ARM JTAG 调试原理

OPEN-JTAG 开发小组

1 前言

这篇文章主要介绍 ARM JTAG 调试的基本原理。基本的内容包括了 TAP (TEST ACCESS PORT) 和 BOUNDARY-SCAN ARCHITECTURE 的介绍，在此基础上，结合 ARM7TDMI 详细介绍了的 JTAG 调试原理。

这篇文章主要是总结了前段时间的一些心得体会，希望对想了解 ARM JTAG 调试的网友们有所帮助。我个人对 ARM JTAG 的理解还不是很透彻，在文章中，难免会有偏失和不准确的地方，希望精通 JTAG 调试原理的大侠们不要拍砖，有什么问题提出来，我一定尽力纠正。同时也欢迎对 ARM JTAG 调试感兴趣的朋友们一起交流学习。

2 IEEE Standard 1149.1 - Test Access Port and Boundary-Scan Architecture

既然是介绍 JTAG 调试，还是让我们从 IEEE 的 JTAG 调试标准开始吧。JTAG 是 JOINT TEST ACTION GROUP 的简称。IEEE 1149.1 标准就是由 JTAG 这个组织最初提出的，最终由 IEEE 批准并且标准化的。所以，这个 IEEE 1149.1 这个标准一般也俗称 JTAG 调试标准。

接下来的这一部分，主要简单的介绍了 TAP (TEST ACCESS PORT) 和 BOUNDARY-SCAN ARCHITECTURE 的基本构架。虽然不是很全面，但对了解 JTAG 的基本原理来说，应该是差不离了。如果希望更全面深入的了解 JTAG 的工作原理，可以参考 IEEE 1149.1 标准。

2-1 边界扫描

在 JTAG 调试当中，边界扫描（Boundary-Scan）是一个很重要的概念。边界扫描技术的基本思想是在靠近芯片的输入输出管脚上增加一个移位寄存器单元。因为这些移位寄存器单元都分布在芯片的边界上（周围），所以被称为边界扫描寄存器（Boundary-Scan Register Cell）。当芯片处于调试状态的时候，这些边界扫描寄存器可以将芯片和外围的输入输出隔离开来。通过这些边界扫描寄存器单元，可以实现对芯片输入输出信号的观察和控制。对于芯片的输入管脚，可以通过与之相连的边界扫描寄存器单元把信号（数据）加载倒该管脚中去；对于芯片的输出管脚，也可以通过与之相连的边界扫描寄存器“捕获”（CAPTURE）该管脚上的输出信号。在正常的运行状态下，这些边界扫描寄存器对芯片来说是透明的，所以正常的运行不会受到任何影响。这样，边界扫描寄存器提供了一个便捷的方式用以观测和控制所需要调试的芯片。另外，芯片输入输出管脚上的边界扫描（移位）寄存器单元可以相互连接起来，在芯片的周围形成一个边界扫描链（Boundary-Scan Chain）。一般的芯片都会提供几条独立的边界扫描链，用来实现完整的测试功能。边界扫描链可以串行的输入和输出，通过相应的时钟信号和控制信号，就可以方便的观察和控制处在调试状态下的芯片。

利用边界扫描链可以实现对芯片的输入输出进行观察和控制。下一个问题是：如何来管理和使用这些边界扫描链？对边界扫描链的控制主要是通过 TAP（Test Access Port）Controller 来完成的。在下一个小节，我们一起来看看 TAP 是如何工作的。

2-2 TAP （TEST ACCESS PORT）

在上一节，我们已经简单介绍了边界扫描链，而且也了解了一般的芯片都会提供几条边界扫描链，用来实现完整的测试功能。下面，我将逐步介绍如何实现扫描链的控制和访问。

在 IEEE 1149.1 标准里面，寄存器被分为两大类：数据寄存器(DR－Data Register)和指令寄存器(IR－Instruction Register)。边界扫描链属于数据寄存器中很重要的一种。边界扫描链用来实现对芯片的输入输出的观察和控制。而指令寄存器用来实现对数据寄存器的控制，例如：在芯片提供的所有边界扫描链中，选择一条指定的边界扫描链作为当前的目标扫描链，并作为访问对象。下面，让我们从 TAP(Test Access Port)开始。

TAP 是一个通用的端口，通过 TAP 可以访问芯片提供的所有数据寄存器（DR）和指令寄存器（IR）。对整个 TAP 的控制是通过 TAP Controller 来完成的。TAP 总共包括 5 个信号接口 TCK、TMS、TDI、TDO 和 TRST ：其中 4 个是输入信号接口和另外 1 个是输出信号接口。一般，我们见到的开发板上都有一个 JTAG 接口，该 JTAG 接口的主要信号接口就是这 5 个。下面，我先分别介绍这个 5 个接口信号及其作用。

- **Test Clock Input (TCK)**
  TCK 为 TAP 的操作提供了一个独立的、基本的时钟信号，TAP 的所有操作都是通过这个时钟信号来驱动的。TCK 在 IEEE 1149.1 标准里是强制要求的。
- **Test Mode Selection Input (TMS)**
  TMS 信号用来控制 TAP 状态机的转换。通过 TMS 信号，可以控制 TAP 在不同的状态间相互转换。TMS 信号在 TCK 的上升沿有效。TMS 在 IEEE 1149.1 标准里是强制要求的。
- **Test Data Input (TDI)**
  TDI 是数据输入的接口。所有要输入到特定寄存器的数据都是通过 TDI 接口一位一位串行输入的（由 TCK 驱动）。TDI 在 IEEE 1149.1 标准里是强制要求的。
- **Test Data Output (TDO)**
  TDO 是数据输出的接口。所有要从特定的寄存器中输出的数据都是通过 TDO 接口一位一位串行输出的（由 TCK 驱动）。TDO 在 IEEE 1149.1 标准里是强制要求的。
- **Test Reset Input (TRST)**
  TRST 可以用来对 TAP Controller 进行复位（初始化）。不过这个信号接口在 IEEE 1149.1 标准里是可选的，并不是强制要求的。因为通过 TMS 也可以对 TAP Controller 进行复位（初始化）。

事实上，通过 TAP 接口，对数据寄存器（DR）进行访问的一般过程是：

- 通过指令寄存器（IR），选定一个需要访问的数据寄存器；
- 把选定的数据寄存器连接到 TDI 和 TDO 之间；
- 由 TCK 驱动，通过 TDI，把需要的数据输入到选定的数据寄存器当中去；同时把选定的数据寄存器中的数据通过 TDO 读出来。

接下来，让我们一起来了解一下 TAP 的状态机。TAP 的状态机如图 1 所示，总共有 16 个状态。在图中，每个六边形表示一个状态，六边形中标有该状态的名称和标识代码。图中的箭头表示了 TAP Controller 内部所有可能的状态转换流程。状态的转换是由 TMS 控制的，所以在每个箭头上有标有 tms = 0 或者 tms = 1。在 TCK 的驱动下，从当前状态到下一个状态的转换是由 TMS 信号决定。假设 TAP Controller 的当前状态为 Select-DR-Scan，在 TCK 的驱动下，如果 TMS ＝ 0，TAP Controller 进入 Capture-DR 状态；如果 TMS = 1，TAP Controller 进入 Select-IR-Scan 状态。

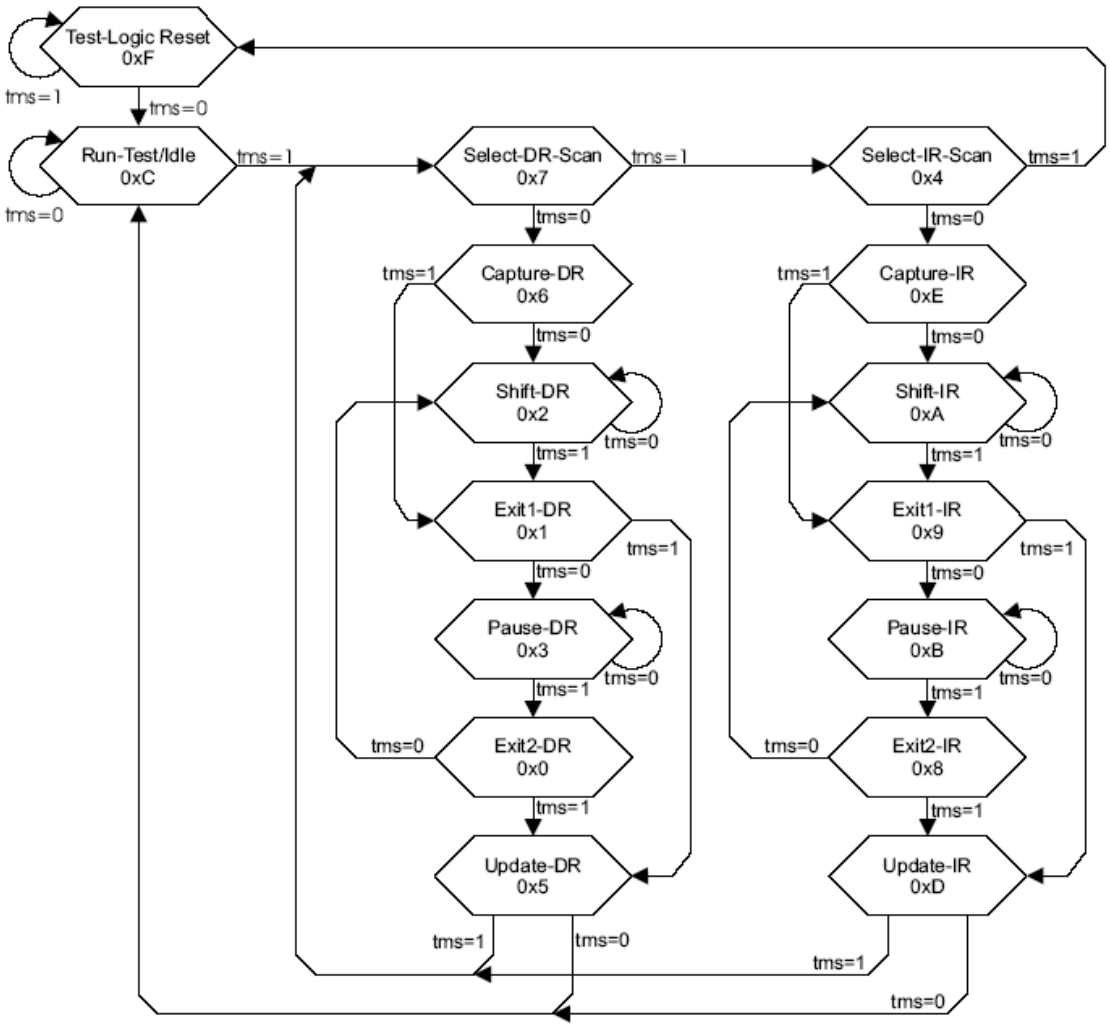


图 1. TAP Controller State Transitions

这个状态机看似很复杂，其实理解以后会发现这个状态机其实很直接、很简单。观察图 1，我们可以发现，除了 Test-Logic Reset 和 Test-Run/Idle 状态外，其他的状态有些类似。例如 Select-DR-Scan 和 Select-IR-Scan 对应，Capture-DR 和 Capture-IR 对应，Shift-DR 和 Shift-IR 对应，等等。在这些对应的状态中，DR 表示 Data Register，IR 表示 Instruction Register。记得我们前面说过吗，寄存器分为两大类，数据寄存器和指令寄存器。其实标识有 DR 的这些状态是用来访问数据寄存器的，而标识有 IR 的这些状态是用来访问指令寄存器的。

在详细描述整个状态机中的每一个状态之前，首先让我们来想一想：要通过边界扫描链来观察和控制芯片的输入和输出，需要做些什么？如果需要捕获芯片某个管脚上的输出，首先需要把该管脚上的输出装载到边界扫描链的寄存器单元里去，然后通过 TDO 输出，这样我们就可以从 TDO 上得到相应管脚上的输出信号。如果要在芯片的某个管脚上加载一个特定的信号，则首先需要通过 TDI 把期望的信号移位到与相应管脚相连的边界扫描链的寄存器单元里去，然后把该寄存器单元的值加载到相应的芯片管脚。下面，让我们一起来看看每个状态具体表示什么意思？完成什么功能？

Test-Logic Reset

系统上电后，TAP Controller 自动进入该状态。在该状态下，测试部分的逻辑电路全部被

禁用，以保证芯片核心逻辑电路的正常工作。通过 TRST 信号也可以对测试逻辑电路进行复位，使得 TAP Controller 进入 Test-Logic Reset 状态。前面我们说过 TRST 是可选的一个信号接口，这是因为在 TMS 上连续加 5 个 TCK 脉冲宽度的“1”信号也可以对测试逻辑电路进行复位，使得 TAP Controller 进入 Test-Logic Reset 状态。所以，在不提供 TRST 信号的情况下，也不会产生影响。在该状态下，如果 TMS 一直保持为“1”，TAP Controller 将保持在 Test-Logic Reset 状态下；如果 TMS 由“1”变为“0”（在 TCK 的上升沿触发），将使 TAP Controller 进入 Run-Test/Idle 状态。

Run-Test/Idle

这个是 TAP Controller 在不同操作间的一个中间状态。这个状态下的动作取决于当前指令寄存器中的指令。有些指令会在该状态下执行一定的操作，而有些指令在该状态下不需要执行任何操作。在该状态下，如果 TMS 一直保持为“0”，TAP Controller 将一直保持在 Run-Test/Idle 状态下；如果 TMS 由“0”变为“1”（在 TCK 的上升沿触发），将使 TAP Controller 进入 Select-DR-Scan 状态。

Select-DR-Scan

这是一个临时的中间状态。如果 TMS 为“0” （在 TCK 的上升沿触发），TAP Controller 进入 Capture-DR 状态，后续的系列动作都将以数据寄存器作为操作对象；如果 TMS 为“1” （在 TCK 的上升沿触发），TAP Controller 进入 Select-IR-Scan 状态。

Capture-DR

当 TAP Controller 在这个状态中，在 TCK 的上升沿，芯片输出管脚上的信号将被“捕获”到与之对应的数据寄存器的各个单元中去。如果 TMS 为“0”（在 TCK 的上升沿触发），TAP Controller 进入 Shift-DR 状态；如果 TMS 为“1” （在 TCK 的上升沿触发），TAP Controller 进入 Exit1-DR 状态。

Shift-DR

在这个状态中，由 TCK 驱动，每一个时钟周期，被连接在 TDI 和 TDO 之间的数据寄存器将从 TDI 接收一位数据，同时通过 TDO 输出一位数据。如果 TMS 为“0” （在 TCK 的上升沿触发），TAP Controller 保持在 Shift-DR 状态； 如果 TMS 为“1” （在 TCK 的上升沿触发），TAP Controller 进入到 Exit1-DR 状态。假设当前的数据寄存器的长度为 4。如果 TMS 保持为 0，那在 4 个 TCK 时钟周期后，该数据寄存器中原来的 4 位数据（一般是在 Capture-DR 状态中捕获的数据）将从 TDO 输出来；同时该数据寄存器中的每个寄存器单元中将分别获得从 TDI 输入的 4 位新数据。

Update-DR

在 Update-DR 状态下，由 TCK 上升沿驱动，数据寄存器当中的数据将被加载到相应的芯片管脚上去，用以驱动芯片。在该状态下，如果 TMS 为“0”，TAP Controller 将回到 Run-Test/Idle 状态；如果 TMS 为“1”，TAP Controller 将进入 Select-DR-Scan 状态。

Select-IR-Scan

这是一个临时的中间状态。如果 TMS 为“0” （在 TCK 的上升沿触发），TAP Controller 进入 Capture-IR 状态，后续的系列动作都将以指令寄存器作为操作对象；如果 TMS 为“1” （在 TCK 的上升沿触发），TAP Controller 进入 Test-Logic Reset 状态。

Capture-IR

当 TAP Controller 在这个状态中，在 TCK 的上升沿，一个特定的逻辑序列将被装载到指令寄存器中去。如果 TMS 为“0”（在 TCK 的上升沿触发），TAP Controller 进入 Shift-IR 状态；如果 TMS 为“1” （在 TCK 的上升沿触发），TAP Controller 进入 Exit1-IR 状态。

Shift-IR

在这个状态中，由 TCK 驱动，每一个时钟周期，被连接在 TDI 和 TDO 之间的指令寄存

器将从 TDI 接收一位数据，同时通过 TDO 输出一位数据。如果 TMS 为“0”（在 TCK 的上升沿触发），TAP Controller 保持在 Shift-IR 状态； 如果 TMS 为“1”（在 TCK 的上升沿触发），TAP Controller 进入到 Exit1-IR 状态。假设指令寄存器的长度为 4。如果 TMS 保持为 0，那在 4 个 TCK 时钟周期后，指令寄存器中原来的 4bit 长的特定逻辑序列（在 Capture-IR 状态中捕获的特定逻辑序列）将从 TDO 输出来，该特定的逻辑序列可以用来判断操作是否正确；同时指令寄存器将获得从 TDI 输入的一个 4bit 长的新指令。

Update-IR

在这个状态中，在 Shift-IR 状态下输入的新指令将被用来更新指令寄存器。

说了那么多，下面，让我们先看看指令寄存器和数据寄存器访问的一般过程，以便建立一个直观的概念。

1. 系统上电，TAP Controller 进入 Test-Logic Reset 状态，然后依次进入：Run-Test/Idle → Select-DR-Scan → Select-IR-Scan → Capture-IR → Shift-IR → Exit1-IR → Update-IR，最后回到 Run-Test/Idle 状态。在 Capture-IR 状态中，一个特定的逻辑序列被加载到指令寄存器当中；然后进入到 Shift-IR 状态。在 Shift-IR 状态下，通过 TCK 的驱动，可以将一条特定的指令送到指令寄存器当中去。每条指令都将确定一条相关的数据寄存器。然后从 Shift-IR → Exit1-IR → Update-IR。在 Update-IR 状态，刚才输入到指令寄存器中的指令将用来更新指令寄存器。最后，进入到 Run-Test/Idle 状态，指令生效，完成对指令寄存器的访问。
2. 当前可以访问的数据寄存器由指令寄存器中的当前指令决定。要访问由刚才的指令选定的数据寄存器，需要以 Run-Test/Idle 为起点，依次进入 Select-DR-Scan → Capture-DR → Shift-DR → Exit1-DR → Update-DR，最后回到 Run-Test/Idle 状态。在这个过程当中，被当前指令选定的数据寄存器会被连接在 TDI 和 TDO 之间。通过 TDI 和 TDO，就可以将新的数据加载到数据寄存器当中去，同时，也可以捕获数据寄存器中的数据。具体过程如下。在 Capture-DR 状态中，由 TCK 的驱动，芯片管脚上的输出信号会被“捕获”到相应的边界扫描寄存器单元中去。这样，当前的数据寄存器当中就记录了芯片相应管脚上的输出信号。接下来从 Capture-DR 进入到 Shift-DR 状态中去。在 Shift-DR 状态中，由 TCK 驱动，在每一个时钟周期内，一位新的数据可以通过 TDI 串行输入到数据寄存器当中去，同时，数据寄存器可以通过 TDO 串行输出一位先前捕获的数据。在经过与数据寄存器长度相同的时钟周期后，就可以完成新信号的输入和捕获数据的输出。接下来通过 Exit1-DR 状态进入到 Update-DR 状态。在 Update-DR 状态中，数据寄存器中的新数据被加载到与数据寄存器的每个寄存器单元相连的芯片管脚上去。最后，回到 Run-Test/Idle 状态，完成对数据寄存器的访问。

上面描述的就是通过 TAP 对数据寄存器进行访问的一般流程。会不会还是觉得很抽象？让我们来看一个更直观的例子。现在假设，TAP Controller 现在处在 Run-Test/Idle 状态，指令寄存器当中已经成功的写入了一条新的指令，该指令选定的是一条长度为 6 的边界扫描链。下面让我们来看看实际如何来访问这条边界扫描链。图 2 所示的是测试芯片及其被当前指令选定的长度为 6 的边界扫描链。由图 2 可以看出，当前选择的边界扫描链由 6 个边界扫描移位寄存器单元组成，并且被连接在 TDI 和 TDO 之间。TCK 时钟信号与每个边界扫描移位寄存器单元相连。每个时钟周期可以驱动边界扫描链的数据由 TDI 到 TDO 的方向移动一位，这样，新的数据可以通过 TDI 输入一位，边界扫描链的数据可以通过 TDO 输出一位。经过 6 个时钟周期，就可以完全更新边界扫描链里的数据，而且可以将边界扫描链里捕获的 6 位数据通过

TDO 全部移出来。

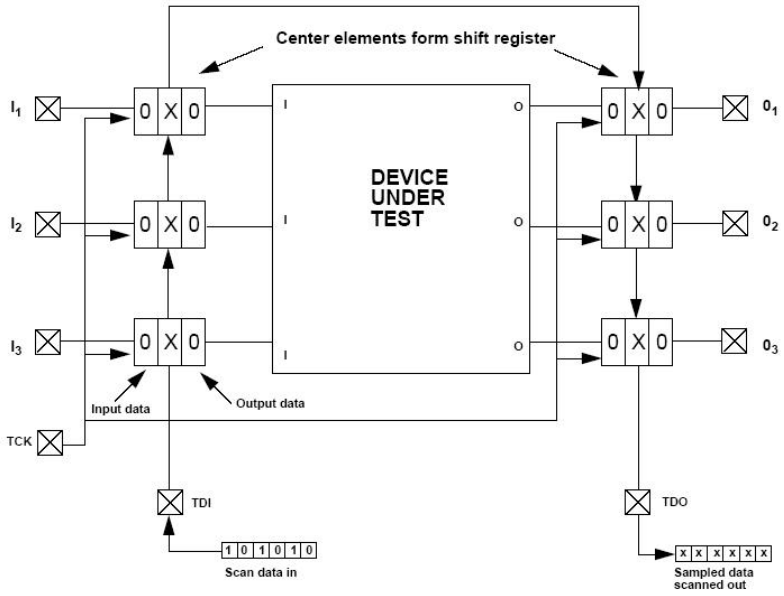
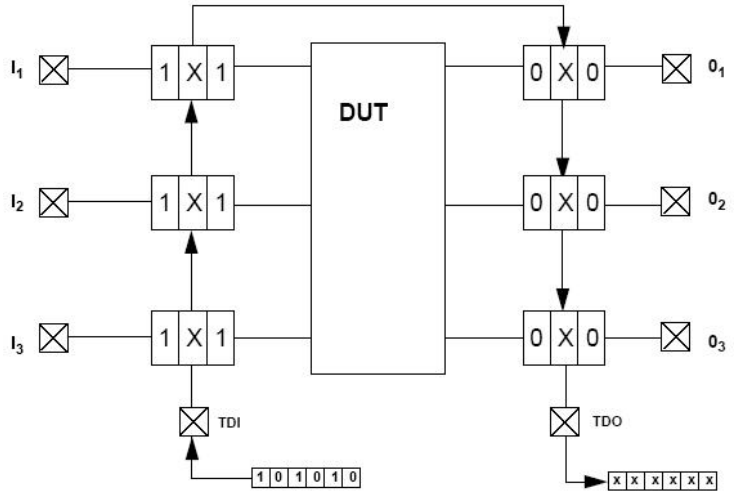
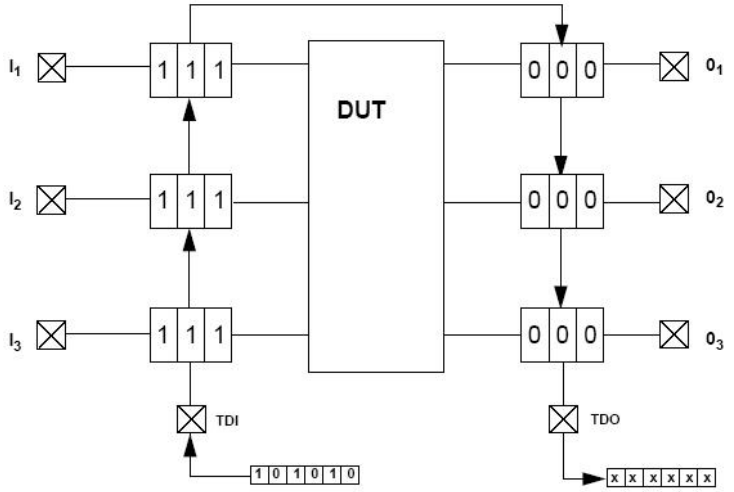


图 2. 测试芯片及其当前选定的边界扫描链

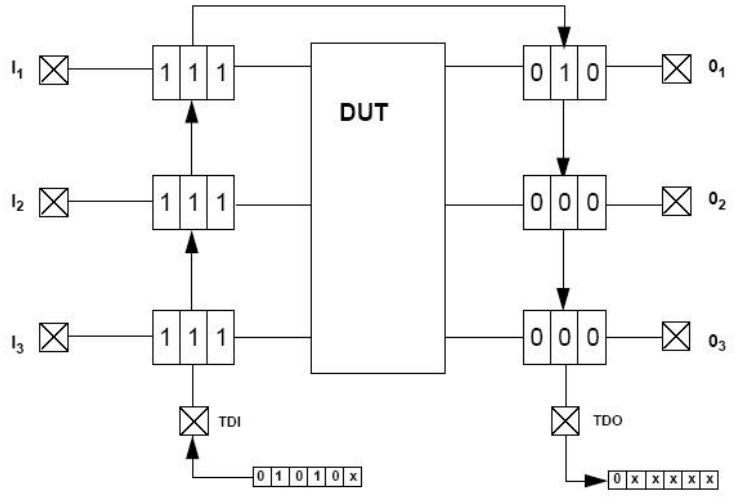
图 3 表示了边界扫描链的访问过程。图 3.1 表示了芯片和边界扫描链的初始化状态，在测试状态下，芯片的外部输入和输出被隔离开了，芯片的输入和输出可以通过相应的边界扫描链来观察和控制。在图 3.1 中，扫描链里的每个移位寄存器单元的数据是不确定的，所以在图中用 X 表示，整个扫描链里的数据序列是 XXXXXX。要从 TDI 输入到测试芯片上的数据序列是：101010. 同时要从 TDO 得到芯片相应管脚上的状态。现在 TAP Controller 从 Run-Test/Idle 状态经过 Select-DR-Scan 状态进入到 Capture-DR 状态，在 Capture-DR 状态当中，在一个 TCK 时钟的驱动下，芯片管脚上的信号状态全部被捕获到相应的边界扫描移位寄存器单元当中去，如图 3.2 所示。从图 3.2 中我们可以看出，在进入 Capture-DR 状态后，经过一个 TCK 时钟周期，现在扫描链中的数据序列变成了：111000. 在数据捕获完成以后，从 Capture-DR 状态进入到 Shift-DR 状态。在 Shift-DR 状态中，我们将通过 6 个 TCK 时钟周期来把新的数据序列（101010）通过 TDI 输入到边界扫描链当中去；同时，将边界扫描链中捕获的数据序列（111000）通过 TDO 输出来。在进入到 Shift-DR 状态后，每经过一个 TCK 时钟驱动，边界扫描链从 TDO 输出一位数据；同时，从 TDI 接收一位新的数据。图 3.3 所示的是在 Shift-DR 状态下，1 个 TCK 时钟周期后的扫描链的变化。图 3.4 所示的是在 Shift-DR 状态下，2 个 TCK 时钟周期后的扫描链的变化。此时，扫描链已经从 TDI 串行得到了两位新数据，从 TDO 也串行输出了两位数据。在 TCK 时钟的驱动下，这个过程一直继续下去。图 3.5 所示的是在经过 6 个 TCK 时钟周期以后扫描链的情况。从图 3.5 中我们可以看到：边界扫描链当中已经包含了新的数据序列：101010. 在 TDO 端，经过 6 个 TCK 时钟驱动以后，也接收到了在 Capture-DR 状态下捕获到的数据序列：111000. 到目前为止，虽然扫描链当中包含了新的数据序列：101010，但测试芯片的管脚上的状态还是保持为：111000. 下一步，需要更新测试芯片相应管脚上的信号状态。要实现更新，TAP Controller 从 Shift-DR 状态，经过 Exit1-DR 状态，进入到 Update-DR 状态。在 Update-DR 状态中，经过一个周期的 TCK 时钟驱动，边界扫描链中的新数据序列将被加载到测试芯片的相应管脚上去，如图 3.6 所示。从图 3.6 可以看出，测试芯片的状态已经被更新，相应管脚上的状态序列已经从 111000 变为 101010. 最后从 Update-DR 状态回到 Run-Test/Idle 状态，完成对选定的边界扫描链的访问。



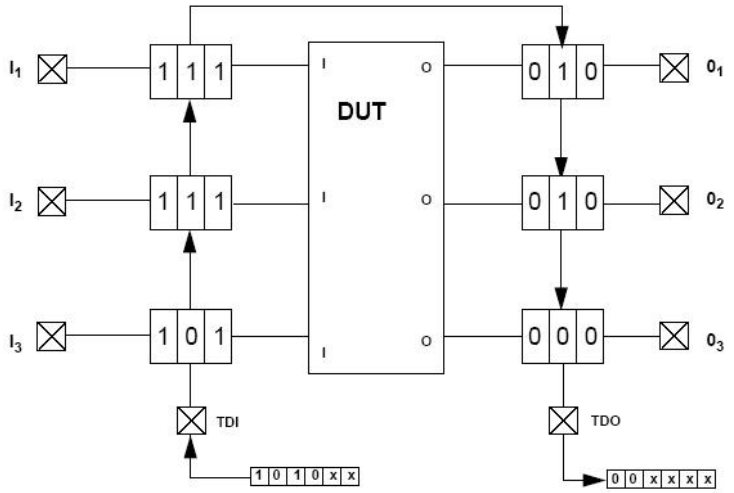
（1）初始化状态



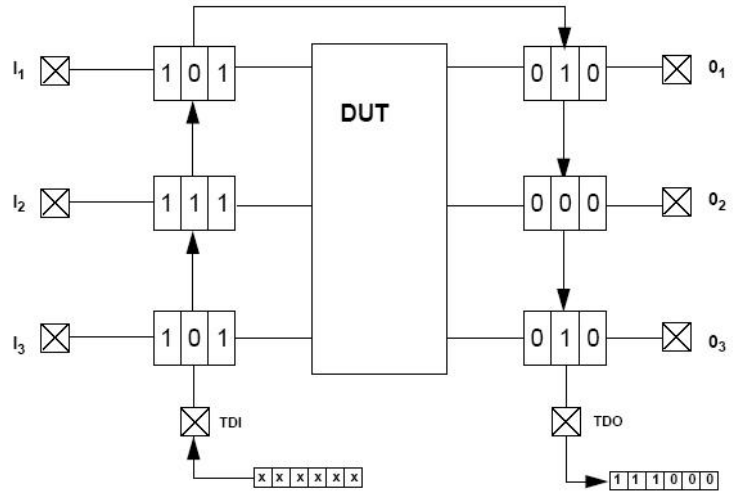
（2）CAPTURE-DR



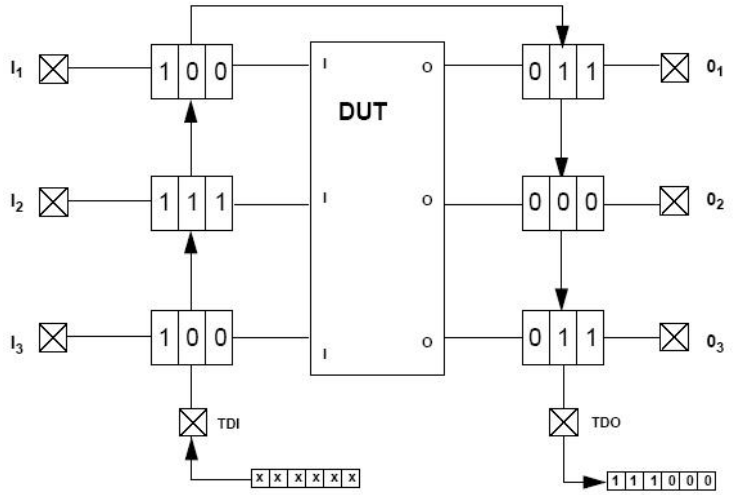
（3）SHIFT-DR ＋ 1 TCK



（4）SHIFT-DR ＋ 2 TCK



（5）SHIFT-DR ＋ 6 TCK



（6）UPDATE-DR

图 3. 边界扫描链的访问过程

在看完上面这个例子以后，对 TAP Controller 的状态机应该大概了解了吧？对如何访问边界扫描链应该也有个直观的概念了吧？虽然上面的这个例子只是说明了如何访问边界扫描链，对其它的数据寄存器、指令寄存器的访问过程也是类似的。要实现对指令寄存器的访问，不同的是 TAP Controller 必须经过不同的状态序列：Run-Test/Idle → Select-DR-Scan → Select-IR-Scan → Capture-IR → Shift-IR → Exit1-IR → Update-IR → Run-Test/Idle.

2-3 指令寄存器、公共指令以及数据寄存器

在 IEEE 1149.1 标准当中，规定了一些指令寄存器、公共指令和相关的一些数据寄存器。对于特定的芯片而言，芯片厂商都一般都会在 IEEE 1149.1 标准的基础上，扩充一些私有的指令和数据寄存器，以帮助在开发过程中进行进行方便的测试和调试。在这一部分，我将简单介绍 IEEE 1149.1 规定的一些常用的指令及其相关的寄存器。与 ARM7TDMI 相关的私有指令和寄存器将在后面的部分专门介绍。

指令寄存器：

指令寄存器允许特定的指令被装载到指令寄存器当中，用来选择需要执行的测试，或者选择需要访问的测试数据寄存器。每个支持 JTAG 调试的芯片必须包含一个指令寄存器。

BYPASS 指令和 Bypass 寄存器：

Bypass 寄存器是一个一位的移位寄存器，通过 BYPASS 指令，可以将 bypass 寄存器连接到 TDI 和 TDO 之间。在不需要进行任何测试的时候，将 bypass 寄存器连接在 TDI 和 TDO 之间，在 TDI 和 TDO 之间提供一条长度最短的串行路径。这样允许测试数据可以快速的通过当前的芯片送到开发板上别的芯片上去。

IDCODE 指令和 Device Identification 寄存器：

Device identification 寄存器中可以包括生产厂商的信息，部件号码，和器件的版本信息等。使用 IDCODE 指令，就可以通过 TAP 来确定器件的这些相关信息。例如，ARM MULTI-ICE 可以自动识别当前调试的是什么片子，其实就是通过 IDCODE 指令访问 Device Identification 寄存器来获取的。

INTEST 指令和 Boundary-Scan 寄存器：

Boundary-Scan 寄存器就是我们前面例子中说到的边界扫描链。通过边界扫描链，可以进行部件间的连通性测试。当然，更重要的是可以对测试器件的输入输出进行观测和控制，以达到测试器件的内部逻辑的目的。INTEST 指令是在 IEEE 1149.1 标准里面定义的一条很重要的指令：结合边界扫描链，该指令允许对开发板上器件的系统逻辑进行内部测试。在 ARM JTAG 调试当中，这是一条频繁使用的测试指令。

我们前面说过，寄存器分为两大类：指令寄存器和数据寄存器。在上面提到的 Bypass 寄存器、Device Identification 寄存器和 Boundary-scan 寄存器（边界扫描链），都属于数据寄存器。在调试当中，边界扫描寄存器（边界扫描链）最重要，使用的也最为频繁。

3 ARM7TDMI 调试构架

在这个小节里，我将介绍一下 ARM7TIMI 的调试构架，让大家对 ARM7TDMI JTAG 调有个大概的认识。细节问题将在下一小节中介绍。

ARM7TDMI 典型的调试系统结构如图 4 所示。一个调试系统一般包括三个部分：调试主机、协议转换器和调试目标。

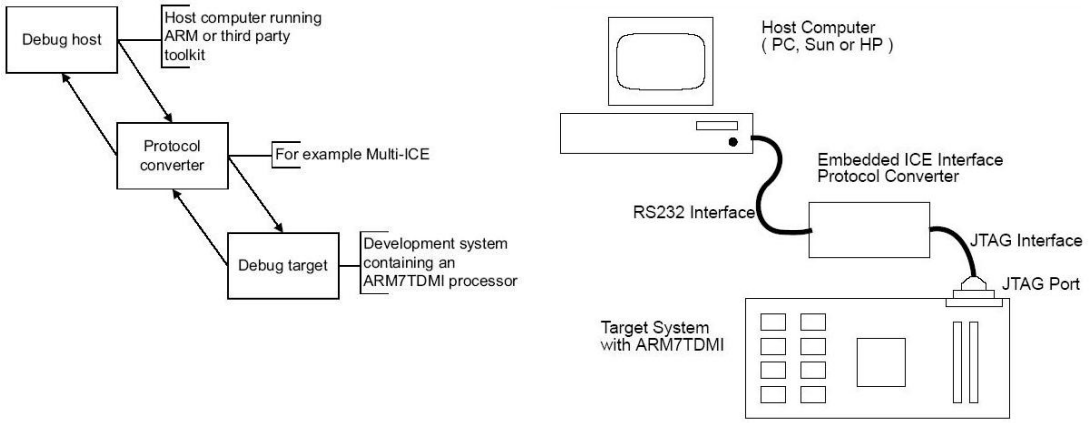


图 4. ARM7TDMI 典型的系统调试结构

调试主机一般是一台运行调试软件（例如 ADS）的计算机。调试主机可以发出一些高层的调试命令，例如设置断点、访问内存等。协议转换器（例如 MULTI-ICE），用来将调试主机发出的高层调试命令转换为底层的 ARM JTAG 调试命令。调试目标一般就是指基于 ARM7TDMI 内核 MCU 目标开发板。经过协议转换器进行命令解释，主机上运行的调试软件就可以通过 JTAG 接口直接和 ARM7TDMI 内核对话。通过扫描链，可以把 ARM/THUMB 指令插入到 ARM7TDMI 的指令流水线当中去执行。通过插入特定 ARM/THUMB 指令，我们可以检查、保存或者改变内核和系统的状态。为了支持底层的调试，ARM7TDMI 处理器提供了硬件上的扩展，。这些调试扩展包括：

- 停止程序的运行
- 检查和修改 ARM7TDMI 的内核状态
- 观察和修改内存
- 恢复程序的运行

ARM7TDMI 处理器的结构框图如下图所示：

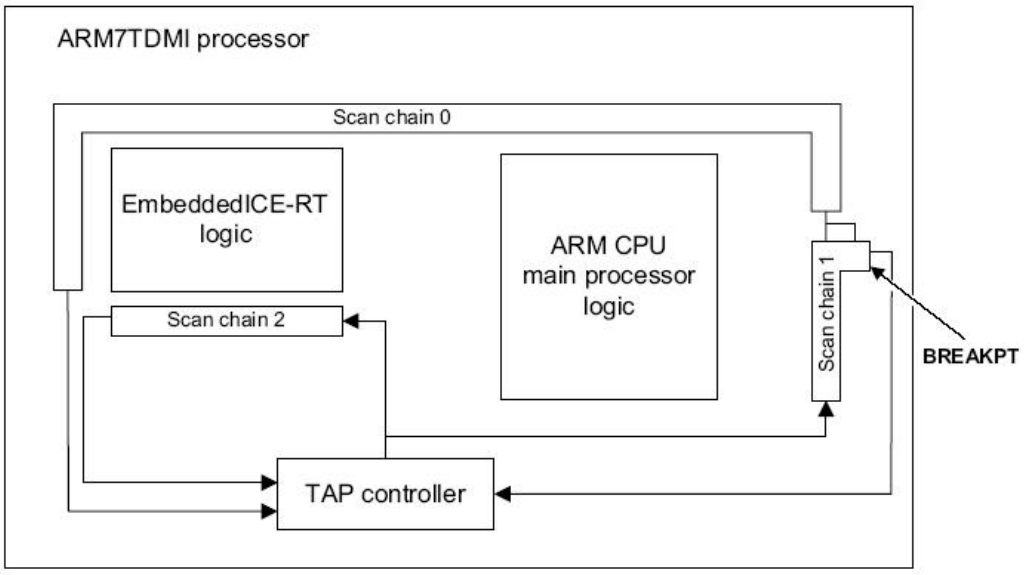


图 5. ARM7TDMI 处理器结构框图

从图 5 可以看到，ARM7TDMI 处理器主要包括三大部分：

- **ARM CPU Main Processor Logic**
  这部分包括了对调试的硬件支持；
- **Embedded ICE-RT Logic**
  这部分包括了一组寄存器和比较器，用来产生调试异常、设置断点和观察点；
- **TAP Controller**
  TAP Controller 通过 JTAG 接口来控制和操作扫描链。

从图 5，我们可以发现 ARM7TDMI 提供了 4 条扫描链：Scan Chain 0, 1, 2 & 3.

- **Scan Chain 0**
  通过扫描链 0 可以访问 ARM7TDMI 内核的外围电路，包括数据总线。通过该扫描链可以进行芯片间的测试（EXTEST）和芯片的内部测试（INTEST）。该扫描链长度为 113 位，具体包括：数据总线的 0－31 位，内核控制信号，地址总线的 31－0 位，EmbeddedICE-RT 的控制信号。
- **Scan Chain 1**
  扫描链 1 是扫描链 0 的子集，长度为 33 位，具体包括：数据总线的 0－31 位、BREAKPT 信号。扫描链 1 比扫描链 0 的长度短了很多，通过扫描链 1 可以更快的插入指令或者是数据到 ARM7TDMI 的内部。
- **Scan Chain 2**
  扫描链 2 长度为 38 位，该扫描链是专门用来访问 EmbeddedICE-RT 内部的寄存器。通过访问 EmbeddedICE-RT 的内部寄存器，可以让 ARM7TDMI 进入调试状态、设置断点、设置观察点。
- **Scan Chain 3**
  通过扫描链 3，ARM7TDMI 可以访问外部的边界扫描链。该扫描链用的很少，在此就不介绍了，想了解的网友可以参考 ARM7TDMI 手册。

在实际的调试过程中，扫描链 1 和扫描链 2 用的最多。基本上所有的调试动作都是通过这两条扫描链来完成的。

ARM7TDMI 还提供了 3 个附加的信号：DBGRQ、DBGACK 和 BREAKPT. 这 3 个信号都被 EmbeddedICE-RT 控制，可以用来使得处理器从正常的运行状态进入调试状态或从调试状态返回到正常的运行状态。

- **DBGREQ**
  调试请求，通过把 DBGREQ 置“1”，可以迫使 ARM7TDMI 进入调试状态。
- **DBGACK**
  调试确认，通过 DBGACK，可以判断当前 ARM7TDMI 是否在调试状态。
- **BREAKPT**
  断点信号，这个信号是输入到 ARM7TDMI 处理器内核的。如果 BREAKPT 被置“1”，当前的内存访问被设置为断点。如果当前的内存访问是取指令的话，当该指令被执行的时候，ARM7TDMI 处理器自动进入调试状态；如果当前的内存访问是存取数据的话，在存储完成以后，ARM7TDMI 处理器自动进入调试状态。

4 ARM7TDMI 调试原理

在这个小节里，我将详细介绍 ARM7TIMI 的调试原理，并对一些比较感兴趣的问题进行讨论。例如，如何设置断点？如何让 ARM7TDMI 进入调试状态？ 在调试状态下，如何访问 ARM7TDMI 的内部寄存器？如何访问内存单元？

在继续之前，看看 ARM7TDMI 的正常运行状态和调试状态的区别。在正常运行状态下，ARM7TDMI 由 MCLK（Memory Clock）驱动，正常运行。在调试状态下，ARM7TDMI 的正常运行被打断，并且和系统的其它部分隔离开来。在调试状态下，我们可以通过插入特定的 ARM/THUMB 的指令来读写 ARM7TDMI 的内部寄存器和修改内存的内容。在调试状态下，ARM7TDMI 由内部的调试时钟 DCLK（Debug Clock）驱动。DCLK 比 MCLK 慢很多，所以在调试状态下，插入的 ARM/THUMB 指令的运行速度会比较慢（现对而言）。在完成需要的操作后，可以用 RESTART JTAG 指令让 ARM7TDMI 返回到正常运行状态，恢复原来的运行。

4-1 ARM7TDMI 的指令寄存器及常用 JTAG 指令

ARM7TDMI 的指令寄存器长度是 4 位，通过 TAP 和 JTAG 接口，可以把指令装载到指令寄存器当中去。在 CAPTURE-IR 状态下，固定值 B0001 总是被装载到指令寄存器当中去。在 SHIFT-IR 状态中，可以把 ARM7TDMI 支持的新指令从 TDI 串行输入，同时固定值 B0001 会从 TDO 串行输出。通过这个输出的固定值，可以判断当前的操作是否正确。在 UPDATE-IR 状态，新输入的指令被装载到指令寄存器当中去。最后回到 RUN-TEST/IDLE 状态后，新指令立即生效。

下面先来看看在 ARM7TDMI 调试当中经常用到的几条 JTAG 指令。

- **IDCODE**
  该指令的二进制代码是 1110。IDCODE 指令将 Device Identification Code 寄存器连接到 TDI 和 TDO 之间。Device Identification Code 寄存器的长度是 32 位，通过 TAP 就可以将 ARM7TDMI 的 ID 给读出来。ARM7TDMI 的 ID 是 0x1F0F0F0F. 在 CAPTURE-DR 状态下，器件的 ID 被捕获到 ID 寄存器里面。在 SHIFT-DR 状态下，先前捕获的 ID 通过 TDO 位移出来，总共需要 32 个 TCK 时钟周期。在 UPDATE-DR 状态下，ID 寄存器不受任何影响。
- **SCAN_N**
  该指令的二进制代码是 0010。ARM7TDMI 提供了 4 条扫描链，通过 SCAN_N 指令可以选择需要访问的扫描链。一般来说，需要访问 Embedded ICE-RT 的寄存器时，选择扫描链 2；需要插入指令到 ARM7TDMI 内核去执行的时候，选择扫描链 1。选择扫描链的过程如下：先把 SCAN_N 指令装载到指令寄存器当中去，该指令会将长度为 4 位的扫描链选择寄存器连接到 TDI 和 TDO 之间；然后进入到 CAPTURE-DR 状态，在这个状态下，固定值 B1000 将被捕获到扫描链选择寄存器当中去；在 SHIFT-DR 状态下，将需要选择的扫描链的号码通过 TDI 输入到扫描链选择寄存器当中去；在 UPDATE-DR 状态下，被选择的扫描链将被连接到 TDI 和 TDO 之间。在 TAP 被复位以后，默认状态下选择的是扫描链 3。在使用 SCAN_N 选定了一条扫描链后，当前选定的扫描链会一直保持不变，直到在下一次调用 SCAN_N 选择另外的扫描链。
- **BYPASS**
  该指令的二进制代码是 1111。BYPASS 指令将 1-BIT 长的 BYPASS 寄存器连接到 TDI 和 TDO 之间。
- **INTEST**
  该指令的二进制代码是 1100。INTEST 指令将通过 SCAN_N 选定的扫描链置于内部测

试模式。

- **RESTART**
  该指令的二进制代码是 0100。RESTART 指令用来使 ARM7TDMI 处理器从调试状态退回到正常的运行状态。

4-2 EmbeddedICE-RT Logic

在这个小节里，主要介绍通过扫描链 2 来访问 EmbeddedICE-RT 内部的寄存器。EmbeddedICE-RT 内部包括了丰富的寄存器，通过这些寄存器，可以控制 ARM7TDMI 进入或者退出调试状态；可以设置断点和观察点。

先来看看 EmbeddedICE-RT 包括那些寄存器，下面这个表里罗列了 EmbeddedICE-RT 中包含的所有寄存器。不同的寄存器有不同的长度，而且被分配了一个长度为 5 的地址。其中 Debug Control Register 用来控制 EmbeddedICE-RT；通过 Debug Status Register 可以查询当前系统的状态；Abort Status Register 用来确定异常的产生原因：断点、观察点还是真的异常；Debug Comms Control Register 和 Debug Comms Data Register 是用来控制和操作 Debug Communication Channel 的；后面的所有寄存器都是关于 WATCH POINT 的，设置断点和观察点就是通过设置这些寄存器来实现的。后面我将逐一的进行介绍。

TABLE I. Mapping of EmbeddedICE-RT Registers

Address	Width	Function
00000	6	Debug control
00001	5	Debug status
00010	1	Abort status
00100	6	Debug comms control register
00101	32	Debug comms data register
01000	32	Watchpoint 0 address value
01001	32	Watchpoint 0 address mask
01010	32	Watchpoint 0 data value
01011	32	Watchpoint 0 data mask
01100	9	Watchpoint 0 control value
01101	8	Watchpoint 0 control mask
10000	32	Watchpoint 1 address value
10001	32	Watchpoint 1 address mask
10010	32	Watchpoint 1 data value
10011	32	Watchpoint 1 data mask
10100	9	Watchpoint 1 control value
10101	8	Watchpoint 1 control mask

要访问 EmbeddedICE-RT 内部的寄存器，必须通过扫描链 2 来实现。扫描链的长度是 38 位，其中 0－31 位为数据位，32－36 位用来标识要访问的寄存器的 5 位宽的地址;；第 37 位用来标识对当前寄存器进行读操作还是写操作。如果是对当前寄存器进行写操作，0－31 的

数据位用来标识需要写入到寄存器的数据；如果对当前寄存器进行读操作，0－31 的数据位用来存储从寄存器中读出的数据。假设我们现在想访问 Debug Control Register，对其进行写操作，首先需要选择扫描链 2，将其连接到 TDI 和 TDO 之间，然后通过扫描链 2 对 Debug Control Register 进行访问。其过程如下所述。通过 TAP 将 SCAN_N 指令写入到指令寄存器当中去，TAP 状态转换如下：RUN-TEST/IDLE → SELECT-DR-SCAN → SELECT-IR-SCAN → CAPTURE-IR → SHIFT-IR → EXIT1-IR → UPDATE-IR → RUN-TEST/IDLE，在 SHIFT-IR 状态下，将 SCAN_N 通过 TDI 写到指令寄存器中去；接下来，访问被 SCAN_N 指令连接到 TDI 和 TDO 直接的扫描链选择寄存器，通过将 2 写入到扫描链选择寄存器当中去，以将扫描链 2 连接到 TDI 和 TDO 之间，TAP 状态转换过程如下：RUN-TEST/IDLE → SELECT-DR-SCAN → CAPTURE-DR → SHIFT-DR → EXIT1-DR → UPDATE-DR → RUN-TEST/IDLE，在 SHIFT-DR 状态下，将数值 2 通过 TDI 写到扫描链选择寄存器当中去。在通过扫描链 2 访问任何 EmbeddedICE-RT 内部寄存器之前，我们还需要用 INTEST 指令将当前通过 SCAN_N 指令选择的扫描链置内部测试状态。写入 INTEST 指令的过程和写入 SCAN_N 指令的过程类似。在此就不多说了。好，至此，我们已经选择了扫描链 2，并且将它置于内部测试状态。接下来，我们就可以通过扫描链 2 访问 EmbeddedICE-RT 内部寄存器了。下面，以写 Debug Control Register 为例，看看怎样实现访问。假设要将 Debug Control Register 的 6 位全部置“1”，按照扫描链 2 的格式，需要写入到扫描链 2 的序列应该为：

1, 00000,0000,0000,0000,0000,0000,0000,0011,1111

在上面这个长度为 38 位的序列当中，5 位红色标识的数列是 Debug Control Register 的地址；最高位蓝色标识的 1 表示对 Debug Control Register 进行写操作；黑色表示的 32 位是要写入到 Debug Control Register 的数据，因为 Debug Control Register 长度为 6，所以只有低 6 为的数据序列 111111 有效。要将上面长度为 38 位的序列写入到扫描链 2 中，TAP 状态转换过程如下：RUN-TEST/IDLE → SELECT-DR-SCAN → CAPTURE-DR → SHIFT-DR → EXIT1-DR → UPDATE-DR → RUN-TEST/IDLE。在 SHIFT-DR 状态下，通过 38 个 TCK 时钟驱动，就可以将上面的序列串行输入到扫描练 2 当中去。在回到 RUN-TEST/IDLE 状态后，Debug Control Register 的值就会被改写为 111111。

我们已经知道如何访问 EmbeddedICE-RT 的内部寄存器了，下面让我们一起来了解了解各个寄存器的作用。

- **Debug Control Register**

  从该寄存器的名称可以看出，该寄存器是用来控制调试的。下面这个表是 Debug Control 寄存器的格式。

5	4	3	2	1	0
EmbeddedICE-RT disable	Monitor mode enable	SBZ/RAZ	**INTDIS**	**DBGRQ**	**DBGACK**

  DBGACK，用来控制 DBGACK 信号的值，通过 Debug Control 寄存器，可以设置 DBGACK 的值。个人的感觉是，不要人为的去设置这个值，让 EmbeddedICE-RT 自动控制。DBGRQ，是调试请求信号，通过将该信号置“1”，可以强制 ARM7TDMI 暂停当前的指令，进入调试状态。INTDIS 用来控制中断。SBZ/RAZ 任何时候都必须被置“0”。Monitor Mode Enable 用来控制是否进入 Monitor 模式，我没有用过这个控制位，按我个人的理解，该模式在调试当中一般不会用到。EmbeddedICE-RT Disable，很明显，这个控制位用来控制整个 EmbeddedICE-RT，是启用还是禁用。如果禁用的话，ARM7TDMI 将一直保持在正常的运行状态。总结一下，通过 Debug Control 寄存器，可以强制让 ARM7TDMI 进入调试状态。这是第一种让 ARM7TDMI 进入调试状态的方法，另外，也可以通过断点或者是观察点来进入调试状态，每当设置的断点或

者观察点被触发后，ARM7TDMI 自动进入调试状态。在我们调试的时候，第一步一般都是先用 DBGRQ 信号使得 ARM7TDMI 进入调试状态，然后将我们需要调试的程序装载到内存里去，接着在需要的地方设置断点和观察点，开始调试（单步运行、全速运行、让断点或者观察点触发）。

- **Debug Status Register**

  从该寄存器的名称可以看出，该寄存器是用来保存当前低调试状态的。虽然参手册上说该寄存器可读可写，不过建议对该寄存器只进行读操作。该寄存器的长度位 5，格式如下：

4	3	2	1	0
TBIT	cgenL	IFEN	DBGRQ	DBGACK

  DBGACK 信号用来标识当前系统是否处于调试状态，当 ARM7TDMI 进入调试状态后，该信号会被自动置“1”。所以，通过查询该位，就可以判断 ARM7TDMI 当前的状态。DBGRQ 信号用来标识 DBGRQ 信号的当前状态，要设置 DBGRQ 信号请访问 Debug Control 寄存器。IFEN 用来标识系统的中断控制状态：启用还是禁用？cgenL，可以用来判断当前对调试器（DEBUGGER）在调试状态下对内存的访问是否完成。TBIT，该位用来判断 ARM7TDMI 是从 ARM 状态还是 THUMB 状态进入到调试状态的。

- **Abort Status Register**

  该寄存器的长度为 1，该寄存器用来判断一个异常的产生的原因：断点触发？观察点触发？还是一个真的异常？

- **Watch Point 0/1 Address Value/Mask Register**

  Watch Point 0/1 Data Value/Mask Register

  Watch Point 0/1 Control Value/Mask Register

  关于 WATCH POINT 的 6 个寄存器，因为联系比较紧密，我在这里对这 6 个寄存器一并介绍。ARM7TDMI 有两组 WATCH POINT 寄存器，所以总共有 12 个寄存器，在这里我只介绍其中的一组，另外一组的功能完全是一样的。看看上面提到的 6 个寄存器，其实是 3 对寄存器，其中 3 个是 Value 寄存器，另外 3 个是 Mask 寄存器，通过 Mask 寄存器的设置，可以屏蔽相应的 Value 寄存器的某些或全部数据位，在比较的时候不予以考虑。WP Address Value/Mask 是用来监控地址总线的，WP Data Value/Mask 是用来监控数据总线的。每组寄存器都与一个比较器相联系，该比较器的比较结果决定了断点条件是否满足。每次对存储空间进行访问，Watch Point 的比较器都会判断当前访问的地址和 WP Address Value 寄存器的中的地址是否匹配？该地址中的数据和 WP Data Value 寄存器中的数据是否匹配？ 如果地址和数据同时都匹配，BREAKPT 信号会被置 1，当前的内存访问被设置为断点。如果当前的内存访问是取指令的话，当该指令被执行的时候，ARM7TDMI 处理器自动进入调试状态；如果当前的内存访问是存取数据的话，在存储完成以后，ARM7TDMI 处理器自动进入调试状态。通过相应的 MASK 寄存器，可以使得比较器的使用很灵活。例如，我们只考虑地址是否匹配，可将 WP Data Mask 寄存器的值设置为：0xFFFFFFFF. 这样的话，数据部分总是被认为是匹配的，是实际上起作用的就是地址和 WP Address Value 寄存器的值是否匹配；反过来，如果我们只希望考虑数据是否匹配，而不考虑地址是否匹配，可以将 WP Address Mask 寄存器的值设置为 0xFFFFFFFF. 这样的话，地址部分总是被认为是匹配的，实际上起作用的就是数据和 WP Data Value 寄存器的值是否匹配。另外一个寄存器，WP Control Value/Mask 是用来控制该组的 WP 寄存器的，同时提供些附加的

比较条件（信息）。下面逐一介绍。

WP Control Value/Mask Register

WP Control Value 寄存器的长度为 9 位，WP Control Mask 寄存器的长度为 8 位，因为 WP Control Value 寄存器的最高位是用来控制启用/禁止 WP 功能的，该位不可以被屏蔽。WP Control Mask 寄存器的作用和前面的两种 MASK 寄存器的作用类似，这里不多说了。下面具体看看 WP Control Value 寄存器，其格式如下：

8	7	6	5	4	3	2	1	0
ENABLE	RANGE	CHAIN	EXTERN	nTRANS	nOPC	MAS[1]	MAS[0]	nRW

ENABLE: 如果该位置 0 的话，意味着断点触发条件永远不成立，也就是把全部断点都给 disable 掉了；
RANGE: 暂时不会用，呵呵；
CHAIN: 暂时不会用，呵呵；
EXTERN: 外部到 EmbeddedICE-RT 的输入，通过该输入，可以使得断点的触发依赖于一定的外部条件；
nTRANS: 用来判断是在用户态下还是非用户态下，用户态下：nTRANS = 0，否则 nTRANS = 1;
nOPC: 检测当前的周期是取指令还是进行数据访问；
MAS[1:0]: 和 ARM7TDMI 的 MAS[1:0]信号进行比较，以探测当前总线的宽度是 8 位、16 位还是 32 位；
nRW: nRW = 0, 当前的是读周期，nRW = 1，当前的是写周期。
在 WP Control Value 寄存器中用的比较多的是 nPOC，用来区分当前的是一个取指令周期，还是普通的数据访问周期。断点和观察点是不一样的。断点用来标识某个地址上的一条指令，如果要将一个 WATCH POINT 寄存器组用作断点设置，首先需要置 WP Control Value 寄存器的 nPOC 位为 0，用来表示：只有在当前的周期是进行取指令的前提下，才触发断点。观察点用来观察某个地址上的数据变化的，如果要将一个 WATCH POINT 寄存器组用作观察点使用，首先需要置 WP Control Value 寄存器的 nPOC 位为 1，用来表示：只有在当前的周期是进行普通的数据访问的前提下，才触发观察点。

WP Address Value/Mask Register 和硬件断点

断点是用来标识某个地址上的指令的，所以要将一个 WATCH POINT 用作断点设置，首先需要将 WP Control Value 寄存器的 nPOC 位置 0，用来表示：只有在当前的周期是进行取指令的条件下，才触发断点。要在一个地址设置一个断点，可以通过 WP Address Value/Mask 这两个寄存器来实现。假设，要在地址 0x0040 设置一个断点，可以将 WP Address Value 寄存器的值设置为 0x0040，同时将 WP Address Mask 寄存器的值设置为 0x0。另外，将 WP Data Mask 设置为 0xFFFFFFFF，这样可以屏蔽掉 WP Data value 寄存器的影响，在进行比较的时候，只考虑地址是否匹配。这样，每次 ARM7TDMI 从地址 0x0040 取指令的话，不管该指令是什么，断点就会被触发，ARM7TDMI 会暂停当前的运行，自动进入调试状态，要清除该断点的话，只要改变 WP Address Value 寄存器的值，或者设置 WP Control Value 寄存器，禁止该功能。如果你想在所有地址的低 16 位的值为 0x0040 的地方设置断点，可以配合使用 WP Address Mask 寄存器，

将它的值设置为 0xFFFF,0000，这样的话，每次比较的时候，高 16 为的地址就会被屏蔽掉。其实这种使用方式应该是比较少的，比较多的情况是，屏蔽掉最低 2 位或者最低 1 位的地址。在 ARM 状态下，因为 ARM 指令的长度是 32 位的，所有指令地址的最低 2 位必须为 0，所以，如果在 ARM 状态下，一般将 WP Address Mask 的低 2 位置 1，在进行地址比较的时候，屏蔽掉地址低最低 2 位，确保断点是被设置在正确的地址上；如果类似，如果是在 THUMB 状态下，一般需要将 WP Address Mask 的最低 1 位置 1。上面描述的是第一种设置断点的方式，这种方式是通过地址比较来实现断点的。这也是我们常说的硬件断点，这类断点可以被设置在任何地址（包括 FLASH 和 SDRAM）。因为 ARM7TDMI 提供了两组 WP Address Value/Mask 寄存器，所以，可以支持两个硬件断点。下面让我们来看看断点的另外一种实现方式。

WP Data Value/Mask Register 和软件断点

在调试过程当中，通过 WP Data Value/Mask 寄存器也可以实现断点设置。利用 WP Data Value/Mask 如何实现断点设置呢？ 和硬件断点的设置一样，首先需要将 WP Control Value 寄存器的 nPOC 位置 0，用来表示：只有在当前的周期是进行取指令的条件下，才触发断点。 然后，将 WP Address Mask 寄存器的值设置为：0xFFFFFFFF, 这样可以屏蔽掉 WP Address value 寄存器的影响，在进行比较的时候，只考虑数据是否匹配。接下来，将 WP Data Value 寄存器的值设置为一个固定的值，例如：0xDEDEDEDE，将 WP Data Mask 寄存器的值设置为 0x00000000。在需要设置断点的地方，将其内容替换为 0xDEDEDEDE。这样，一但程序运行到该位置，尝试从该位置取指令或者数据的时候，因为取得的数据值和 WP Data Value 寄存器的值相同，ARM7TDMI 会暂停当前的运行，自动进入调试状态。要清除该位置上的断点，我们只需要将该位置原来的指令恢复就可以了。这样，退出调试状态后，程序可以继续正常的运行。通过 WP Data Value/Mask 寄存器，我们可以在任何需要设置断点的地方，将其内容替换为一个固定的序列，就可以达到设置断点的目的。这种断点就是我们通常所说的软件断点。软件断点的设置方式使得 ARM7TDMI 可以支持任意数量的软件断点。但也决定了软件断点的局限性：软件断点不能设置在 ROM/FLASH 里面。因为软件断点的实现需要替换要设置断点的位置的内容，这点在 ROM/FLASH 做不到。（虽然从某种意义上来说，FLASH 也是可读写的，但是需要特殊的指令）。

WP Address Value/Mask Register 和观察点

WP Data Value/Mask Register 也可以用来设置观察点，用以观察某个地址的数据变化。每当系统访问（读写）完在被观察的地址的数据的时候，ARM7TDMI 就会进入调试状态，这样，我们就可以马上检查该地址上的数据。要设置一个观察点，将 WP Address Value 寄存器的值设置为需要观察的数据的地址，将 WP Address Mask 寄存器的值设置为 0x00000000，将 WP Data Mask 寄存器的值设置为 0xFFFFFFFF。同时不要忘记设置 WP Control Value 寄存器：nPOC = 1（数据访问，而非取指令）；还可以设置 nRW，表示是对读操作进行观察，还是对写操作进行观察；另外，也可以通过 MAS[1:0]设置合适的数据宽度（8 位、16 位、32 位）。如果希望 WP Control Value 寄存器的某些位不起作用，可以恰当的设置 WP Control Mask 寄存器。例如，如果希望对读操作和写操作都进行观察，那么就可以设置 WP Control Mask 寄存器的第 0 位为 1。

在这一小节里，我们介绍了如何通过扫描链 2 访问 EmbeddedICE-RT 的内部寄存器以及 EmbeddedICE-RT 的内部寄存器的作用。同时我们还介绍了通过 EmbeddedICE-RT 内部寄存器

使 ARM7TDMI 进入调试状态的 3 种方式：控制 DBGRQ 信号、断点、观察点。要使得 ARM7TDMI 推出调试状态，可以使用 RESTART JTAG 指令。

4-3 通过扫描链 1 访问 ARM7TDMI 的通用寄存器和系统存储空间

在这个小节里，我们将讨论如何通过边界扫描链 1 来访问处于调试状态的 ARM7TDMI. 这个小节的所有讨论都基于这样的一个默认条件：ARM7TDMI 已经进入到调试状态（DBGRQ、断点、观察点）。除非明确指出，否则都默认为 ARM7TDMI 处于调试状态。

边界扫描链 1 分布在 ARM7TDMI 的 32 数据总线周围，另外包括 BREAKPT 信号，总长度为 33 位。其扫描寄存器单元的分布如下图所示：

Number	Signal	Type
1	D[0]	Input/output
2	D[1]	Input/output
3	D[2]	Input/output
4	D[3]	Input/output
5	D[4]	Input/output
6	D[5]	Input/output
	. . .	
29	D[28]	Input/output
30	D[29]	Input/output
31	D[30]	Input/output
32	D[31]	Input/output
33	BREAKPT	Input

通过边界扫描链 1 的前 32 位，我们可以插入 THUMB/ARM 指令（不是 JTAG 指令）到 ARM7TDMI 的指令流水线当中去，让 ARM7TDMI 在 DCLK 的驱动下，执行这些指令。通过这种方式，我们可以检查和修改 ARM7TDMI 的通用寄存器和系统内存。为什么通过扫描链可以将指令插入到 ARM7TDMI 的流水线当中去？ 可以访问 ARM7TDMI 的内部同意寄存器？这从边界扫描链 1 的结构可以得到答案。边界扫描链分布在 ARM7TDMI32 位数据总线的周围，每次 ARM7TDMI 取指令或者进行数据存储，都要通过 32 位数据总线进行。这样，通过边界扫描链 1，就可以插入新指令或者新数据，同时也可以捕获出现在数据总线上的数据。在调试状态下，ARM7TDMI 支持如下的这些指令：数据处理指令、load, store, load multiple, and store multiple 指令、MSR，MRS 指令。扫描链 1 中的第 33 位，BREAKPT 输入，在调试过程当中有很重要的作用，主要体现在：

- 在 ARM7TDMI 刚进入调试状态的时候，通过读取该位的值，可以判断进入调试状态的原因：如果 BREAKPT = 0, 由断点触发；如果 BREAKPT = 1, 由观察点触发。
- 在调试的时候，ARM7TDMI 允许特定的指令以系统速度执行，由 MCLK 驱动。这是通过边界扫描链 1 的 BREAKPT 位来完成的。如果 BREAKPT 置 0，意味着下一条指令以调试速度执行（DCLK 驱动）；如果 BREAKPT 置 1，意味着下一条指令以系统速度执行（MCLK 驱动）。当执行到被标识为以系统速度执行的指令时，ARM7TDMI 会同步于 MCLK，由 MCLK 驱动该指令的执行，在执行完毕后，ARM7TDMI 会重新回到调试状态，后面的执行重新由 DCLK 驱动。关于扫描链 1 的 BREAKPT 的具体使

用方式，在后面我会结合具体例子进行说明。

ARM7TDMI 采用的是 3 级流水线，指令的执行分为 3 个阶段（如图 6 所示）：

- **Fetch** 取指令，从内存中取指令；
- **Decode** 译指令，解析指令中需要用到的寄存器；
- **Execute** 执行指令。

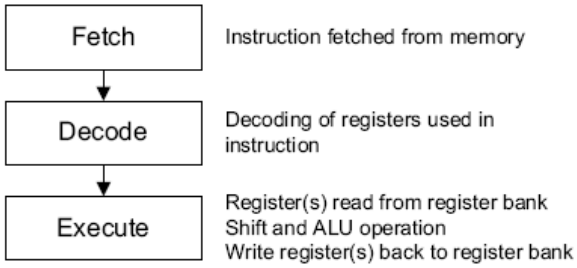


图 6. ARM7TDMI 指令流水线

在插入 ARM 指令前，先通过 JTAG 指令 SCAN_N 选择边界扫描链 1，并用 JTAG 指令 INTEST 将边界扫描链 1 置于 INTEST 状态。接下来，就可以通过 JTAG 接口将需要的指令或者数据放到 ARM7TDMI 32 位数据总线 (边界扫描链 1) 上去。插入指令/数据的过程如下：TAP Controller 的状态变化如下：RUN-TEST/IDLE → SELECT-DR-SCAN → CAPTURE-DR → SHIFT-DR → EXIT1-DR → UPDATE-DR → RUN-TEST/IDLE。在 CAPTURE-DR 状态下，可以通过边界扫描链 1 捕获 ARM7TDMI 32 位数据总线上的数据。在 SHIFT-DR 状态下，通过 33 个 TCK 时钟驱动，就可以将 32 位长的新指令或者需要的数据加上 BREAKPT 输入到扫描链 1 当中去。同时，在 CAPTURE-DR 状态下捕获的数据总线上的数据也可以通过 TDO 输出。到 UPDATE-DR 状态下，输入的指令/数据就会被放到 32 位的数据总线上去。回到 RUN-TEST/IDLE 状态后，插入的指令/数据会在 DCLK 的驱动下执行。另外提醒一点，因为扫描链 1 的第一位对应于 D[0]，而在将指令插入前将整个指令＋BREAKPT 共 33 位序列的比特顺序颠倒过来。这样，通过 TDI 先送入的是 BREAKPT，然后是指令的第 31 位、30 位、29 位 … 第 0 位。

下面，通过几个具体的例子，来看看如何通过插入 ARM 指令/数据来访问 ARM7TDMI 的内部通用寄存器和系统内存。

(1). 读取通用寄存器 R0 的值：

要读取寄存器 R0 的值，可以用指令 STR R0, [R0] = 0xE5800000 来实现。该指令将寄存器 R0 的值存储到内存单元 R0 中去。因为在 ARM7TDMI 处于调试状态的时候，ARM7TDMI 和外部是隔离开来的，所以该指令实际上不能访问内存单元，也不会对内存单元产生任何影响。使用指令 STR R0, [R0] 的目的是使得寄存器 R0 的值出现在数据总线上，这样我们就可以通过扫描链 1 将其捕获，然后从 TDO 输出。指令 STR R0, [R0]的执行需要 2 个指令执行周期（CYCLE）。在第 1 个指令执行周期，执行地址计算；在第 2 个指令执行周期，将寄存器 R0 的值放到数据总线上去。要读取寄存器 R0 的值，对边界扫描链 1 的操作过程如下（在下面描述的每一个步骤，TAP 状态都是从 RUN-TEST/IDLE 状态出发，最后回到 RUN-TEST/IDLE 状态）：

- 插入指令 STR R0, [R0] **&** BREAKPT = 0：
  — 这步相当于指令 STR R0, [R0]的取指令周期（3 级流水线）；
- 插入空指令 MOV R0, R0 **&** BREAKPT = 0：

— 这一步读取新指令 MOV R0, R0，同时，相当于指令 STR R0, [R0]的译指令周期（3 级流水线）；

- 插入空指令 MOV R0, R0 **&** BREAKPT = 0：

  — 这一步读取新指令 MOV R0, R0，同时，插入的 STR R0, [R0]指令开始执行（3 级流水线）。在这一步，STR R0, [R0]指令处在第 1 个指令执行周期，在该周期，先执行地址计算；

- 通过扫描链 1 读出捕获的数据总线上的数据：

  — 在这一步，指令 STR R0, [R0]继续执行，指令 STR R0, [R0]处在第 2 个指令执行周期。在这一步，寄存器 R0 的值会被放到数据总线上去，指令 STR R0, [R0]执行完毕。所以，在这一步，从扫描链中读出的数据就是寄存器 R0 的值。另外，在这一步，因为还处在 STR R0, [R0]的指令执行周期内，所以访问扫描链 1 的时候，通过 TDI 输入的空指令实际上是会被 ARM7TDMI 忽略。

至此，我们成功获得寄存器 R0 的值。因为除了 STR R0, [R0] 指令外，插入的另外 2 条指令都是空指令，所以后面我们可以不考虑这 2 条指令的执行结果。

(2). 修改通用寄存器 R0 的值：

要修改寄存器 R0 的值，可以用指令 LDR R0, [R0] = 0xE5900000 来实现。该指令将内存单元 R0 的值存储到寄存器 R0 中去。因为在 ARM7TDMI 处于调试状态的时候，ARM7TDMI 和外部是隔离开来的，所以该指令实际上不能访问内存单元，也不会对内存单元产生任何影响。使用指令 LDR R0, [R0] 的目的是：该指令要从内存单元 R0 中取数据放到数据总线上去，这样我们就可以通过扫描链 1 将任意需要的值放到数据总线上去，达到修改寄存器 R0 的目的。指令 LDR R0, [R0]的执行需要 3 个指令执行周期（CYCLE）。在第 1 个指令执行周期，执行地址计算；在第 2 个指令执行周期，从内存单元 R0 中读取数据，并将数据放到数据总线上去；在第 3 个指令执行周期，将数据总线上的数据写到寄存器 R0 中去。要修改寄存器 R0 的值，对边界扫描链 1 的操作过程如下（在下面描述的每一个步骤，TAP 状态都是从 RUN-TEST/IDLE 状态出发，最后回到 RUN-TEST/IDLE 状态）：

- 插入指令 LDR R0, [R0]**&** BREAKPT = 0：

  — 这步相当于指令 LDR R0, [R0]的取指令周期（3 级流水线）；

- 插入空指令 MOV R0, R0 **&** BREAKPT = 0：

  — 这一步读取新指令 MOV R0, R0，同时，相当于指令 LDR R0, [R0]的译指令周期（3 级流水线）；

- 插入空指令 MOV R0, R0 **&** BREAKPT = 0：

  — 这一步读取新指令 MOV R0, R0，同时，插入的 LDR R0, [R0]指令开始执行（3 级流水线）。在这一步，LDR R0, [R0]指令处在第 1 个指令执行周期，在该周期，先执行地址计算；

- 通过扫描链 1 将寄存器 R0 的新值放到数据总线上去：

  — 在这一步，指令 LDR R0, [R0]继续执行，LDR R0, [R0]指令处在第 2 个指令执行周期。在这一步，通过扫描链 1，将 R0 的新值放到数据总线上去。对 ARM7TDMI 而言，相当于从系统内存中取得了所需的数据。

- 插入空指令 MOV R0, R0 **&** BREAKPT = 0：

  — 在这一步，指令 LDR R0, [R0]继续执行，指令 LDR R0, [R0]处在第 3 个指令执行周期。在这一步，数据总线上的数据写到寄存器 R0 中去，完成对寄存器 R0 的修改，指令 LDR R0, [R0]执行完毕。另外，在这一步，因为还处在 LDR R0, [R0]的指令执行周期内，所以访问扫描链 1 的时候，通过 TDI 输入的空指令实际上是会被 ARM7TDMI

忽略。

至此，我们成功修改了寄存器 R0 的值。因为除了 STR R0, [R0] 指令外，插入的另外 2 条指令都是空指令，所以后面我们可以不考虑这 2 条指令的执行结果。

(3). 读内存：

在前面说过，当 ARM7TDMI 处于调试状态的时候，不能访问系统的存储空间，这是因为在调试状态下，指令的执行是由 DCLK 驱动的。而对存储空间（内存）的访问需要 MCLK 的驱动。那在调试状态下，如何实现对存储空间的访问呢？这个时候，边界扫描链 1 的 BREAKPT 位就派上用场了。在通过扫描链 1 插入一条指令的时候，如果将 BREAKPT 位置 1，意味这条指令的下面一条指令将在 MCLK 的驱动下执行，执行完毕后，自动返回调试状态。这样，通过扫描链 1 的 BREAKPT 位，我们就可以实现对系统存储空间的访问。让我们来看看要读取内存地址 ADDR 上的 32 位数据的大概步骤。首先，我们将要访问的地址 ADDR 写到寄存器 R0 当中去；然后用指令 LDR R1, [R0]将地址 ADDR 处的 32 位长的数据拷贝到寄存器 R1 当中去；最后，将寄存器 R1 的值读出来，就得到了存储空间地址 ADDR 处的值。修改寄存器 R0 的值和读取寄存器 R1 的值的步骤在前面都详细介绍了，在下面，我只介绍结合 BREAKPT 如何用指令 LDR R1, [R0] 来真正的访问系统存储空间，将地址 R0 处的内容拷贝到寄存器 R1 当中去。具体过程如下（在下面描述的每一个步骤，TAP 状态都是从 RUN-TEST/IDLE 状态出发，最后回到 RUN-TEST/IDLE 状态）：

- 插入空指令 MOV R0, R0 **&** BREAKPT = 1：
  — 首先插入一条空指令，该指令的主要目的是将扫描链 1 的 BREAKPT 信号置 1。这样，这条空指令的下一条指令将在 MCLK 的驱动下，回到正常状态下运行。
- 插入指令 LDR R1, [R0]**&** BREAKPT = 0：
  — 因为上一条指令将扫描链 1 的 BREAKPT 置 1 了，所以，这条指令将在 MCLK 的驱动下，回到正常的系统状态下去执行；
- 将 JTAG RESTART 指令插入到 ARM7TDMI 的 JTAG 指令寄存器当中去：
  — 将 RESTART 插入到 JTAG 指令寄存器当中去有几个作用。一是，使 ARM7TDMI 重新同步于 MCLK 信号；二是，使刚才插入的指令 LDR R1, [R0]在系统正常状态下，在 MCLK 的驱动下执行；三是，在系系统正常状态下执行完指令 LDR R1, [R0]后，使 ARM7TDMI 自动返回到调试状态下；

在将 RESTART 写入到 JTAG 指令寄存器，并且在 TAP Controller 回到 RUN-TEST/IDLE 状态后，ARM7TDMI 将会暂时返回到正常的运行状态。在正常的运行状态下，ARM7TDMI 将以系统速度（MCLK 驱动）完成指令 LDR R1, [R0]的执行。执行完毕后，ARM7TDMI 将自动返回到调试状态。这样，寄存器 R1 中就已经有了地址 ADDR 处数值的一个拷贝了。最后需要做的就是通过前面介绍的方法，在调试状态下，将 R1 的值读出来。在需要访问内存的时候，需要注意的就是扫描链 1 中 BREAKPT 位的设置，以及 JTAG RESTART 指令的使用。

(4). 写内存：

修改内存的过程和读取内存的过程类似，大概步骤如下：将内存地址 ADDR 写到寄存器 R0 当中去，将新的值写到寄存器 R1 当中去，最后利用指令 STR R1 [R0]完成实际的内存访问。在将指令 STR R1 [R0] 插入插入到扫描链 0 之前前，将扫描链 1 的 BREAKPT 位置 1，以使得关键指令 STR R1 [R0] 能在 MCLK 的驱动下，回到正常的系统状态下去执行，以便能正常访问系统存储。具体的过程在此就不在罗嗦了。

在这个小节里，我们主要介绍了通过边界扫描链 1 如何访问 ARM7TDMI 的通用寄存器和系统存储空间。在我们的讨论当中，为了简单起见，我们使用了最简单的 LDR 和 STR 指令。在实际的调试过程中，可以使用 LDM 和 STM 指令，这样一次可以读写多个寄存器或者是多个地址上的数据，提高效率。从上面的讨论可以看出，在调试状态下，每次对通用寄存器和系统存储系统进行访问都需用多条指令来实现。每次插入一条指令到扫描链 1 中去，都需要 33 个 TCK 时钟周期。因为并口的速率限制，通过并口进行调试时，速度会比较的慢。如果使用 USB 接口或者网络接口，调试的速度将能够获得很大的提升。

5 总结

在实际的调试过程中，有很多的细节问题需要考虑，例如：从 ARM 状态还是 THUMB 状态进入调试状态？系统工作在 LITTLE ENDIAN 模式还是 BIG ENDIAN 模式？ 在此就一一略过了，感兴趣的朋友可以参考相关的资料。只是希望大家看了这篇文章后，对了解 ARM JTAG 调试的原理和过程有一定的帮助。另外，如果文章中有什么地方理解有不对的地方，希望大伙能发 EMAIL 给我指出来，相信通过你的 EMAIL，我也能进一步的加深对 ARM JTAG 调试的理解。同时，我也会尽力的纠正。

参考资料：

[1] IEEE Standard 1149.1 - Test Access Port and Boundary-Scan Architecture
[2] Application Note 28: The ARM7TDMI Debug Architecture
[3] ARM7TDMI (Rev 4) Technical Reference Manual

TWENTYONE
Oct-2004