ES-Debugger : the flexible Embedded System Debugger based on JTAG technology

Ingeol Chun, Chaedeok Lim
Embedded S/W Research Division,
Electronics and Telecommunications Research Institute (ETRI)
161, Kajong-dong, Yusong-Gu, Taejon, 305-350, KOREA
igchun{cdlim}@etri.re.kr

Abstract — **In the past the Embedded Software was used for controlling industrial apparatus but now is used in control systems for military, home appliances and automatic sensor systems and so on at present. Embedded systems are different from conventional computer system and this distinction makes it much difficult to debug as well as develop the embedded systems because of the lack of resources and sensitiveness in target system such as memory, power, timing constraint, etc. Especially huge integrated system like SoC(System-on-a-Chip) - that has memory, I/O port, etc. - has no space to locate debugging gadget so that the debugging of the embedded system is more difficult.**

To solve above problem, Boundary scan testing (also named JTAG) is appeared and we proposed the embedded system debugger based on JTAG technology in this paper.

Keywords* —*Embedded System, Debugging Tool

1. Introduction

Recent marketing drive for reduced product size, such as portable phones and digital cameras, higher functional integration, faster clock rates, shorter product life-cycle with dramatically faster time to market, has created new technology trends. These new technology trends include increased device complexity, fine pitch components such as SMTs, MCMs, and BGAs, increased IC-pin count, and smaller PCB traces. However, these trends make the development of modern hardware systems be one of the difficult areas[1].

To solve the problems, the IEEE standard number 1149 "Standard Test Access Port and Boundary-Scan Architecture" is appeared. This standard defines a 5-pin serial protocol for accessing and controlling the signal-levels on the pins of a digital circuit, and has some extensions for testing the internal circuitry on the chip itself. The standard was written by Joint Test Action Group(JTAG) and the architecture defined by it is known as "JTAG boundary-scan" or as "IEEE 1149"[2][3].

As the acceptance of boundary-scan as the main technology for interconnect testing and in-circuit programming has increased, the various boundary-scan test and in-system programming tools have matured as well. But these tools that usually consist of software and hardware are so expensive that novice or small business developer or student cannot afford the expense[4][5].

In this paper, we propose the inexpensive and flexible Embedded System Debugger(ES-Debugger) supporting to debug software in embedded systems.

2. What is Boundary-Scan?

Boundary-scan, as defined by the IEEE Std. 1149.1 standard, is an integrated method for testing interconnects on printed circuit boards that are implemented at the IC level. The inability to test highly complex and dense printed circuit boards using traditional in-circuit testers and bed of nail fixtures was already evident in the mid eighties. Due to physical space constraints and loss of physical access to fine pitch components and BGA devices, fixturing cost increased dramatically while fixture reliability decreased at the same time.

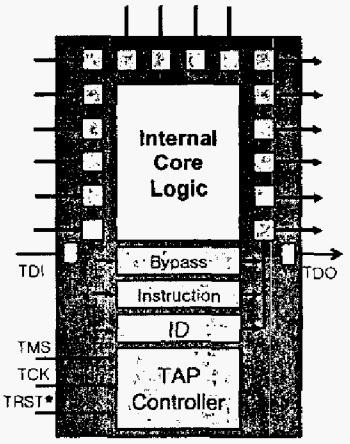

Figure 1. IEEE 1149.1 Device Architecture

The boundary-scan test architecture provides a means to test interconnects between integrated circuits on a board without using physical test probes. It adds a boundary-scan cell that includes a multiplexer and latches, to each pin on the device. Boundary-scan cells in a device can capture data from pin or core logic signals, or force data onto pins. Captured data is serially shifted out and externally compared to the expected results. Forced test data is serially shifted into the

boundary-scan cells. All of this is controlled from a serial data path called the scan path or scan chain. By allowing direct access to nets, boundary-scan eliminates the need for large number of test vectors, which are normally needed to properly initialize sequential logic. Tens or hundreds of vectors may do the job that had previously required thousands of vectors. Potential benefits realized from the use of boundary-scan are shorter test times, higher test coverage, increased diagnostic capability and lower capital equipment cost[6][7].

3. ES-Debugger : the flexible Embedded System Debugger based on JTAG technology

In this chapter, we describe the flexible Embedded System Debugger based on JTAG technology(ES-Debugger). Figure 2 depicts the internal architecture of the proposed system.

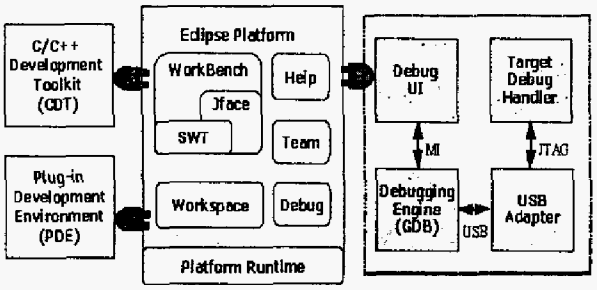

Figure 2. The Architecture of the ES-Debugger

The ES-Debugger is divided into 4 parts (User Interface, Debugging Engine, USB Adapter[9], Target Debug Handler[8]). First, User Interface installed at a host system is developed based on the eclipse platform. The UI is plugged in to Eclipse CDT(C/C++ Development Tools). As the original Eclipse CDT supports local debugging and standard GDB server, some modification is needed. Second, the Debugging Engine also is located at a host system. It is the key part of the ES-Debugger and developed based on GNU debugger(GDB). The major function of the Debugging Engine is to translate debugging commands to JTAG commands that are propagated to USB Adapter via USB port equipped in PCs. Next, USB adapter is needed for the difference of signals and the voltage between the USB port and the JTAG port. It generates JTAG signals that is being sent to a target system when receives control signals via USB interface. Finally, Target Debug Handler that is provided by CPU manufacturer but needs some modification for appropriate operations is located at the cache memory in the CPU of a target system. When it receives the JTAG control signals via JTAG interface, it operates the CPU according to them.

We developed whole system using open source so that it have abundant scalability and flexibility.

3.1 User Interface based on Eclipse Platform

As mentioned above, we developed user interface as eclipse plug-ins for user friendliness and integration with other tools. Because original CDT is only applied to local and standard remote debugging, we needed to modify CDT to communicate with the debugging engine. Especially, the target control method should be included. Figure 3 shows the modified Eclipse CDT, but resembles original CDT.

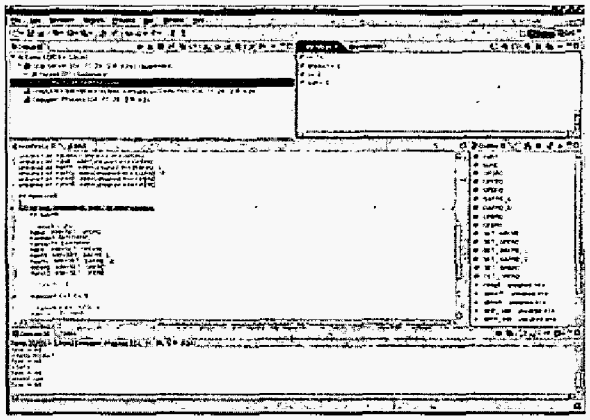

Figure 3. The screenshot of ES-Debugger User Interface

3.2 Debugging Engine

We design the debugging engine using GNU GDB. The debugging engine interacts with User Interface and USB adapter shown in Figure 4. We can use general debugging commands such as setting and unsetting breakpoint or tracepoint at source codes using UI based on Eclipse CDT (see Figure 3). The basic usage and operation of ES-Debugger is alike to remote GDB. However the internal operation is much different from the original GDB[10]. As all command from UI must be propagated to the target system using JTAG standard, GDB MI command is transformed into JTAG command[11]. The CPU controller that is located intermediately makes JTAG command in accordance with CPU type. In most case, one gdb command is divided into various JTAG commands because one JTAG command expresses atomic operation.

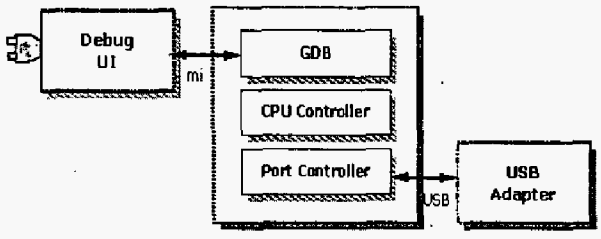

Figure 4. The internal architecture of Debugging Engine

The port controller initiates usb port of the host system and makes connection between the host system and the usb adapter. The usb controller chip used in usb adapter has a special

feature with which it is possible to download the firmware via USB and then disconnect/reconnect as a new device. It gives us many profits under and after development. The detailed information about usb connection and the role of the usb adapter is described in next chapter.

3.3 USB Adapter

Generally the CPU in embedded systems equipped with JTAG port, but host systems have no JATG port. The special device that connects host systems with target system is needed inevitably. Because of this problem, we design USB adapter as figure 5.

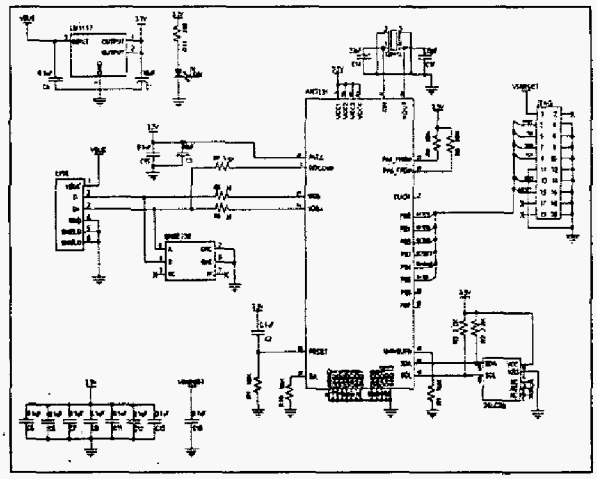

Figure 5. The Circuit Diagram of the USB Adapter

The cypress Ezusb series an2131qc is used in the usb adapter. Cypress AN2131QC has some smart features. One of these was its Re-numeration(TM) which allows it's processor to operate without ROM, EPROM or FLASH. It does this by automatically enumerating without firmware as a "Default Anchor Device". This then allows you to download 8051 code to the processor, then renumerate with your newly downloaded code. This is not only a sought after feature during development, but can also be used in the field as a means of a re-configurable device or having the ability to download the code each time the device is used to ensure the firmware is up to date.

The usb adapter is only usb device from host systems. As soon as we plug it to host systems, usb device driver is loaded on host systems. Then the firmware for usb adapter is downloaded to and installed at the usb adapter.

The usb adapter has major two functions. First is to make usb connection between the host system and target system. It is considered as general usb device. Second is to generating JTAG signal from JTAG command that is converted from GDB command at the host system. Because the JTAG signal is sensitive about the timing, the generating algorithm of firmware in the usb adapter is developed in accordance with CPU manufacturers[12].

3.4 Target Debug Handler

The target debug handler is basically used for initializing the CPU of the target system. Especially the SDRAM controller is initialized by the debug handler. It's hard coded and there is absolutely no autodetection. If you want to get a new configuration, visit Intel Web Site. The debug handler is running from the instruction cache. We can't access code in data mode. Instruction like: ldr r0, =0xFF00FF00 will not work. Use instead the macro lreg, defined in debug handler source.

4. Implementation Result

In this chapter, we show the implementation result. First of all, we introduce testing environment. It consists of the host system, the usb adapter and the target system. The host system that is installed linux operating system needs JDK, eclipse (including CDT), cross compiler for the target system and ES-Debugger. The usb adapter is handmade device. Finally, we choose the target system that Intel PXA series is installed in. There are many development board using PXA series but we have one of them fortunately named Tynux box of PalmPalm. So it's the reason that it is selected as the target system in testing environment. We make testing environment as shown Figure 6. The right side of Figure 6 is the usb adapter that has cypress AN2131QC and some other parts.

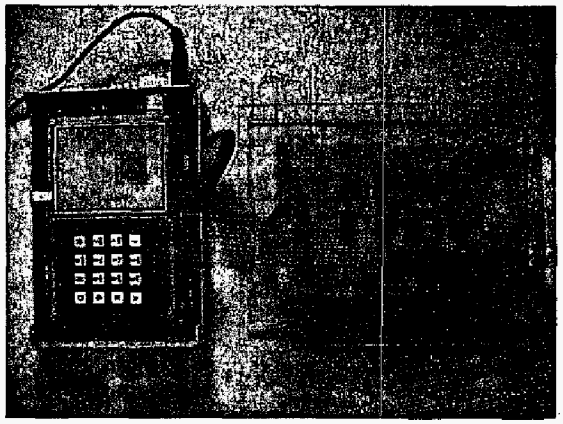

Figure 6. Testing Environment

For testing, we define testing procedure as follows because of the difference from general systems. First, make connection between the usb adapter and the target system using JTAG cable. Then plug usb cable from the usb adapter into the host system. The sequence of the connection is very important. As soon as we launch ES-debugger, it downloads the firmware for the usb adapter and the target debug handler that is located at the cache of the target CPU via usb adapter. Second, code

sample program as shown in Figure 7. The target system has no software such as bootloader so that we must set the CPU initialization in the sample code. Then it is complied using the static physical memory address. The first section of the sample code is defining the gpio register address and the next the gpio initial value. The third section declares variables and links them with the physical addresses. The body of the sample code controls the LED and the value of specified memory in the target board.

```
//                          define gpio register
#define GPSR0 0x40e00018
#define GPCR0 0x40e00024
#define GPDR0 0x40e0000c
#define GAFR0_L 0x40e00054
#define GAFR0_U 0x40e00058
#define GRER0 0x40e00030
#define GFER0 0x40e0003c
//                          define gpio initial values
#define SET_GPCR0 0x08022088
#define SET_GPDR0 0xc382a8bc
#define SET_GAFR0_L 0x80000000
#define SET_GAFR0_U 0xa51a8010
#define SET_GRER0 0x00000100
#define SET_GFER0 0x00000100
//                          define variable address
unsigned int *result=(unsigned int*)0xa0100000;
unsigned int *gpiooff=(unsigned int*)GPSR0;
unsigned int *gpioon=(unsigned int*)GPCR0;
unsigned int *gpdr_add=(unsigned int*)GPDR0;
unsigned int *gafrl_add=(unsigned int*)GAFR0_L;
unsigned int *gafru_add=(unsigned int*)GAFR0_U;
unsigned int *grer0_add=(unsigned int*)GRER0;
unsigned int *gfer0_add=(unsigned int*)GFER0;
int main(void)
{
        int i=0, product=0, j=0;  // local variable
        int sum=0;
        // initialize gpio
        *gpdr_add=SET_GPDR0;
        *gpioon= 0xffffffff;
        *gpiooff= 0xffffffff;
        *gpdr_add=SET_GPDR0;
        *gafrl_add=SET_GAFR0_L;
        *gafru_add=SET_GAFR0_U;
        *grer0_add=SET_GRER0;
        *gfer0_add=SET_GFER0;
        *gpioon= 0x1 << 5; // turn on LED
        *gpiooff= 0x1 << 5; // turn off LED
        // LED Test
        for (i=0 ; i < 10; i++) {
                *gpioon= 0x1 << 5;
                for (j=0 ; j < 10000 ; j++);
                *gpiooff= 0x1 << 5;
                for (j=0 ; j < 10000 ; j++);
        }
        // product example
        for (i=1 ; i < num2+1 ; i++)
        {
                *result=*result + num1;
                if ( i % 2 )    *gpioon= 0x1 << 5;
                else            *gpiooff= 0x1 << 5;
        }
        return 0;
}
```

Figure 7. The sample code

Third adjust compile/debug option and set breakpoint where we want using Eclipse Esto (for detailed information, see http://esto.etri.re.kr). Finally, compile the sample program using arm-cross compiler and run it. Then we can see turning on or off the LED and increasing local variables.

According to the testing result, ES-Debugger is working properly as we expected. This test system is displayed at SoftExpo 2004 and many people recognize the efficiency of this system.

5. Conclusion

As almost electronic products include embedded systems recently, it is very difficult and time consuming jobs to debug software in it. For debugging JTAG technology is very good solution, but JTAG tools are very expensive so that many developers don't take advantage of it.

In this paper, to help these developers we proposed inexpensive and flexible debugger(ES-Debugger) that has all function in commercial off the shelf debuggers. The debugging engine originated from open source (such as GNU debugger, Eclipse platform) and handmade USB adapter make it possible. At this time, the prototype system is developed and we go on additional tests for stabilizing the ES-Debugger.

In the future, we plan to show examples of the kernel and device driver debugging using ES-Debugger. Also the capability of supporting various processes will be studied.

REFERENCES

[1] IG Chun, Developing Flexible JTAG Debugger for Embedded Systems, Proceeding of Fall KIPS Conference, 2004

[2] IEEE : IEEE Standard Test Access Port and Boundary-Scan Architecture, Std 1149.1-1990, 1993.

[3] Asset InterTech, Inc., and R. G. Bennefits : Boundary-Scan Tutorial, 2000.

[4] Vink, G. : Trends in Debugging Technology, Embedded Systems Conference East, March. 1998.

[5] Gott, Robert A. : Debugging Embedded Software, Computer Design's: Electronic Systems Technology & Design, Feb98, Vol. 37 Issue 2, 1998.

[6] Lauterbach Ltd. : RTOS-Linux, 2003.

[7] Akgul T., P. Kuacharoen, V. J. Mooney and V. K. Madisetti : A Debugger RTOS for Embedded Systems, Euromicro Conference, pro. 27th, sep. 2001.

[8] Galiff, W. : Implementing a Remote Debugging Agent Using the GNU Debugger, 2001.

[9] A Technical Information to USB 2.0, 2004.

[10] Stallman, R., R. Pesch, and S. Shebs, et al. : Debugging With GDB, 2003.

[11] Minheng Tan : A minimal GDB stub for embedded remote debugging, 2002.

[12] Intel Co. : Intel PXA255 Processor, Developer's Manual, 2004.