MANAGING AND AUTOMATING HW/SW TESTS FROM IP TO SOC

Matthew Ballance
matt_ballance@mentor.com
Mentor, A Siemens Business
Wilsonville, OR

***Abstract*-This paper presents the key features of a portable micro-executor framework that makes a key set of services required by low-level driver code portable across IP, subsystem, and SoC verification environments. It will also show how combining this technique with portable stimulus techniques, supported by the emerging Accellera Portable Stimulus standard, enables reuse of high-level test intent and low-level driver code across IP, subsystem, and SoC-level environments. Benefits, including finding hardware/software bugs earlier, increasing reuse of test intent and driver code, and accelerating the SoC validation process, will be illustrated using examples.**

I. INTRODUCTION

The functionality of today's SoC designs is increasingly driven and directed by software, and much of that software is tightly coupled with behavior of the hardware that it controls. Test software is a critical component of verification environments at the SoC level, but bringing software into the verification process as early as the IP level can be beneficial both in terms of finding Hw/Sw interface bugs earlier and making test creation at the SoC level more productive.

There are two key challenges to making software a key and consistent element of the verification process from IP to SoC level. First, the test-automation capabilities present in IP, subsystem, and SoC environments is very different. IP-level environments (and to some extent subsystem-level environments) benefit from constrained-random stimulus generation. In contrast, SoC environments typically provide no automated means of test creation. Secondly, the way that IP-specific content interacts with the design and with other pieces of IP-specific test content is quite different at IP, subsystem, and SoC level. At IP level, test content only deals with a single IP block and uses SystemVerilog mechanisms for threading and access IP registers via verification IP APIs or transactions. At SoC, the test scenario must deal with multiple IPs, and use either bare metal or OS mechanisms for threading.

These differences in test-creation capabilities and mechanisms for software to interact with hardware pose a significant obstacle to reusing software-driven test content from IP to SoC level. This paper describes how the emerging Accellera Portable Stimulus Specification standard enables automated creation of test content from IP to SoC level, and how a hardware abstraction layer for test software simplifies integration of test content

EXAMPLE

The examples in this paper will primarily be based around the Wishbone DMA IP from opencores.org [1]. A block diagram of this IP is shown in Figure 1. The DMA engine provides 31 DMA channels which can perform memory-to-memory transfers, as well as transfers controlled by hardware handshake signals.

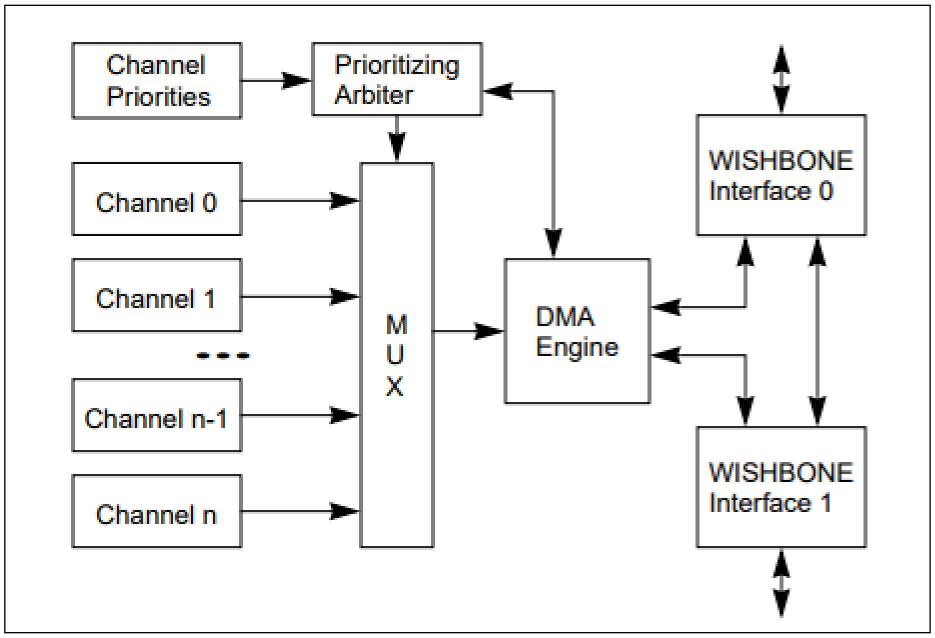

Figure 1 - Opencores Wishbone DMA Engine

Detailed IP-level verification of this block, as any other, would be performed in a UVM environment using directed, constrained-random, and other methods, to exercise the implementation to confirm that it conforms to the design specification. Thinking ahead to the verification needs at subsystem and SoC level, developing some basic firmware for exercising the IP as a part of the IP verification process would greatly help when a system containing the DMA engine is assembled.

PSS Scenario Creation

Portable Stimulus is a key enabler for reusable test content across IP, subsystem, and SoC levels. The emerging Accellera PSS standard input language [2] provides a standard way for users to capture portable stimulus content, such that it can be consumed by multiple vendor tools. Several commercial tools, including Questa inFact [3] from Mentor, A Siemens Company, provide portable stimulus functionality.

Modeling DMA Transfers with Portable Stimulus

Portable stimulus techniques apply from IP to SoC level, but add particular value at SoC level where there is little in the way of test-creation automation. At SoC level, interactions between IP blocks are typically the target of verification and validation. Consequently, test scenarios use devices, like a DMA engine, in the ways that they will be used in the target system. This is quite different from IP-level verification, where all possible use models of an IP are verified.

```
component dma_c {
        buffer mem_seg_b {
                rand bit[31:0]            addr;
                rand bit[31:0]            size;
        }

        resource dma_channel_r { }

        pool dma_channel_r     channels[31];

        action dma_mem2mem_xfer_a {
                input mem_seg_b           src;
                output mem_seg_b    dst;
                lock dma_channel_r  channel;

                constraint size_match {
                        src.size == dst.size;
                }

                // Target implementation left unspecified
        }
}
```

Figure 2 - PSS DMA Component and Transfer Action

Figure 2 shows an example description in the PSS domain-specific language that captures a DMA transfer. In PSS, a component captures the set of resources available to a device, and the actions (operations) that can be performed on the device. In this case, the ‘channels’ field in the component captures the fact that our DMA engine has 31 channels and can execute up to 31 parallel transfers. We can build higher-level scenarios on top of the DMA component and action to, for example, execute a number of parallel transfers on randomly-selected channels.

Specifying Test Realization with PSS

The PSS description shown in Figure 2 doesn’t specify anything about how the behavior will be carried out (realized). The PSS description deliberately allows the mapping to a specific realization mechanism to be specified separately from the core description. This allows the same PSS test intent to be targeted to environments that have very different interface mechanisms.

```
package dma_c_pkg {

        import void wb_dma_xfer(
                bit[31:0]           channel,
                bit[31:0]           src_addr,
                bit[31:0]           dst_addr,
                bit[31:0]           size
        );

        extend action dma_c::dma_mem2mem_xfer_a {
                exec body {
                        wb_dma_xfer(
                                channel.instance_id,
                                src.addr,
                                dst.addr,
                                src.size
                        );
                }
        }
}
```

Figure 3 - PSS Test Realization Mapping

Figure 3 shows a specific test-realization mapping that is described using a type extension to the dma_c::dma_mem2mem_xfer_a action. This realization mapping directs the PSS processing tool to call the wb_dma_xfer function to carry out activity of the dma_mem2mem_xfer_a action.

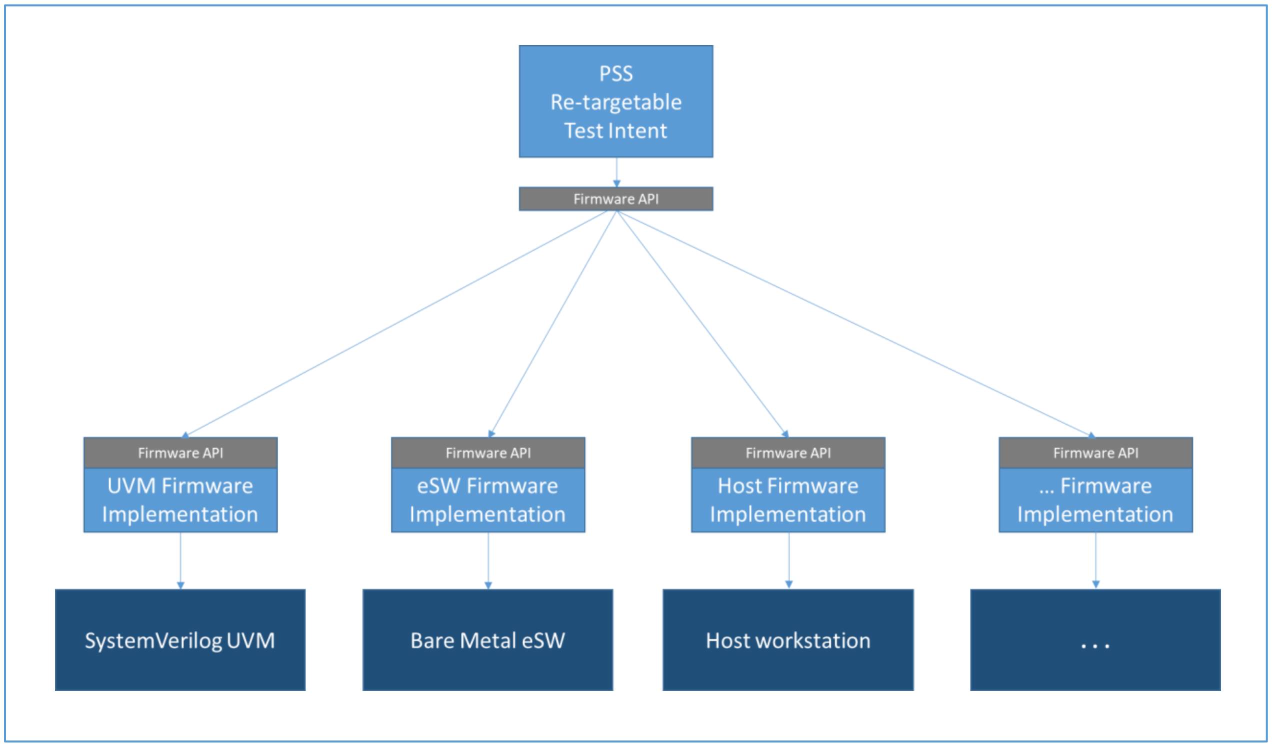

Figure 4 - Reusing PSS Test Intent with a Common API

Now, if we wish to execute the same test intent in our IP, subsystem, and SoC environment, we will at least require a SystemVerilog and an embedded software version of the wb_dma_xfer function. This situation is illustrated by Figure 4. A common test API is used, which maximizes the reusability of the PSS description. However, the result is a different implementation of that firmware API for each test environment. But, what if we wanted to start testing the actual firmware for interacting with our DMA engine at IP level, and reuse that same firmware at subsystem and SoC level? Finding simple bugs in basic firmware is much simpler at IP level, than when we're also trying to bring up a full SoC and embedded-software stack.

UEX SOFTWARE ABSTRACTION LAYER

This paper describes the micro-executor (UEX) API [4], a software abstraction layer that enables the same firmware-level code to provide test-realization utility to test scenarios across a range of verification environments.

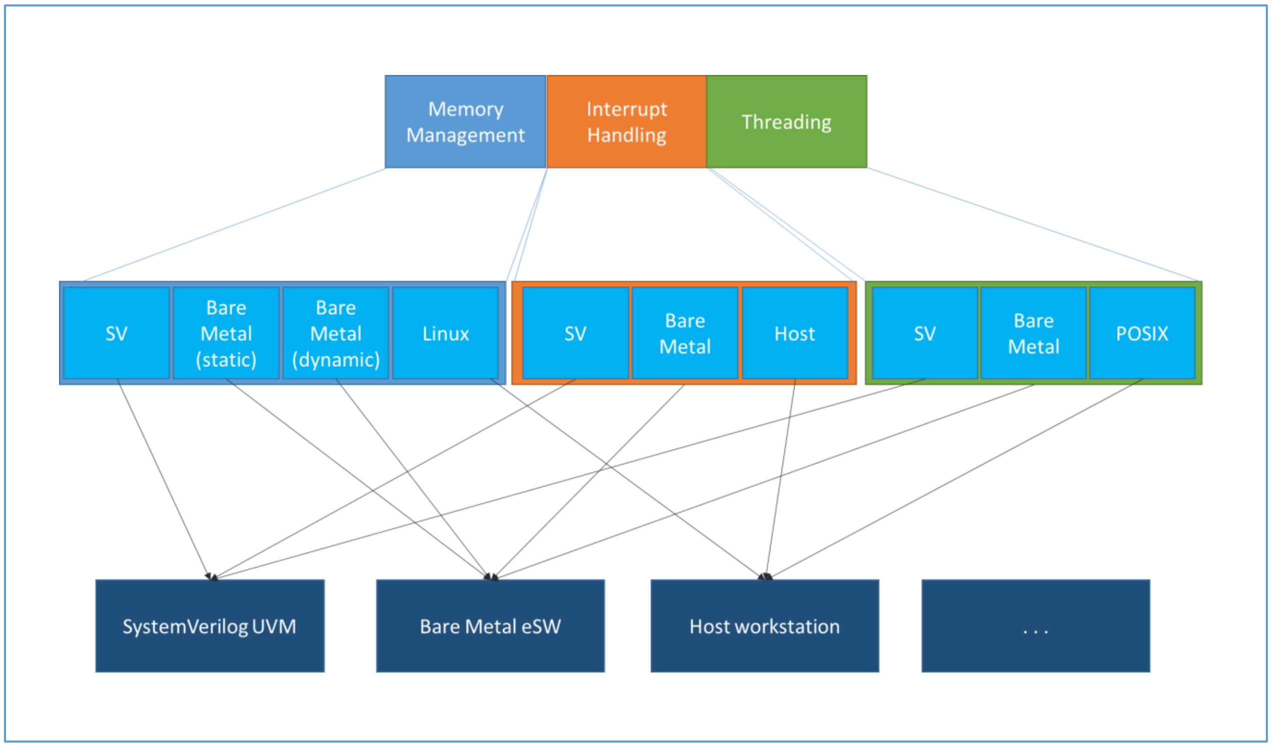

Figure 5 - UEX Block Diagram

Figure 5 shows a block diagram of the UEX API and implementations. The goals and requirements for UEX are deliberately kept simple, since UEX must support a wide variety of platforms with a range of capabilities and restrictions. The UEX API specifies a set of APIs that provides services for accessing and managing device memory, abstracting interrupts, and working with threads. Multiple environment-specific implementations exist for each service category. For example, SystemVerilog DPI-based implementations of the memory management, interrupt, and threading APIs. These implementations are mixed together to provide a complete platform-specific set of services.

UEX SERVICES

Memory Management

Firmware that interacts with hardware IP tends to require support for memory management in three areas: accessing memory-mapped registers, managing device memory, and managing memory used within the firmware.

Of these three areas, access to memory-mapped registers is the most important. UEX requires firmware to explicitly map I/O regions they wish to access. In early testing of firmware, this is beneficial in detecting firmware that attempts to access illegal regions. This model is also compatible with the requirements of a full OS, such as Linux. The I/O map and unmap functions are shown in Figure 6.

```
/**
 * uex_ioremap()
 *
 * Creates a mapping for an I/O address range. In the simplest case,
 * this will be a NOP and returns the address
 */
void *uex_ioremap(void *p, uint32_t sz, uint32_t flags);

/**
 * uex_iounmap()
 *
 * Unmaps a previous I/O address mapping. In the simplest case,
 * this will be a NOP
 */
void uex_iounmap(void *p);
```

Figure 6 - I/O Map and Unmap functions

Once an I/O region has been mapped using the uex_ioremap function, it is accessed using a set of accessor functions which are modeled on the accessor functions provided by the Linux device driver framework [5].

```
void uex_iowrite8(uint8_t v, void *p);

void uex_iowrite16(uint16_t v, void *p);

void uex_iowrite32(uint32_t v, void *p);

void uex_iowrite64(uint64_t v, void *p);

uint8_t uex_ioread8(void *p);

uint16_t uex_ioread16(void *p);

uint32_t uex_ioread32(void *p);

uint64_t uex_ioread64(void *p);
```

Figure 7 - I/O read/write functions

UEX also provides methods for bulk data transfer, shown in Figure 8. uex_iomemset sets the specified portion of I/O memory to a specific value. uex_memcpy_toio copies data from application memory to I/O memory, while uex_memcpy_fromio does the reverse.

```
void uex_iomemset(void *p, uint8_t v, uint32_t count);

void uex_memcpy_toio(void *d, void *s, uint32_t count);

void uex_memcpy_fromio(void *d, void *s, uint32_t count);
```

Figure 8 - I/O bulk operation functions

Design Memory Allocation

Device-control firmware sometimes needs to allocate memory that is accessible to the device. UEX provides the uex_ioalloc and uex_iofree methods to allocate and free this type of memory, as shown in Figure 9. In the case of a DMA engine, I/O memory might be used to store the linked list of DMA descriptors. Memory allocated by uex_ioalloc must be accessed using the I/O access functions described above.

```
void *uex_ioalloc(
             uint32_t     sz,
             uint32_t     align,
             uint32_t     flags);

void uex_iofree(void *p);
```

Figure 9 - I/O Memory Allocation

Local Memory Allocation

Sometimes firmware needs to allocate local memory that is used during firmware execution. This memory is distinct from design memory, since it is not necessarily accessible to the device. One way to view the distinction between design and local memory is that, in an IP-level environment, local memory is accessed only by the test code (eg UVM) while design memory is accessed by both the test code and the design.

UEX provides the uex_malloc and uex_free methods to manage local memory, as shown in Figure 10.

```
void *uex_malloc(uint32_t sz);


void uex_free(void *p);
```

Figure 10 - Local Memory Allocation

Threading

Firmware often performs time-consuming operations, so it is important to be able describe the operation of multiple threads in device-access firmware. UEX provides a simple API to dynamically create and manage threads, as shown in Figure 11.

```
uex_thread_t uex_thread_create(
             uex_thread_f       main_f,
             void              *ud);

int uex_thread_join(uex_thread_t t);

void uex_yield(void);

uex_thread_t uex_thread_self(void);
```

Figure 11 - UEX Thread Creation and Management API

UEX also provides APIs for initializing and interacting with mutex and condition objects, as shown in Figure 12.

```
void uex_mutex_init(uex_mutex_t *m);

void uex_mutex_lock(uex_mutex_t *m);

void uex_mutex_unlock(uex_mutex_t *m);

void uex_cond_init(uex_cond_t *c);

void uex_cond_wait(uex_cond_t *c, uex_mutex_t *m);

void uex_cond_signal(uex_cond_t *c);
```

Figure 12 - Thread Mutex and Condition API

It is important to note in both cases that the core UEX API does not define the thread, mutex, or condition data types. These are defined by the specific implementation layers, providing an implementation layer the freedom to, for example, represent a thread with an integer ID or represent a thread using a dynamically-allocated instance of a class.

Different threading models are more appropriate at different points in the verification cycle. During simulation-based verification at subsystem level, using cooperative SystemVerilog threads is most appropriate. When first running bare-metal embedded software in simulation, cooperative multithreading is most appropriate because it has lower overhead than full preemptive multithreading. It's also simpler to debug. By the time a more-complex software stack is running in emulation or on a prototype, preemptive multithreading makes the most sense because it allows for more-complex interactions between threads.

To enable the same firmware to support both threading models, code written against the UEX thread API must be written assuming that the thread implementation could be either cooperative or preemptive. Busy-wait must be implemented by periodically calling uex_yield(). Access to data shared by multiple threads must be protected by a mutex.

Interrupts

Firmware often needs to interact with interrupts as it interacts with the device it controls. UEX provides a very simple API for interacting with interrupts, as shown in Figure 13, that allows a device to register a handler function for an interrupt that is identified by an integer ID.

```
typedef void (*uex_irq_f)(void *);


void uex_set_irq_handler(
             uint32_t           irq,
             uex_irq_f          f,
             void               *ud);
```

Figure 13 - UEX Interrupt API

WRITING DMA FIRMWARE WITH UEX

Writing our DMA-control firmware using the UEX API allows us to begin testing the same firmware at IP level that can be reused at SoC level. The complexity of writing firmware against the UEX API is certainly no greater than writing any other embedded software, and is often simpler because of the available built-in services.

```
void wb_dma_uex_drv_init(
             wb_dma_uex_drv_t          *drv,
             void                      *base) {
      uint32_t i;
      uex_device_t *dev;

      memset(drv, 0, sizeof(wb_dma_drv_t));
      drv->regs = (wb_dma_regs_t *)uex_ioremap(base, (5*4)+(8*4), 0);

      // Route enabled channels to INTA
      uex_iowrite32(&drv->regs->int_msk_a, 0xFFFFFFFF);
      uex_iowrite32(&drv->regs->int_src_a, 0xFFFFFFFF);

      for (i=0; i<31; i++) {
             // Enable for 256-byte chunks
             uex_iowrite32(&drv->regs->channels[i].size, ((256/4) << 16));

             // Configure address mask
             uex_iowrite32(&drv->regs->channels[i].src_msk, 0xFFFFFFFC);
             uex_iowrite32(&drv->regs->channels[i].dst_msk, 0xFFFFFFFC);
      }

      // Initialize thread-control fields
      for (i=0; i<31; i++) {
             uex_mutex_init(&drv->xfer_mutex[i]);
             uex_cond_init(&drv->xfer_cond[i]);
      }

      // Find the device and connect the interrupt based on the base address
      dev = uex_find_device(base);
      uex_set_irq_handler(dev->irqs[0], &wb_dma_uex_irq, drv);
}
```

Figure 14 - UEX DMA Firmware Initialization

Figure 14 shows the initialization code for the UEX DMA firmware. The uex_ioremap function is used to map the I/O address of the DMA registers. Then, uex_iowrite functions are used to initialize the DMA registers. Per-channel mutex and condition objects are initialized. Finally, an interrupt handler is registered using the uex_set_irq_handler function.

```
void wb_dma_uex_drv_single_xfer(
             wb_dma_uex_drv_t          *drv,
             uint32_t                        channel,
             uint32_t                        src,
             uint32_t                        inc_src,
             uint32_t                        dest,
             uint32_t                        inc_dst,
             uint32_t                        sz) {
      uint32_t csr, sz_v;
      uex_mutex_lock(&drv->xfer_mutex[channel]);

      // Program channel registers
      csr = uex_ioread32(&drv->regs->channels[channel].csr);

      csr |= (1 << 18); // interrupt on done
      csr |= (1 << 17); // interrupt on error
      csr =  (inc_src)?(csr | (1 << 4)):(csr & ~(1 << 4));
      csr =  (inc_dst)?(csr | (1 << 3)):(csr & ~(1 << 3));

      csr |= (1 << 2); // use interface 0 for source
      csr |= (1 << 1); // use interface 1 for destination

      csr |= (1 << 0); // enable channel

      // Setup source and destination addresses
      uex_iowrite32(&drv->regs->channels[channel].src, src);
      uex_iowrite32(&drv->regs->channels[channel].dst, dest);

      sz_v = uex_ioread32(&drv->regs->channels[channel].size);
      sz_v &= ~(0xFFF); // Clear tot_sz
      sz_v |= (sz & 0xFFF);
      uex_iowrite32(&drv->regs->channels[channel].size, sz_v);

      // Start the transfer
      uex_iowrite32(&drv->regs->channels[channel].csr, csr);

      drv->status[channel] = 1;

      // Wait for completion
      uex_cond_wait(&drv->xfer_cond[channel], &drv->xfer_mutex[channel]);

      uex_mutex_unlock(&drv->xfer_mutex[channel]);
}
```

Figure 15 - DMA Transfer

Figure 15 shows a DMA transfer implemented using the UEX API. Aside from the normal calls to program registers, note that each call to wb_dma_uex_drv_single_xfer locks the requested channel on entry. The tail of the function then waits for the transfer-done condition.

```
static void wb_dma_uex_irq(void *ud) {
        wb_dma_uex_drv_t *drv = (wb_dma_uex_drv_t *)ud;
        uint32_t i;

        // Need to spin through the channels to determine
        // which channel to activate
        for (i=0; i<31; i++) {
                uex_mutex_lock(&drv->xfer_mutex[i]);
                if (drv->status[i] == 1) {
                        // Check this channel's CSR
                        uint32_t csr = uex_ioread32(&drv->regs->channels[i].csr);

                        if ((csr & 1) == 0) {
                                // Transfer is done
                                drv->status[i] = 0;
                                uex_cond_signal(&drv->xfer_cond[i]);
                        }
                }
                uex_mutex_unlock(&drv->xfer_mutex[i]);
        }
}
```

Figure 16 - DMA Interrupt Service Routine

Figure 16 shows the code for the DMA firmware interrupt service routine. For each DMA channel, the ISR checks whether the channel is active (status[i] == 1), then reads the DMA channel control register to determine whether the transfer is complete. If so, the xfer_cond mutex is signaled, which unblocks the uex_cond_wait call in the wb_dma_uex_drv_single_xfer function.

SUMMARY

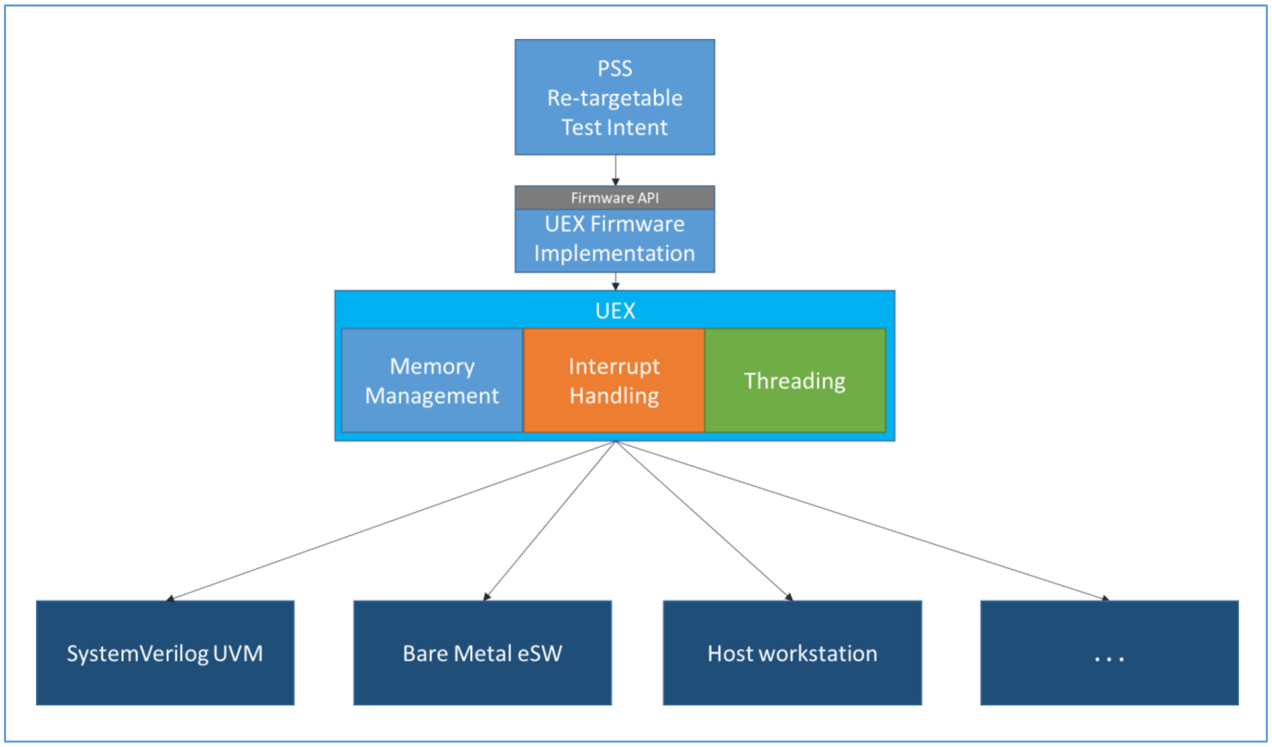

Figure 17 - Reusable Test Intent and Firmware with UEX

Developing device firmware as the IP is developed and verified helps to discover hardware/software interface bugs as early as possible in the development process, and ensures the existence of reusable firmware for subsystem and SoC-level verification. Portable stimulus techniques and tools provide a way to execute the same high-level test intent across IP, subsystem, and SoC environments. However, portable stimulus users are typically responsible for implementing the "connection" code that interfaces high-level intent to the specifics of the target environment. This gives significant flexibility in connecting portable stimulus to an enormous range of environments, but also requires the user to maintain multiple environment-specific implementations of the firmware logic.

The UEX API presented in this paper specifies an API used by device firmware to interact with hardware and with other modules of device firmware, as well as implementations of that API that are appropriate for simulation-based UVM, bare-metal embedded software, host workstation, as well as other environments. This allows the same firmware code to be reused across these platforms without modification (Figure 17), eliminating mismatches in behavior between different implementations and saving the effort required to re-implement firmware.

Together, portable stimulus reusable test intent and a software-centric portability API bring significant benefits to early development and co-verification of device firmware.

REFERENCES

[1] Wishbone DMA Core, https://opencores.org/project,wb_dma
[2] Accellera Portable Stimulus Working Group, https://workspace.accellera.org/apps/org/workgroup/pswg/
[3] Questa inFact, https://www.mentor.com/products/fv/infact/
[4] UEX, https://github.com/mballance/uex
[5] A. Pugalia, *Device Drivers, part 7: Generic Hardware Access*, http://opensourceforu.com/2011/06/generic-hardware-access-in-linux/