Blackbox JTAG Reverse Engineering

Felix Domke tmbinc@elitedvb.net

November, 27th 2009

Abstract

JTAG's IEEE 1149.1 standard is a well-known method to access on-chip scan chains for test-mode functionality. But a large part of the accessible test-modes are usually not documented. It will be shown that often these test-modes can be reverse-engineered by looking at the JTAG inputs and outputs. Undocumented test-modes can be pretty powerful tools in gaining "back door" access to a system.

1 Introduction

With the introduction of multi-layer circuit boards and much finer pitches, having a decent test coverage with a bed-of-nails type device got harder and harder. In the 1980's *Joint Test Action Group*, a group composed of several electronic manufacturers, developed a solution called *IEEE 1149.1: Standard Test Access Port and Boundary-Scan Architecture*, often simplified just as *JTAG*. For the sake of readability, we will use this acronym to refer to the actual implementation.

JTAG is a standard that defines access to test-mode functionality on an integrated circuit. While certain JTAG functionality, like *Boundary Scan*, are widely documented[1], JTAG often allows accessing functionality much deeper inside the chip. Often, these test-modes are undocumented to the general public, even when signing NDAs. Nevertheless, they are available and can be used once their functionality is known.

This paper will first give an overview over the basics to understand what JTAG actually is. After that, methods to explore undocumented functionality will be shown. Finally, an example test-mode, which has been reverse-engineered, will be shown and explained. In this example, a test-mode was found that allows reading and writing to memory in a 32 bit address space of the device.

2 Basics

An IC implementing JTAG implements access to several on-die *scan chains*, **special register used for testing**, in a standardized way. The content of these registers, however, are (other than for a few exceptions) not part of the standard. A scan chain is a shift register with parallel load and store functionality. Functionally, it can be described as a register that can be read and written to, pretty much like a memory-mapped register accessible from a CPU, although a scan chain has some additional properties which can be helpful for reverse-engineering. Such a register can have any number of bits, usually from 1 (like *BYPASS*, see below) to several hundred (like a *Boundary Scan* register).

JTAG defines a standardized way to access the test registers using **4 special IO pins** (*TMS*, *TCK*, *TDI*, *TDO*, plus an optional *TRST*)

By controlling *TMS* (while clocking *TCK*), the test system can **navigate in the *JTAG State Machine***. The *TMS* pin is used to determine which path the state machine will move, whenever there is a raising edge on the *TCK* pin. The optional *TRST* pin will bring the state machine back into the initial *Test Logic Reset* state (*TLR*), but the same effect can be achieved by scanning in (at most) 5 ones.

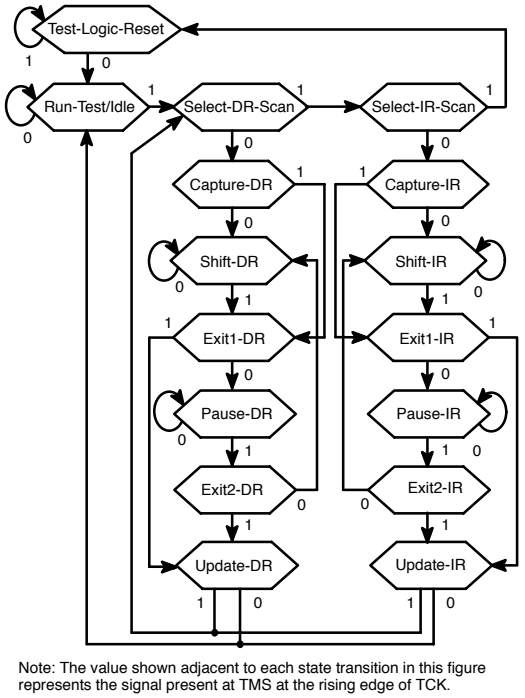

Figure 1: TAP Controller State Diagram[3, Page 3-5]

Since *TRST* is level-low-active, TRST must not be tied low to make the JTAG interface operational.

The state machine allows access to **two special registers**, the *IR* (*Instruction Register*), and a symbolic register called *DR* (*Data Register*), which is a placeholder for to the register that is currently selected with the current content of the IR. Thus, IR is an **index into a number of registers**, and DR is the currently selected register. The contents of IR are also called *instruction*, even though selecting an instruction will usually not directly trigger any action, but merely select the proper register as DR. The size of the IR is implementation specific, and usually varies between 4 and 32 bit. Since the selected register is directly accessed when scanning the DR, the size of the DR depends on the current instruction.

When a scan chain is selected and the JTAG state machine is in the *shift* state, the **register is connected between *TDI* and *TDO* pins**. The rightmost bit is connected to the *TDO* pin, and on each *TCK* clock cycle, the register contents will be shifted one bit to the right. The *TDI* pin will be used as input for the leftmost bit. Shifting a register is an inherent bidirectional operation. For registers with a positional notation, this means that scanning - both input and output - will be done from LSB to MSB.

Since shifting a register will modify the content (due to the shifting), a scan chain is usually not directly connected to other logic, but latched first. The *Capture* state is used to capture the current state of the logic into the scan chain, and the *Update* state is used to update the logic with the scan chain contents. The JTAG state machine ensures that first the logic register contents are latched into the scan chain (*Capture*), then the scan chain can be read out and modified (*Scan*), and finally, the updated content will be written back (*Update*). Thus, both capture and update are **atomic operations**.

The meanings of the referenced registers are partially standardized; they are called *public* registers, opposed to the **implementation-specific** *private* registers.

The **most basic public register** is called *BYPASS*. It is a single bit and doesn't have any functionality. It is used to daisy-chain multiple ICs, each sharing the same *TCK* and *TMS* pins. The JTAG state machines will be synchronized, and the *TDI* and *TDO* lines are connected in series, so whenever a register is scanned, all ICs will appear as one big scan chain. Since when scanning the instruction, all the IRs are also chained, its possible to select a different instruction on each IC. In order to just talk to a specific ICs JTAG chain, all other ICs will be placed in the *BYPASS* instruction. This means that other than delaying the scan chain for one cycle, nothing will happen (even on capture and update). The *BYPASS* register is always accessible with setting the instruction to all-ones.

Another register, which is technically optional, but most of the time implemented is **the *IDCODE* register**. It is 32bit in width, and identifies the manufacturer and device using a standardized code. The exact instruction to select this register is not standardized, but when-

ever the state machine is in the Test-Logic-Reset state, the instruction will be loaded with the IDCODE instruction. Thus, it's possible to read out the IDCODE value from a device by going straight into the *DR scan* phase, without ever modifying the instruction. By scanning 32 bit at a time, it's possible to identify each individual chip in a daisy-chained operation. By determining the length of the combined IDCODE registers, it's also possible to detect the number of devices in a chain.

3 Exploration

If **absolutely no documentation** is available for a given target IC, the first thing to do is to figure out the actual JTAG pins. Tools like JTAG Finder[2] can be used for this.

To test whether the JTAG connection **is working ok**, the IDCODE should be read. This is easy, since it doesn't involve scanning the IR, and the IDCODE register has a fixed size of 32 bit. If the IDCODE isn't all-zero or all-one and stable over multiple runs, this is already a pretty good sign that the electrical JTAG connection is working fine.

The next thing to determine is the number of theoretically accessible registers, i.e. the **length of the IR**. After that, each of the indexable registers should be individually explored.

In order to determine the **length of any register** (both of IR or individually selected DR), the following algorithm can be used:

1. Go into the scan state for that register.
2. Scan in a large number of ones (i.e. set *TDI* to high, toggle *TCK*). After some delay, depending on the previous state of the register, these ones should be shifted out on *TDO*. Make sure that the whole register is flooded with ones. Shifting in several thousand ones is usually a sufficient number.
3. Scan in a single zero.
4. Clock *TCK* until that zero reaches *TDO*. The number of clock cycles required (plus one) will be the length of the register.

Some devices have IR of up to 32 bit. That of course doesn't mean that they implement 2^{32} different registers - most of these are unused. Unused registers usually behave like BYPASS, i.e. as a function-less 1 bit register, or produce a static value at *TDO*. When exploring an unknown chip, it's usually a good idea to first map the entire IR space, and determining a few characteristics of each register. Most characteristic is the length of the register. Other important properties are if certain bits of a register stays across an update/capture cycle, i.e. if they are readable and writeable. Since accessing a register through the capture/scan/update states is always a rewrite operation, read-only access is not easily possible. Some chips implement different instructions for reading and writing of the same registers. Sometimes, an additional bit in the DR will be used to determine if the register will be updated or not (in the update-phase). Another scheme is to have a scratch register, which will be transferred into the final register once a certain *command* bit (in the same or in another register) will be toggled. These are **implementation specific behaviors** that need to be determined. It is hard to give a universal algorithm to determine these exact properties, so playing around with registers to determine the behavior is usually the easiest way. Also, instructions next to each other are often part of the same functional block. For example, said *command* bits are usually directly next to the actual data register.

There are a few **typical patterns**. One is the already-mentioned BYPASS instruction. It will behave as a read/writeable 1 bit register.

Another typical pattern is a usually **very large** (hundreds of bits) register. The same length of register will usually appear for multiple instructions, with different properties. This is a sign that this is the boundary scan register, used for the SAMPLE/PRELOAD, INTEST and EXTEST instructions. Care must be taken since they can override the output pins of an IC, thus driving pins which might not be configured as an output originally, potentially conflicting with the output of another chip. This will cause excess current, and might even destroy the output driver of any of the two

chips. Fortunately, these instructions are often documented, so it's easy to avoid them.

Sometimes, register have a **zero width**. That means that just selecting the instruction itself or entering the *update* state of the DR triggers an action. This behavior is common for *reset*-style instruction, which will do a partial or complete reset of the chip.

"Interesting" registers, for example those used for **bus read/write operations**, usually exist in a pattern of a few consecutive 32 bit (sometimes a few bits more) registers. The next section documents such a real-world example. Extra bits are sometimes provided for individual byte-lane write enables, or selection of different address spaces.

Depending on the actual implementation, the TLR state might actually **reset part of the test logic**. For example, it might clear an address register back to zero. This can be avoided by not entering the TLR state when accesing registers, but instead staying in the *run test idle* mode. But observing changes in certain registers when going through the TLR state also provides valuable information.

4 Real-World Example

On this example device, the JTAG port usually provides boundary scan functionality. No further test-mode functionality could be found. However, it was determined that when certain pins labeled as *TMODEn* would be pulled into a configuration that was described as *reserved*, the JTAG functionality would change completely. While the original IR length was 32 bit (with most of the IR bits unused), when entering the undocumented test mode, the IR length did change to 5 bit, giving access to a completely different test-mode functionality. An automated length-scan over the 32 possible instructions gave the following result:

```
IR  Length
00: 4096+
01: 4096+
02: 4096+
03: 32
04: 4096+
05: 4096+
06: 1
07: 1
08: 32
09: 32
0a: 4
0b: 4096+
0c: 4096+
[...]
1d: 4096+
1e: 4096+
1f: 1
```

Clearly, instruction *0x1f* (all-ones) is a BYPASS instruction, as required by the JTAG specifications. This corresponds to the measured length of 1 bit. Unimplemented registers are not implemented as BYPASS on this chip, but give a static high level on *TDO*. This means that the search operation to count the cycles until *TDO* becomes 0 has timed out (in this case after a maximum length of 4096), since the register didn't behave like a shift register at all, so the input zero was never propagated to the output. These instructions are unimplemented, and can be ignored.

Instruction 3 was determined to be the IDCODE instruction, by first capturing the IDCODE (by going into the *shift DR* state without modifying the instruction before), and then comparing it with the value that can be scanned out while selecting instruction 3. Also, this register is 32 bit and completely read-only, and has a stable and constant pattern, which is a good sign for being the IDCODE register. Finally, decoding the value according to the JTAG rules did yield a plausible result.

Instructions 8 and 9 are interesting, because they are 32 bit in size. Registers in this length are often used for higher level tests, like accessing a memory bus, rather than low-level test-modes, which individually connect to obscure on-chip functionality. Instruction 8 was manually checked and found to be read/write, while instruction 9 was found to be read-only.

For a busmaster capable test mode, usually one or more registers are used to enter the address, the data. Upon updating a separate *command* register, usually with a bit set to high, the

transaction will be started. The exact semantics are of course implementation specific, but can usually be found with trial and error easily.

In this case, the semantics to do a busmaster read transaction have been found out to be the following:

1. Scan in the address into instruction 8
2. Toggle the rightmost bit in instruction 0xA (i.e. scan in 0x0, then 0x1)
3. Scan out the data from instruction 9

A similar pattern was found to also do writes.

Certainly, the chip in question has a very simple accessible test-mode, and also the length of the IR is pretty small. But the same technique have been successfully applied to chips with a much larger IR and more complicated test-mode functionality.

5 Conclusion

It has been shown that test-modes hidden in ICs can be found and used, even when no documentation is available. Since test-mode functionality is designed to provide broad test coverage, a lot of functionality might be accessible. Once again, security by obscurity does not work, and silicon designers need to be aware that even hidden test-modes will ultimately be discovered and abused by hackers.

References

[1] JTAG - Teach new tricks to your FPGA, KNJN LLC., `http://www.fpga4fun.com/JTAG.html`

[2] JTAG Finder, hunz, `http://www.elinux.org/JTAG_Finder`

[3] IEEE Std. 1149.1 (JTAG) Testability, Primer, Texas Instruments `http://simplemachines.it/doc/jtag-tutorial.pdf`