Application Note 205

Writing JTAG Sequences for ARM9 Processors

Document number: ARM DAI 205A

Issued: 1st June 2008

Copyright ARM Limited 2008

Application Note 205

Writing JTAG Sequences for ARM9 Processors

1.1.1. Release information

Change history

Date	Issue	Change
June 2008	A	First release

1.1.1. Proprietary notice

1.1.1. Confidentiality status

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

1.1.1. 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.

1.1.1. ARM web address

`http://www.arm.com`

Table of Contents

1 About this document

This document describes the JTAG sequences required to perform particular operations on ARM9 based processor cores. Rather than list every single clock cycle, sequences in this document will appear in the following format:

TAP Path	Data to shift	Comment
IR	SCAN_N	Shift SCAN_N into IR
DR	5’h10	Shift 5 bits into DR, MSB first (1, 0, 0, 0, 0)
RTI		
RESET		

TAP Path can be “IR” indicating that you should generate TMS patterns required to take you from the current position through to Shift-IR, shift the data indicated and then move to Update-IR. TAP Path “DR” indicates that you should generate TMS patterns required to take you from the current state through to Shift-DR, shift the data indicated and then move to Update-DR.

For “IR” and “DR”, you should not enter Run-Test/Idle state either on your way to Shift or after you have finished in Update.

“RTI” indicates that you should generate TMS patterns required to take you from the current state to Run-Test/Idle, and stay there for only one cycle unless otherwise indicated in the comment. You stay in Run-Test/Idle state for as many consecutive RTI’s that there are in the table (one for each TCK).

“RESET” indicates that you should generate TMS = 1 for 5 TCK cycles. This will take you to the Reset state of the tap controller.

The “Data to shift” should be shifted in to TDI Most Significant Bit first, as shown in the example. If the Data to Shift is an IR instruction (e.g. SCAN_N) then the code for this can be found in the technical reference manual. If the Data to Shift is displayed as groups, such as {32’h0, 3’h0, 32’h0} the MSB of the left most element is shifted first, and the LSB or the right most element is shifted last.

1.1 Note about shift order

Shift patterns shown in this document should always be presented MSB first. That is, if this document instructs you to shift in 5’h10, you should shift 1 followed by 0, 0, 0 and 0.

After you have shifted values through TDI, the first bit you shifted in can either come to rest at the most significant or the least significant bit of the destination register, depending on which register you are accessing.

For example, the bit pattern for the IDCODE instruction is 4’b1110 (4’hE). However, when you wish to shift this command into the IR of the ARM core TAP register, the first bit you present to TDI will end up at the LSB position of the instruction register. Therefore to shift in the IDCODE instruction you actually need to shift in the reverse bit pattern: 4’b0111 (4’h7).

The same is true of the scan chain selection register when SCAN_N has been shifted into the IR: the first bit shifted in will end up at the LSB position of the scan chain selection register. Therefore to select scan chain #1 you will need to shift in 1 followed by 0, 0, 0 and 0 (5’h10).

The technical reference manual explains the bit ordering for the scan chains in detail.

2 Background

Before attempting to write your own JTAG stimulus, you should be familiar with the debug architecture as described by the ARM Technical Reference Manuals (for example, the ARM966E-S Rev.2 is described in the ARM966E-S Rev.2 TRM and the ARM9E-S Rev.2 TRM). In particular you should be familiar with the “Debug Interface and EmbeddedICE-RT” and “Debug in Depth” chapters in the Integer Core TRM.

2.1 Debug Visibility

A debugger connected to the ARM core debug port cannot see directly into the memory. A debugger wishing to access external memory must tell the ARM core what to do on its behalf by shifting in ARM instructions to the pipeline.

For example, if you wish to execute a store to a particular memory location, you must first shift the address and data value(s) into the ARM registers, followed by telling the ARM core to execute a system speed store using these registers. The ARM core will come out of debug to perform the store (at system speed) and then immediately jump back into debug.

A debugger connected to the ARM core also cannot tell the ARM core to execute a Coprocessor instruction at debug speed due to the handshaking that goes on between the ARM core and the Coprocessor. This also applies to CP15, the configuration and control coprocessor. CP14 and CP15 coprocessors have separate scan chains for accesses to these devices.

2.2 About ARM Debug Modes

ARM cores have two debug modes: Halt mode, where the ARM core is completely stopped and Monitor mode where the ARM core continues running code during debug (these are described fully in the TRM).

In order to have the ARM core execute instructions issued by the debugger, you must use Halt mode debugging.

During Halt Mode debugging there are two methods of executing instructions: (1) Debug Speed and (2) System Speed. All memory accesses (ARM <-> Memory) and exiting from debug is done at system speed. Everything else is done at debug speed. The ARM core can execute most instructions at debug speed.

On Scan Chain 1, there is a "System Speed" bit. As well as pushing instructions on to this scan chain for execution by the core, you must either set or clear the System Speed bit. When it is set, this indicates two things to that ARM core:

1. This instruction will run at system speed

2. The debug request bit traveling up the pipeline with this instruction will be set, so that after the instruction has been executed the ARM core will re-enter debug

3 Halting the ARM core

To halt the ARM core through a watchpoint or breakpoint, you must alter some bits in the Debug Control Register (ARM9E-S B.10.5, page B-32) - bit 4 must be cleared (to enable Halt mode debug). This can be done while the ARM core is in reset (HRESETn held low, DBGnTRST held high). Failure to do this will cause the ARM core to enter monitor mode when a breakpoint or watchpoint is hit.

Asserting the EDBGRQ signal or setting the DBGRQ bit (bit 1) in the Embedded-ICE debug control register will also HALT the ARM core.

Once the ARM core has halted, you can then start to inject instructions into the ARM pipeline for the ARM core to execute.

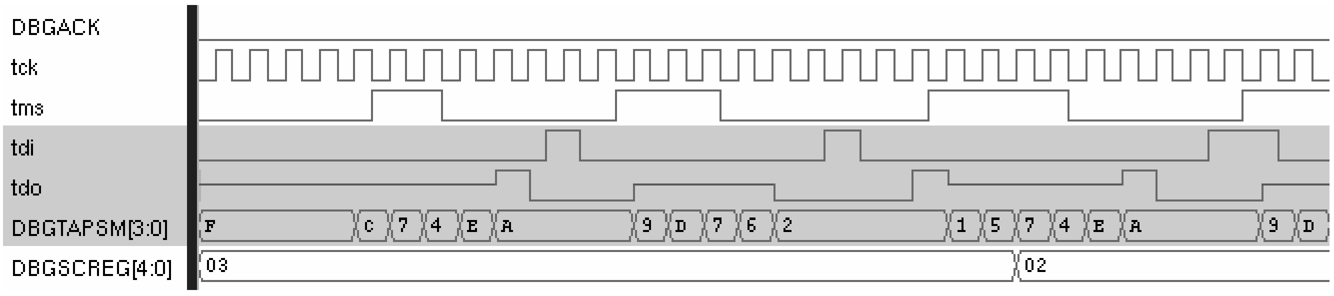
3.1 Example: Halting the ARM core with the Debug Control Register DBGRQ

The following is the sequence used to set bit 1 in the debug and control register, and therefore HALT the ARM core.

To access the EmbeddedICE-RT logic, you must first select scan chain 2:

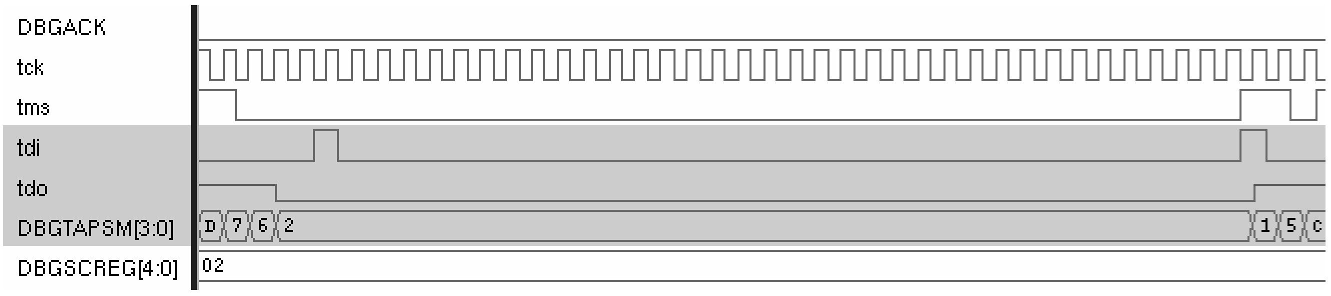
IR	SCAN_N	Shift SCAN_N into IR
DR	5’h08	Select scan chain 2
IR	INTEST	Enable the scan chain

After this pattern you do not proceed to Run-Test/Idle – you should immediately proceed to Shift-DR to start shifting the first data item onto the scan chain.



Once you have selected scan chain 2, the following sequence will set the debug control register, which is register address 0x0 (ARM9E-S B.10.1, page B-27):

DR	{6’h2, 26’h0, 5’h0, 1’b1}	Shift in value to write to the register, then 26 0’s, then Register address (0), finally Write enable
RTI	-	Perform the access



The ARM core will now halt after the pipeline has moved a few times (there is always a delay, and the ARM core pipeline **must** be moving in order for the ARM core to enter debug).

Once the ARM core has entered HALT debug, DBGACK will be asserted.

Once the ARM core has halted, you **must** clear the DBGRQ bit, otherwise you will not be able to exit debug:

DR	{6'h0, 26'h0, 5'h0, 1'b1}	Shift in value to write to the register, then 26 0's, then Register address (0), finally Write enable
RTI	-	Perform the access

3.2 Conditions for entering debug

The ARM processor core can only enter debug if the pipeline is moving and DBGEN is HIGH.

If the ARM core is not executing instructions (e.g. HRESETn is tied low, or the ARM core is waiting for some data to come back from external memory) then it won't enter debug and DBGACK will not go high.

If you are planning on debugging from reset because you have no code in memory, you should assert EDBGRQ, or the DBGRQ bit in the EmbeddedICE-RT debug control register before the ARM core comes out of reset. The ARM core will enter Halt mode debug before trying to execute anything after coming out of reset, but only after the ARM core has attempted to fetch a few instructions. Therefore the memory system should not hold the ARM core off, or return error responses during these initial instruction fetches.

4 Executing simple instructions

This section excludes:

- Reading or writing ARM register bank registers through the JTAG port
- Reading or writing memory by the ARM core or through the JTAG port
- Coprocessor instructions
- Exiting debug

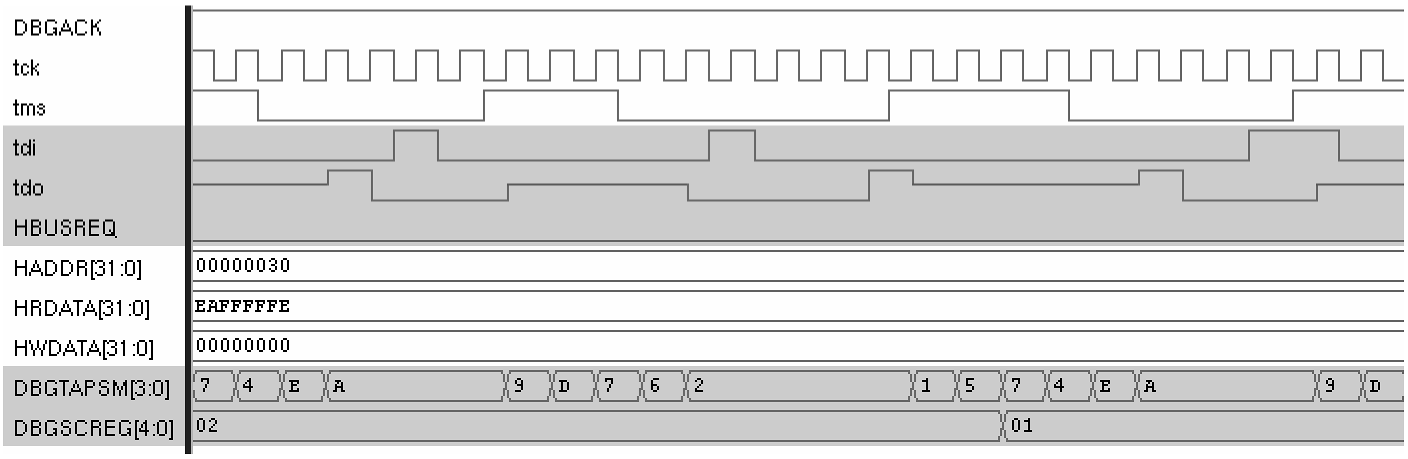
Once the ARM core is in Halt mode debug, you can get the ARM core to execute instructions by shifting the instructions into the JTAG port. This functionality is provided through scan chain 1.

When scan chain 1 is selected, the ARM processor core (and pipeline) is clocked for every TCK that the TAP controller is in Run-Test/Idle state. Therefore care must be taken when shifting through Run-Test/Idle state, since being in this state longer or shorter than necessary will produce undesirable results.

It is for this reason that after you have selected scan chain 1 and shifted the INTEST instruction into the TAP IR you should proceed directly to Shift-DR to perform the first scan to avoid unnecessarily advancing the pipeline.

The following sequence will select scan chain 1:

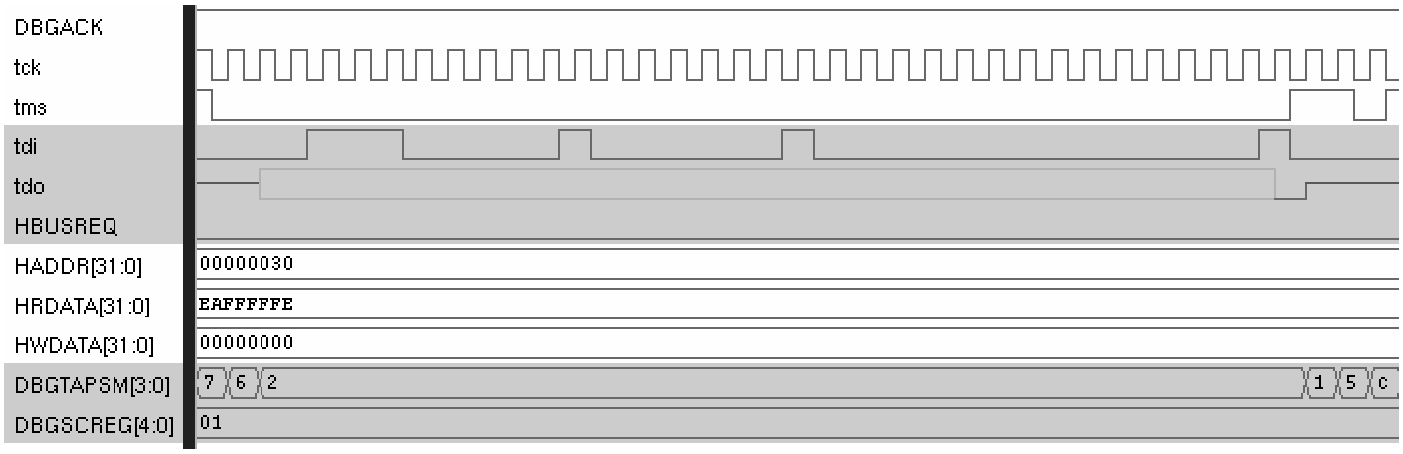
IR	SCAN_N	Shift SCAN_N into IR
DR	5’h10	Select scan chain 1
IR	INTEST	Enable the scan chain



Instructions can now be shifted in. Since the ARM core ignore scan chain 1 bits [66:33] unless it’s reading/writing data, simple instruction scans only need to shift 33 bits – the SYSSPEED bit and the opcode of the instruction you wish to execute. Therefore, the first bit shifted into TDI during the Shift-DR state is the intended value for SYSSPEED, the next bit is bit 31 of the opcode of the instruction you wish to execute, then 30, 29, 28… 0.

The following example is how to shift a debug speed ADD r0, r1, r2 instruction in to the ARM core (opcode: 0xE0810002) at debug speed

DR	{1’h0, 32’hE0810002}	Shift SYSSPEED = 0, the opcode for ADD
RTI	-	Commit the instruction to the pipeline



If you stay in Run-Test/Idle for more than one TCK cycle then the command will be issued multiple times into the pipeline and the result may not be what you expect (e.g. if you are using the same register for source and destination).

You need to then repeat the steps above for each instruction you wish to execute. As you do this, the preceding instructions will make their way through the pipeline.

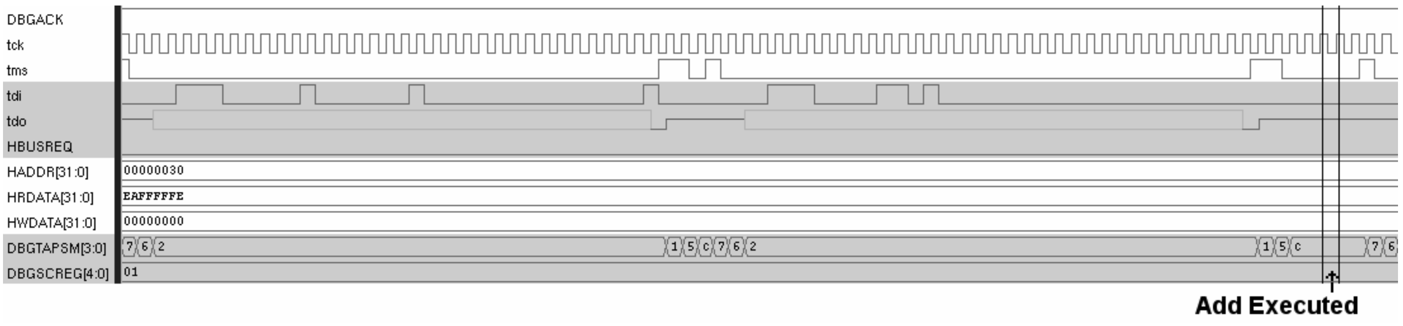
Once all instructions have been shifted in, you need to advance the pipeline to ensure that all of the instructions are executed. You do this by issuing NOP instructions (MOV r0, r0, opcode 0xE1A00000) and clocking the pipeline.

DR	{1'h0, 32'hE1A00000}	Shift SYSSPEED = 0, the opcode for NOP
RTI	-	Commit the instruction to the pipeline You can safely stay in RTI for many TCK cycles with a NOP instruction

4.1 The complete sequence for executing a single ADD instruction

Here is the complete sequence for executing a single ADD instruction

IR	SCAN_N	Shift SCAN_N into IR	
DR	5'h10	Select scan chain 1	
IR	INTEST	Enable the scan chain	
DR	{1'h0, 32'hE0810002}	Shift SYSSPEED = 0, the opcode for ADD	
RTI	-	Commit the ADD instruction to the pipeline	Decode: ADD
DR	{1'h0, 32'hE1A00000}	Shift SYSSPEED = 0, the opcode for NOP	
RTI	-	Commit the NOP instruction to the pipeline, advance ADD to EXECUTE stage of pipeline	Execute: ADD Decode: NOP
RTI	-	Commit the NOP instruction to the pipeline, advance ADD to MEMORY stage of pipeline	Memory: ADD Execute, Decode: NOP
RTI	-	Commit the NOP instruction to the pipeline, advance ADD to WRITEBACK stage of pipeline	Writeback: ADD Memory, Execute, Decode: NOP
RTI	-		Pipeline full of NOPs



4.2 Multi-Cycle Instructions

Note that instructions that require more than one cycle to execute will stall the pipeline, and therefore the instruction that you've shifted in will overwrite the previous instruction committed. You can overcome this by using NOP instructions liberally to flush the pipeline.

5 Reading and Writing Registers through the JTAG port

Reading and writing registers is done by performing debug-speed stores and loads (respectively).

For loads (data into the register), the data is sampled from the data bus portion of scan chain 1 when the instruction is in the memory stage of the pipeline and the TAP controller is in Run-Test/Idle state.

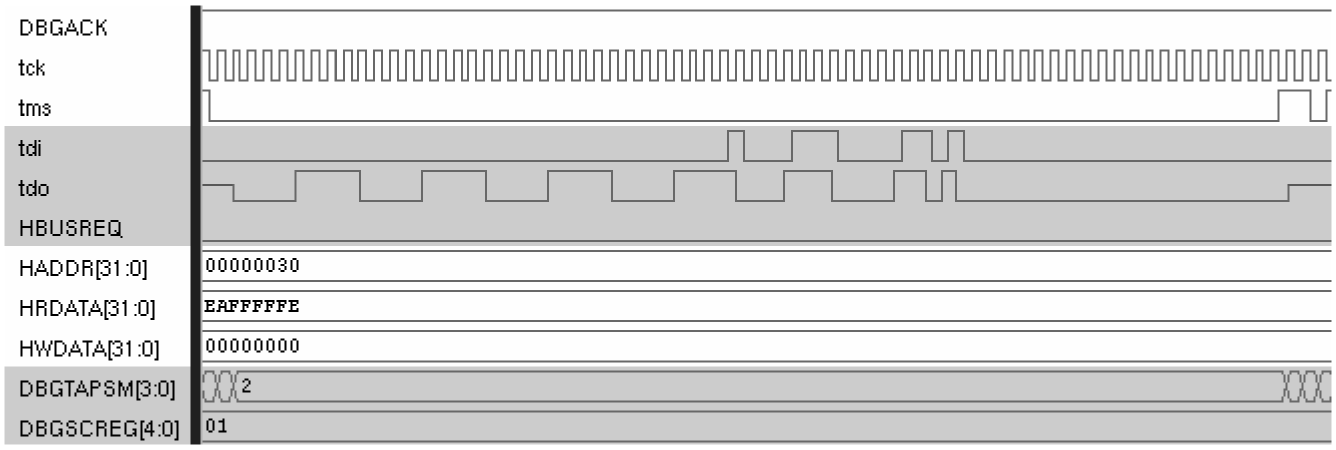
For stores, the data is placed onto the data portion of scan chain 1 when the instruction is in the execute stage of the pipeline and the TAP controller is in Run-Test/Idle state.

For register loads and stores through the JTAG port you should use the program counter for the address although this isn't mandatory – you may use any register for the "address" since it is ignored during this operation and external memory is not updated or accessed in any way.

5.1 For reading a register:

Reading a register through the JTAG port is achieved with a store instruction. In this example, we are reading out the value of r0.

IR	SCAN_N	Shift SCAN_N into IR	
DR	5'h10	Select scan chain 1	
IR	INTEST	Enable the scan chain	
DR	{1'h0, 32'hE58f0000}	Shift SYSSPEED = 0, the opcode for STR r0, [pc]	
RTI	-	Commit the STR instruction to the pipeline	Decode: STR
DR	{1'h0, 32'hE1A00000}	Shift SYSSPEED = 0, the opcode for NOP	
RTI	-	Commit the NOP instruction to the pipeline, advance STR to EXECUTE stage of pipeline	Execute: STR Decode: NOP
RTI	-	Commit the NOP instruction to the pipeline; advance STR to MEMORY stage of pipeline. **The ARM core makes the data available on scan chain 1 after this RTI cycle.**	Memory: STR Execute, Decode: NOP
DR	{1'h0, 32'hE1A00000}	The first 32 bits out of TDO is the value of r0, bit 0 out **first**	
RTI	-		Writeback: STR Memory, Execute, Decode: NOP



Shifting out the data value – in this case F0F0F0F0 (LSB out **first**)

5.2 For writing a register

Writing a register through the JTAG port is achieved with a load instruction. In this example, we are writing the value of r0 with data 0x11223344. Remember that the data portion of scan chain 1 is reverse, so bit 0 of this data value is shifted in first (effectively shifting the value 0x22CC4488).

IR	SCAN_N	Shift SCAN_N into IR	
DR	5'h10	Select scan chain 1	
IR	INTEST	Enable the scan chain	
DR	{1'h0, 32'hE59f0000}	Shift SYSSPEED = 0, the opcode for LDR r0, [pc]	
RTI	-	Commit the LDR instruction to the pipeline	Decode: LDR
DR	{1'h0, 32'hE1A00000}	Shift SYSSPEED = 0, the opcode for NOP	
RTI	-	Commit the NOP instruction to the pipeline, advance LDR to EXECUTE stage of pipeline	Execute: LDR Decode: NOP
RTI	-	Commit the NOP instruction to the pipeline; advance LDR to MEMORY stage of pipeline.	Memory: LDR Execute, Decode: NOP
DR	{32'h22CC4488 3'h0, 32'hE1A00000}	The first 32 bits into of TDI is the value of r0 (reversed, bit 0 in **first**).	
RTI	-	Data value is committed	Writeback: LDR Memory, Execute, Decode: NOP
RTI	-	Data value is written back to r0	Pipeline full of NOPs

6 Reading/Writing Multiple Registers through JTAG

Reading and writing multiple registers can be accomplished by executing store or load multiples (respectively). When executing an LDM/STM instruction, you must note that it occupies both the Execute and Memory stages of the pipeline up until just before the last register is accessed. This will affect which instructions shifted will be committed to the pipeline.

The LDM/STM instruction always accesses the lowest register first, ascending to the highest register last.

6.1 For reading multiple registers

When reading multiple registers, you can use the STM commands instead of the STR command. This will have the effect of shifting out the data of each register in turn.

IR	SCAN_N	Shift SCAN_N into IR	
DR	5’h10	Select scan chain 1	
IR	INTEST	Enable the scan chain	
DR	{1’h0, 32’hE88F00FF}	Shift SYSSPEED = 0, the opcode for STMIA pc, {r0-r7}	
RTI	-	Commit the STM instruction to the pipeline	Decode: STM
DR	{1’h0, 32’hE1A00000}	Shift SYSSPEED = 0, the opcode for NOP	
RTI	-	Commit the NOP instruction to the pipeline, advance STM to EXECUTE stage of pipeline	Execute: STM Decode: NOP
RTI	-	Advance STM to MEMORY stage of pipeline. **The ARM core makes the data for the first register available on scan chain 1 after this RTI cycle.**	Memory, Execute: STM Decode: NOP
DR	{1’h0, 32’hE1A00000}	The first 32 bits out of TDO is the value of r0, bit 0 out **first**	
RTI	-		
DR	{1’h0, 32’hE1A00000}	The first 32 bits out of TDO is the value of r1, bit 0 out **first**	
RTI	-		
DR	{1’h0, 32’hE1A00000}	The first 32 bits out of TDO is the value of r2, bit 0 out **first**	
RTI	-		
DR	{1’h0, 32’hE1A00000}	The first 32 bits out of TDO is the value of r3, bit 0 out **first**	
RTI	-		
DR	{1’h0, 32’hE1A00000}	The first 32 bits out of TDO is the value of r4, bit 0 out **first**	

RTI	-		
DR	{1'h0, 32'hE1A00000}	The first 32 bits out of TDO is the value of r5, bit 0 out **first**	
RTI	-		
DR	{1'h0, 32'hE1A00000}	The first 32 bits out of TDO is the value of r6, bit 0 out **first**	
RTI	-	Commit a NOP to the pipeline	Memory: STM Execute, Decode: NOP
DR	{1'h0, 32'hE1A00000}	The first 32 bits out of TDO is the value of r7, bit 0 out **first**	
RTI	-		

6.2 For writing multiple registers

When writing multiple registers, you can use the LDM commands instead of the LDR command. In the following example, the data shifted in to r0-r7 will be hex values 0x1 to 0x8 (r0 = 0x1, r1 = 0x2, etc.)

IR	SCAN_N	Shift SCAN_N into IR	
DR	5'h10	Select scan chain 1	
IR	INTEST	Enable the scan chain	
DR	{1'h0, 32'hE89F00FF}	Shift SYSSPEED = 0, the opcode for LDMIA pc, {r0-r7}	
RTI	-	Commit the LDM instruction to the pipeline	Decode: LDM
DR	{1'h0, 32'hE1A00000}	Shift SYSSPEED = 0, the opcode for NOP	
RTI	-	Commit the NOP instruction to the pipeline, advance LDM to EXECUTE stage of pipeline	Execute: LDM Decode: NOP
RTI	-	Advance LDM to MEMORY stage of pipeline.	Memory, Execute: LDM Decode: NOP
DR	{32'h80000000, 1'h0, 32'hE1A00000}	The first 32 bits into TDI is the value of r0, bit 0 in **first**	
RTI	-	Register r0	Writeback, Memory, Execute: LDM Decode: NOP
DR	{32'h40000000, 1'h0, 32'hE1A00000}	The first 32 bits into TDI is the value of r1, bit 0 in **first**	
RTI	-	Register r1, commit r0 to register bank	
DR	{32'hC0000000, 1'h0, 32'hE1A00000}	The first 32 bits into TDI is the value of r2, bit 0 in **first**	

RTI	-	Register r2, commit r1 to register bank	
DR	{32’h20000000, 1’h0, 32’hE1A00000}	The first 32 bits into TDI is the value of r3, bit 0 in **first**	
RTI	-	Register r3, commit r2 to register bank	
DR	{32’hA0000000 1’h0, 32’hE1A00000}	The first 32 bits into TDI is the value of r4, bit 0 in **first**	
RTI	-	Register r4, commit r3 to register bank	
DR	{32’h60000000 1’h0, 32’hE1A00000}	The first 32 bits into TDI is the value of r5, bit 0 in **first**	
RTI	-	Register r5, commit r4 to register bank	
DR	{32’hE0000000 1’h0, 32’hE1A00000}	The first 32 bits into TDI is the value of r6, bit 0 in **first**	
RTI	-	Register r6, commit r5 to register bank	Writeback, Memory: LDM Execute, Decode: NOP
DR	{32’h10000000 1’h0, 32’hE1A00000}	The first 32 bits into TDI is the value of r7, bit 0 in **first**	
RTI	-	Register r7, commit r6 to register bank	Writeback: LDM Memory, Execute, Decode: NOP
RTI	-	Commit r7 to register bank	Pipeline full of NOPs

7 Reading/Writing to memory through the JTAG port

The ARM debug logic can only see inside the ARM core. Therefore, to read and write memory you must use registers in the ARM core as a buffer and use system speed ARM instructions to read/write from the ARM registers into memory.

7.1 About System Speed Instructions

The ARM core must perform memory accesses (including caches/TCMs) at system speed. From the ARM cores point of view, System Speed means that it must exit debug, execute the instruction to be run and then re-enter debug.

This is all managed by the ARM core provided that you perform the correct sequence to execute a system speed instruction. Failure to supply the correct sequence will result in either the ARM core leaving debug completely, or not performing the instruction at system speed.

On the ARM9 core, the correct sequence to execute a system speed instruction is:

DR	{1'h0, 32'hxxxxxxxx}	Shift SYSSPEED = 0, the opcode for instruction to run at system speed
RTI	-	Commit the instruction to the pipeline
DR	{1'h1, 32'hE1A00000}	Shift SYSSPEED = 1, the opcode for NOP
RTI	-	Commit the NOP instruction to the pipeline
IR	RESTART	Shift RESTART instruction into the TAP IR
RTI	-	Go! Stay in RTI for as long as you need.

This assumes that you have already selected scan chain 1 and placed the INTEST instruction in the TAP IR.

When you enter this sequence, the ARM core will drop out of debug and lower DBGACK, execute the instruction shifted in at step 1 and then reenter debug, reasserting DBGACK. You do not need to assert EDBGRQ during this sequence, as the ARM core takes care of debug exit/entry.

While the ARM core is executing the instruction you should not attempt to select scan chain 1. You can hold the TAP controller in Run-Test/Idle until the ARM core has reentered debug.

Once the ARM core has reentered debug, you will need to reselect the next scan chain that you next wish to access, since the TAP controller will still contain the RESTART instruction.

7.2 About Reading and Writing to Memory in Debug

The correct process for reading a data value from memory out of the JTAG port is to shift in the address into an ARM register, execute a system speed LDR, which places the value you wish to view in an ARM register, followed by a debug speed STR, which shifts the value out of the JTAG port.

The correct process for writing a data value through the JTAG port into memory is to shift the address and data into ARM registers and execute a system speed STR.

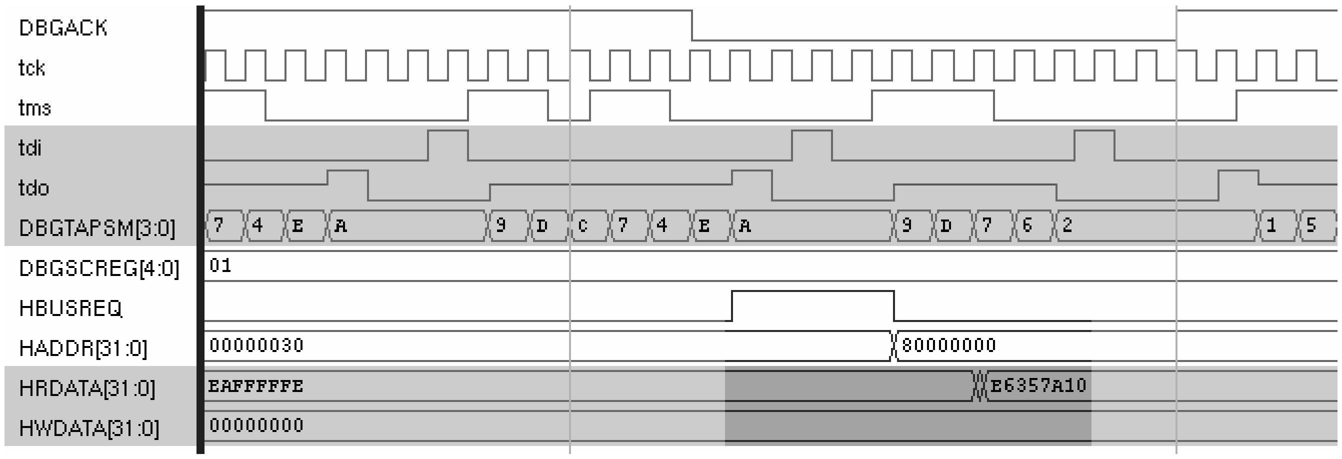
7.3 For reading one memory location (word)

Note: In the following example it is assumed that you have placed the address of the data item you wish to read into r0. Follow "Writing a register" (section 5.2) for details on how to accomplish this.

DR	{1'h0, 32'hE5901000}	Shift SYSSPEED = 0, the opcode for LDR r1, [r0]

RTI	-	Commit the LDR instruction to the pipeline
DR	{1’h1, 32’hE1A00000}	Shift SYSSPEED = 1, the opcode for NOP
RTI	-	
IR	RESTART	Shift RESTART instruction into the TAP IR
RTI	-	Go! Stay in RTI for as long as you need.

After the ARM core has reentered debug, r1 will contain the value from memory location [r0]. Follow the instructions under “Reading a register” (section 5.1) to read this value out of the JTAG port.



A memory access occurring as the result of debug activity (shown from RESTART instruction) – in this example, debug activity continues during memory access, but this isn’t necessary

7.4 For writing memory

Note: In the following example it is assumed that you have placed the address of the data item you wish to write into r0 and the data that you wish to write to that address into r1. Follow “Writing a register” (section 5.2) or “Writing multiple registers (section 6.2) for details on how to accomplish this.

DR	{1’h0, 32’hE5801000}	Shift SYSSPEED = 0, the opcode for STR r1, [r0]
RTI	-	Commit the STR instruction to the pipeline
DR	{1’h1, 32’hE1A00000}	Shift SYSSPEED = 1, the opcode for NOP
RTI	-	
IR	RESTART	Shift RESTART instruction into the TAP IR
RTI	-	Go! Stay in RTI for as long as you need.

Once the ARM core has completed the system speed STR and reentered debug, the external memory, cache or TCM at address [r0] will contain the data r1.

8 Exiting Debug

8.1 Exiting Debug – Calculating Return Address

Leaving Debug state involves performing a sequence of tasks which are described in detail in ARM9E-S B.7.3, page B-21. In general, a debug sequence upon completion would need to restore the internal state of the ARM9E-S core, branch to the next instruction to be executed in the main program and synchronize back to the core CLK. In order to branch back to the next instruction to be executed, a debug sequence or debugger needs to compute the return address to branch to upon exiting debug.

Most commercial debuggers automatically stores the program counter (PC) value on entry to debug and uses it for returning to the main program when debugging completes. If you are writing developing your own debugger, care has to be taken to ensure that your system returns to the correct address in the main program flow.

8.2 PC behavior during Debug Halt mode

Debug entry can be initiated from breakpoints, watchpoints or debug requests. In general, the behavior of PC in debug is similar for all three methods of debug entry. The only difference is that a watchpoint triggered debug allows the next instruction that caused the watchpoint to complete execution. For entry with breakpoints and debug requests, the return address is the instruction which generates the breakpoint or the instruction in the Execute pipeline stage when the debug request is asserted. For a watchpoint entry, the return address is the watchpointed address + 2 addresses. (More information can be found in ARM9E-S B.8, page B-23)

Upon entry to debug irregardless of the debug entry method, the PC advances by 16 bytes / 4 addresses in ARM state or 8 bytes / 4 addresses in Thumb state. Every debug instruction executed in Debug Halt mode adds 1 address to the PC and each system speed instruction adds 5 addresses to the PC. These are the general guidelines when calculating the return address. The next two sections describe two approaches of handling the PC on entry to debug state and how the return address can be calculated in each case.

8.3 Not saving PC value on entry into Debug state

This approach does not save the PC value on entry to debug. In order to calculate the return address, the number of debug and system speed instructions that have been executed has to be taken into consideration as each executed instruction causes PC to advance. This approach is only recommended when your debug sequence is a manageable size and to a certain extent predictable. The calculation of the return address when entered from ARM or Thumb state can be summarized as:

$$PC – (4+N+5S)$$

where N is the number of debug speed instructions executed (including the final branch), and S is the number of system speed instructions executed. The equation above gives you the number of addresses to subtract from PC. If you are operating in ARM state, the value obtained is multiplied by 4 and in Thumb mode by 2 to obtain the number of bytes to subtract from PC.

For example, if 2 system speed instruction (STMFD and LDMFD) and 5 debug speed instructions (2 ADD, 2 MOV and 1 SUB) were executed during debug state,

Substituting S = 2 and N = 5 into the above equation gives,

PC – (4+N+5S) = PC – 19 addresses = PC – 76 bytes (in ARM state, 1 address = 4 bytes)

Hence, the branch instruction for exiting debug state is **SUB pc, pc, #76**

8.4 Saving PC value on entry into Debug state

The second approach saves PC immediately to the stack on entry to debug halt mode and restores PC just before exiting debug. This approach allows a return address independent to the main debug sequence to be calculated. This relieves the programmer from having to keep track of the number of instructions that have been executed during debug.

The following instructions are example instructions that can be used to store the PC or PC + all general purpose registers to stack. These instructions have to be executed in system speed mode as the stack is assumed to reside in external memory with R13 pointing to its address location.

- **Store PC value to stack**

```
STR pc, [R13]

LDR pc, [R13]
```

- **Store PC value to and all general purpose registers to stack**

```
STMFD R13!, {R0-R12,R14-R15}

LDMFD R13!, {R0-R12,R14-R15}
```

As the PC is saved during the execution of a store or store multiple system speed instruction, the equation stated in section 8.3 cannot be directly employed. The PC value is saved to stack before the system speed instruction completes, hence cannot be calculated as one complete system speed instruction. If your debug sequence saves PC immediately upon entry to debug using a system speed instruction and restores PC just before exiting debug, you can assume that the PC has advanced 12 addresses. Therefore, your branch instruction might be **SUB pc, pc, #48** if you have executed ARM instructions.

If your scenario is different, a trial and error approach can always be used to determine the appropriate return address. Note that the LDR / LDRMFD instruction to restore the original value of the registers and PC value has to be issued just before the branch instruction for exiting debug state.

9 Hints and Tips

9.1 Saving tester time

You can save cycles by interleaving instructions. However, this is not a trivial task as you must understand how the ARM core pipeline works, when it will take a register from scan chain 1 and you must also manually work out each instructions time in the pipeline.

9.2 How to find out instruction opcodes

You have two choices here: you could either use the ARM Architecture Manual to manually work out the opcode for a particular instruction, or you can use the ARM assembler to compile the instruction into the opcode.

For using the ARM assembler, you will need to place the commands into an assembler file (e.g. temp.s, see below) and run:

```
armasm -unsafe -cpu 5TE -width 128 -list temp.lst -o temp.o \
  -Errors temp.err
```

Once this instruction has been run, you can view the opcode in temp.lst

9.2.1 temp.s example assembler file for armasm

```
AREA TestCode, CODE

CODE32

ADD r0, r1, r2
```