AN 947: Testing the Nios® II Software with Unit-Test Framework
Testing the Nios® II Software with Unit-Test Framework

This document describes a unit-test framework for testing embedded software known as System Mock that runs on the Intel FPGA Nios® II processor. At a high level, it simulates the FPGA environment that interacts with Nios II software. Under this framework, you can test Nios II code with little to no modification. Through System Mock, developers can inject simulated hardware stimuli to exercise different code paths.

System Mock unit-test framework is a wrapper that allows Nios II user code to execute as if it is on hardware. This framework focuses on modeling the hardware interaction and memory access of a Nios II process. In FPGA designs, Nios II system is usually connected to other hardware components via an Avalon® Memory-Mapped Interface. System Mock overwrites the Nios II intrinsic functions to intercept the Nios II outgoing Avalon memory-mapped interface memory accesses. This key technique allows you to test unmodified Nios II user code from beginning to end. The intercepted memory accesses require appropriate responses. In System Mock, a list of IP mock modules which model hardware behaviors respond to them. These are the core building blocks of this unit-test framework.

Intrinsic Function Overwrite

Nios II software operates on top of the board support package (BSP) for the Nios II processor. BSP provides many useful application programming interfaces (APIs) which assist software to execute on the Nios II processor and interact with hardware. For Nios II software, the hardware interaction means reading from or writing to certain registers in a certain way. Therefore, the System Mock can intercept these read or write (RW) operations to model the Nios II software interactions with hardware.

In the Nios II BSP, there are macros defined for register RW. The commonly used ones are IORD, IORD_32DIRECT, IOWR, and IOWR_32DIRECT. These macros call built-in functions __builtin_ldwio and __builtin_stwio. System Mock can provide new definitions for these built-in functions. Overwrite the definitions of these built-in functions to redirect the memory access and augment it with side effects.

One caveat of this technique is that developers must write code in a certain way. All Avalon memory-mapped interface RW operations must use function calls (for example: IORD/IOWR). Memory access through pointers is allowed in this unit-test framework. The prerequisite is that the System Mock has allocated a memory block and replaced the base address in Nios II code appropriately. However, System Mock cannot intercept or augment these pointer-based memory access.

Example code for intrinsic function overwrite:

```c
// NIOS II software code
// main.c
alt_u32 mb_val = IORD_32DIRECT(U_MAILBOX_AVMM_BRIDGE_BASE, 0);
// NIOS II BSP provides macros for generic read and write
```
// HAL/inc/io.h
#define __IO_CALC_ADDRESS_DYNAMIC(BASE, OFFSET) \  
((void *)((alt_u8*)BASE) + (OFFSET)))

#define IORD_32DIRECT(BASE, OFFSET) 
__builtin_ldwio (__IO_CALC_ADDRESS_DYNAMIC ((BASE), (OFFSET))))
#define IOWR_32DIRECT(BASE, OFFSET, DATA) 
__builtin_stwio (__IO_CALC_ADDRESS_DYNAMIC ((BASE), (OFFSET)), (DATA))

// In unit-test, System Mock framework provides the definitions of these built-in functions
// bsp_mock.h
static alt_u32 __builtin_ldwio(void* src)
{
    return SYSTEM_MOCK::get()->get_mem_word(src);
}
static void __builtin_stwio(void* dst, alt_u32 data)
{
    return SYSTEM_MOCK::get()->set_mem_word(dst, data);
}

// System Mock goes through the list of IP Mock modules and passes on the Read/Write requests.
// system_mock.cpp
alt_u32 SYSTEM_MOCK::get_mem_word(void* addr)
{
    alt_u32 ret = 0;
    // Dispatch to the appropriate handler based on the address
    for (auto& mock : m_memory_mocks)
    {
        if (mock->is_addr_in_range(addr))
        {
            ret = mock->get_mem_word(addr);
            break;
        }
    }
    return ret;
}
void SYSTEM_MOCK::set_mem_word(void* addr, alt_u32 data)
{
    // skipped ...
}

// In Mailbox IP mock module, define IP behavior in get/set_mem_word functions
// mailbox_mock.h
class MAILBOX_MOCK : public MEMORY_MOCK_IF
{
public:
    MAILBOX_MOCK();
    virtual ~MAILBOX_MOCK();
    void reset() override;
    alt_u32 get_mem_word(void* addr) override;
    void set_mem_word(void* addr, alt_u32 data) override;
    bool is_addr_in_range(void* addr) override;
private:
    // A simple array to store register values in Mailbox
    std::array<alt_u32, U_MAILBOX_AVM Bridge_SPAN> m_mailbox_regs;
};

Consider a simple FPGA design with a Nios II system and a mailbox on an Avalon memory-mapped interface. Here, a mailbox is a simple collection of hardware registers that allows Intel FPGA Nios II processor to communicate with the outside
world. These mailbox registers reside in 0x2000–0x2fff of the Avalon memory-mapped interface address range. When the Nios II software executes in an x86 unit-test environment, there are RW operations on this address range, which is arbitrary and probably illegal for CPU processes. To avoid that, in System Mock, user can define a 4KB array representing the mailbox. System Mock overwrites the intrinsic functions and redirects the memory access to this array.

Figure 1. **Intrinsic Function Overwrite Example**

![Diagram showing the concept of intrinsic function overwrite example.]

---

**IP Modeling**

Besides support for connectivity, the System Mock also provides several C++ interfaces to define software models that mimic behaviors of hardware components. As mentioned earlier, these models are known as IP mocks. In an IP mock, you can implement IP’s functionality and establish the expected handshake with the Nios II software. System Mock instantiates and maintains all IP mocks. When System Mock intercepts a Nios II Avalon memory-mapped interface memory access, it goes through a list of IP mocks and passes the memory access to the appropriate IP mock. Then, this IP mock responds to the Nios II memory access accordingly. In addition, System Mock keeps lists of user-defined pre-read, post-read, pre-write, and post-write callback functions. By calling those functions, System Mock can inject events before or after passing on the memory access to IP mock.

To illustrate this further, take an example of a Crypto IP, which provides cryptographic services in an FPGA design. This Crypto IP has a control and status registers (CSR) interface. Beneath this communication interface, there is a state machine to deal with requests and responses. This IP also has core algorithms to perform cryptographic functions such as hashing. In the mock module for this Crypto IP, developers can define the same CSR interface and use switch statements to represent the state machine. For the cryptographic functions, developers can call OpenSSL C++ library APIs to implement the same functionality. From Nios II code’s perspective, interaction with this IP mock is the same as with the actual Crypto IP.

It can be a big overhead to implement IP mocks for all hardware components in an FPGA design. Developers can choose to only implement the relevant features to cut down this overhead. IPs are usually designed to support a wide range of use cases. For example, a flash memory IP can support many configuration settings and hundreds of commands. But a Nios II software may only send read, write, and erase commands in a particular mode. Developers can focus on implementing these commands and flash mode in their flash memory IP mock.
Unit-Test Structure with System Mock

Figure 2. Unit-test Framework High-level View

This diagram provides visualization on how to integrate Nios II code into a System Mock unit-test framework. During compilation, Nios II code must include System Mock library files to access mock IP components and the right definitions of built-in functions. Intel recommends using a common header file to include Nios II BSP library in your Nios II code. This allows you to insert System Mock library from one location. Afterwards, individual unit-test suite can include the Nios II source code, either partially or fully depending on its test scope.

Conclusion

The System Mock framework is tailored to the unit-testing needs of a Nios II software. It provides the observability into state changes, event injectability and side effect creation. Beside the usability aspects, System Mock is highly scalable and reusable due to its modular design. IP Mock modules can be re-used or expanded for future projects. Developers can swiftly swap in or out any IP Mock modules, in line with project requirements. With this framework, developers can easily achieve a high level of coverage, because you can execute the Nios II software from its entry point.

Example Design

Refer to the FPGA Soft Processor Unit Test Example Design, an example FPGA design with Nios II system, along with an example System Mock unit-test framework.

Related Information

FPGA Soft Processor Unit Test Example Design
Document Revision History for AN 947: Testing the Nios II Software with Unit-Test Framework

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2021.04.30</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>