David Baca

Efficient Control of JTAG TAP

JTAG TAP stepping made easy

Introduction

Many of Freescale's microprocessors feature a JTAG port–a port compatible with the IEEE® 1149.1 standard for performing boundary scans and debugging embedded applications. The development of the IEEE 1149.1 standard was started by the Joint Test Action Group in mid 80s, and the activity was later moved under the IEEE organization. As a result, a test access port compliant with the IEEE 1149.1 standard is usually called a JTAG port after the group that initiated the standardization process.

Today, the JTAG port is used for chip boundary scans and programming and debugging purposes. Any JTAG implementation compatible with the IEEE 1149.1 standard is not limited to the functions specified by the standard—it can freely extend functionality. For example, the IEEE-ISTO 5001 Forum™ standard defining a debug interface for embedded processors based its interface on JTAG.

The IEEE 1149.1 standard defines a general-purpose test access port (TAP) as a physical layer for accessing JTAG functions. Further, the standard specifies the TAP controller that controls the signals used for accessing JTAG functions and other possible incorporated circuitries such as a debugging module. The TAP controller is a synchronous finite state machine that responds to the changes at the TAP pins. To access a particular JTAG function, you have to step the TAP controller through its states. For learning the purpose of each TAP controller state and other JTAG specification details, refer to the IEEE 1149.1 standard.

This article presents a simple and efficient algorithm for stepping the TAP controller along the shortest path from the current TAP state to any other of its states. Programmers usually step through the TAP controller by writing sequences of instructions manipulating the TAP input pins. As a result, writing the stepping code for a comprehensive software module mediating JTAG's and associated functionalities can easily become a difficult, tedious and error-prone task. Our algorithm eliminates the sequences of TAP controlling instructions by a routine that navigates the TAP controller to the desired state. As a result, you just call the routine, for example `GoTo(dstate)` and the TAP controller is stepped to the desired state `dstate`.

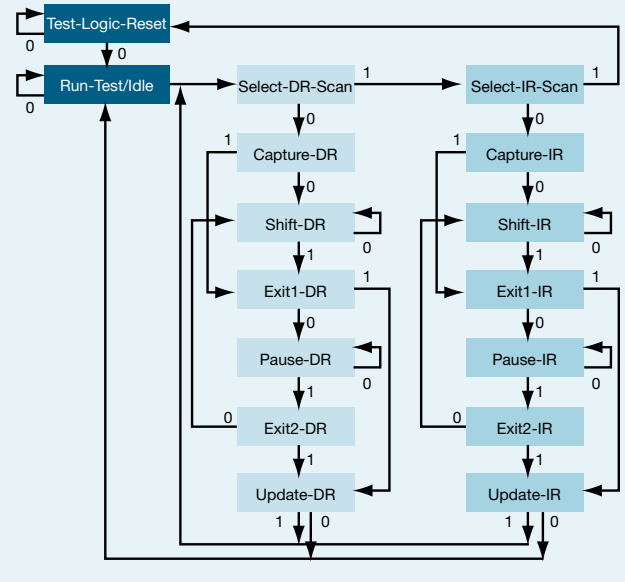
The Stepping Algorithm

In the TAP controller state machine, depicted in Figure 1, a transition is followed at the time of the clock rising edge (pin TCK) if it is enabled by the logical signal level (0 or 1) at the TAP TMS pin and if the TAP is in the transition's initial state. The arrow labels 0 and 1 in Figure 1 represent the enabling logical signal levels.

The diagram in Figure 1 can be viewed as graph consisting of vertices and oriented edges where a vertex is a TAP controller state and an oriented edge is a transition. Finding the shortest path between any two vertices is a classical, solved problem. In the TAP state machine, the shortest paths are quite obvious and can be found manually. Thus, we need an efficient mechanism for storing the shortest paths, so they can be followed without being discovered again.

For storing the shortest paths, we utilize the following corollary. Suppose there is the shortest path from a state s_1 to a state s_n and the path goes through a state s_k. Thus, the path is $s_1,\ldots,s_k,\ldots s_n$. Then, the path $s_k,\ldots,s_n$ is the shortest path between the states s_k and s_n. If this path wasn't the shortest,

Figure 1: TAP Controller State Diagram



then we could shorten the path between s_1 and s_n and the original path wouldn't be the shortest, which is a contradiction.

As a result, it is sufficient for any of the shortest paths starting at s_1 and ending at s_n to store the very next state following the state s_1 on the path. The repository that stores the very next state for each of the shortest paths is called a routing table, similarly defined in networks to route messages. Unlike networks where every node stores a vector of the very next nodes, our routing table is a matrix because the traversing is controlled via the TAP interface. The TAP routing table is depicted in Table 1.

The algorithm that steps the TAP controller via the shortest path to the desired state is very simple. Suppose that the current TAP state is maintained in the variable `tap_state` and the two dimensional array `RoutingTable` holds the content of Table 1. The function `GoTo(dstate)` works as follows.

Table 1: TAP Routing Table

States	Test-logic-reset	Run-test/idle	Select-DR-scan	Capture-DR	Shift-DR	Exit1-DR	Pause-DR	Exit2-DR	Update-DR	Select-IR-scan	Capture-IR	Shift-IR	Exit1-IR	Pause-IR	Exit2-IR	Update-IR
Test-logic-reset	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Run-test/idle	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Select-DR-scan	1	1	x	0	0	0	0	0	0	1	1	1	1	1	1	1
Capture-DR	1	1	1	x	0	1	1	1	1	1	1	1	1	1	1	1
Shift-DR	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
Exit1-DR	1	1	1	1	0	x	0	0	1	1	1	1	1	1	1	1
Pause-DR	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
Exit2-DR	1	1	1	1	0	0	0	x	1	1	1	1	1	1	1	1
Update-DR	1	0	1	1	1	1	1	1	x	1	1	1	1	1	1	1
Select-IR-scan	1	1	1	1	1	1	1	1	1	x	0	0	0	0	0	0
Capture-IR	1	1	1	1	1	1	1	1	1	1	x	0	1	1	1	1
Shift-IR	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
Exit1-IR	1	1	1	1	1	1	1	1	1	1	1	0	x	0	0	1
Pause-IR	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
Exit2-IR	1	1	1	1	1	1	1	1	1	1	1	0	0	0	x	1
Update-IR	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	x

```
GoTo(dstate)
{
    while(tapstate != dstate)
        step(RoutingTable[tapstate][dstate]);
    return;
}
step(tms)
{
        set TMS pin to tms (typically on a parallel port);
    generate clock pulse on the TCK pin;
    tap_state = next_state(tap_state,tms); //
update the current state
}
```

The function `next_state(tap_state,tms)` uses another table that provides the very next successor for each state given the TMS pin value.

Implementation

We implemented the described algorithm in C++. A singleton class called `JTAG` controls the signals for the TAP controller, maintains the current state, and steps the TAP controller to the desired state. When the class is being instantiated, the routing table along with other tables is loaded in memory and the TAP controller is reset to its initial state test-logic-reset. The `JTAG` singelton interface provides a method called `AssertTAPState(tapstate)` to make sure the current TAP controller state is `tapstate`. If the current state is different, the method steps the TAP controller to the right state.

The `JTAG` singleton steps the TAP controller by manipulating the pins of the parallel port that is connected to the JTAG physical interface on the embedded processor. The methods that control the JTAG pins via the parallel port are implemented in another singleton called `JTAGPort`, which is accessed solely from `JTAG`.

Conclusion

We showed that writing the tedious sequence of instructions stepping the TAP controller can be eliminated by keeping the shortest paths between any two TAP controller states in a small routing table. The associated overhead caused by performing methods or function calls and by looking up the TMS pin value is negligible considering the current computer's speed and the maximal frequency achievable at the parallel port (around 1 MHz). Furthermore, the JTAG port communication speed is limited by the processor clock frequency.

David Baca is an application engineer at Freescale. He has been with the company for almost two years. He holds masters degrees in electrical engineering and business administration and is a PhD candidate in artificial intelligence. Before joining Freescale, Baca worked on R&D projects sponsored by agencies such as NASA and the Air Force.