Debugging Embedded Software

Minsoo Ryu

College of Information and Communications
Hanyang University

msryu@hanyang.ac.kr

Topics Covered

- ❒ Definition of Debugging
- ❒ Debugging Techniques
  - ▪ General Facilities Required for Debugging
  - ▪ Simple Techniques
  - ▪ Logic Analyzer
  - ▪ ROM Monitor
  - ▪ In-Circuit Emulator (ICE)
  - ▪ On-Chip Debugging
- ❒ Techniques for Avoiding Bugs
  - ▪ Scientific Debugging
  - ▪ Incremental Development
  - ▪ Defensive Programming
  - ▪ Delta Debugging

What is Debugging?

- “Debugging is the process of identifying the root cause of an error and correcting it. It contrasts with testing, which is the process of detecting the error initially.” – Steven McConnell

- “Debugging is twice as hard as writing the code in the first place. Therefore, if you write code as cleverly as possible, you are, by definition, not smart enough to debug it.” – Brian Kernighan

What is Debugging?

- ❒ **Symptom – Something wrong that you can see**
- ❒ **Bug – Software error that causes a symptom**
- ❒ **Debugging –**
  1. **Finding the bug that causes a symptom – hard!**
  2. **Fixing the bug – usually not too hard**

- ❒ **Failure-inducing input – Input that causes a bug to execute**
  - **Complication: The time at which events occur must be considered part of the input**
    - e.g., It can matter when an interrupt fires
    - Hard to reproduce this

Importance of Debugging

❒ Research carried out in 2005 indicates that approximately 40% of embedded development projects run behind schedule

- VDC 2005 Volume III: Embedded Systems Market Statistics Report

❒ Debugging is generally accepted as the most time consuming and costly phase of the development process with some estimates putting the debug cost as high as 50% of the total development cost

- Code Complete by Steven Mc Connell

❒ By it's very definition, it is also difficult if not impossible to predict when debugging is complete

- Bugs or defects are insidious by nature and can take from minutes to months to resolve

Difficulties with Debugging

- ❒ Visibility into the executing system is limited
- ❒ Several bugs are collaborating to cause the symptom that you see
- ❒ The bug and symptom are widely separated in space or time
- ❒ The “bug” is a mistaken assumption
- ❒ The bug is in a library, hardware, OS, or compiler
- ❒ The system is nondeterministic
- ❒ Worst case scenario: Many of these complications are present at the same time
  - ▪ You will need to stare at the code very hard
  - ▪ Or throw it all away and start over

Common Mistakes and Bugs

- ❒ Often the bug is something so stupid that you’ll have a hard time thinking of it
  - ▪ Accidentally running an old version of the software
  - ▪ Environment variable (e.g. PATH) is hosed
  - ▪ Buggy Makefile resulted in a file not being recompiled
  - ▪ Software upgrade added a DLL that is buggy
  - ▪ File not created since disk is full or you’re over quota
  - ▪ File unavailable due to network glitch
- ❒ There is an infinite variety of these – develop quick sanity checks to rule them out
  - ▪ Rebuild entire application
  - ▪ Reboot or switch to a different machine
  - ▪ Etc.

Topics Covered

- Definition of Debugging
- Debugging Techniques
  - General Facilities Required for Debugging
  - Simple Techniques
  - Logic Analyzer
  - ROM Monitor
  - In-Circuit Emulator (ICE)
  - On-Chip Debugging
- Techniques for Avoiding Bugs
  - Scientific Debugging
  - Incremental Development
  - Defensive Programming
  - Delta Debugging

General Facilities Required for Debugging (1)

- ❒ **Code Download**
  - **Allows program download from the host PC to the target system memory (RAM or flash based ROM) for the purposes of execution**
- ❒ **Go/Halt/Step** execution control
  - **Known as Run-time control, this allows you to control execution of the target code from within the Debugger**
- ❒ **Code Breakpoints**
  - **Allows you to halt program execution when a specified instruction or statement is executed**
  - **Breakpoints typically work by inserting (“patching”) a special instruction at the location of interest (so called Software breakpoints) or by monitoring the relevant internal microcontroller buses using a comparator and halting when a specific instruction is executed (Hardware breakpoints)**

General Facilities Required for Debugging (2)

❒ Data Access Breakpoints

- Allows you to halt program execution when a specified memory location is accessed for the purposes of reading or writing to it

❒ Complex/Advanced Breakpoints

- These allow you to qualify breakpoints e.g. a range of addresses may be specified (as opposed to a single address) or a mask may be used to allow the setting of “don’t care” conditions
- In addition, a breakpoint may be specified only to occur when a defined sequence of events has happened e.g. when execution from a specified code address is followed by a write to a specific data address then assert the breakpoint (i.e. halt the application)

General Facilities Required for Debugging (3)

- ❒ **Memory/Register Access**
  - This allows Registers and Memory to be read and written to by the Debugger
- ❒ **"On-the-fly" Access**
  - Generally means that specific debug functions can be done while the target microcontroller is executing
- ❒ **Real-time Trace and Watchpoints**
  - Provide a mechanism for capturing program flow (or data access) in real-time
  - Typically, trace information is emitted from the microcontroller, captured and time stamped and stored by the Emulator
  - Watchpoints operate similar to Breakpoints, however, instead of halting the microcontroller they simply indicate that the specified condition has occurred and can be used by the Emulator to filter or trigger capture of trace information

Simple Techniques (1)

❒ **LEDs under software control**

- **Minimal workable debugging environment**
- **A most unpleasant way to debug complex software**

❒ **printf() to serial console or LCD**

printf - 사용 쉽고, 이해하기 쉽다.
probe effect - printf 사용으로 인해 기존 코드의 행동에 영향을 준다.

- **Severely perturbs timing, typically**
- **Generally, a debug printf() is synchronous**
  - Hangs the system until the printf completes

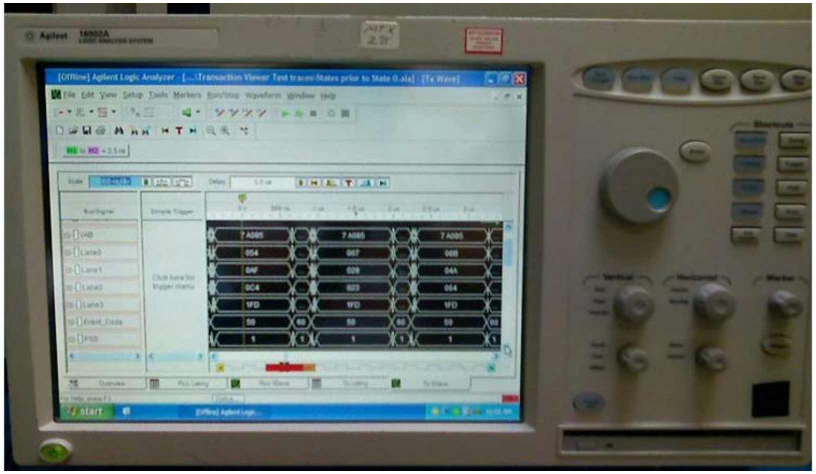
Logic Analyzer (1) - hardware

- A logic analyzer is an electronic instrument that displays signals in a digital circuit that are too fast to be observed and presents it to a user so that the user can more easily check correct operation of the digital system
  - They are typically used for capturing data in systems that have too many channels to be examined with an oscilloscope
    - A logic analyzer usually has between 34 and 136 signals (input channels)
    - An oscilloscope usually has up to four channels
  - Software running on the logic analyzer can convert the captured data into timing diagrams, protocol decodes, state machine traces, assembly language, or correlate assembly with source-level software
  - Agilent and Tektronix make up over 95% of the industry's revenue

Logic Analyzer (2)

❒ A logic analyzer measures and analyzes signals differently than an oscilloscope

- The logic analyzer doesn't measure analog details
- Instead, it detects logic threshold levels
- The oscilloscope is used for analog measurements such as rise- and fall-times, peak amplitudes, and the elapsed time between edges

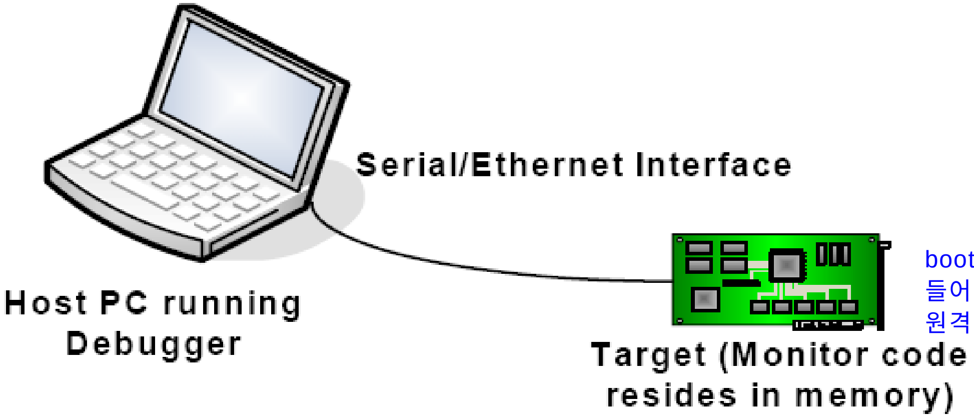


symbolic debug - 변수 수준으로 debuging
logic analyzer - register&hardware binary bit 수준으로 debuging

The four steps to using a logic analyzer:
1. Probe: Connect to the System Under Test – SUT
2. Setup (clock mode and triggering)
3. Acquire
4. Analyze and display

ROM Monitor (1)

- A monitor is a piece of code that runs on the target system that is being debugged
  - The Debugger, running on the host PC communicates to the monitor via a dedicated port (a serial or ethernet port)
  - The Debugger sends commands to the Monitor which carries them out and returns a response
  - A typical example is a request to read target memory at a defined memory location for a specified number of bytes


Serial/Ethernet Interface
Host PC running Debugger
Target (Monitor code resides in memory)

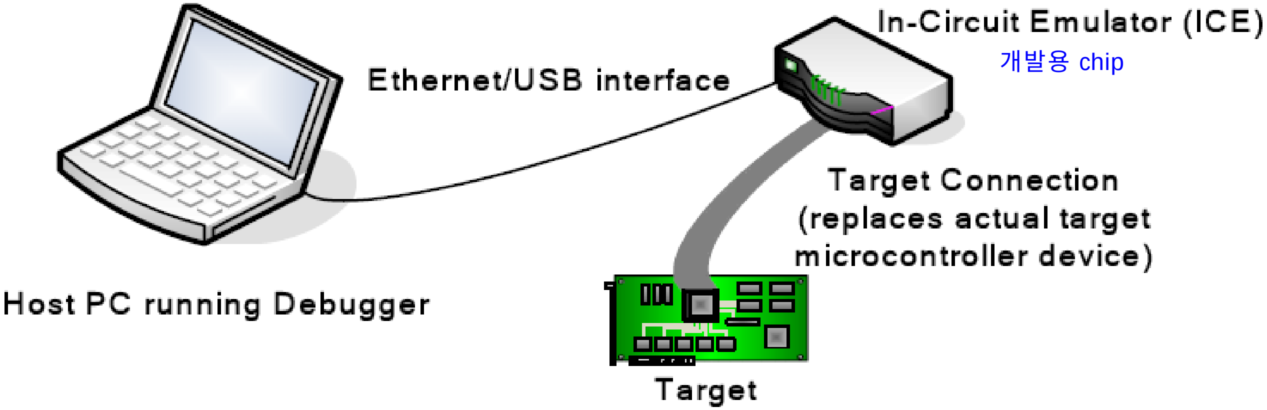
bootloader에 monitor 코드가 대부분
들어가 있다.
원격 debuger

ROM Monitor (2)

- Monitors are useful and cheap, however, they are limited as follows:
  - The Monitor resides in target memory, therefore it takes up application space and must be programmed into memory before debugging can take place
  - The Monitor requires a dedicated port on the target device to communicate with the host PC
  - The Monitor can only run if the target is itself running, therefore debugging a “dead” target or one that does not boot is not possible
  - Debugging Flash or ROM based code with a Monitor may not be possible given the need for the Monitor to be able to write to target memory (“patch”) when setting software breakpoints

In-Circuit Emulator (1)

- ❒ **An In-Circuit Emulator (ICE) works by actually replacing the target's microcontroller in-circuit and "emulating" its functionality**
  - ▪ **The ICE is connected to the host PC and typically controlled by a Debugger**
  - ▪ **The ICE provides full visibility into the inner workings of the "emulated" microcontroller and allows fine-grained control of program execution and access to all its inner workings**


In-Circuit Emulator (ICE)
개발용 chip
Ethernet/USB interface
Target Connection (replaces actual target microcontroller device)
Host PC running Debugger
Target

In-Circuit Emulator (2)

- **Emulating a microcontroller is a very complex task and is typically implemented using a special version of the microcontroller chip called a “bond-out” device which provides the core functionality of the ICE**
  - **The bond-out device is developed and supplied by the microcontroller vendor; the term “bond-out” derives from the fact that additional internal buses and signals not present on the standard device are “bonded-out” to pins to provide the visibility and control needed by the ICE**
  - **Bond-outs are expensive to produce, especially considering that they are only used by ICE vendors and may need subsequent revisions as errata are discovered and fixed in the standard device**
  - **The associated costs have prompted some microcontroller vendors to look at alternatives to bond-outs such as the use of Field Programmable Gate Arrays (FPGAs)**
    - FPGAs are a viable solution in some cases, however, they will not run as fast as bond-out devices or emulate analog circuitry (e.g. Digital to Analog Converters) and thus need additional external circuitry

On-Chip Debugging

- ❒ Due to the costs and limitations associated with ICEs, many semiconductor vendors integrate dedicated debug circuitry into their chips

- ❒ Motorola (Freescale) was one of the first vendors to do so and its Background Debug Mode (BDM) is one of the more widely known on-chip debug standards
  - The on-chip debug circuitry typically interfaces to the outside world via a serial interface with JTAG being the most popular standard used
  - JTAG, stands for Joint Test Action Group, the name used for the IEEE 1149.1 standard entitled Standard Test Access Port and Boundary-Scan Architecture

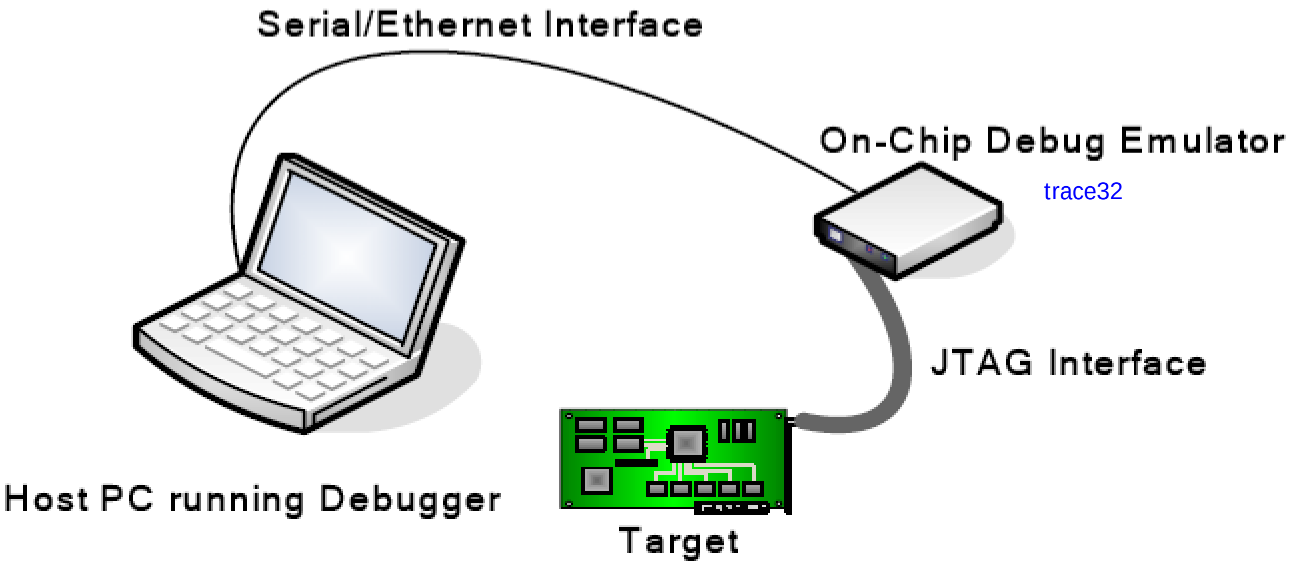
Why On-Chip Debugging?

- On-chip debugging support logic is required in today's (and tomorrow's) microcontrollers because without such support logic it would be impossible or prohibitively expensive to produce debugging tools capable of supporting increasing clock speeds
  - The use of on-chip Flash memory for program storage make it difficult or impossible for external tools (Logic Analyzers or ICEs, for example) to determine the actual instruction being currently executed as there is no external or off-chip visibility of the program address bus on such “Single-Chip” microcontrollers
  - High microcontroller clock frequencies mean that an in-circuit emulator cannot halt target-program execution before it executes a particular instruction of interest
  - Deep instruction pipelines, multiple-issue RISC architectures and on-chip caches can make it very difficult to determine which instructions were fetched and which were actually executed
  - The introduction of application-specific or customer-specific Systems-On-a Chip (SOCs), some with multiple processors, means that a uniform, reliable debugging interface is essential to eliminate the need to redesign the Microprocessor Emulator for each individual SOC design

JTAG Emulator

- The user’s PC interfaces to the Target via a JTAG Emulator which provides a JTAG interface and a USB/Ethernet connection to the host PC

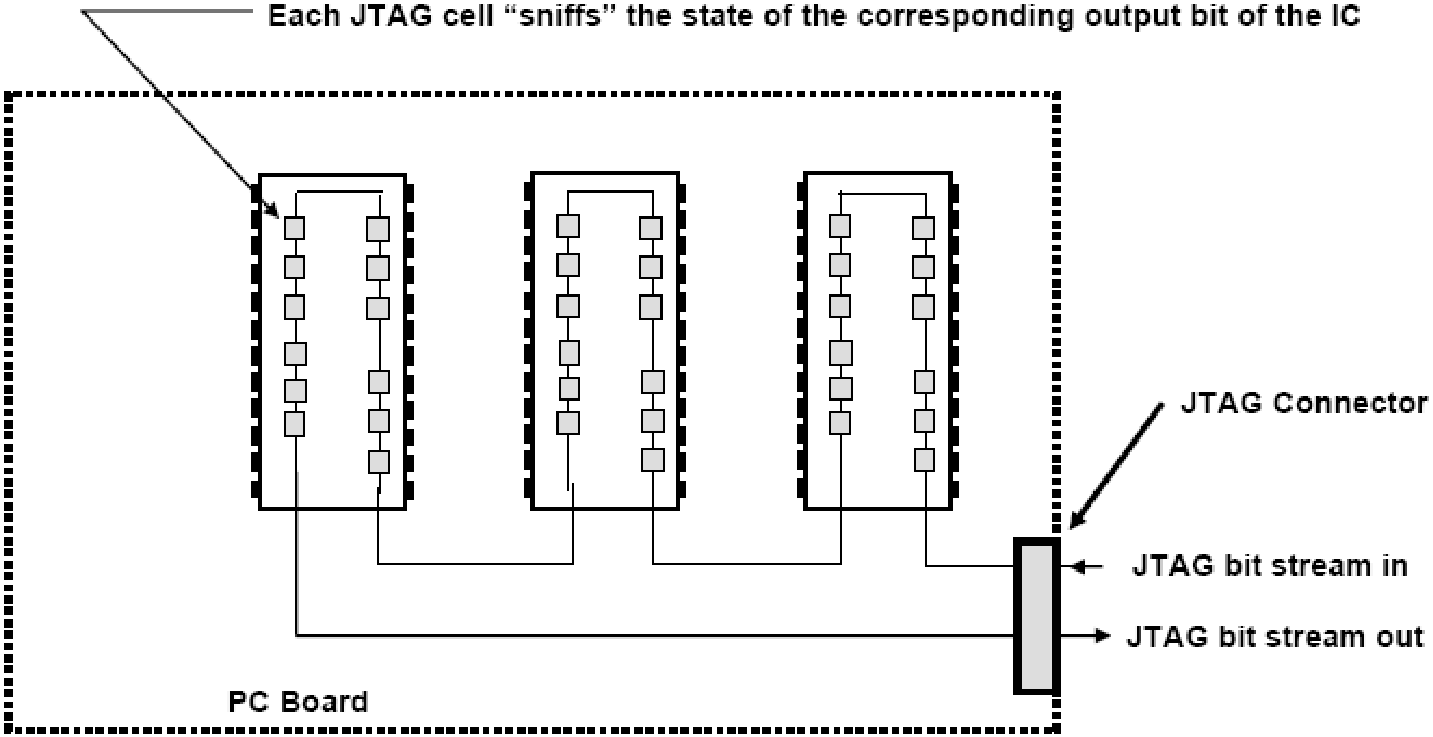
arm926EJS - J 의미 : jtag



IEEE 1149.1 Standard

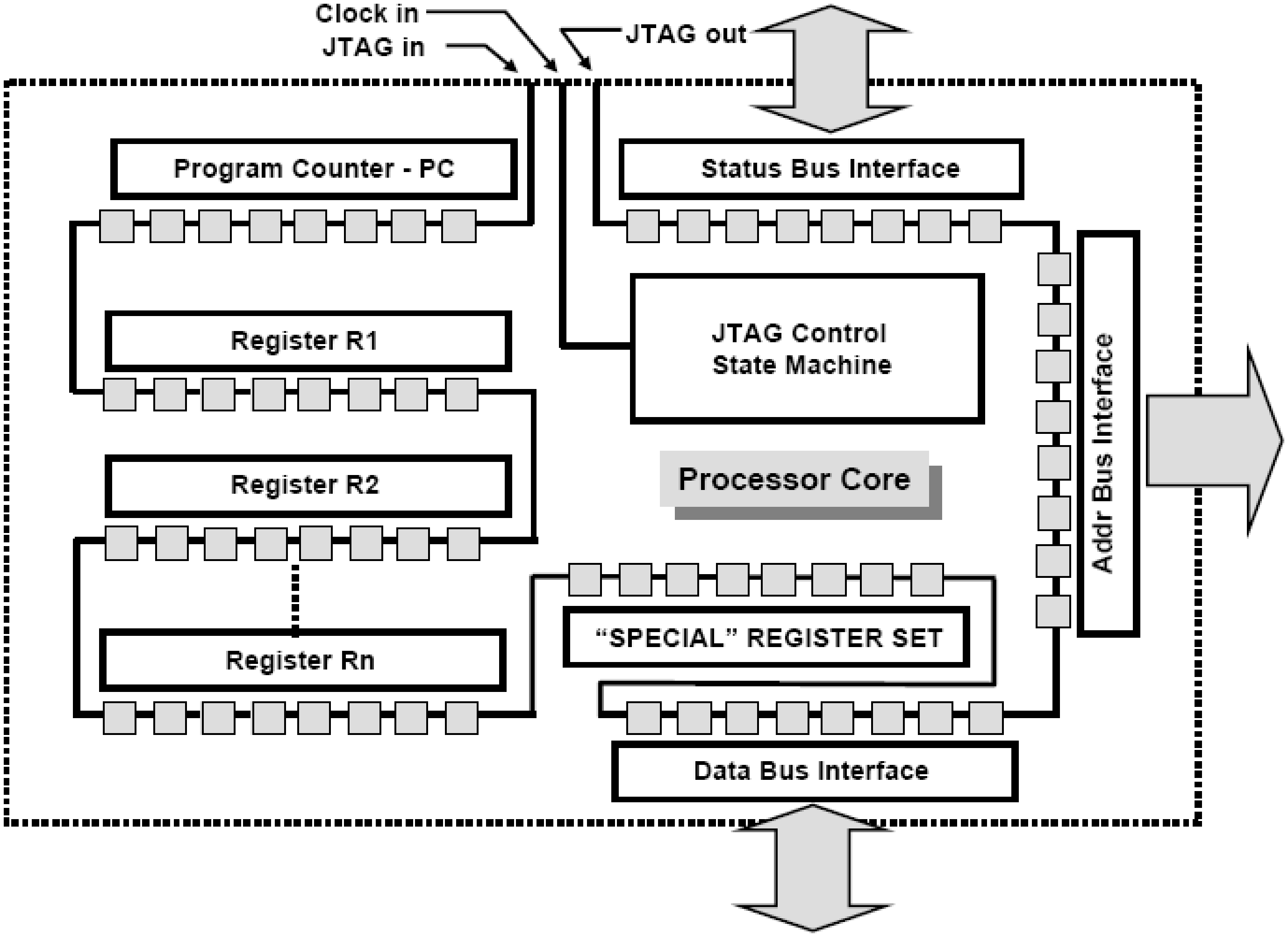
- ❒ Joint Test Action Group (JTAG) is the usual name used for the IEEE 1149.1 standard entitled Standard Test Access Port and Boundary-Scan Architecture for test access ports used for testing printed circuit boards using boundary scan
- ❒ Basic ideas
  - Each I/O pin, register, etc. can be “sniffed” by a JTAG cell
  - JTAG cells are connected in a “JTAG loop”
  - Contents of entire JTAG loop can be read using a shift register
    - Can also be written
  - External tool can reconstruct machine state from the JTAG bit stream

Basic Concept of JTAG

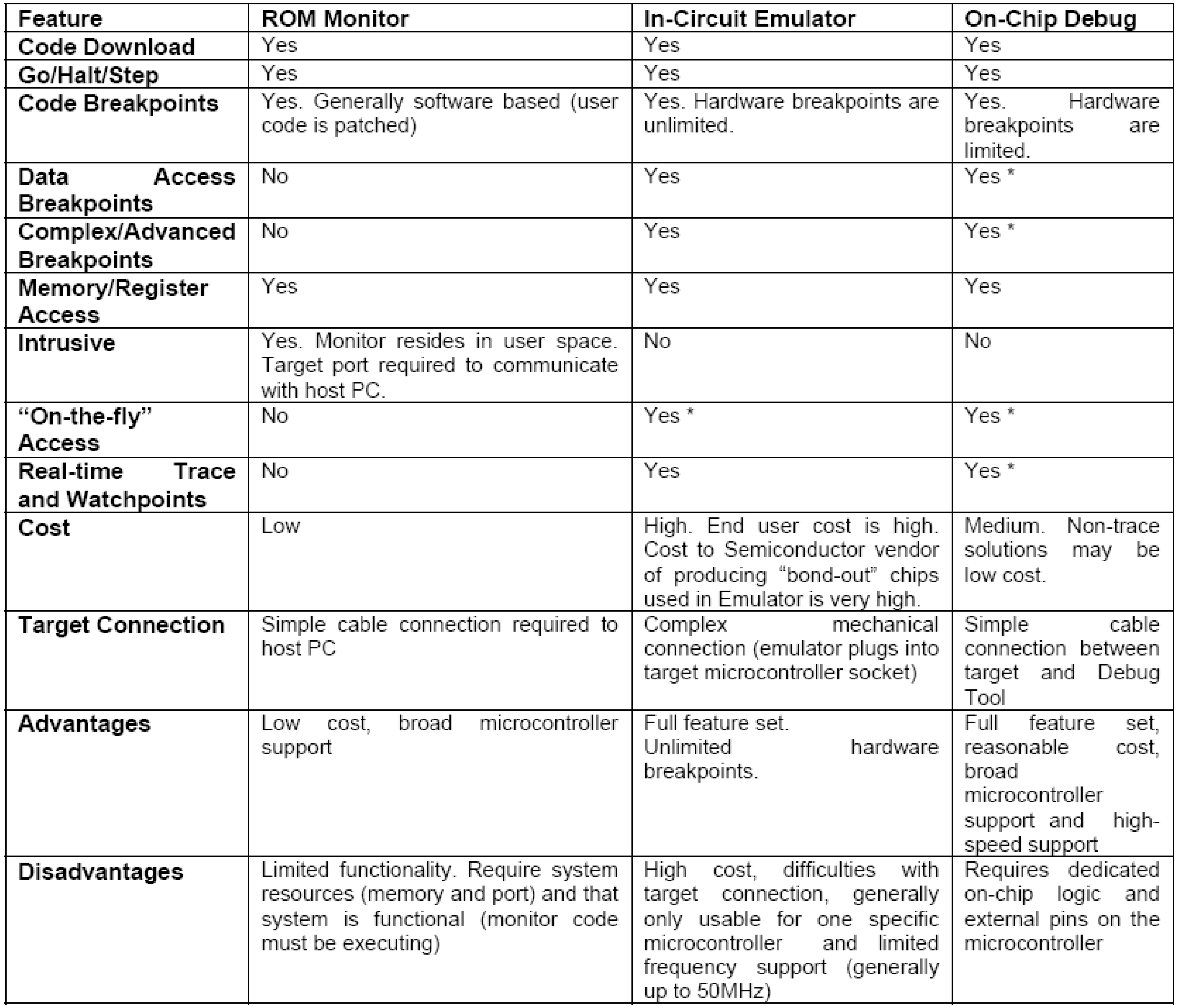

Each JTAG cell "sniffs" the state of the corresponding output bit of the IC
JTAG Connector
JTAG bit stream in
JTAG bit stream out
PC Board

Bit stream forms one long shift-register

Basic Concept of JTAG



Comparison of Debugging Techniques



Feature	ROM Monitor	In-Circuit Emulator	On-Chip Debug
Code Download	Yes	Yes	Yes
Go/Halt/Step	Yes	Yes	Yes
Code Breakpoints	Yes. Generally software based (user code is patched)	Yes. Hardware breakpoints are unlimited.	Yes. Hardware breakpoints are limited.
Data Access Breakpoints	No	Yes	Yes *
Complex/Advanced Breakpoints	No	Yes	Yes *
Memory/Register Access	Yes	Yes	Yes
Intrusive	Yes. Monitor resides in user space. Target port required to communicate with host PC.	No	No
"On-the-fly" Access	No	Yes *	Yes *
Real-time Trace and Watchpoints	No	Yes	Yes *
Cost	Low	High. End user cost is high. Cost to Semiconductor vendor of producing "bond-out" chips used in Emulator is very high.	Medium. Non-trace solutions may be low cost.
Target Connection	Simple cable connection required to host PC	Complex mechanical connection (emulator plugs into target microcontroller socket)	Simple cable connection between target and Debug Tool
Advantages	Low cost, broad microcontroller support	Full feature set. Unlimited hardware breakpoints.	Full feature set, reasonable cost, broad microcontroller support and high-speed support
Disadvantages	Limited functionality. Require system resources (memory and port) and that system is functional (monitor code must be executing)	High cost, difficulties with target connection, generally only usable for one specific microcontroller and limited frequency support (generally up to 50MHz)	Requires dedicated on-chip logic and external pins on the microcontroller

Topics Covered

- ❐ Definition of Debugging
- ❐ Debugging Techniques
  - ▪ General Facilities Required for Debugging
  - ▪ Simple Techniques
  - ▪ Logic Analyzer
  - ▪ ROM Monitor
  - ▪ In-Circuit Emulator (ICE)
  - ▪ On-Chip Debugging
- ❐ Techniques for Avoiding Bugs
  - ▪ Scientific Debugging
  - ▪ Incremental Development
  - ▪ Defensive Programming
  - ▪ Delta Debugging

The Process of Scientific Debugging

1. Stabilize error가 발생한 상황을 연출
   - Pin down system inputs and as many other variables as possible
   - Buggy behavior must be repeatable (not necessarily deterministic)
2. Isolate
   - Get rid of as much extraneous hardware and software as possible while still retaining the buggy behavior
3. Hypothesize
   - Make educated guesses about causes that might explain the observed behavior
   - Initial hypothesis is often pretty vague – this is ok
   - Write down the hypothesis
   - Rank possibilities in order of likelihood

The Process of Scientific Debugging

4. Experiment
   - Should be designed to reject some subset of hypothesis no matter what the result of the experiment is
   - Then go back to Step 3
   - Eventually the hypothesis zeroes in on the actual bug
5. Fix the bug
6. Verify the solution
   - Ensure that it doesn't break anything else
7. Undo extraneous changes introduced to the system
8. Re-test the system
   - Often errors come in groups
9. Create a regression test

자동차 급발진

2014 barr group (embedded sw consoltants)

- 도요타 급발진의 원인은 소프트웨어에 있다!!

과거 자동차 S/W OSECK -> AUTOSAR

도요타 OSECK 자사 customize 시킨 os 사용

bitmap의 bit 변경이 h/w 기타 다른 s/w에 의해서 변경 가능

misra-C 코드 규칙 - 규칙 지키지 않음

A-spice - 규칙 지키지 않음

BOSCH - 급발진 확률이 적은 안정적인 s/w 사용하는 기업

Scientific Debugging

- **A good experiment rules out about 50% of the "bug probability space"**
  - **Debugging by binary search**
    - Requires log N experiments if there are originally N hypotheses
  - **So, what bugs are possible and what are their probabilities of occurring?**
    - This is pretty much the central question!
    - Your intuition is your main tool here
- **Occam's Razor: When given two equally valid explanations for a phenomenon, one should embrace the less complicated explanation**

Two Heads Are Better

- If a bug you wrote is due to a conceptual error, then by definition you can’t find the bug
  - This is common in practice
  - Often talking or emailing helps you find the bug

Incremental Development

- Starting with working code:
  - Make small change
  - Test
  - If it still works, go to step 1
  - If it doesn’t work, back out the change and to go step 1

- Why don’t we always do this?
  - We think we can get it right
  - Incremental development is a hassle

Defensive Programming

- Reduce code complexity
  - Don’t optimize until you know it’s needed

if else - 수가 적을 수록 좋다.
일정 부분 짜고 돌려보는 습관이 중요하다.
한꺼번에 짠 후 한꺼번에 돌리는 것이 아니라.

- Test early and often
- Avoid hidden dependencies
  - E.g., writing an accelerometer driver that assumes the timer is already initialized
- Be very careful when reusing code
  - Write test cases
  - Check values that come from external code
- Include sanity checks
  - Assume your code runs in a hostile environment
- Don’t ignore failures and errors

Programming with Asserts

- Sanity check: Making sure something that “must be true” is actually true
  - Placing asserts well requires judgement calls

- An assertion is a predicate (i.e., a true–false statement) placed in a program to indicate that the developer thinks that the predicate is always true at that place

- Assertions are used to help specify programs and to reason about program correctness
  - A precondition — an assertion placed at the beginning of a section of code — determines the set of states under which the code is expected to be executed
  - A postcondition — placed at the end — describes the expected state at the end of execution

Programming with Asserts

- When an assertion failure occurs, the programmer is immediately notified of the problem
- Many assertion implementations will also halt the program's execution — this is useful, since if the program continued to run after an assertion violation occurred, it might corrupt its state and make the cause of the problem more difficult to locate
- Using the information provided by the assertion failure, the programmer can usually fix the problem
  - Thus, assertions are a very powerful tool in debugging

Programming with Asserts

- In Java, assertions have been a part of the language since version 1.4
- In C, they are added on by the standard header assert.h defining assert (assertion) as a macro that signals an error in the case of failure, usually terminating the program
- In standard C++ the header cassert is required instead
  - However, some C++ libraries still have the assert.h available

Programming with Asserts

depensive 프로그램의 기법중 하나

❒ Bad assert:

- y = x + z; assert (y == (x + z));

precondition assert
code
postcondition assert

❒ Good asserts:

- assert (x >= 0); y = sqrt(x);
- assert (p); *p = 5;
- assert (i < 10); a[i]++;
- assert (sizeof(int) == 4); int의 size는 machin&language&compile dependent한 요소이다.
- x = malloc(sizeof(foo)); assert ((x & 0x7) == 0);
- function_that_doesnt_return(); assert (0);
- assert (SP > (_bss_end + 75));

Programming with Asserts

- Good asserts check
  - Preconditions – must be true for code to run successfully
  - Postconditions – must be true after code runs
  - Invariants – must be true at all times (except while data structure is being manipulated)
    - E.g., length field of a linked list ADT stores the length

Delta Debugging

- ❐ **Delta algorithm:**
  1. **Run the program on input that makes the bug happen**
  2. **Delete some of the input**
  3. **Run again the program on reduced input that makes the bug happen**
  4. **See if the bug still happens**
     - If yes, go to step 2
     - If no, go back to old input and go to step 3

- ❐ **Result: Smaller input that makes the bug happen**