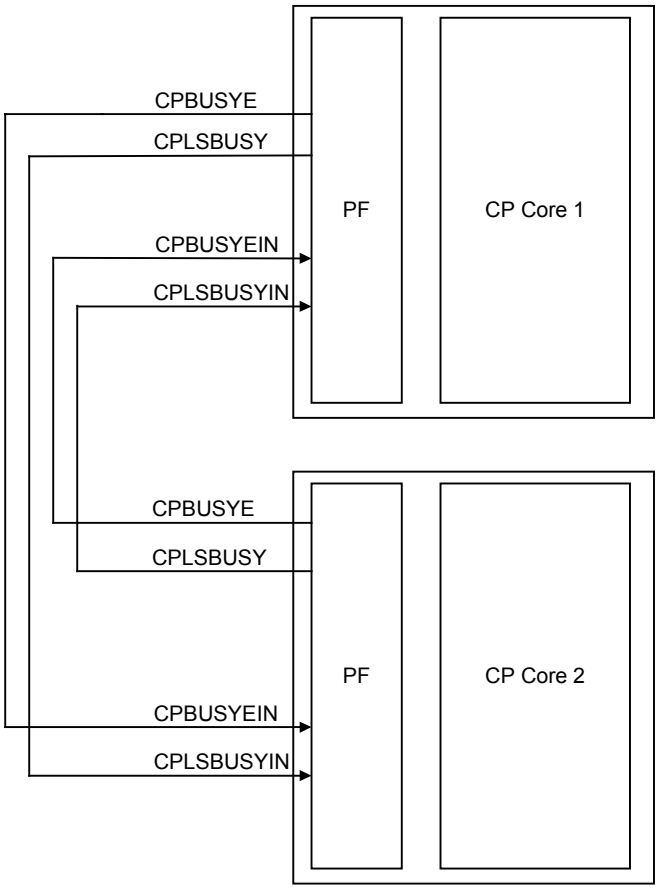

Figure 2.6 Two coprocessors

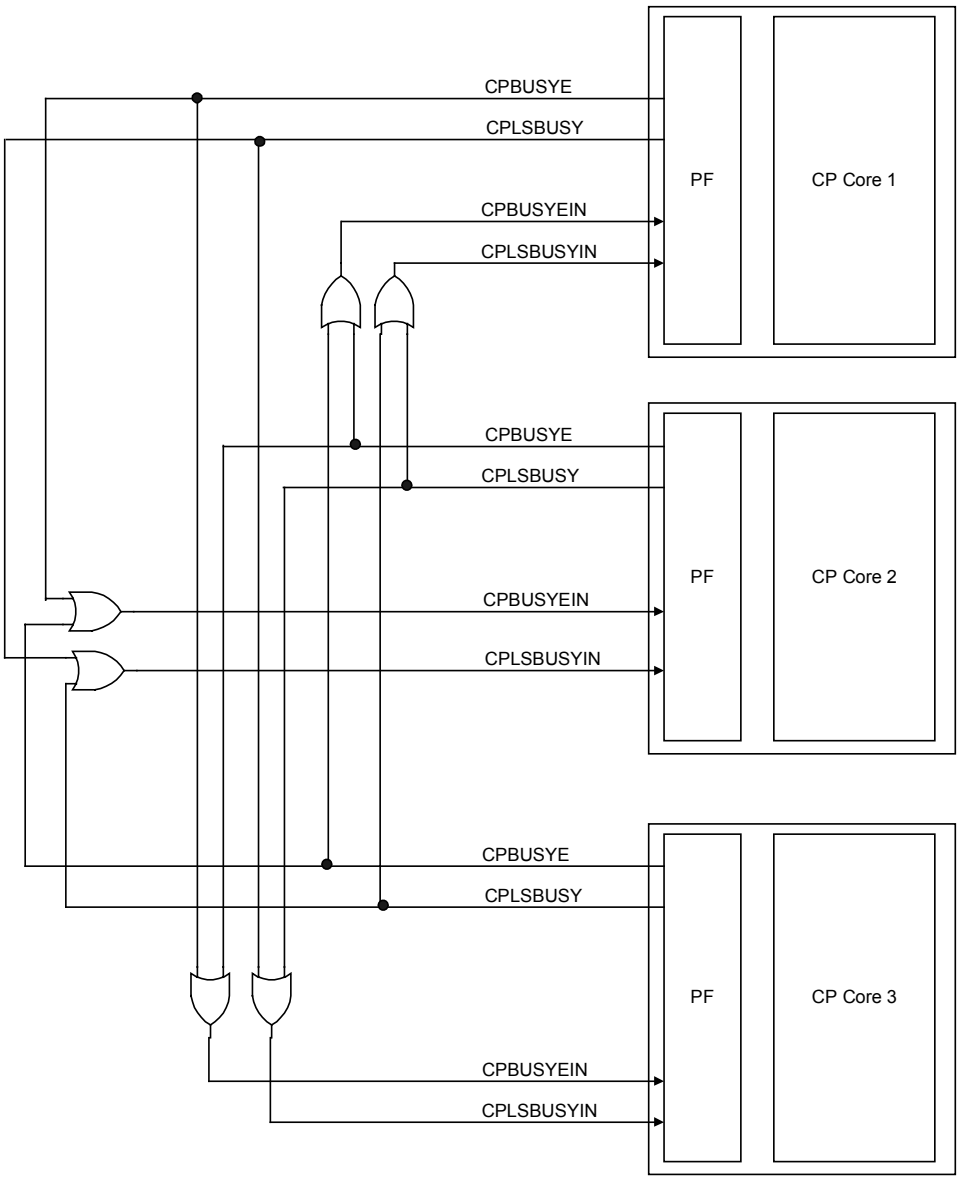

Figure 2.7 Three coprocessors

The same approach is used to hook up more than three coprocessors. For each coprocessor, the **CPBUSYE** signals from the other coprocessors are ORed and connected to the **CPBUSYEIN** signal. Similarly, the **CPLSBUSY** signals from the other coprocessors are ORed and connected to the **CPLSBUSYIN** signal.

3 ARM10 pipeline follower RTL code in Verilog

This section gives an example of an ARM10 pipeline follower that has been extensively tested and synthesized. The design is used in the ARM1026 test chip. It is suggested that, in the interests of minimizing the possibility of introducing subtle bugs, you do not alter this in any way. Ideally it can be instantiated as a sub-module of a coprocessor.

NOTE: In the code flip-flops are represented by module instantiations. These are named as follows:

a10dffx1	A single d-type flip-flop
a10dffx8	An 8-bit wide d-type flip flop
a10gdffx8	An 8-bit wide d-type flip flop with a clock gate input
a10gdffrx8	An 8-bit wide d-type flip flop with a clock gate input and asynchronous reset input

The flip-flop ports have the following ordering:

(Output, Input, Clock, <Gate>, <Reset>)

"< >" signify a port that might not be required

```
//------------------------------------------------------------------------------
//     The confidential and proprietary information contained in this file may
//     only be used by a person authorised under and to the extent permitted
//     by a subsisting licensing agreement from ARM Limited.
//
//            (C) COPYRIGHT 2003 ARM Limited.
//                ALL RIGHTS RESERVED
//
//     This entire notice must be reproduced on all copies of this file
//     and copies of this file may only be made by a person if such person is
//     permitted to do so under the terms of a subsisting license agreement
//     from ARM Limited.
//
//
//------------------------------------------------------------------------------


//# The Pipeline follower interfaces with the ARM10's coprocessor interface on one
//# side and a coprocessor on the other. The utility of having the pipeline follower
//# is that coprocessor designers do not have to understand the ARM10's coprocessor
//# interface. Instead they can use the simple tag based interface provided by the
//# pipeline follower.

//# The pipeline follower supports multi-coprocessor configuration.


module a10PipeFollower(

              CPCLK,
              CPRST,
              CPEN,
              CPINSTR,
              CPINSTRV,
              CPVALIDD,
              ASTOPCPD,
              ASTOPCPE,
              ACANCELCP,
              AFLUSHCP,
              CPBIGEND,
              CPSUPER,
              LSHOLDCPM,
              LSHOLDCPE,
              CPABORT,
              LDCMCRDATA,
              CPBUSYEIN,
              CPLSBUSYIN,
              Stall,
              Bounce,
              DataOut,
```

```
              LSLEN,
              LSSWP,
              LSDBL,
              CPID,


              CPBUSYE,
              CPLSBUSY,
              CPBOUNCEE,
              CPLSLEN,
              CPLSSWP,
              CPLSDBL,
              STCMRCDATA,
              CPInst,
              CPValid,
              CPTag,
              HoldDataOut,
              InvalidInstTag,
              DataIn,
              RetiredInstTag,
              RetiredDataTag
              );


 // Inputs from the ARM
input [63:0]  LDCMCRDATA;
input [25:0]  CPINSTR;
input  [3:0]  CPID;

input         CPCLK;
input         CPRST;
input         CPINSTRV;
input         CPVALIDD;
input         ASTOPCPD;
input         ASTOPCPE;
input         ACANCELCP;
input         AFLUSHCP;
input         CPBIGEND;
input         CPSUPER;
input         LSHOLDCPM;
input         LSHOLDCPE;
input         CPABORT;
input         CPBUSYEIN;
input         CPLSBUSYIN;

// Inputs from the coprocessor
input [63:0]  DataOut;
input  [5:0]  LSLEN;
input         Stall;
input         Bounce;
input         LSDBL;
input         LSSWP;

// Coprocessor enable
input         CPEN;


// Outputs to the ARM
output [63:0] STCMRCDATA;
output  [5:0] CPLSLEN;
output        CPBUSYE;
output        CPLSBUSY;
output        CPBOUNCEE;
output        CPLSSWP;
output        CPLSDBL;


// Outputs to the coprocessor
output [63:0] DataIn;
output [25:0] CPInst;
output  [7:0] InvalidInstTag;
output  [7:0] RetiredInstTag;
output  [7:0] RetiredDataTag;
output  [2:0] CPTag;
output        HoldDataOut;
output        CPValid;



wire   [63:0] STCMRCDATA;
wire   [63:0] LDCMCRDATA;
wire   [63:0] DataOutReg;
```

```
wire   [63:0] DataOut;
wire   [63:0] DataIn;
wire   [25:0] CPINSTR;
wire   [25:0] CPInstIs;
wire   [25:0] CPInstDe;
wire   [25:0] CPInstEx;
wire   [25:0] CPInstMe;
wire   [25:0] CPInstWr;
wire   [25:0] CPInst;
wire   [25:0] CPInstOld;
wire    [7:0] CPControlDe;
wire    [7:0] CPControlEx;
wire    [7:0] CPControlMe;
wire    [7:0] CPControlWr;
wire    [7:0] Control;
wire    [7:0] TagDe;
wire    [7:0] TagEx;
wire    [7:0] TagMe;
wire    [7:0] InvalidInstTag;
wire    [7:0] RetiredInstTag;
wire    [7:0] RetiredDataTag;
wire    [5:0] LSLEN;
wire    [5:0] LSLen1;
wire    [5:0] LSLen2;
wire    [5:0] CPLSLEN;
wire    [5:0] BurstLen;
wire    [4:0] BurstCnt;
wire    [4:0] NextBurstCnt;
wire    [2:0] CPTagDe;
wire    [2:0] CPTagEx;
wire    [2:0] CPTagMe;
wire    [2:0] CPTagWr;
wire    [2:0] CPTag;
wire    [2:0] CPTagOld;
wire    [2:0] CurrentTag;
wire    [2:0] NextTag;
wire          State;
wire          NState;
wire          CPValidIs;
wire          CPValidDe;
wire          CPValidEx;
wire          CPValidMe;
wire          CPValidWr;
wire          CPValid;
wire          EnInstFe;
wire          EnInstIs;
wire          EnInstDe;
wire          EnInstEx;
wire          EnInstMe;
wire          EnValidFe;
wire          EnValidIs;
wire          EnValidDe;
wire          EnValidEx;
wire          EnValidMe;
wire          nHoldFe;
wire          nHoldIs;
wire          nHoldDe;
wire          nHoldEx;
wire          nHoldMe;
wire          NextCPBUSYE;
wire          EnCPBUSYE;
wire          NextCPBOUNCEE;
wire          EnCPBOUNCEE;
wire          CPBOUNCEE;
wire          Stall;
wire          MCRInst;
wire          MRCInst;
wire          MCRRInst;
wire          MRRCInst;
wire          LDCInst;
wire          STCInst;
wire          LDCMInst;
wire          STCMInst;
wire          EnOld;
wire          LSDBL;
wire          LSSWP;
wire          nHold;
wire          LdBurstCntr;
wire          BurstHold;
wire          BurstHoldReg;
wire          EnBurstCntr;
wire          HoldDataOut;
```

```
wire          LSMInterlock;
wire          InstValid;
wire          Stall1;
wire          Stall2;
wire          ValidIs;
wire          CPCntrl;
wire          CntrlOld;


function [7:0] OneHotEncoder;
input [2:0]     Index;
begin
   case(Index)
      3'b000: OneHotEncoder = 8'b00000001;
      3'b001: OneHotEncoder = 8'b00000010;
      3'b010: OneHotEncoder = 8'b00000100;
      3'b011: OneHotEncoder = 8'b00001000;
      3'b100: OneHotEncoder = 8'b00010000;
      3'b101: OneHotEncoder = 8'b00100000;
      3'b110: OneHotEncoder = 8'b01000000;
      3'b111: OneHotEncoder = 8'b10000000;
      default : OneHotEncoder = {8{1'bx}};
   endcase
end
endfunction


//*************************************************************
//                       PIPELINE FOLLOWER
//*************************************************************

//# CO-PROCESSOR FETCH

//# Stall the FETCH on ASTOPCPD or ASTOPCPE or CPBUSYE or CPBUSYEIN
//# AFLUSHCP has the highest priority, and it resets the VALID bit.

assign nHoldFe    = nHoldIs;
assign EnInstFe   = (CPINSTR[11:8]==CPID)&CPINSTRV&CPEN&(!AFLUSHCP)&nHoldFe;
assign EnValidFe  = nHoldFe|AFLUSHCP;

a10gdffrx26  uFetchInst(CPInstIs,CPINSTR,CPCLK,EnInstFe,CPRST);
a10gdffrx1   uFetchValid(CPValidIs,EnInstFe,CPCLK,EnValidFe,CPRST);



//# CO-PROCESSOR ISSUE

//# Stall the ISSUE on ASTOPCPD or ASTOPCPE or CPBUSYE or CPBUSYEIN
//# AFLUSHCP has the highest priority, and it resets the VALID bit.

//# CPVALIDD signal qualifies the instruction in the FETCH/ISSUE latch
//# of the pipeline follower.

assign ValidIs    = CPVALIDD ? CPValidIs : 1'b0;

assign nHoldIs    = !(ASTOPCPD|ASTOPCPE|CPBUSYE|CPBUSYEIN);
assign EnInstIs   = ValidIs&(!AFLUSHCP)&nHoldIs;
assign EnValidIs  = nHoldIs|AFLUSHCP|((!nHoldIs)&nHoldDe);

// synopsys translate_off
a10gdffx26  uIssueInst(CPInstDe,CPInstIs,CPCLK,EnInstIs);
// synopsys translate_on
a10gdffrx1  uIssueValid(CPValidDe,EnInstIs,CPCLK,EnValidIs,CPRST);
a10gdffx8   uIssueControl(CPControlDe,Control,CPCLK,EnInstIs);
a10gdffx3   uIssueTag(CPTagDe,CurrentTag,CPCLK,EnInstIs);



//# CO-PROCESSOR DECODE

//# Stall the DECODE on ASTOPCPD or ASTOPCPE or CPBUSYE
//# AFLUSHCP has the highest priority, and it resets the VALID bit.

assign nHoldDe    = !(ASTOPCPD|ASTOPCPE|CPBUSYE);
assign EnInstDe   = CPValidDe&(!AFLUSHCP)&nHoldDe;
assign EnValidDe  = nHoldDe|AFLUSHCP|((!nHoldDe)&nHoldEx);

// synopsys translate_off
a10gdffx26  uDecodeInst(CPInstEx,CPInstDe,CPCLK,EnInstDe);
// synopsys translate_on
a10gdffrx1  uDecodeValid(CPValidEx,EnInstDe,CPCLK,EnValidDe,CPRST);
a10gdffx8   uDecodeControl(CPControlEx,CPControlDe,CPCLK,EnInstDe);
```

```
a10gdffx3   uDecodeTag(CPTagEx,CPTagDe,CPCLK,EnInstDe);


//# CO-PROCESSOR EXECUTE

//# Stall the EXECUTE on ASTOPCPE or LSHOLDCPE or LSHOLDCPM.
//# AFLUSHCP and ACANCELCP have the highest priority, and they reset the VALID bit.


assign nHoldEx    = !(ASTOPCPE|LSHOLDCPE|LSHOLDCPM|BurstHold);
assign EnInstEx   = CPValidEx&nHoldEx&(!(ACANCELCP|AFLUSHCP));
assign EnValidEx  = nHoldEx|(AFLUSHCP&(!BurstHold))|((!nHoldEx)&nHoldMe&(!BurstHold));

// synopsys translate_off
a10gdffx26  uExeInst(CPInstMe,CPInstEx,CPCLK,EnInstEx);
// synopsys translate_on
a10gdffrx1  uExeValid(CPValidMe,EnInstEx,CPCLK,EnValidEx,CPRST);
a10gdffx8   uExeControl(CPControlMe,CPControlEx,CPCLK,EnInstEx);
a10gdffx3   uExeTag(CPTagMe,CPTagEx,CPCLK,EnInstEx);



//# The LDCM/STCM instructions are held in the MEM stage (EXE/MEM latch)
//# of the pipeline follower until the burst transfer is over. CPValidMe,
//# CPTagMe and CPControlMe are also held until the burst transfer is over.


assign LdBurstCntr  = EnInstEx&CPControlEx[2];
assign EnBurstCntr  = !(LSHOLDCPE|LSHOLDCPM);
assign NextBurstCnt = LdBurstCntr ? 5'b0_0001:(BurstCnt+ 5'b0_0001);
assign BurstHold    = (BurstCnt<CPControlMe[7:3])&CPControlMe[2]&CPValidMe;


a10gdffrx5  uBurstCnt(BurstCnt,NextBurstCnt,CPCLK,EnBurstCntr,CPRST);
a10gdffrx1  uBurstHold(BurstHoldReg,BurstHold,CPCLK,1'b1,CPRST);



//# CO-PROCESSOR MEM

//# Stall the MEM on LSHOLDCPM.

assign nHoldMe    = BurstHold ? !(LSHOLDCPE|LSHOLDCPM) :!LSHOLDCPM;
assign EnInstMe   = CPValidMe&nHoldMe&(!CPABORT);
assign EnValidMe  = 1'b1;

// synopsys translate_off
a10gdffx26  uMemInst(CPInstWr,CPInstMe,CPCLK,EnInstMe);
// synopsys translate_on
a10gdffrx1  uMemValid(CPValidWr,EnInstMe,CPCLK,EnValidMe,CPRST);
a10gdffx8   uMemControl(CPControlWr,CPControlMe,CPCLK,EnInstMe);
a10gdffx3   uMemTag(CPTagWr,CPTagMe,CPCLK,EnInstMe);



//# CO-PROCESSOR WRITE



//*****************************************************************
//                       INSTRUCTION DECODE
//*****************************************************************


//# MCR  :  CPInstIs[25] = 1, CPInstIs[4]  = 1, CPInstIs[20] = 0
//# MRC  :  CPInstIs[25] = 1, CPInstIs[4]  = 1, CPInstIs[20] = 1
//# MRRC :  CPInstIs[25:20] = 6'b000101
//# MCRR :  CPInstIs[25:20] = 6'b000100
//# LDC  :  CPInstIs[25] = 0, CPInstIs[20] = 1
//# STC  :  CPInstIs[25] = 0, CPInstIs[20] = 0
//# LDCM :  CPInstIs[25] = 0, CPInstIs[20] = 1, CPInstIs[22] = 1
//# STCM :  CPInstIs[25] = 0, CPInstIs[20] = 0, CPInstIs[22] = 1

 assign MCRInst   = CPInstIs[25]&CPInstIs[4]&(!CPInstIs[20]);
 assign MRCInst   = CPInstIs[25]&CPInstIs[4]&CPInstIs[20];
 assign MRRCInst  = (CPInstIs[25:20]==6'b000101);
 assign MCRRInst  = (CPInstIs[25:20]==6'b000100);
 assign LDCInst   = (!CPInstIs[25])&CPInstIs[20]&(!MRRCInst);
```

```
 assign LDCMInst  = LDCInst&CPInstIs[22];
 assign STCInst   = (!CPInstIs[25])&(!CPInstIs[20])&(!MCRRInst);
 assign STCMInst  = STCInst&CPInstIs[22];
 assign BurstLen  = (CPLSLEN[5:0]+6'b00_0001);

 //# Control[0]   - Data transfer from coprocessor to ARM
 //# Control[1]   - Data transfer from ARM to coprocessor
 //# Control[2]   - Load/Store Multiple
 //# Control[7:3] - Load/Store Multiple Burst Length

 assign Control = {BurstLen[5:1],LDCMInst|STCMInst,MCRInst|MCRRInst|LDCInst,
                   MRCInst|MRRCInst|STCInst};

 assign nHold   = !(ASTOPCPD|ASTOPCPE);


//*****************************************************************
//                  COPROCESSOR INTERFACE SIGNALS
//*****************************************************************

//# The pipeline follower, in the ISSUE stage, passes a coprocessor instruction
//# along with the valid signal and the tag to the coprocessor.


assign EnOld   = !(CPBUSYE|ASTOPCPD|ASTOPCPE);
a10gdffx26  uCPInstOld(CPInstOld,CPInstIs,CPCLK,EnOld);
a10gdffx3   uCPTagOld(CPTagOld,CurrentTag,CPCLK,EnOld);

assign CPInst        = CPBUSYE  ? CPInstOld : CPInstIs;
assign CPTag         = CPBUSYE  ? CPTagOld  : CurrentTag;
assign InstValid     = (CPBUSYE ? 1'b1: ValidIs)&(!AFLUSHCP);
assign CPValid       = InstValid&(!LSMInterlock)&(!CPLSBUSYIN)&(!CPBUSYEIN)&nHold;

assign HoldDataOut   = (CPValidDe&CPControlDe[0]&(ASTOPCPD|ASTOPCPE)&(!CPBUSYE))|
                       (CPValidEx&CPControlEx[2]&(!nHoldEx))|
                       (CPValidMe&CPControlMe[2]&(!nHoldMe));



//*****************************************************************
//                     INTERLOCK
//*****************************************************************


//# Once an LDCL/STCL instruction reaches the EXECUTE stage, the
//# DECODE, ISSUE, and FETCH stages of the pipeline follower are
//# held until the LDCL/STCL instruction completes data transfer.
//# Absence of this interlock sometimes lead to pipeline hazards.


//# Valid LDCL/STCL instruction in DE/EX/ME/WR stages
assign LSMInterlock = (CPValidWr&CPControlWr[2])|
                      (CPValidMe&CPControlMe[2])|
                      (CPValidEx&CPControlEx[2])|
                     (CPValidDe&CPControlDe[2]&(!CPBUSYE));


//*****************************************************************
//                      ARM10 INTERFACE SIGNALS
//*****************************************************************

//# CPBUSYE (registered output) generated in the ISSUE/DECODE boundary.
//# Coprocessor generates 'Stall' signal if it cannot accept a new
//# instruction into its DECODE stage due to data dependencies.

//# Pipeline follower generates 'LSMInterlock' signal if there
//# is an LDCL/STCL instruction in the pipeline. Upstream instructions
//# have to wait until the LDCL/STCL completes data transfer.


assign Stall1        = Stall;
assign Stall2        = LSMInterlock|CPLSBUSYIN;
assign NextCPBUSYE   = InstValid&(!CPBUSYEIN)&(Stall1|Stall2);
assign EnCPBUSYE     = nHold|AFLUSHCP;
a10gdffrx1 uCPBUSYE(CPBUSYE,NextCPBUSYE,CPCLK,EnCPBUSYE,CPRST);



//# CPBOUNCEE (registered output) generated in the ISSUE/DECODE boundary
//# Instruction validity is driven by the decode unit in the ISSUE stage.

assign NextCPBOUNCEE = InstValid&(!CPBUSYEIN) ? Bounce : 1'b1;
```

```
assign EnCPBOUNCEE   = nHold|AFLUSHCP;
a10gdffrx1 uCPBOUNCEE(CPBOUNCEE,NextCPBOUNCEE,CPCLK,EnCPBOUNCEE,CPRST);



//# STCMRCDATA bus driven in the ISSUE/DECODE boundary

assign CPCntrl    = CPBUSYE  ? CntrlOld : Control[0];
assign DataOutReg = (CPValid&(!Stall)&CPCntrl)|LSMInterlock ? DataOut : 64'd0;

a10gdffx1   uCPCntrlOld(CntrlOld,Control[0],CPCLK,EnOld);
a10gdffx64  uSTCMRCDATA(STCMRCDATA,DataOutReg,CPCLK,!HoldDataOut);



//# LDCMCRDATA bus is sampled in every cycle
a10gdffx64  uLDCMCRDATA(DataIn,LDCMCRDATA,CPCLK,1'b1);


//# Load/Store-Multiple control signals
//# LSLEN, LSSWP and LSDBL are driven (hardwired) by  the coprocessor.

//# The signals have to be driven in the pipeline follower ISSUE stage
//# and ARM10 DECODE stage.

//# In ARM10, even a normal LDC/STC with an addressing mode that produces
//# a sequence of addresses (Ex. immediate pre-indexed) produces a burst
//# access. CPLSLEN has to be made 1 to prevent this. CPLSLEN should be
//# given a value greater than 1 only in the case of LDCL/STCLs. Bit 22
//# differentiates LDCL/STCLs from LDC/STCs. The signal LSLen2 is generated
//# in the FETCH stage of the pipeline follower since CPLSLEN has to be
//# driven out in the ISSUE stage.


//# Normally, LSLEN driven (hardwired) by the coprocessor core is used
//# to generate the LSLen1 signal.
//# Ex. assign LSLen1 = CPINSTR[22] ? LSLEN : 6'b00_0001;
//# But, for the validation coprocessor, the LDCL/STCL burst length is
//# encoded within the instruction (CPINSTR[7:5]).

assign LSLen1 = CPINSTR[22] ? LSLEN : 6'b00_0001;
//# assign LSLen1 = CPINSTR[22] ? {3'b000,CPINSTR[7:5]} + 6'b00_0001 : 6'b00_0001;
assign LSLen2 = (CPINSTR[11:8]==CPID)&CPINSTRV&CPEN ? LSLen1 : 6'd0;


a10gdffx6 uCPLSLEN(CPLSLEN,LSLen2,CPCLK,nHoldFe);
a10dffx1  uCPLSSWP(CPLSSWP,LSSWP,CPCLK);
a10dffx1  uCPLSDBL(CPLSDBL,LSDBL,CPCLK);

assign CPLSBUSY   = LSMInterlock;

//*************************************************************
//                    TAG GENERATION
//*************************************************************

assign NextTag = CurrentTag + 3'b001;
a10gdffrx3 uTagCounter(CurrentTag,NextTag,CPCLK,EnInstIs,CPRST);


assign TagDe             = CPValidDe&(!CPBUSYE)&AFLUSHCP  ? OneHotEncoder(CPTagDe):{8{1'b0}};
assign TagEx             = CPValidEx&(AFLUSHCP|ACANCELCP) ? OneHotEncoder(CPTagEx):{8{1'b0}};
assign TagMe             = CPValidMe&(CPABORT)            ? OneHotEncoder(CPTagMe):{8{1'b0}};

assign InvalidInstTag    = TagDe|TagEx|TagMe;

//# Tag is sent out whenever an instruction retires. For a load/store multiple instruction
//# this happens in the last cycle of the burst transfer.

assign RetiredInstTag    = CPValidWr&(!BurstHoldReg) ? OneHotEncoder(CPTagWr):{8{1'b0}};

//# Tag is sent out if data is valid on the DataIn bus. In the case of a load multiple
//# instruction, the same tag is sent out multiple times - whenever the load data is
//# valid on the bus.

assign RetiredDataTag    = CPValidWr&CPControlWr[1] ? OneHotEncoder(CPTagWr):{8{1'b0}};

endmodule
```

4 Reference Design

This chapter describes the design of a Validation CoProcessor (VCP) used for testing the ARM1026EJ-S coprocessor interface. It is provided here as an example use of the ARM10 pipeline follower. A coprocessor targeted for validation might not be an ideal starting point for designing a high-performance data processing coprocessor, but it does have the advantage that it is well tested and exhibits a wide range of behavior on the interface. The VCP supports all types of coprocessor instructions (CDP, MRC, MCR, MCRR, MRCC, LDC, LDCL, STC, and STCL instructions). It was synthesized to core speed and used in FPGA and test chip validation.

4.1 Functionality and instruction encoding

Coprocessors are normally designed to support complex data processing operations. But, the emphasis of the VCP is to validate ARM10's coprocessor interface by exercising the various interface signals rigorously. The VCP has 8 general-purpose registers (CR0-CR7). Register CR7 also serves a configuration register. (Note: Clearly it is not elegant to use a general-purpose register as a configuration register. In the special circumstances of a coprocessor designed for validation it makes sense because it keeps the design simple.)

The VCP core can deal with issued instructions in the following ways:

- accept immediately
- accept after a specific number of busy-wait cycles
- bounce immediately
- bounce after a specific number of busy-wait cycles.

Because the VCP contains only 8 registers, the MSB of the coprocessor register identifier in the instruction is not used. Instead, this bit is used in the VCP to decide whether to bounce the instruction or not. For CDP, MCR, MRC, MCRR, and MRRC instructions fields within the instructions contain the busy-wait counts. For STC, STCL, LDC, and LDCL instructions, coprocessor register CR7 contains the busy-wait counts. The burst length for an LDCL/STCL instruction is also encoded within the instruction. The encoding is summarized below.

Instruction	Busy-wait count	Bounce	Length
CDP coproc, opcode_1, CRd, CRn, CRm, opcode_2	opcode_1[3:0]	CRd[3]	-
MRC coproc, opcode_1, Rd, CRn, CRm, opcode_2	CRm[3:0]	CRn[3]	-
MCR coproc, opcode_1, Rd, CRn, CRm, opcode_2	CRm[3:0]	CRn[3]	-
MRRC coproc, opcode, Rd, Rn, CRm	opcode	CRm[3]	-
MCRR coproc, opcode, Rd, Rn, CRm	opcode	CRm[3]	-
LDC coproc, CRd, addressing mode	CR7[11:8]	CRd[3]	-
STC coproc, CRd, addressing mode	CR7[3:0]	CRd[3]	-
LDCL coproc, CRd, addressing mode	CR7[15:12]	CRd[3]	offset[7:5] +1
STCL coproc, CRd, addressing mode	CR7[7:4]	CRd[3]	offset[7:5] +1

CDP instruction is implemented as a register move from CRn to CRd. LDC instructions with the offset[7] bit 1 have side effects[2] to enable testing that **CPABORT** functions correctly. (Note: LDC and LDCL instructions with side effects are strongly discouraged because it is unlikely that any coprocessors that use them will be adaptable for use with future ARM microprocessors). Along with loading the value to the specified register, the subsequent register gets incremented by 1. If CRd is 7, then register 0 gets incremented.

4.2 Design

The VCP consists of the ARM10 pipeline follower and a core specific to the functionality described above. The details are shown on Figure 4.1.

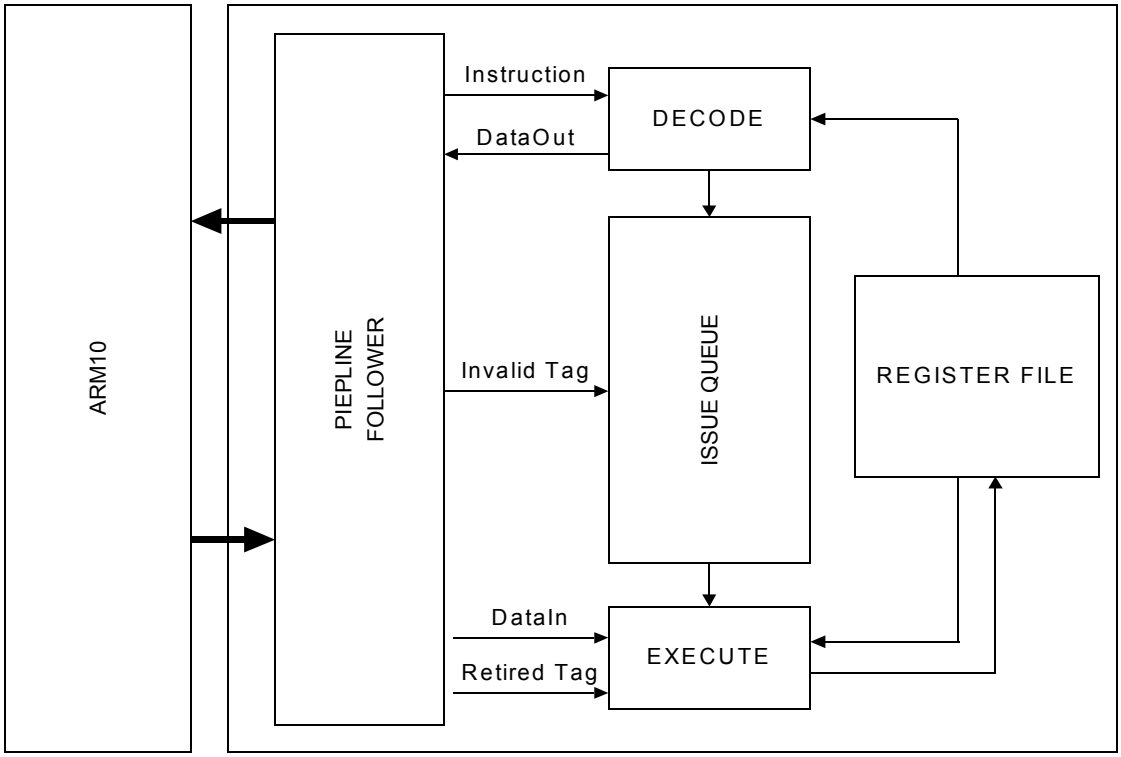

Figure 4.1 Validation CoProcessor (VCP)

The VCP core does nothing but track the ARM10 pipeline using the simple interface provided by the pipeline follower until it is sure that a given instruction will not be cancelled, and only then does it start to execute it. High-performance cores can speculatively use the issued instructions

[2] That is, these instructions not only load a value into the specified register or registers but also cause the contents of other configuration or general-purpose registers to be altered. A simple load instruction can be re-executed with no ill effects. A load instruction with a side effect that, say, increments another register cannot be re-executed without the re-execution being visible to software through a double increment of the value in the register.

even before they are retired in the ARM10 pipeline. These have the potential for higher performance but will be more complex.

4.2.1 Pipeline organization

The VCP core has three pipeline stages – Decode, Issue, and Execute. The pipeline organization is shown in Figure 4.1.

Decode

In the Decode stage the VCP core decodes the instruction on the **CPInst** bus. The **CPValid** signal becomes stable only toward the middle of the clock cycle. Therefore, instruction decode starts at the beginning of every clock cycle irrespective of **CPValid**. During the decode of an instruction, the **Stall** signal is asserted for the number of cycles specified by the busy-wait count field, followed by the **Bounce** signal based on the bounce field. A stall-counter in the Decode stage keeps track of the busy-wait count. The counter is reset to zero when an instruction is decoded for the first time, and it is incremented every time the same instruction is decoded again (same **CPInst** and **CPTag** as the previous cycle). The **Stall** signal is asserted for as long as the stall-counter value is less than the busy-wait count field in the instruction.

If both **Stall** and **Bounce** signals are LOW and **CPValid** is HIGH toward the end of the cycle, the Decode stage registers the instruction and writes it to the issue queue. For instructions that involve data transfer from the core to the pipeline follower (MRC, MRRC, STC, and STCL), the data is driven in the same cycle in which the Decode stage writes the instruction into the issue queue. The data to be driven out might depend on some outstanding instruction in the VCP core. If that is the case, then the **Stall** signal is asserted until all outstanding instructions in the VCP core have completed their execution.

For an STCL (store multiple) instruction, the first doubleword is driven out in the same cycle in which the instruction is written to the issue queue. The remaining data is driven out in the subsequent cycles. In case the pipeline follower is unable to accept the data in some cycle, it asserts the **DataOutHold** signal, and the Decode stage holds that data on the **DataOut** bus.

Issue

Instructions issued by the Decode stage are written to the Issue queue. The Issue queue can hold up to 8 instructions. Issued instructions wait until they are read by the Execute stage.

Every cycle the issue stage samples the **InvalidInstTag** signal. The instructions in the Issue queue whose tag matches with the tag encoded in the **InvalidInstTag** are cancelled.

Execute

The Execute stage decodes the **RetiredInstTag** and **RetiredDataTag** every cycle.

When an instruction involving data transfer from the VCP core to the pipeline follower (MRC, MRRC, STC, and STCL) is retired by the pipeline follower, the tag is driven on the **RetiredInstTag** signal. The data transfer from the VCP core is already done at the Decode stage, and therefore the Execute stage cancels the instruction from the issue queue.

When a CDP instruction is retired by the pipeline follower, its tag is driven on **RetiredInstTag**. The Execute stage reads the CDP instruction from the issue queue, reads the value of register CRn, and writes the same value to register CRd.

When an instruction involving data transfer from the pipeline follower to the VCP core (MCR, MCRR, and LDC) is retired by the pipeline follower, the tag is driven on both **RetiredInstTag** and **RetiredDataTag** in the same cycle. **RetiredDataTag** is an indication that data is available on the **DataIn** bus. The Execute stage reads the associated instruction, writes the data from **DataIn** bus into the register file, and then cancels the instruction from the Issue queue. If the

offset[7] bit of an LDC instruction is 1, then along with loading the value to register CRd, the subsequent register is incremented by 1.

The execution of an LDCL (load multiple) instruction is slightly different. The tag of the LDCL instruction is driven on **RetiredDataTag** whenever the data associated with the instruction is available on the **DataIn** bus. And, only in the last cycle of the burst transfer is the tag driven on both the **RetiredInstData** and the **RetiredDataTag**. On seeing the LDC instruction's tag on the **RetireDataTag** the first time, the Execute stage reads the instruction from the Issue queue and writes the **DataIn** value to the register file. Also, a counter is loaded with the CRd value. The counter is incremented whenever the LDCL tag appears on **RetiredDataTag** next, and the data gets written into subsequent registers. In the last cycle of the transfer, the tag is driven on both **RetiredInstTag** and **RetiredDataTag**. The Execute stage writes the last register and cancels the instruction from the Issue queue.

4.2.2 ARM10 validation coprocessor example RTL in Verilog

This section contains the RTL for the VCP core, the VCP top-level which instantiates the pipeline follower and the core, and the VCP hook-up with the ARM1026EJ-S processor.

The ARM10 pipeline follower RTL given in Chapter 3 is used for the validation coprocessor. However, there is a minor modification required to the RTL. The pipeline follower uses the **LSLEN** signal driven by the coprocessor core as the length of the LDCL/STCL transfer. For the validation coprocessor the transfer length is encoded within the instruction. Therefore, the assignment

assign LSLen1 = CPINSTR[22] ? LSLEN : 6'b00_0001;

should be replaced by

assign LSLen1 = CPINSTR[22] ? {3'b000, CPINSTR[7:5]} + 6'b00_0001 : 6'b00_0001;

- **VCP Core**

```
//-------------------------------------------------------------------------------
//     The confidential and proprietary information contained in this file may
//     only be used by a person authorised under and to the extent permitted
//     by a subsisting licensing agreement from ARM Limited.
//
//            (C) COPYRIGHT 2003 ARM Limited.
//                ALL RIGHTS RESERVED
//
//     This entire notice must be reproduced on all copies of this file
//     and copies of this file may only be made by a person if such person is
//     permitted to do so under the terms of a subsisting license agreement
//     from ARM Limited.
//
//
//
//-------------------------------------------------------------------------------


//# The validation coprocessor is based on the ARM10 pipeline follower. It uses
//# a simple tag based interface provided by the pipeline follower. It supports all
//# types of coprocessor instructions - CDP, MRC, MCR, MRRC, MCRR, LDC, STC, LDCL
//# and STCL instructions. CDP is implemented as a move between two registers.
//# The validation coprocessor has a register file with 8 entries.

//# The validation coprocessor can deal with incoming instructions in the
//# following ways:
//# a) accept immediately
//# b) accept after certain busy-wait cycles
//# c) bounce immediately
//# d) bounce after certain busy-wait cycles
```

```
//# The decision to bounce the incoming instruction is encoded within the instruction.
//# The number of cycles to busy-wait can be encoded within the instruction (CDP, MRC,
//# MCR, MCRR, MRRC), or it is read from coprocessor register CR7 (LDC, STC, LDCL, and
//# STCL). The burst length for the LDCL/STCLs are encoded within the instruction. The
//# details are given below.


//#-------------------------------------------------------------------
//# Instruction      Busy-wait count     Bounce       Burst length
//#-------------------------------------------------------------------
//# CDP              opcode_1[3:0]       CRd[3]
//# MRC              CRm[3:0]            CRn[3]
//# MCR              CRm[3:0]            CRn[3]
//# MRRC             opcode              CRm[3]
//# MCRR             opcode              CRm[3]
//# STC              CR7[3:0]            CRd[3]
//# STCL             CR7[7:4]            CRd[3]         Offset[7:5]+1
//# LDC              CR7[11:8]           CRd[3]
//# LDCL             CR7[15:12]          CRd[3]         Offset[7:5]+1
//#-------------------------------------------------------------------

//# LDC instruction with Offset[7] bit 1 has a side effect. Along with loading
//# the value to the specified register, the value in the subsequent register is
//# incremented by 1.

module a10VCPCore(
                   CPInst,
                  CPTag,
                  CPValid,
                  InvalidInstTag,
                  RetiredInstTag,
                  RetiredDataTag,
                  HoldDataOut,
                  DataIn,
                  CPCLK,
                  CPRST,



                  Bounce,
                  Stall,
                  DataOut,
                  LSLEN,
                   LSSWP,
                   LSDBL
                 );



input  [63:0] DataIn;
input  [25:0] CPInst;
input   [7:0] InvalidInstTag;
input   [7:0] RetiredInstTag;
input   [7:0] RetiredDataTag;
input   [2:0] CPTag;
input         CPValid;
input         HoldDataOut;
input         CPCLK;
input         CPRST;


output [63:0] DataOut;
output  [5:0] LSLEN;
output        LSSWP;
output        LSDBL;
output        Stall;
output        Bounce;




wire   [63:0] DataOut;
wire   [63:0] DataIn;
wire   [31:0] ReadData1;
wire   [31:0] ReadData2;
wire   [31:0] ReadData3;
wire   [31:0] ReadData4;
wire   [31:0] ConfigReg;
```

```
wire   [25:0] Entry0;
wire   [25:0] Entry1;
wire   [25:0] Entry2;
wire   [25:0] Entry3;
wire   [25:0] Entry4;
wire   [25:0] Entry5;
wire   [25:0] Entry6;
wire   [25:0] Entry7;
wire   [25:0] CPInst;
wire   [25:0] RetiredInst;
wire   [25:0] OldCPInst;
wire    [7:0] Remove;
wire    [7:0] InvalidInstTag;
wire    [7:0] RetiredInstTag;
wire    [7:0] RetiredDataTag;
wire    [7:0] Retired;
wire    [7:0] OldTag;
wire    [7:0] EnValid;
wire    [7:0] EnEntry;
wire    [7:0] Valid;
wire    [7:0] NextTag;
wire    [7:0] STCLTag;
wire    [7:0] STCLEn;
wire    [7:0] Config;
wire    [7:0] NextConfig;
wire    [5:0] LSLEN;
wire    [3:0] StallCount;
wire    [3:0] NextCount;
wire    [2:0] LDCLCntReg;
wire    [2:0] LDCLCnt;
wire    [2:0] ReadAddr1;
wire    [2:0] ReadAddr2;
wire    [2:0] ReadAddr3;
wire    [2:0] ReadAddr4;
wire    [2:0] BAddr1;
wire    [2:0] BAddr2;
wire    [2:0] RAddr1;
wire    [2:0] RAddr2;
wire    [2:0] LDCLAddr1;
wire    [2:0] LDCLAddr2;
wire    [2:0] CPTag;
wire    [2:0] OldCPTag;
wire    [2:0] STCLStartReg;
wire    [2:0] STCLBurstLen;
wire    [2:0] STCLCnt;
wire    [2:0] NextCnt;
wire          LDCLCmp;
wire          Stall;
wire          RetiredValid;
wire          LSSWP;
wire          LSDBL;
wire          RetiredMRRC;
wire          RetiredLDC;
wire          MCRRInst;
wire          STCInst;
wire          LDCInst;
wire          STCLInst;
wire          LDCLInst;
wire          MRCInst;
wire          MCRInst;
wire          MRRCInst;
wire          STCLBurst;
wire          STCLValid;
wire          SameInst;
wire          Hazard1;
wire          Hazard2;
wire          STCLLd;
wire          LDCLCntEn;
wire          ConfigWr;
wire          ValidInst;
wire          BurstLen;



reg    [31:0] WriteData1;
reg    [31:0] WriteData2;
reg    [25:0] Queue[7:0];
reg     [7:0] Enable;
reg     [7:0] QueueValid;
reg     [3:0] BusyWaitCnt;
```

```
reg     [2:0] WriteAddr1;
reg     [2:0] WriteAddr2;
reg           Bounce;
reg           WriteEn1;
reg           WriteEn2;



function [7:0] OneHotEncoder;
input [2:0]     Index;
begin
   case(Index)
      3'b000: OneHotEncoder = 8'b00000001;
      3'b001: OneHotEncoder = 8'b00000010;
      3'b010: OneHotEncoder = 8'b00000100;
      3'b011: OneHotEncoder = 8'b00001000;
      3'b100: OneHotEncoder = 8'b00010000;
      3'b101: OneHotEncoder = 8'b00100000;
      3'b110: OneHotEncoder = 8'b01000000;
      3'b111: OneHotEncoder = 8'b10000000;
      default : OneHotEncoder = {8{1'bx}};
   endcase
end
endfunction



a10VCPRegFile ua10VCPRegFile (
                     .CPCLK (CPCLK),
                     .writeenable1 (WriteEn1),
                     .writeaddr1 (WriteAddr1),
                     .writedata1 (WriteData1),
                    .writeenable2 (WriteEn2),
                     .writeaddr2 (WriteAddr2),
                     .writedata2 (WriteData2),
                     .readaddr1 (ReadAddr1),
                     .readaddr2 (ReadAddr2),
                    .readaddr3 (ReadAddr3),
                    .readaddr4 (ReadAddr4),
                    .CPRST (CPRST),

                     .readdata1 (ReadData1),
                     .readdata2 (ReadData2),
                    .readdata3 (ReadData3),
                    .readdata4 (ReadData4),
                    .configreg (ConfigReg)


  );



//Generation of Bounce & Stall signals

//# Valid instruction in the ISSUE stage if CPValid is high
//# CDP  :  CPInst[25] = 1, CPInst[4]  = 0
//# MCR  :  CPInst[25] = 1, CPInst[20] = 0, CPInst[4]  = 1
//# MRC  :  CPInst[25] = 1, CPInst[20] = 1, CPInst[4]  = 1
//# MRRC :  CPInst[25:20] = 6'b000101
//# MCRR :  CPInst[25:20] = 6'b000100
//# LDC  :  CPInst[25] = 0, CPInst[20] = 1, CPInst[22] = 0
//# STC  :  CPInst[25] = 0, CPInst[20] = 0, CPInst[22] = 0
//# LDCL :  CPInst[25] = 0, CPInst[20] = 1, CPInst[22] = 1
//# STCL :  CPInst[25] = 0, CPInst[20] = 0, CPInst[22] = 1

assign MRRCInst  = CPInst[25:20]==6'b000101;
assign MCRRInst  = CPInst[25:20]==6'b000100;
assign MRCInst   = CPInst[25]&CPInst[4]&CPInst[20];
assign MCRInst   = CPInst[25]&CPInst[4]&(!CPInst[20]);
assign LDCInst   = (!CPInst[25])&(!CPInst[22])&CPInst[20];
assign STCInst   = (!CPInst[25])&(!CPInst[22])&(!CPInst[20]);
assign LDCLInst  = (!CPInst[25])&CPInst[22]&CPInst[20]&(!MRRCInst);
assign STCLInst  = (!CPInst[25])&CPInst[22]&(!CPInst[20])&(!MCRRInst);


always @(CPInst or ConfigReg)
begin
   casez({CPInst[25:20],CPInst[4]})
      // CDP
      7'b1??_???0 : {Bounce,BusyWaitCnt} = {CPInst[15], CPInst[23:20]};
      // MCR/MRC
      7'b1??_???1:  {Bounce,BusyWaitCnt} = {CPInst[19], CPInst[3:0]};
```

```
      // MCRR/MRRC
      7'b000_10??:  {Bounce,BusyWaitCnt} = {CPInst[3],  CPInst[7:4]};
      // STC
      7'b0??_0?0?:  {Bounce,BusyWaitCnt} = {CPInst[15], ConfigReg[3:0]};
      // LDC
      7'b0??_0?1?:  {Bounce,BusyWaitCnt} = {CPInst[15], ConfigReg[11:8]};
      // STCL
      7'b0??_1?0?:  {Bounce,BusyWaitCnt} = {CPInst[15], ConfigReg[7:4]};
      // LDCL
      7'b0??_1?1?:  {Bounce,BusyWaitCnt} = {CPInst[15], ConfigReg[15:12]};
      // Instructions not supported by the CP
      default    :  {Bounce,BusyWaitCnt} = 5'b1_0000;
   endcase
end

// All outstanding instructions in the CP pipeline should be completed
// before any data transfer from the CP to ARM. This takes care of data
// dependencies without any special interlock logic.

assign ValidInst  = (|Valid[7:0]);
assign Hazard1    = (MRRCInst|MRCInst|STCInst|STCLInst)&ValidInst;


// This interlock signal takes care that any change in the configuration
// register is seen by all the subsequent instructions.

assign Hazard2    = (|Config[7:0]);



a10gdffrx26  uOldCPInst(OldCPInst,CPInst,CPCLK,1'b1,CPRST);
a10gdffrx3   uOldCPTag(OldCPTag,CPTag,CPCLK,1'b1,CPRST);

assign SameInst  = (OldCPInst==CPInst)&(OldCPTag==CPTag);
assign NextCount = SameInst ? (StallCount + 4'b0001) : 4'b0000;
assign Stall     = (BusyWaitCnt > NextCount)|Hazard1|Hazard2;

a10gdffrx4  uCounter(StallCount,NextCount,CPCLK,1'b1,CPRST);




//Adding new the instruction to the queue

always @(CPTag)
begin
   casez(CPTag)
      3'b000: Enable = 8'b0000_0001;
      3'b001: Enable = 8'b0000_0010;
      3'b010: Enable = 8'b0000_0100;
      3'b011: Enable = 8'b0000_1000;
      3'b100: Enable = 8'b0001_0000;
      3'b101: Enable = 8'b0010_0000;
      3'b110: Enable = 8'b0100_0000;
      3'b111: Enable = 8'b1000_0000;
     default: Enable = {8{1'bx}};
   endcase
end

assign EnEntry[7:0] = Enable[7:0]&{8{CPValid&(!Stall)}};

a10gdffx26  uEntry0(Entry0,CPInst,CPCLK,EnEntry[0]);
a10gdffx26  uEntry1(Entry1,CPInst,CPCLK,EnEntry[1]);
a10gdffx26  uEntry2(Entry2,CPInst,CPCLK,EnEntry[2]);
a10gdffx26  uEntry3(Entry3,CPInst,CPCLK,EnEntry[3]);
a10gdffx26  uEntry4(Entry4,CPInst,CPCLK,EnEntry[4]);
a10gdffx26  uEntry5(Entry5,CPInst,CPCLK,EnEntry[5]);
a10gdffx26  uEntry6(Entry6,CPInst,CPCLK,EnEntry[6]);
a10gdffx26  uEntry7(Entry7,CPInst,CPCLK,EnEntry[7]);



//Handling the Valid bit

assign Remove      = InvalidInstTag|RetiredInstTag;
assign EnValid     = Remove|EnEntry;

a10gdffrx1  uValid0(Valid[0],!Remove[0],CPCLK,EnValid[0],CPRST);
a10gdffrx1  uValid1(Valid[1],!Remove[1],CPCLK,EnValid[1],CPRST);
a10gdffrx1  uValid2(Valid[2],!Remove[2],CPCLK,EnValid[2],CPRST);
a10gdffrx1  uValid3(Valid[3],!Remove[3],CPCLK,EnValid[3],CPRST);
```

```
a10gdffrx1  uValid4(Valid[4],!Remove[4],CPCLK,EnValid[4],CPRST);
a10gdffrx1  uValid5(Valid[5],!Remove[5],CPCLK,EnValid[5],CPRST);
a10gdffrx1  uValid6(Valid[6],!Remove[6],CPCLK,EnValid[6],CPRST);
a10gdffrx1  uValid7(Valid[7],!Remove[7],CPCLK,EnValid[7],CPRST);


//Handling the config bit

//No new instruction is added to the queue until the instruction
//which updates the configuration reg (CR7) is retired. There is
//a config bit associated with each instruction in the queue, and
//it is set if the instruction can potentially update CR7.


assign ConfigWr = (MCRInst&CPInst[18]&CPInst[17]&CPInst[16]) |
                  (MCRRInst&CPInst[2]&CPInst[1]&CPInst[0])   |
                  (MCRRInst&CPInst[2]&CPInst[1]&(!CPInst[0]))|
                 (LDCInst&CPInst[14]&CPInst[13]&CPInst[12]) |
                 (LDCLInst&(({1'b0,CPInst[14:12]}+{1'b0,CPInst[7:5]})>4'b0110));

assign NextConfig[7:0] = EnEntry[7:0]&{8{ConfigWr}};

a10gdffrx1  uConfig0(Config[0],NextConfig[0],CPCLK,EnValid[0],CPRST);
a10gdffrx1  uConfig1(Config[1],NextConfig[1],CPCLK,EnValid[1],CPRST);
a10gdffrx1  uConfig2(Config[2],NextConfig[2],CPCLK,EnValid[2],CPRST);
a10gdffrx1  uConfig3(Config[3],NextConfig[3],CPCLK,EnValid[3],CPRST);
a10gdffrx1  uConfig4(Config[4],NextConfig[4],CPCLK,EnValid[4],CPRST);
a10gdffrx1  uConfig5(Config[5],NextConfig[5],CPCLK,EnValid[5],CPRST);
a10gdffrx1  uConfig6(Config[6],NextConfig[6],CPCLK,EnValid[6],CPRST);
a10gdffrx1  uConfig7(Config[7],NextConfig[7],CPCLK,EnValid[7],CPRST);



//Processing instructions which are retired by the pipeline follower

assign RetiredMRRC  = (RetiredInst[25:20]==6'b000101);
assign RetiredLDC   = (!RetiredInst[25])&(!RetiredInst[22])&RetiredInst[20];
assign Retired      = RetiredInstTag|RetiredDataTag;

assign RetiredInst  = ({26{Retired[0]}} & Entry0) |
                      ({26{Retired[1]}} & Entry1) |
                     ({26{Retired[2]}} & Entry2) |
                     ({26{Retired[3]}} & Entry3) |
                     ({26{Retired[4]}} & Entry4) |
                      ({26{Retired[5]}} & Entry5) |
                     ({26{Retired[6]}} & Entry6) |
                     ({26{Retired[7]}} & Entry7);



assign ReadAddr1    = RetiredInst[18:16];
assign ReadAddr2    = RetiredInst[14:12]+3'b001;

assign LDCLAddr1  = RetiredInst[14:12] + LDCLCntReg;
assign LDCLAddr2  = RetiredInst[14:12] + LDCLCntReg + 3'b001;
assign LDCLCmp    = (LDCLCntReg!=RetiredInst[7:5]);


always @(RetiredInst or ReadData1 or ReadData2 or DataIn or LDCLAddr1 or LDCLAddr2 or LDCLCmp or RetiredMRRC
or RetiredDataTag)
begin
   casez({RetiredInst[25:20],RetiredInst[4]})

     7'b1??_???0: begin                                 // MOVE
                    WriteData1 = ReadData1;
                   WriteAddr1 = RetiredInst[14:12];
                   WriteEn1   = 1'b1;
                   WriteData2 = {32{1'bx}};
                   WriteAddr2 = {3{1'bx}};
                   WriteEn2   = 1'b0;
                 end

     7'b1??_??01: begin                                 //MCR
                    WriteData1 = DataIn[31:0];
                   WriteAddr1 = RetiredInst[18:16];
                   WriteEn1   = 1'b1;
                   WriteData2 = {32{1'bx}};
                   WriteAddr2 = {3{1'bx}};
                   WriteEn2   = 1'b0;
                  end
```

```
     7'b0??_0?1?: if(!RetiredInst[7])                //LDC
                  begin
                    WriteData1 = DataIn[31:0];
                    WriteAddr1 = RetiredInst[14:12];
                   WriteEn1   = 1'b1;
                   WriteData2 = {32{1'bx}};
                   WriteAddr2 = {3{1'bx}};
                   WriteEn2   = 1'b0;
                  end
                 else                               //LDC with side effect
                 begin
                    WriteData1 = DataIn[31:0];
                    WriteAddr1 = RetiredInst[14:12];
                   WriteEn1   = 1'b1;
                   WriteData2 = ReadData2+32'd1;
                   WriteAddr2 = RetiredInst[14:12]+3'b001;
                   WriteEn2   = 1'b1;
                  end


     7'b0??_1?1?: if((!RetiredMRRC)&(|RetiredDataTag[7:0]))
                  begin                               //LDCL
                    WriteData1 = DataIn[31:0];
                   WriteData2 = DataIn[63:32];
                    WriteAddr1 = LDCLAddr1;
                   WriteAddr2 = LDCLAddr2;
                   WriteEn1   = 1'b1;
                   WriteEn2   = LDCLCmp;
                  end
                 else
                 begin
                   WriteData1 = {32{1'bx}};
                    WriteAddr1 = {3{1'bx}};
                    WriteEn1   = 1'b0;
                    WriteData2 = {32{1'bx}};
                    WriteAddr2 = {3{1'bx}};
                    WriteEn2   = 1'b0;
                 end

     7'b000_100?: begin                                //MCRR
                    WriteData1 = DataIn[31:0];
                   WriteData2 = DataIn[63:32];
                    WriteAddr1 = RetiredInst[2:0];
                   WriteAddr2 = RetiredInst[2:0]+3'b001;
                   WriteEn1   = 1'b1;
                   WriteEn2   = 1'b1;
                  end

        default:  begin
                    WriteData1 = {32{1'bx}};
                    WriteAddr1 = {3{1'bx}};
                    WriteEn1   = 1'b0;
                    WriteData2 = {32{1'bx}};
                    WriteAddr2 = {3{1'bx}};
                    WriteEn2   = 1'b0;
                 end
   endcase
end


//LDCL Counter
assign LDCLCntEn = (|Retired[7:0])|(!ValidInst);
assign LDCLCnt   = ((|RetiredInstTag[7:0])|(!ValidInst)) ? 3'b000 : LDCLCntReg+3'b010;

a10gdffrx3  uLDCLCntr(LDCLCntReg,LDCLCnt,CPCLK,LDCLCntEn,CPRST);



//STCL Counter
assign STCLLd       = CPValid&(!Stall);
assign NextCnt      = (STCLLd ? 3'b000 : STCLCnt) + 3'b010;

a10gdffrx3  uSTCLStartReg(STCLStartReg,CPInst[14:12],CPCLK,STCLLd,CPRST);
a10gdffrx3  uSTCLBurstLen(STCLBurstLen,CPInst[7:5],CPCLK,STCLLd,CPRST);
a10gdffrx3  uSTCLCnt(STCLCnt,NextCnt,CPCLK,!HoldDataOut,CPRST);

assign NextTag  = OneHotEncoder(CPTag)&{8{STCLInst}}&(~Remove);
assign STCLEn   = {8{STCLLd}}|Remove;
```

```
a10gdffrx1  uSTCLTag0(STCLTag[0],NextTag[0],CPCLK,STCLEn[0],CPRST);
a10gdffrx1  uSTCLTag1(STCLTag[1],NextTag[1],CPCLK,STCLEn[1],CPRST);
a10gdffrx1  uSTCLTag2(STCLTag[2],NextTag[2],CPCLK,STCLEn[2],CPRST);
a10gdffrx1  uSTCLTag3(STCLTag[3],NextTag[3],CPCLK,STCLEn[3],CPRST);
a10gdffrx1  uSTCLTag4(STCLTag[4],NextTag[4],CPCLK,STCLEn[4],CPRST);
a10gdffrx1  uSTCLTag5(STCLTag[5],NextTag[5],CPCLK,STCLEn[5],CPRST);
a10gdffrx1  uSTCLTag6(STCLTag[6],NextTag[6],CPCLK,STCLEn[6],CPRST);
a10gdffrx1  uSTCLTag7(STCLTag[7],NextTag[7],CPCLK,STCLEn[7],CPRST);

assign STCLBurst    = (|STCLTag[7:0])&(STCLCnt<=STCLBurstLen);



//Drive the DataOut bus for MRC, MRRC, STC and STC Multiple instructions

assign BAddr1 = STCLStartReg + STCLCnt;
assign BAddr2 = STCLStartReg + STCLCnt + 3'b001;

assign RAddr1 = MRRCInst ? CPInst[2:0]: (MRCInst ? CPInst[18:16]:CPInst[14:12]);
assign RAddr2 = (MRRCInst ? CPInst[2:0]: CPInst[14:12]) + 3'b001;

assign ReadAddr3 = STCLBurst ? BAddr1 : RAddr1;
assign ReadAddr4 = STCLBurst ? BAddr2 : RAddr2;


assign DataOut   = {ReadData4,ReadData3};


//Configuring the Pipeline follower
assign LSLEN   = 6'b00_0111;
assign LSDBL   = 1'b0;
assign LSSWP   = 1'b0;




//***************
//Error messages
//***************


// synopsys translate_off
// VCS coverage off

//Adding new instruction into the queue and setting the valid bit
always @(posedge CPCLK)
   if(!CPRST)
      if(EnEntry&Valid)
                $display("%d  Error: Valid entry overwritten in the queue",$time);

// VCS coverage on
// synopsys translate_on



endmodule
```

- **VCP Top-level**

```
//-------------------------------------------------------------------------
//     The confidential and proprietary information contained in this file may
//     only be used by a person authorised under and to the extent permitted
//     by a subsisting licensing agreement from ARM Limited.
//
//            (C) COPYRIGHT 2003 ARM Limited.
//                ALL RIGHTS RESERVED
//
//     This entire notice must be reproduced on all copies of this file
//     and copies of this file may only be made by a person if such person is
//     permitted to do so under the terms of a subsisting license agreement
//     from ARM Limited.
//
//
//-------------------------------------------------------------------------
```

```
// Overview
//=========

// This file contains the top level for the validation coprocessor (VCP). It
// instantiates the ARM10 pipeline follower and the VCP core. The pipeline
// follower interfaces with the ARM10's coprocessor interface on one side and
// the VCP core on the other. The VCP core is based on the simple tag based
// interface provided by the pipeline follower. The validation coprocessor is
// fully synthesizable.

// The VCP uses a pipeline follower that supports multi-coprocessor configuration.

module a10VCP(
              CPCLK,
              CPRST,
               CPEN,
              CPID,
              CPINSTR,
              CPINSTRV,
              CPVALIDD,
              ASTOPCPD,
              ASTOPCPE,
              ACANCELCP,
              AFLUSHCP,
              CPBIGEND,
              CPSUPER,
              LSHOLDCPM,
              LSHOLDCPE,
               CPABORT,
              LDCMCRDATA,
              CPBUSYEIN,
              CPLSBUSYIN,

              CPBUSYE,
              CPLSBUSY,
              CPBOUNCEE,
              CPLSLEN,
              CPLSSWP,
              CPLSDBL,
              STCMRCDATA
              );

//#
//# Interface Signals
//# =================
//#


//In from ARM (broadcast)
input         CPCLK;
input         CPRST;
input [25:0]  CPINSTR;
input         CPINSTRV;
input         CPVALIDD;
input         ASTOPCPD;
input         ASTOPCPE;
input         ACANCELCP;
input         AFLUSHCP;
input         CPBIGEND;
input         CPSUPER;
input         LSHOLDCPM;
input         LSHOLDCPE;
input         CPABORT;
input [63:0]  LDCMCRDATA;

//In from other CPs
input         CPBUSYEIN;
input         CPLSBUSYIN;

//Coprocessor enable
input         CPEN;

//Coprocessor ID
input   [3:0] CPID;

//Out to ARM and other CPs
output        CPBUSYE;
output        CPLSBUSY;
output        CPBOUNCEE;
output [5:0]  CPLSLEN;
output        CPLSSWP;
```

```
output        CPLSDBL;
output [63:0] STCMRCDATA;



wire   [63:0] STCMRCDATA;
wire   [63:0] LDCMCRDATA;
wire   [63:0] DataIn;
wire   [63:0] DataOut;
wire   [25:0] CPINSTR;
wire   [25:0] CPInst;
wire    [7:0] InvalidInstTag;
wire    [7:0] RetiredInstTag;
wire    [7:0] RetiredDataTag;
wire    [5:0] CPLSLEN;
wire    [5:0] LSLEN;
wire    [3:0] CPID;
wire    [2:0] CPTag;
wire          Stall;
wire          Bounce;
wire          LSSWP;
wire          LSDBL;
wire          CPValid;
wire          HoldDataOut;



a10PipeFollower ua10PipeFollower(

                          .CPCLK (CPCLK),
                          .CPRST (CPRST),
                          .CPEN (CPEN),
                          .CPINSTR (CPINSTR),
                          .CPINSTRV (CPINSTRV),
                          .CPVALIDD (CPVALIDD),
                          .ASTOPCPD (ASTOPCPD),
                          .ASTOPCPE (ASTOPCPE),
                          .ACANCELCP (ACANCELCP),
                          .AFLUSHCP (AFLUSHCP),
                          .CPBIGEND (CPBIGEND),
                          .CPSUPER (CPSUPER),
                          .LSHOLDCPM (LSHOLDCPM),
                          .LSHOLDCPE (LSHOLDCPE),
                            .CPABORT   (CPABORT),
                          .LDCMCRDATA (LDCMCRDATA),
                          .CPBUSYEIN (CPBUSYEIN),
                          .CPLSBUSYIN (CPLSBUSYIN),
                          .Stall (Stall),
                          .Bounce (Bounce),
                          .DataOut (DataOut),
                          .LSLEN (LSLEN),
                          .LSSWP (LSSWP),
                          .LSDBL (LSDBL),
                          .CPID (CPID),

                          .CPBUSYE (CPBUSYE),
                          .CPLSBUSY (CPLSBUSY),
                          .CPBOUNCEE (CPBOUNCEE),
                          .CPLSLEN (CPLSLEN),
                          .CPLSSWP (CPLSSWP),
                          .CPLSDBL (CPLSDBL),
                          .STCMRCDATA (STCMRCDATA),
                          .CPInst (CPInst),
                          .CPValid (CPValid),
                          .CPTag (CPTag),
                           .HoldDataOut (HoldDataOut),
                          .InvalidInstTag (InvalidInstTag),
                          .DataIn (DataIn),
                          .RetiredInstTag (RetiredInstTag),
                           .RetiredDataTag (RetiredDataTag)
             );



a10VCPCore ua10VCPCore(
                  .CPInst (CPInst),
                   .CPTag (CPTag),
                   .CPValid (CPValid),
                   .InvalidInstTag (InvalidInstTag),
                   .RetiredInstTag (RetiredInstTag),
```

```
                    .RetiredDataTag (RetiredDataTag),
                    .HoldDataOut (HoldDataOut),
                    .DataIn (DataIn),
                    .CPCLK (CPCLK),
                    .CPRST (CPRST),


                    .Bounce (Bounce),
                    .Stall (Stall),
                    .DataOut (DataOut),
                    .LSLEN (LSLEN),
                   .LSSWP (LSSWP),
                   .LSDBL (LSDBL)
                   );


endmodule
```

- **VCP hookup with the ARM1026EJ-S**

```
////////////////////////////
// VALIDATION COPROCESSOR  //
////////////////////////////

    a10VCP
      ua10VCP(
              //In from ARM
              .CPCLK       (CLK),
              .CPRST       (CPRST),
                .CPEN        (1'b1),
              .CPID        (4'b0110),
              .CPINSTR     (CPINSTR),
              .CPINSTRV    (CPINSTRV),
              .CPVALIDD    (CPVALIDD),
              .ASTOPCPD    (ASTOPCPD),
              .ASTOPCPE    (ASTOPCPE),
              .ACANCELCP   (ACANCELCP),
              .AFLUSHCP    (AFLUSHCP),
              .CPBIGEND    (CPBIGEND),
              .CPSUPER     (CPSUPER),
              .LSHOLDCPM   (LSHOLDCPM),
              .LSHOLDCPE   (LSHOLDCPE),
                .CPABORT     (CPABORT),
              .LDCMCRDATA  (LDCMCRDATA),

              //In from other CPs
              .CPBUSYEIN   (1'b0),
              .CPLSBUSYIN  (1'b0),

              //Out to ARM and other CPs
              .CPBUSYE     (CPBUSYE2),
              .CPLSBUSY    ( ),
              .CPBOUNCEE   (CPBOUNCEE2),
              .CPLSLEN     (CPLSLEN2),
              .CPLSSWP     (CPLSSWP2),
              .CPLSDBL     (CPLSDBL2),
              .STCMRCDATA  (STCMRCDATA)
              );


//////////////////////////////////////////
//         ARM1026EJ-S                  //
//////////////////////////////////////////


 ARM1026EJS_TCM uARM1026EJS_TCM(

    .CLK          (CLK),

    .nFIQ         (nFIQ),
```

```
.nIRQ          (nIRQ),
.IRQADDR       (IRQADDR[31:2]),
.IRQADDRV      (IRQADDRV),
.IRQACK        (IRQACK),
.STANDBYWFI    (STANDBYWFI),
.VINITHI       (HIVECS),
.BIGENDINIT    (plusargBigEndianMemory),
.CFGBIGEND     (CFGBIGEND),
.TAPID         (TAPID[31:0]),

.MMUnMPU       (plusargMMUnMPU),
.I64n32        (1'b1),
.D64n32        (1'b1),

.HRESETn       (HRESETN),

.HCLKEND       (HCLKEND),
.HADDRD        (HADDRD[31:0]),
.HTRANSD       (HTRANSD[1:0]),
.HBURSTD       (HBURSTD[2:0]),
.HWRITED       (HWRITED),
.HSIZED        (HSIZED[2:0]),
.HBSTRBD       (HBSTRBD[7:0]),
.HPROTD        (HPROTD[3:0]),
.HREADYD       (HREADYD),
.HRESPD        (HRESPD),
.HWDATAD       (HWDATAD[63:0]),
.HRDATAD       (HRDATAD[63:0]),
.HMASTLOCKD    (HMASTLOCKD),

.HCLKENI       (HCLKENI),
.HADDRI        (HADDRI[31:0]),
.HTRANSI       (HTRANSI[1:0]),
.HBURSTI       (HBURSTI[2:0]),
.HWRITEI       (HWRITEI),
.HSIZEI        (HSIZEI[2:0]),
.HBSTRBI       (HBSTRBI[7:0]),
.HPROTI        (HPROTI[3:0]),
.HREADYI       (HREADYI),
.HRESPI        (HRESPI),
.HRDATAI       (HRDATAI[63:0]),
.HMASTLOCKI    (HMASTLOCKI),


.SIMTESTMDI64n32   (SIMTESTMDI64n32),   // output
.SIMTESTMDD64n32   (SIMTESTMDD64n32),   // output

.SIMTESTMDRNDDRWT     (SIMTESTMDRNDDRWT),     // input
.SIMTESTMDRNDDDMA     (SIMTESTMDRNDDDMA),     // input

.SIMTESTMDRNDIRWT     (SIMTESTMDRNDIRWT),     // input
.SIMTESTMDRNDIDMA     (SIMTESTMDRNDIDMA),     // input

.CPEN          (CPEN),
.CPRST         (CPRST),
.CPINSTRV      (CPINSTRV),
.CPVALIDD      (CPVALIDD),
.AFLUSHCP      (AFLUSHCP),
.CPBIGEND      (CPBIGEND),
.ASTOPCPD      (ASTOPCPD),
.ASTOPCPE      (ASTOPCPE),
.CPSUPER       (CPSUPER),
.CPINSTR       (CPINSTR[25:0]),
.ACANCELCP     (ACANCELCP),
.CPABORT       (CPABORT),
.LDCMCRDATA    (LDCMCRDATA[63:0]),
.LSHOLDCPE     (LSHOLDCPE),
.LSHOLDCPM     (LSHOLDCPM),

.STCMRCDATA    (STCMRCDATA[63:0]),
.CPBOUNCEE1    (1'b1),
.CPBOUNCEE2    (CPBOUNCEE2),
.CPLSLEN1      (6'b000000),
.CPLSLEN2      (CPLSLEN2[5:0]),
.CPBUSYD1      (1'b0),
.CPBUSYD2      (1'b0),
.CPBUSYE1      (1'b0),
.CPBUSYE2      (CPBUSYE2),
.CPLSSWP1      (1'b0),
.CPLSSWP2      (CPLSSWP2),
.CPLSDBL1      (1'b0),
.CPLSDBL2      (CPLSDBL2),
```

```
    .COMMRX         (),
    .COMMTX         (),
    .DBGACK         (DBGACK),         // output from ARM10EJS
    .DBGEN          (DBGEN),
    .EDBGRQ         (EDBGRQ),         // input to ARM10EJS

    .DBGnTRST       (DBGnTRST),       // input to ARM10EJS
    .DBGTCKEN       (DBGTCKEN),       // input to ARM10EJS
    .DBGSCREG       (),
    .DBGIR          (),
    .DBGTAPSM       (),
    .DBGSDOUT       (ETMTDO),

    .DBGTDI         (TDI),
    .DBGTMS         (TMS),
    .DBGTDO         (DBGTDO),
    .DBGnTDOEN      (DBGnTDOEN),

    .ETMPWRDOWN     (ETMPWRDOWNFAKE),
    .ETMCORECTL     (ETMCORECTL[30:0]),
    .ETMDATAVALID   (ETMDATAVALID[1:0]),
    .ETMDATA        (ETMDATA[63:0]),
    .ETMDA          (ETMDA[31:0]),
    .ETMIA          (ETMIA[31:0]),
    .ETMR15EX       (ETMR15EX[31:0]),
    .ETMR15BP       (ETMR15BP[31:0]),

    .INITRAM        (INITRAM),


    // MBIST
    .MBISTCLKEN          (1'b0),
    .MTESTON             (1'b0),
    .MBISTDSHIFT         (1'b0),
    .MBISTSHIFT          (1'b0),
    .MBISTRXTCM          (),
    .MBISTRXCGR          (),
    .MBISTTX             (11'bxxxxxxxxxxx),

  // ATPG
    .RSTSAFE             (1'b0),
    .SE                  (1'b0),
    .SI                  (56'h00_0000_0000_0000),
    .SO                  (),
    .SCANMODE            (1'b0),
    .MUXINSEL            (1'b0),
    .MUXOUTSEL           (1'b0),
    .MBISTRAMBYP         (1'b0),
    .MBISTRESETN         (1'b0),

    .CHECKTEST           (1'b0),
    .SCANMUX             (2'b00),
    .SCORETEST           (1'b0),
    .WMUX                (2'b00),

    .WSON                (),
    .WSI                 (6'b000000),
    .WSO                 (),
    .WSEI                (1'b0),
    .WSEO                (1'b0)
  );
```

5 Appendix

This section presents an example in which two coprocessors are attached to the ARM1026EJ-S coprocessor interface. The first one is the validation coprocessor (VCP), which is described in detail in chapter 4. The second one is a trickbox coprocessor (TBOXCP), which has other validation functions required to test the ARM1026EJ-S processor. Both the coprocessors are based on the ARM10 pipeline follower. The hooking-up of the coprocessors with the ARM1026EJ-S is given below.

```
////////////////////////////////////////////////////
// VALIDATION COPROCESSOR  //
////////////////////////////////////////////////////

    a10VCP
      ua10VCP(
              //In from ARM
              .CPCLK       (CLK),
              .CPRST       (dftCPRST),
              .CPEN        (ENVCP),
              .CPINSTR     (CPINSTR),
              .CPINSTRV    (CPINSTRV),
              .CPVALIDD    (CPVALIDD),
              .ASTOPCPD    (ASTOPCPD),
              .ASTOPCPE    (ASTOPCPE),
              .ACANCELCP   (ACANCELCP),
              .AFLUSHCP    (AFLUSHCP),
              .CPBIGEND    (CPBIGEND),
              .CPSUPER     (CPSUPER),
              .LSHOLDCPM   (LSHOLDCPM),
              .LSHOLDCPE   (LSHOLDCPE),
              .CPABORT     (CPABORT),
              .LDCMCRDATA  (LDCMCRDATA),

              //In from other CPs
              .CPBUSYEIN   (CPBUSYE2),
              .CPLSBUSYIN  (CPLSBUSY2),


              //Out to ARM and other CPs
              .CPBUSYE     (CPBUSYE1),
              .CPLSBUSY    (CPLSBUSY1),
              .CPBOUNCEE   (CPBOUNCEE1),
              .CPLSLEN     (CPLSLEN1),
              .CPLSSWP     (CPLSSWP1),
              .CPLSDBL     (CPLSDBL1),
              .STCMRCDATA  (STCMRCDATA1)
              );

////////////////////////////////////////////////////
// TRICKBOX   COPROCESSOR  //
////////////////////////////////////////////////////

   a10TBOXCP
     ua10TBOX(

              //In from ARM
              .CPCLK       (CLK),
              .CPRST       (dftCPRST),
              .CPEN        (ENTBCP),
              .CPINSTR     (CPINSTR),
              .CPINSTRV    (CPINSTRV),
              .CPVALIDD    (CPVALIDD),
              .ASTOPCPD    (ASTOPCPD),
              .ASTOPCPE    (ASTOPCPE),
              .ACANCELCP   (ACANCELCP),
              .AFLUSHCP    (AFLUSHCP),
              .CPBIGEND    (CPBIGEND),
              .CPSUPER     (CPSUPER),
              .LSHOLDCPM   (LSHOLDCPM),
              .LSHOLDCPE   (LSHOLDCPE),
              .CPABORT     (CPABORT),
              .LDCMCRDATA  (LDCMCRDATA),
              .IRQACK      (IRQACK),

             //In from other CPs
              .CPBUSYEIN   (CPBUSYE1),
```

```
                .CPLSBUSYIN  (CPLSBUSY1),

                 //Out to ARM and other CPs
                .CPBUSYE     (CPBUSYE2),
                .CPLSBUSY    (CPLSBUSY2),
                .CPBOUNCEE   (CPBOUNCEE2),
                .CPLSLEN     (CPLSLEN2),
                .CPLSSWP     (CPLSSWP2),
                .CPLSDBL     (CPLSDBL2),
                .STCMRCDATA  (STCMRCDATA2),
                .nIRQ        (CPnIRQ),
                .IRQADDR     (IRQADDR),
                .IRQADDRV    (IRQADDRV),
                .MDENWST     (MDENWST),
                .MDRNDWST    (MDRNDWST),
                .MDNUMWST    (MDNUMWST),
                .MDNUMNWST   (MDNUMNWST),
                .MIENWST     (MIENWST),
                .MIRNDWST    (MIRNDWST),
                .MINUMWST    (MINUMWST),
                .MINUMNWST   (MINUMNWST),
                .MDENDMA     (MDENDMA),
                .MDRNDDMA    (MDRNDDMA),
                .MDNUMDMA    (MDNUMDMA),
                .MDNUMNDMA   (MDNUMNDMA),
                .MIENDMA     (MIENDMA),
                .MIRNDDMA    (MIRNDDMA),
                .MINUMDMA    (MINUMDMA),
                .MINUMNDMA   (MINUMNDMA)
                );

assign STCMRCDATA = STCMRCDATA1|STCMRCDATA2;


////////////////////////////////////////////////////////
//         ARM1026EJ_88  Block          //
////////////////////////////////////////////////////////

  ARM1026EJ_88
    uARM1026EJ_88
      (
       // Outputs
       .IRQACK         (IRQACK),
       .STANDBYWFI     (),
       .CFGBIGEND      (CFGBIGEND),

       .HRESETn        (HRESETn),

       .HCLKEND        (HCLKEN),
       .HADDRD         (HADDRD[31:0]),
       .HTRANSD        (HTRANSD[1:0]),
       .HBURSTD        (HBURSTD[2:0]),
       .HWRITED        (HWRITED),
       .HSIZED         (HSIZED[2:0]),
       .HBSTRBD        (HBSTRBD[7:0]),
       .HPROTD         (HPROTD[3:0]),
       .HWDATAD        (HWDATAD[63:0]),
       .HMASTLOCKD     (HMASTLOCKD),
       .HREADYD        (HREADYD),
       .HRESPD         (HRESPD),
       .HRDATAD        (HRDATAD[63:0]),

       .HCLKENI        (HCLKEN),
       .HADDRI         (HADDRI[31:0]),
       .HTRANSI        (HTRANSI[1:0]),
       .HBURSTI        (HBURSTI[2:0]),
       .HWRITEI        (HWRITEI),
       .HSIZEI         (HSIZEI[2:0]),
       .HBSTRBI        (HBSTRBI[7:0]),
       .HPROTI         (HPROTI[3:0]),
       .HMASTLOCKI     (HMASTLOCKI),
       .HREADYI        (HREADYI),
       .HRESPI         (HRESPI),
       .HRDATAI        (HRDATAI[63:0]),

       .COMMRX         (COMMRX),
       .COMMTX         (COMMTX),
       .DBGACK         (DBGACK),
       .DBGSCREG       (),
       .DBGIR          (),
       .DBGTAPSM       (),
       .DBGSDOUT       (ETMTDO),
```

```
.DRnRW          (DRnRW),
.DRADDR         (DRADDR[16:0]),
.DRWD           (DRWD[63:0]),
.DRCS           (DRCS[1:0]),
.DRWBL          (DRWBL[7:0]),
.DRWPAR         (),
.IRnRW          (IRnRW),
.IRADDR         (IRADDR[16:0]),
.IRWD           (IRWD[63:0]),
.IRCS           (IRCS[1:0]),
.IRWBL          (IRWBL[7:0]),
.IRWPAR         (),

.CPEN           (),
.CPRST          (CPRST),
.CPINSTRV       (CPINSTRV),
.CPVALIDD       (CPVALIDD),
.AFLUSHCP       (AFLUSHCP),
.CPBIGEND       (CPBIGEND),
.ASTOPCPD       (ASTOPCPD),
.ASTOPCPE       (ASTOPCPE),
.CPSUPER        (CPSUPER),
.CPINSTR        (CPINSTR),
.ACANCELCP      (ACANCELCP),
.CPABORT        (CPABORT),
.LDCMCRDATA     (LDCMCRDATA),
.LSHOLDCPE      (LSHOLDCPE),
.LSHOLDCPM      (LSHOLDCPM ),

.SIMTESTMDDRSIZE   (),
.SIMTESTMDDRSZVAL  (),
.SIMTESTMDIRSIZE   (),
.SIMTESTMDIRSZVAL  (),
.SIMTESTMDI64n32   (),
.SIMTESTMDD64n32   (),

// Inputs
.CLK            (CLK),
.nFIQ           (nFIQint),
.nIRQ           (nIRQint),
.IRQADDR        (IRQADDR),
.IRQADDRV       (IRQADDRV),
.VINITHI        (VINITHI),
.BIGENDINIT     (BIGENDIN),
.MMUnMPU        (MMUnMPU),
.I64n32         (I64n32),
.D64n32         (D64n32),
.TAPID          (TAPID[31:0]),
.DBGEN          (DBGEN),
.EDBGRQ         (EDBGRQint),
.DBGnTRST       (DBGnTRSTsync),
.DBGTCKEN       (DBGTCKEN),
.DRDMAEN        (DRDMAEN),
.DRRD           (DRRD[63:0]),
.DRWAIT         (DRWAIT),
.DTCMSIZE       (DTCMSIZE[3:0]),

.IRDMAEN        (IRDMAEN),
.IRRD           (IRRD[63:0]),
.IRWAIT         (IRWAIT),
.ITCMSIZE       (ITCMSIZE[3:0]),
.INITRAM        (INITRAM),

.TCMVALInIMPL   (1'b0),

.STCMRCDATA     (STCMRCDATA[63:0]),
.CPBOUNCEE1     (CPBOUNCEE1),
.CPBOUNCEE2     (CPBOUNCEE2),
.CPLSLEN1       (CPLSLEN1[5:0]),
.CPLSLEN2       (CPLSLEN2[5:0]),
.CPBUSYD1       (1'b0),
.CPBUSYD2       (1'b0),
.CPBUSYE1       (CPBUSYE1),
.CPBUSYE2       (CPBUSYE2),
.CPLSSWP1       (CPLSSWP1),
.CPLSSWP2       (CPLSSWP2),
.CPLSDBL1       (CPLSDBL1),
.CPLSDBL2       (CPLSDBL2),

// JTAG inputs and outputs
.DBGTDI         (DBGTDI),
```

```
.DBGTMS          (DBGTMS),
.DBGTDO          (DBGTDO),
.DBGnTDOEN       (DBGnTDOEN),

// ETM inputs and outputs
.ETMPWRDOWN      (ETMPWRDOWN),
.ETMCORECTL      (ETMCORECTL[30:0]),
.ETMDATAVALID    (ETMDATAVALID[1:0]),
.ETMDATA         (ETMDATA[63:0]),
.ETMDA           (ETMDA[31:0]),
.ETMIA           (ETMIA[31:0]),
.ETMR15EX        (ETMR15EX[31:0]),
.ETMR15BP        (ETMR15BP[31:0]),

// MBIST inputs and outputs
.MBISTCLKEN(MbistCLKEN),
.MBISTDSHIFT(MbistDSHIFT),
.MBISTRAMBYP(MbistRAMBYP),
.MBISTRESETN(MbistRESETN),
.MBISTSHIFT(MbistSHIFT),
.MBISTTX(MbistTX[10:0]),
.MTESTON(MTESTON),
.MBISTRX(MbistRX1026[2:0]),

// ATPG inputs and outputs
.RSTSAFE             (arm1026RSTSAFE),
.WSI                 ({1'b0,arm1026WSI[2:0]}),
.WSO                 ({dummy[0],arm1026WSO[2:0]}),
.WSON                (arm1026WSON),
.WSEI                (arm1026WSEI),
.WSEO                (arm1026WSEO),
.SE                  (arm1026SE),
.SCANMODE            (arm1026SCANMODE),
.MUXINSEL            (arm1026MUXINSEL),
.MUXOUTSEL           (arm1026MUXOUTSEL),

.SI                  ({28'd0,ScanIN[27:0]}),
.SO                  ({dummy[27:0],arm1026SCANOUT[27:0]}),

.CHECKTEST           (CheckTest),
.SCANMUX             (ScanMux),
.SCORETEST           (ScoreTest),
.WMUX                (WMux[1:0])

);
```