Essential Topics

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

This chapter provides information about various topics which are relevant to everyone who is writing or debugging applications to run on OCTEON processors:

- Physical Addresses versus Virtual Addresses
- Free Pool Allocator (FPA) Buffers and Buffer Pools
- Scratchpad Areas
- Fetch and Add Unit
- Definitions: SE Configuration, Unit Initialization, Unit Enable, Pool Population
- Write Buffers
- Synchronization Instructions
- How to Measure the Cycles Used by a Section of Code
- Common Software Issues

This information is especially important for readers new to multicore programming or new to OCTEON processors. For example, a common problem is writing data to shared memory from one core and attempting to read it from another core, only to find that the data read is not the expected value which was written. This chapter provides details on the synchronization instructions which are necessary to ensure shared data is visible to other cores when required.

This chapter also contains an example of using the SDK API functions to time how long it takes to execute a section of code, which is extremely useful in debugging performance issues (see Section 9 – "How to Measure the Cycles Used by a Section of Code").

Advanced information for readers who are adding code to the Simple Executive API, writing a custom API, or reading the SDK code is provided in the *Advanced Topics* chapter. This chapter contains information about register access, race conditions to avoid, etc. Readers who are simply using the provided APIs do not need to read the *Advanced Topics* chapter.

Directions on how to configuration resources (such as the FPA buffer pools, scratchpad areas, and FAU registers) are provided in the *Configuration* chapter.

1.1 Conventions Used In this Chapter

Note that in most cases the format `REGISTER[FIELD]` in this chapter refers to a hardware register and field combination, not a software `ARRAY[INDEX]`.

2 Physical Addresses versus Virtual Addresses (Pointers)

At the hardware level, transactions requiring addresses use physical addresses. For example, the FPA unit provides "allocate" and "free" operations (discussed in Section 3 – "Buffers and Buffer Pools"). These operations use the physical address of the buffer in DRAM, not a virtual address. The FPA is a hardware unit: it has no concept of the TLB or of virtual address space.

When using the API functions such as `cvmx_fpa_alloc()` and `cvmx_fpa_free()`, many of the functions will handle address translation between Virtual and Physical addresses, so application functions calling the API often use virtual addresses.

It is essential to stay alert to which type of address is being used when using the API. The following API functions are provided to convert between the two types of addresses as needed:

- `cvmx_ptr_to_phys()` – convert virtual address to physical address
- `cvmx_phys_to_ptr()` – convert physical address to virtual address

See the *Software Overview* chapter for a discussion of physical versus virtual addresses.

3 Buffers and Buffer Pools

Packet input and output accelerators (IPD, SSO, cores, and PKO) within the OCTEON processor require three different types of FPA-managed buffers for various purposes as the packet transits OCTEON from ingress to egress port:

- Packet Data Buffers
- Work Queue Entry (WQE) Buffers (this type of buffer is optional on some OCTEON models)
- PKO Command Buffers

In addition to these packet IO (and core processing of the packet in transit) related uses, the TIMER, DFA/HFA, ZIP, RAID units, and the PCI DMA Engines also use FPA-managed buffers as part of their normal operation. Users may also configure custom FPA-managed buffers for other uses (for example, pure software core-to-core messaging).

All FPA-managed buffers must be multiples of 128 bytes (the cache line size), and must be aligned on the 128-byte boundary or else errors which are difficult to debug may result. When using the Simple Executive configuration program and functions, these requirements are automatically met. This is only an issue when writing custom software.

The Free Pool Allocator (FPA) manages pools of available buffers. The cores and some of the hardware units as described above may request/allocate the address of an available buffer from the FPA via the `buffer_allocate` operation. The FPA will return the memory address of an available buffer to the caller. When the buffer is no longer needed, the cores and some of the hardware units can return/free buffer addresses back to a specified FPA pool via the `buffer_free` operation. There is both a synchronous (software blocks waiting for operation to complete) and asynchronous (non-blocking) version of the `buffer allocate` operation.

> ***Note: The asynchronous `buffer_allocate` operation allows the user to request multiple FPA-managed buffers, but this is not generally a good idea: if 8 buffers are requested, and only 6 buffers are available, the FPA will return 0 buffers.***

The FPA manages up to 8 pools of buffer addresses. Typically, each pool is dedicated to a different purpose. Dedicating an FPA pool to manage buffers of a single type reduces programming errors and debugging complexity. (Note that the PKO Command buffers are usually shared by various hardware units, but all of these have the common purpose “Instruction/Command buffer”. Due to hardware requirements, Packet Data Buffers always come from FPA pool 0.

The buffer size and FPA pool number used for a given buffer type are set at Simple Executive configuration time. The default configuration of the three FPA pools essential to packet IO is shown in the following table:

Table 1: Three Essential Buffer Pools

Purpose	Pool Name	Pool Number	Buffer Size
Packet Data Buffers	`CVMX_FPA_PACKET_POOL`	0	2048 bytes (16 cache lines)
Work Queue Entry Buffers	`CVMX_FPA_WQE_POOL`	1	128 bytes (1 cache line)
PKO Queue Command Buffers	`CVMX_FPA_OUTPUT_BUFFER_POOL`	2	1024 bytes (8 cache lines)
Notes			
Note1: `CVMX_FPA_PACKET_POOL` is required by hardware to be pool 0. Note2: Each buffer is a multiple of cache line size (`CVMX_CACHE_LINE_SIZE`) which is 128 bytes. Buffers must be 128-byte aligned.			

During the Simple Executive configuration step, macros are created for the buffer size and buffer pool number, but the number of FPA-managed buffers in the pool is not set until runtime. For example, after the configuration step, the Work Queue Entry buffer size and pool number are defined:

```
// Work Queue Entry buffer size in bytes, a multiple
// of CVMX_CACHE_LINE_SIZE
#define CVMX_FPA_POOL_1_SIZE (1 * CVMX_CACHE_LINE_SIZE)
// pool number
#define CVMX_FPA_WQE_POOL              (1) /**< Work queue entrys */
#define CVMX_FPA_WQE_POOL_SIZE         CVMX_FPA_POOL_1_SIZE
```

(Note the extra characters "*<" seen in the "/**< Work queue entrys */" line, and the misspelling of "entries" is because this line is generated by the configuration utility from data in the `cvmx-resources.config` file. See the Configuration chapter for more information.)

Software can then use these macros, as seen in the `passthrough` example:

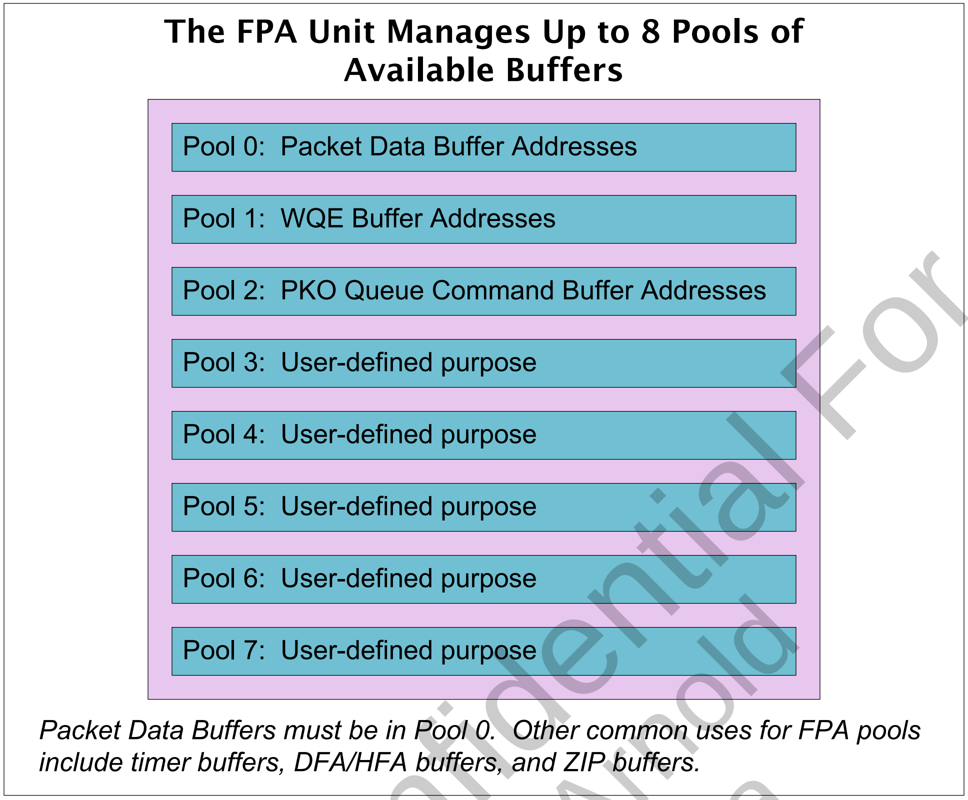
```
cvmx_fpa_free(work, CVMX_FPA_WQE_POOL, 0);
```

At runtime, one (and only one) core populates the pool by allocating memory, dividing it into buffers, and freeing the buffers to the specified pool. When using the Simple Executive function `cvmx_helper_initialize_fpa()` , the user simply specifies the desired count of each type of buffer:

```
int cvmx_helper_initialize_fpa(int packet_buffers,
                               int work_queue_entries,
                               int pko_buffers, int tim_buffers,
                               int dfa_buffers)
```

For each type of buffer, the function will allocate a chunk of 128-byte aligned memory, divide it into buffers, and free the buffers to the appropriate pool.

Figure 1: FPA Supports Eight Buffer Pools


The FPA Unit Manages Up to 8 Pools of Available Buffers
Pool 0: Packet Data Buffer Addresses
Pool 1: WQE Buffer Addresses
Pool 2: PKO Queue Command Buffer Addresses
Pool 3: User-defined purpose
Pool 4: User-defined purpose
Pool 5: User-defined purpose
Pool 6: User-defined purpose
Pool 7: User-defined purpose

Packet Data Buffers must be in Pool 0. Other common uses for FPA pools include timer buffers, DFA/HFA buffers, and ZIP buffers.

As mentioned in Section 2 – "Physical Addresses versus Virtual Addresses (Pointers)", the `buffer_allocate` and `buffer_free` operations performed by the FPA always use the physical addresses of buffers, and many API functions such as the `cvmx_fpa_alloc()` and `cvmx_fpa_free()` use virtual addresses.

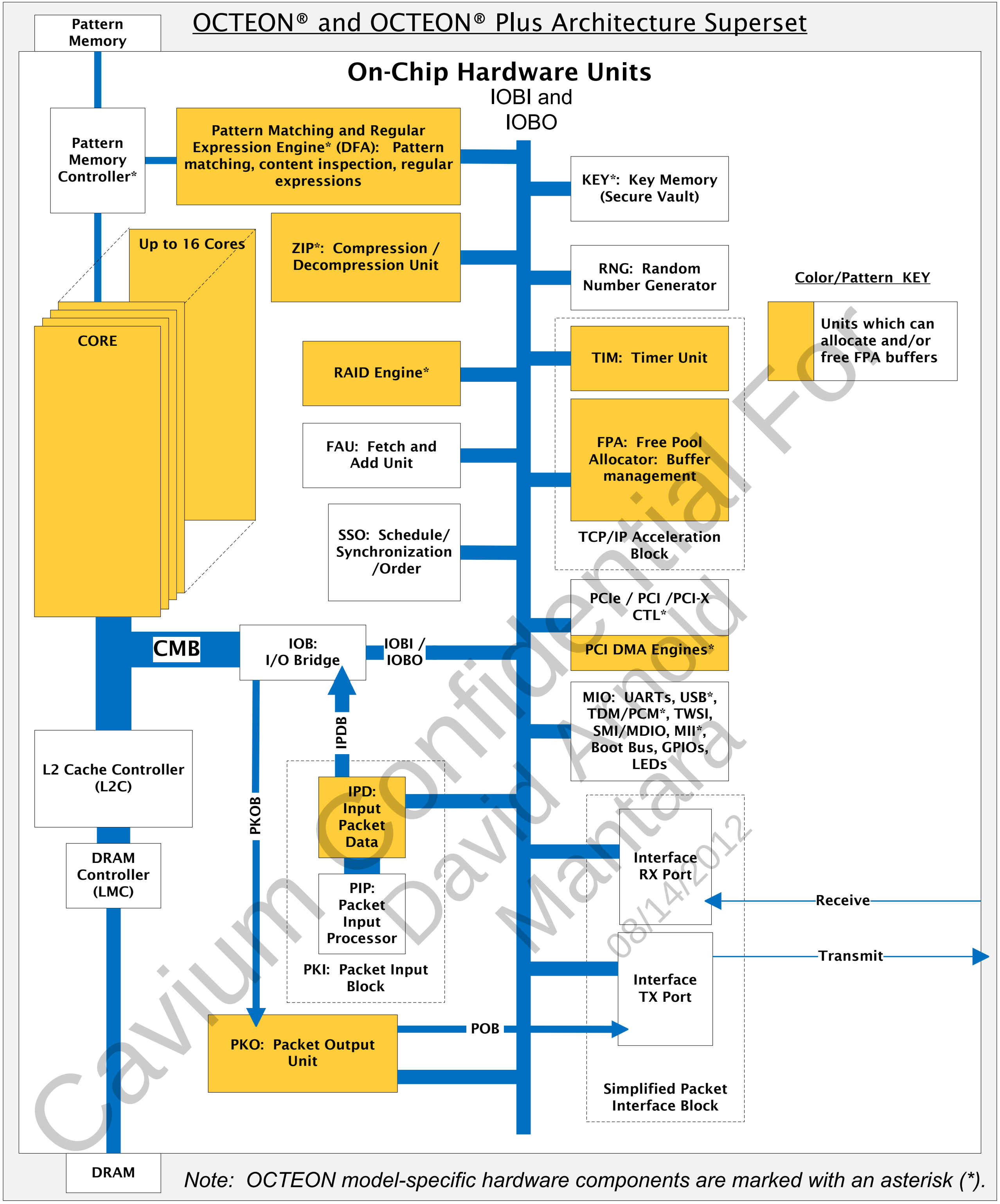
In addition to the synchronous `buffer_allocate` operation, the FPA provides an asynchronous `buffer allocation` (prefetch) operation. The SDK function `cvmx_fpa_async_alloc()` performs this operation. When using asynchronous buffer allocation, software continues with other processing while the operation is performed by the FPA asynchronously. The FPA unit stores the physical address of the requested buffer in a specified scratchpad area where software can later retrieve it. After software retrieves the physical address, it must use `cvmx_phys_to_ptr()` to convert it to a virtual address. Asynchronous buffer allocation is discussed in more detail in Section 4 – "Scratchpad Memory".

> ***Note that after the application uses the buffer, it is essential to free the buffer to the appropriate FPA pool, and to only free the buffer once. Also note that a write beyond the end of the buffer will corrupt adjacent memory. Please see the FPA chapter for details of these and other common errors. These types of errors can be difficult to debug because***

when memory is corrupted, the symptoms of memory corruption can occur a long time after the erroneous write that resulted in the actual corruption.

As shown in the following figure, many different units can use the FPA pools: they are an essential component of OCTEON applications.

Figure 2: Units Which Use the FPA Pools


OCTEON® and OCTEON® Plus Architecture Superset
On-Chip Hardware Units
IOBI and IOBO
Pattern Memory
Pattern Memory Controller*
Pattern Matching and Regular Expression Engine* (DFA): Pattern matching, content inspection, regular expressions
KEY*: Key Memory (Secure Vault)
Up to 16 Cores
ZIP*: Compression / Decompression Unit
RNG: Random Number Generator
Color/Pattern KEY
Units which can allocate and/or free FPA buffers
CORE
TIM: Timer Unit
RAID Engine*
FAU: Fetch and Add Unit
FPA: Free Pool Allocator: Buffer management
SSO: Schedule/ Synchronization /Order
TCP/IP Acceleration Block
PCIe / PCI /PCI-X CTL*
CMB
IOB: I/O Bridge
IOBI / IOBO
PCI DMA Engines*
MIO: UARTs, USB*, TDM/PCM*, TWSI, SMI/MDIO, MII*, Boot Bus, GPIOs, LEDs
IPDB
L2 Cache Controller (L2C)
PKOB
IPD: Input Packet Data
DRAM Controller (LMC)
PIP: Packet Input Processor
Interface RX Port
Receive
Transmit
PKI: Packet Input Block
Interface TX Port
PKO: Packet Output Unit
POB
Simplified Packet Interface Block
DRAM

Note: OCTEON model-specific hardware components are marked with an asterisk (*).

4 Scratchpad Memory (CVMSEG) and IOBDMA Operations

The OCTEON processor-specific CVMSEG virtual address segment, *scratchpad* memory, and IOBDMA operations were introduced in the *Software Overview* chapter.

The scratchpad is a core-local memory, accessible to software via CVMSEG LM virtual addresses. Some of the hardware units (FPA, SSO, RNG, FAU, RAID, etc.) can DMA directly to the core's scratchpad. This DMA operation is called an *IOBDMA* operation because the DMA is from hardware units on the I/O Bus. IOBDMA operations are initiated by the core.

The scratchpad can also be used for software-specific purposes. In this case, it is not configured to be the target of an IOBDMA operation. (Note: Software uses for scratchpad are rare. If a cache line is frequently accessed, it is already in Dcache, limiting the value gained from using the scratchpad.)

When running Linux, scratchpad memory is saved and restored on each context switch.

IOBDMA operations are asynchronous; compared to a normal IO read which stalls the core for some time until the CSR targeted by the read has supplied the requested data, an IOBDMA allows a core to continue with program execution, while in parallel the accelerator that was targeted by the IOBDMA writes the requested data directly into the specified location of the initiating core's scratchpad memory.

In pseudo code (which contains fragments of real code, but is in no way complete):

```
start IOBDMA operation; // do not wait for result
do other processing;    // during this time the target hardware unit DMAs
                        // the data to the scratchpad area
CVMX_SYNCIOBDMA;        // use this macro to issue the synciobdma instruction.
                        // This will stall the core until the IOBDMA operation
                        // has completed.
retrieve the result of the IOBDMA operation from the scratchpad area
```

(The CVMX_SYNCIOBDMA macro is discussed in Section 8 – "Synchronization Instruction (sync) Variations".)

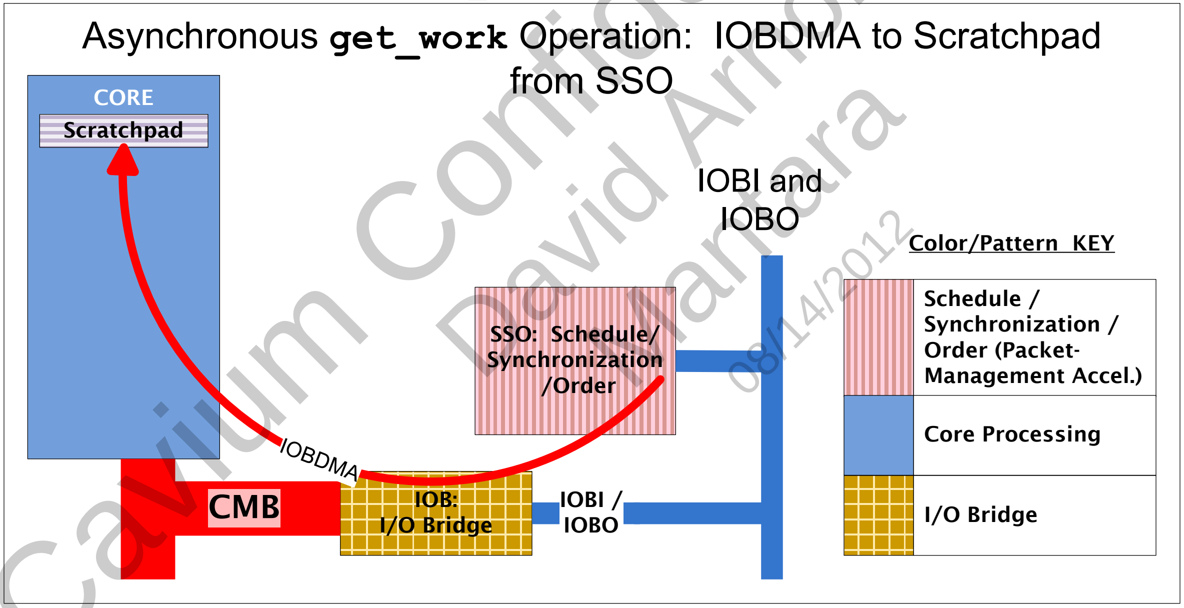
These operations are commonly called *prefetch* operations, because data is requested before it is actually needed. On successful completion of an IOBDMA operation, the requested data is stored in the specified location in the scratchpad. Before retrieving the data, the core must issue a synciobdma instruction to ensure the IOBDMA operation has completed. (The alternative to using synciobdma instruction is to write a known-bad value to the scratchpad and poll until the value changes. This option is shown in Section 9.2 – "Asynchronous Operation Timed".) Variations on the MIPS sync instruction supported by OCTEON processors are shown in section 8 – "Synchronization Instruction (sync) Variations".

IOBDMA operations which may be initiated by the core include:

- FPA Buffer Prefetch: Software may prefetch FPA buffer pointers (the asynchronous `buffer allocate` operation). The API function is `cvmx_fpa_async_alloc()`.
- SSO Work Prefetch: Software may prefetch work from the SSO (POW). The API function is `cvmx_pow_work_request_async()`.
- Random Number Prefetch: Software can prefetch a random number from the Random Number Generator (RNG). The API function is `cvmx_rng_request_random_async()`. (The Random Number Generator hardware unit is sometimes referred to as Random Number Memory (RNM)). Up to 128 bytes of random data can be requested with one IOBDMA command.
- Asynchronous FAU Register Add: The Fetch and Add Unit (FAU) supports asynchronous requests from the core to add a value to a FAU register. On request, the FAU will delay the operation until after the prior tag switch completes (the SSO notifies the FAU when the tag switch completes). If required, the FAU DMAs the new value of the register to the scratchpad when the operation completes.

Note: Sometimes the extra code needed for IOBDMA is not worth the savings, particularly in the case of buffer prefetch, which is not very expensive. For high-performance code, timing studies with and without IOBDMA might be needed to determine which is best for the specific application. See Section 9 – "How to Measure the Cycles Used by a Section of Code".

Figure 3: Asynchronous **`get_work`** Operation: SSO Writes to Scratchpad


Asynchronous get_work Operation: IOBDMA to Scratchpad from SSO
CORE
Scratchpad
IOBI and IOBO
Color/Pattern KEY
SSO: Schedule/ Synchronization /Order
IOBDMA
CMB
IOB: I/O Bridge
IOBI / IOBO
Schedule / Synchronization / Order (Packet-Management Accel.)
Core Processing
I/O Bridge

4.1 Scratchpad Memory Allocation

Scratchpad memory is allocated from the per-core L1 data cache (Dcache). Up to 54 blocks of L1 Dcache can be configured as scratchpad memory instead of normal L1 Dcache. Each cache block is 128 bytes (matching cache line size). By default, the SDK configures 4 Dcache lines for the scratchpad.

> ***Note that since space for the scratchpad comes from Dcache, keeping the size of the scratchpad to a minimum will help maximize system performance for applications that access large amounts of data by leaving more Dcache blocks available for the application stack & local variables.***
>
> ***Warning: If an illegal address is provided in an IOBDMA instruction, or the requested number of bytes will exceed the allocated cache lines for the scratchpad, but is within the range of `CVMSEG LM` in the virtual address map, then the adjacent Dcache memory may be overwritten. (An address error will occur, but stores to these illegal addresses may not be stopped by the hardware, so they may corrupt the Dcache.)***

For directions on how to change the default scratchpad size, see the *Advanced Topics* chapter.

4.2 Naming Scratchpad Areas

The scratchpad can be divided into areas designated for different purposes. Scratchpad areas are specified in the Simple Executive configuration files: name, element size, number of elements, and whether it will be used for IOBDMA operations. The Simple Executive configuration utility (discussed in the *Configuration* chapter) will allocate the requested scratchpad area (given the number of bytes in the scratchpad and whether the scratchpad area will be the target of an IOBDMA operation). The configuration utility will create a macro equating the scratchpad name to the offset of the named scratchpad area relative to the beginning of the entire scratchpad.

For example, the default configuration provides a generic scratchpad: 8 bytes, the target of an IOBDMA. This scratchpad is named `CVMX_SCR_SCRATCH`. In this example, the offset of `CMVX_SCR_SCRATCH` is 0 bytes (the configuration tool will create the offsets, locating the first scratchpad area at offset 0 relative to the start of the scratchpad area).

```
#define CVMX_SCR_SCRATCH          (0) /* Generic scratch IOBDMA area          */
```

API functions which cause the core to issue IOBDMA instructions (such as `cvmx_fpa_async_alloc()`) use the scratchpad area's name to calculate the physical address of the scratchpad. This physical address is included in the IOBDMA instruction which is issued to the target hardware unit. API functions which retrieve the result of the IOBDMA operation from the scratchpad area (`cvmx_scratch_read*()`) use the scratchpad area's name to calculate the virtual address to be read.

4.3 Reading Scratchpad Memory

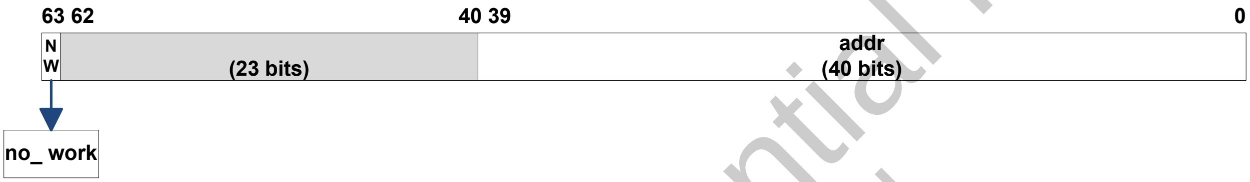
Scratchpad memory may be read using the Simple Executive functions `cvmx_scratch_read8()`, `cvmx_scratch_read16()`, `cvmx_scratch_read32()`, or `cvmx_scratch_read64()`. The 8, 16, 32, and 64 in the function names represent the number of bits to read from the specified scratchpad area.

4.4 The Format of the Data Returned by the IOBDMA Operation

For each IOBDMA operation, the format of the data returned depends on the specific operation performed. In the case of the IOBDMA used to prefetch work from the SSO (an asynchronous `get_work` operation), the data returned has the following format:

Figure 4: Data Stored in Scratchpad when **get_work** IOBDMA Completes

The Result of an Asynchronous **get_work** Operation



This result is written to the specified scratchpad are when the IOBDMA operation completes. Reserved areas are gray and are set to all zeroes.

no_work: set when no new WQE address was returned
addr:
    Case 1: If **no_work==0**, **addr** is the physical address of the WQE buffer returned by the SSO.
    Case 2: If **no_work==1**, **addr** is unpredictable.

Note: The **no_work** bit is set if there the SSO scheduler did not find work for this core. In this case work for *other* cores could be available. For instance, if there is work for group 2, but this core *only* processes work for group 1, if there is no group 1 work available, then there is no work for this core, and the **no_work** bit will be set in the result.

The corresponding software data structure is `cvmx_pow_tag_load_resp_t` (defined in `executive/cvmx-pow.h`). The SSO (POW) unit supports multiple IOBDMA operations, so `cvmx_pow_tag_load_resp_t` is a union containing the different data structures corresponding to the different IOBDMA operations. In the case of the `get_work` IOBDMA operation, this data structure is:

```
struct {
        uint64_t    no_work        :  1; // Set when no new WQE buffer
                                         // address was returned
        uint64_t    reserved_40_62 : 23; // Must be zero
        uint64_t    addr           : 40; // The Work Queue Entry
                                         // buffer physical address
} s_work;
```

4.5 Example IOBDMA Operation: Prefetch Work from the SSO

The scratchpad is commonly used to prefetch work (a pointer to a Work Queue Entry (WQE) buffer) from the SSO. The following pseudo code is taken from the `traffic-gen` example. (Note that the WQE data structure is discussed in more detail in the *PIP/IPD* chapter. This information is not needed to understand the use of IOBDMA to prefetch the WQE buffer.)

In the following pseudo code, confusion can occur because the term *prefetch* is used in two different ways:

1. The physical address of a WQE is prefetched from the SSO (POW) using an IOBDMA operation.
2. The MIPS `pref` instruction is used to prefetch the WQE buffer contents into the cache.

The following pseudo code is not the complete `packet_transmitter()` function: the code was simplified to highlight the use of prefetch. The format of the data returned by the IOBDMA operation is shown in Figure 4 – "Data Stored in Scratchpad when `get_work` IOBDMA Completes".

Note that the Work Queue Entry data type, `cvmx_wqe_t`, is discussed in detail in the *PIP/IPD* Chapter.

```
static void packet_transmitter(int low_port)
{
    cvmx_wqe_t *work = NULL;  // the WQE buffer pointer, initialized to NULL

    // write a 0 to the scratchpad area.  When the IOBDMA successfully
    // completes, a non-zero value is written to the scratchpad area, so
    // if code reads a zero, the IOBDMA operation is not complete
    // TRAFFICGEN_SCR_WORK is the offset in the scratchpad where the result of
    // the IOBDMA operation will be stored
    cvmx_scratch_write64(TRAFFICGEN_SCR_WORK, 0);

    //start work prefetch IOBDMA operation
    cvmx_pow_work_request_async(TRAFFICGEN_SCR_WORK, CVMX_POW_NO_WAIT);

    while (1)  // forever transmit loop
    {
        // the first time through the loop, work will be NULL
        if (work != NULL)
        {
            process the work
            work = NULL;
        }

        // Don't use the normal get work routines to avoid a CVMX_SYNCIOBDMA
        // Instead of CVMX_SYNCIOBDMA, use polling until result != 0

        cvmx_pow_tag_load_resp_t result; // create result data structure

        // read the scratchpad to retrieve the result of the IOBDMA operation
        result.u64 = cvmx_scratch_read64(TRAFFICGEN_SCR_WORK);
        // polling for IOBDMA to complete
        if (result.u64 != 0) // if non-zero, IOBDMA operation has completed
        {
            // reinitialize scratchpad area to 0 for next IOBDMA
            cvmx_scratch_write64(TRAFFICGEN_SCR_WORK, 0);

            if (!result.s_work.no_work) // if the SSO returned work to do
            {
                // convert physical address of the WQE buffer to a
                // virtual address
                work = (cvmx_wqe_t*)cvmx_phys_to_ptr(result.s_work.addr);

                // use MIPS pref instruction to prefetch WQE contents
                // into L1 cache
                CVMX_PREFETCH(work, 0); // (address, byte offset)

            } // end if the SSO returned work to do
            // start work prefetch IOBDMA operation
            cvmx_pow_work_request_async(TRAFFICGEN_SCR_WORK,
                                                CVMX_POW_NO_WAIT);
        } // end if IOBDMA operation has completed
    }  // end while forever loop
} // end packet_transmitter()
```

Note: In the actual `traffic-gen` code, the function used to initiate the `get_work` IOBDMA operation is `cvmx_pow_work_request_async_nocheck()`. The `cvmx_pow_work_request_async_nocheck()` function is an advanced function which should not generally be used because it is essential that no tag switches are pending when the function is called. Performing the `get_work` operation while a tag switch is pending is an illegal operation, and the result is undefined. In general, users should not use any of the "`nocheck`" functions.

4.6 The Scratchpad and Linux SE-UM Applications

When accessing the scratchpad from Linux user-mode (SE-UM applications), use the same functions as for SE-S applications. The scratchpad is saved/restored on process context switch. Multiple threads in the same process share access to the scratchpad without protection. Details on how to configure the scratchpad for Linux are provided in the *Software Overview* chapter.

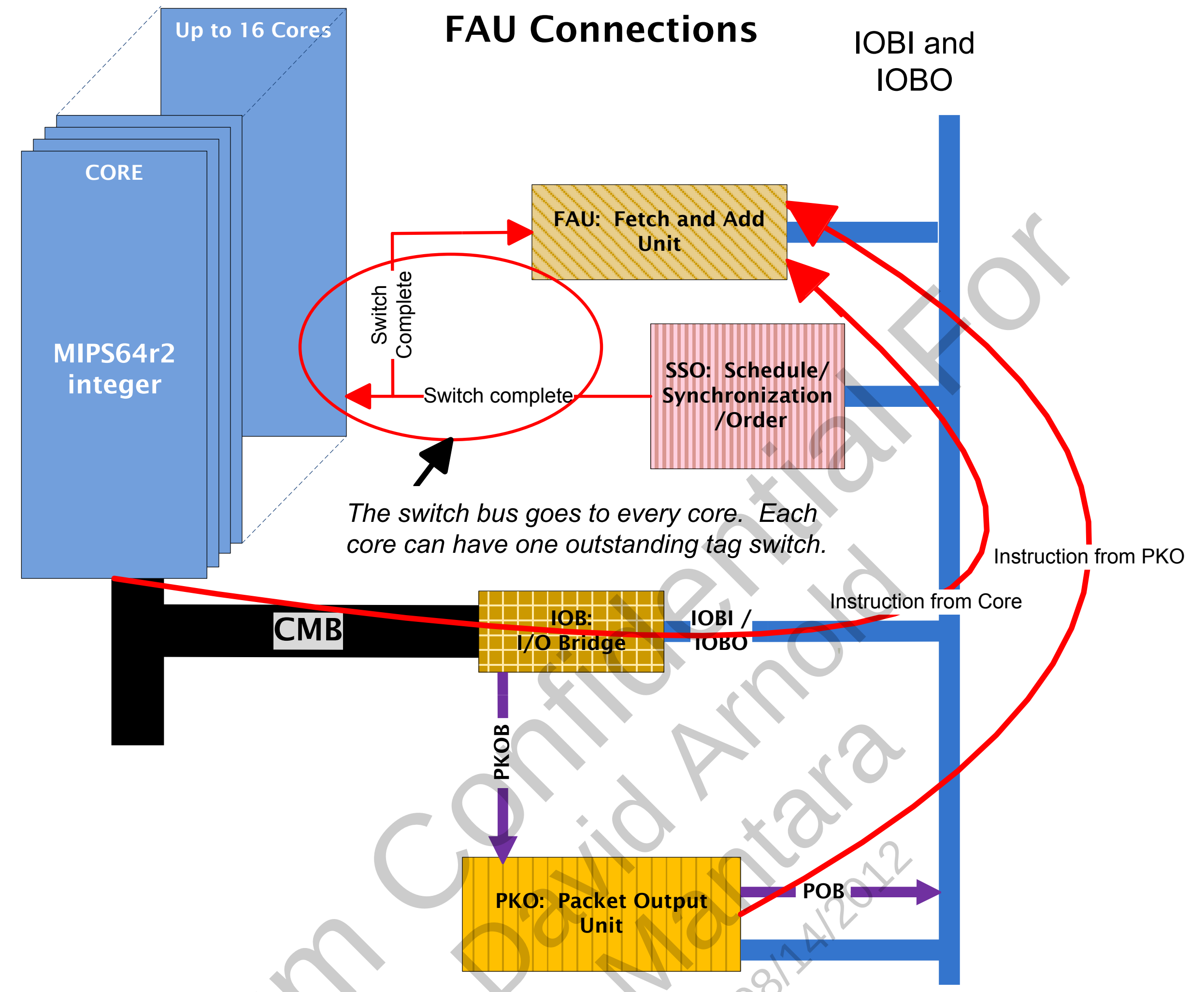
4.7 Where to Find More Information About the Scratchpad

For more information about the scratchpad and IOBDMA operations, see the *Software Overview* and *Advanced Topics* chapters.

5 Fetch and Add Unit (FAU)

The FAU is a 2 KByte register bank supporting atomic read-modify-write operations to 8, 16, 32 or 64 bit counters.

Figure 5: FAU Connections



Example Use 1: Software can monitor the length of backlog of packets awaiting transmission (performed by the PKO) using the following method:
1) Prior to adding the PKO command to the PKO Output Queue, core software increments a FAU counter.
2) The PKO command can optionally cause the PKO to decrement the same counter after the PKO command has been processed (i.e. the packet has egressed).

Example Use 2: An alternative use of FAU counters is as a shared counter/variable that is updated asynchronously without need for spinlocks by multiple cores. Use of this feature ensures that updates of the shared FAU register are performed in the order enforced by the SSO.
1) The core requests a tag switch
2) The same core sends a request to the FAU to update a counter after the tag switch completes. The tag switch complete is signaled by the SSO directly to the FAU via the switch bus.

The cores and the PKO can both send commands to the FAU. For example, PKO can issue up to two FAU operations on every PKO command, which can be used to count bytes and packets as they are transmitted. The SSO is also able to tie tag switches to FAU operations, which can be used to ensure that an atomic operation happens in synchronization with a tag switch. FAU register offsets (within the 2 KByte register bank) can be given names via Simple Executive configuration.

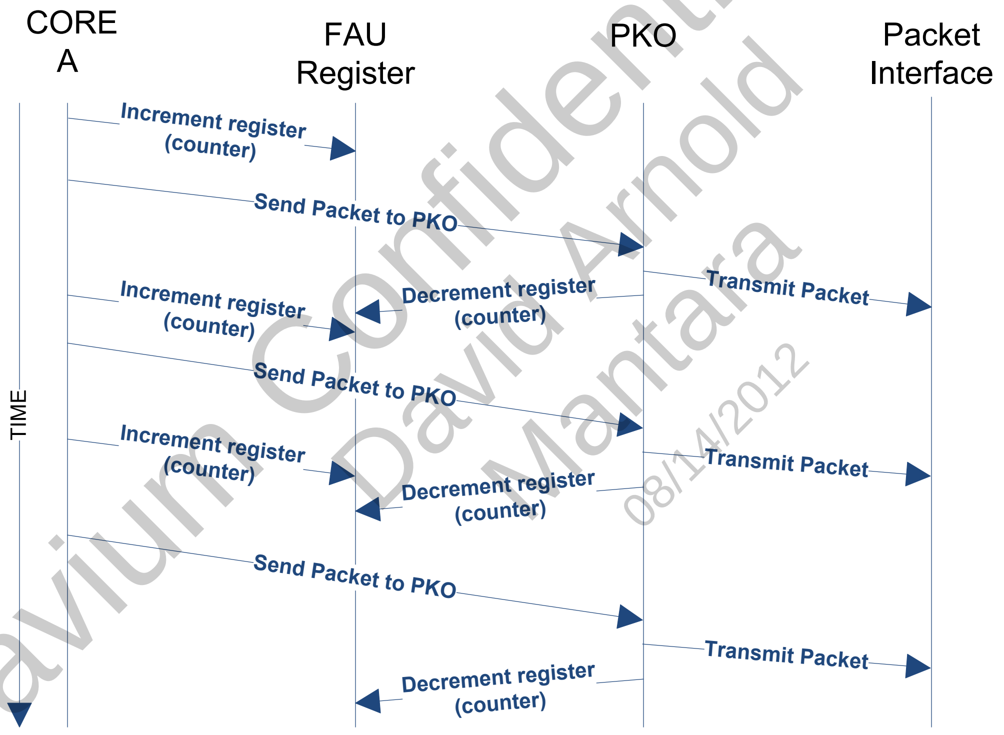
Figure 6: Core and PKO Use FAU Register to Monitor Output Queue Size

The Core and PKO Use a FAU Register to Monitor PKO Output Queue Length

1. The Core updates a counter in the FAU indicating a packet has been sent to the PKO output queue.

2. The PKO decrements the counter when the packet ships.

This allows the core to track the length of the PKO output queue.


CORE A
FAU Register
PKO
Packet Interface
TIME
Increment register (counter)
Send Packet to PKO
Increment register (counter)
Decrement register (counter)
Transmit Packet
Send Packet to PKO
Increment register (counter)
Decrement register (counter)
Transmit Packet
Send Packet to PKO
Transmit Packet
Decrement register (counter)

FAU IOBDMA operations always return a 64-bit result to the core, but the actual operation performed by the hardware, and the effective result returned, may be 8-bit, 16-bit, 32-bit, or 64-bit (an 8-bit result is returned in bits <7:0> of the 64-bit register, and the other bits are zero). The result is the value of the register before the update occurs.

The FAU is only used in specialized situations, since other instructions, such as the OCTEON processor-specific SAA and SAAD instructions can perform atomic memory read-modify-write operations (see the *Hardware Reference Manual* (*HRM*) for details). An example showing use of the FAU is in Section see Section 9 – "How to Measure the Cycles Used by a Section of Code". For more information, see the *HRM* (I/O Bridge section).

6 Definitions

When configuring Simple Executive, initializing the units, enabling the units, and populating the FPA pools, it is essential to understand the order in which these steps must be done. Especially when using the FPA, doing these steps in the wrong order can result in corrupted FPA pools.

6.1 Simple Executive Configuration (build time)

Configuring the Simple Executive is a build-time step which configures resources such as buffer sizes and pool numbers for FPA pools, and names and number of bytes to reserve for specific scratchpad areas. Configuring the Simple Executive requires an understanding of the application design, especially when configuring FPA pools. FPA pool design decisions are discussed in the *FPA* chapter.

6.2 Unit Initialization (runtime)

The Simple Executive configuration options presented in this chapter do not include all possible ways to configure the hardware. Custom configuration registers are discussed in the specific hardware unit's, for example, the *PIP/IPD* chapter. Custom configuration must be done before the hardware unit is enabled.

6.3 Unit Enable (runtime)

Enabling the hardware unit is done at runtime by the application initialization function after the hardware unit has been configured.

6.4 FPA Pool Population (runtime)

Buffers are added to the FPA pools at runtime by allocating a chunk of memory, dividing it into buffers, and freeing the buffers to a specific FPA pool using the `cvmx_fpa_free()` function.

Pool population must be done *after* the hardware unit has been initialized and enabled.

Note that the API functions sometimes refer to this as *initialization*. It is important to distinguish between the different steps because the order is critical: Simple Executive configuration; FPA initialization; FPA enable; and FPA pool population.

When using the Cavium Networks Ethernet driver, FPA pool population is done by the Ethernet driver for the Packet Data buffers, WQE buffers, and PKO command buffers.

When not running the Cavium Networks Ethernet driver, the application usually populates the FPA pools FPA at runtime by the application function.

See the *FPA* chapter for a discussion about adding more FPA-managed buffers at a later time.

7 Per-Core Write Buffers

This section describes the per-core Write Buffer. Understanding the Write Buffer is essential for correct multicore programming.

Each core has a Write Buffer. The Write Buffer enhances system performance by merging multiple L2/DRAM stores to a cache line instead of committing each store immediately to L2/DRAM. Each Write Buffer contains 16 entries, corresponding to 16 128-byte DRAM memory blocks that are normally loaded into separate cache lines.

When a core writes to memory, the write is not immediately visible to other cores because it may be temporarily delayed in the Write Buffer (see Section 7.2 – "When are Write Buffer Contents Committed to L2/DRAM?"). A `sync` instruction must be executed to guarantee the Write Buffer has been flushed. The `sync` instruction will flush the Write Buffer to the Coherent Memory Bus (CMB), destined for the L2 cache, and from there to possibly to DRAM. (The data will not be flushed from L2 cache to DRAM if the dirty bit was cleared by a DWB operation. See the *Advanced Topics* chapter for more information.) If other cores also have a copy of the data in their L1 Dcache, those copies are invalidated by the L2 Cache controller. Then if other cores later attempt to read data, they will automatically read the latest copy from L2 cache. See Section 8 – "Synchronization Instruction (`sync`) Variations" for details about the `sync` instruction.

Figure 7: Write Buffer Entry

The Write Buffer Entry Merges Writes

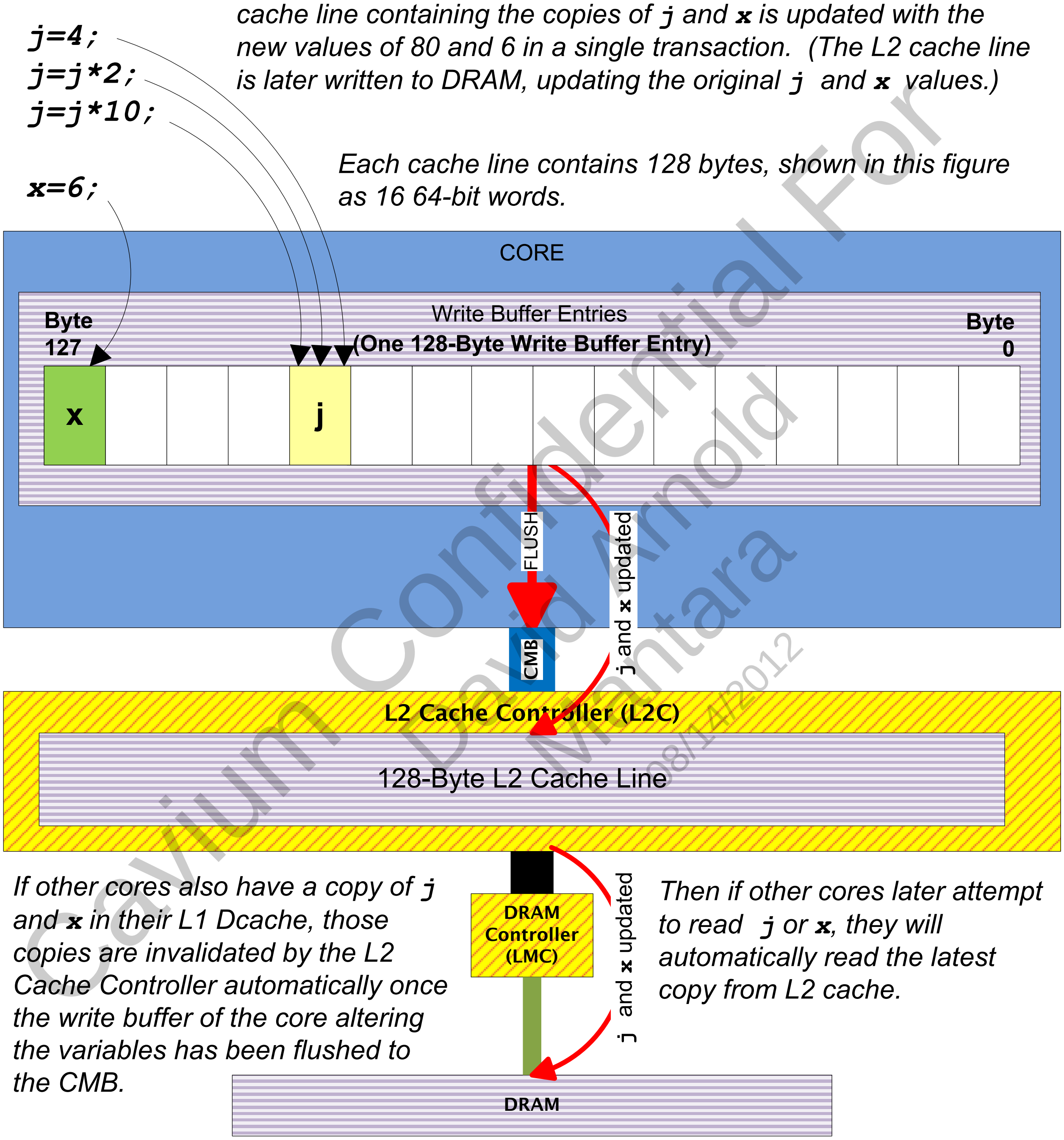
The Write Buffer entry merges writes which would go to the same hardware cache line. Later, these writes are flushed to L2 cache/DRAM.

This increases performance by reducing the number of writes to L2 cache.

In this example, when the Write Buffer Entry is flushed, the L2 cache line containing the copies of `j` *and* `x` *is updated with the new values of 80 and 6 in a single transaction. (The L2 cache line is later written to DRAM, updating the original* `j` *and* `x` *values.)*

```
j=4;
j=j*2;
j=j*10;

x=6;
```

Each cache line contains 128 bytes, shown in this figure as 16 64-bit words.



If other cores also have a copy of `j` *and* `x` *in their L1 Dcache, those copies are invalidated by the L2 Cache Controller automatically once the write buffer of the core altering the variables has been flushed to the CMB.*

Then if other cores later attempt to read `j` *or* `x`*, they will automatically read the latest copy from L2 cache.*

7.1 Marked and Unmarked Memory

For SE-S applications, the bootloader *marks* some memory and leaves other memory *unmarked* when it creates the TLB entry for the memory. (The terms *marked* and *unmarked* are OCTEON processor-specific terms.):

- Marked: Private memory (heap, stack, and read/write data). (Note: In a load set, each core has its own copy of the heap, stack, and read/write data. See the *Software Overview* chapter for information on load sets.)
- Unmarked: Shared memory (memory that might be the target of a DMA) and the *cvmx_shared* region. Shared memory is located in DRAM. See the *Software Overview* chapter for information on the *cvmx_shared* region. (Code and read/only data are also unmarked, but since they are not writable, they are not included in this discussion: `sync` instructions do not affect them.)

  Note: This discussion does not apply to Linux kernel, drivers, or applications. When running the SDK Linux, all memory is marked, and there is no separate synchronization instruction for unmarked memory. This discussion only applies to cores running SE-S applications.

Writes to shared memory may be buffered in the cores' Write Buffer. These writes must be flushed to L2/DRAM before the change can be seen by other cores. Writes to process-private memory are also buffered in the Write Buffer, but do not need to be visible to other cores. The separation of marked and unmarked writes allows writes to shared memory to be flushed without flushing all Write Buffer entries for marked memory.

Variations on the standard MIPS `sync` instruction are provided to support the flush of marked and/or unmarked memory. Synchronization instructions are discussed in Section 8 – "Synchronization Instruction (`sync`) Variations". See Figure 8 – "Marked and Unmarked memory, store and the `sync` Instruction" for an illustration of how `sync` variations affect marked and unmarked Write Buffer entries.

7.2 When are Write Buffer Contents Committed to L2/DRAM?

The selected contents of a Write Buffer entry are committed to L2/DRAM when (from the CN54-5-6-7XX-HM-2.4E version of the *HRM* – this list may be slightly different on different OCTEON models):

1. A `sync` or `syncw` instruction is issued.
2. A `syncws` instruction is issued and at least one store in the Write Buffer entry contains at least one unmarked store.
3. The Write Buffer entry times out. (The register field `CvmMemCtl[WBFLTIME]` controls the expiration interval of unmarked L2/DRAM Write Buffer entries.)
4. A Write Buffer entry contains updates to a memory location that is not currently present in Dcache, and a prefetch or load instruction targets the same memory block (i.e., the same Write Buffer entry). In this case, the stores in the Write Buffer entry are first committed to memory. (Note: if it is not present in L2 cache already, the L2 Cache Controller will first identify a free L2 cache block and then read the targeted address from DRAM into the L2 cache, and then complete the updates per the Write Buffer entry). Then an available

Dcache block is identified and loaded with the now updated L2 cache block contents. (Note that only the affected Write Buffer entry is flushed.)

5. A MIPS store conditional (sc or scd) instruction executes and hits in the Dcache block holding a memory block (address) that is already targeted for an update by an existing Write Buffer entry.
6. The Write Buffer entry is ejected to make space for other write-buffer entries.

Write operations to I/O space are not buffered in the Write Buffer; they go directly to the I/O bus. When writing code for OCTEON, it is essential to use sync appropriately to avoid the race condition shown in the following section.

See the *HRM* for more information about the Write Buffer.

8 Synchronization Instruction (sync) Variations

This section describes the different synchronization instructions provided on OCTEON. Understanding the use of synchronization instructions is essential for correct high-performance multicore applications. Note that for SDK Linux, all memory is unmarked. See Section 7.1 – "Marked and Unmarked Memory" for more information.

Cavium Networks has implemented the MIPS sync and synci (flush core Icache) instructions on the OCTEON processors, and has also provided special variations of the sync instruction to maximize hardware-specific features.

The sync instruction variations are used to create a barrier: load and store operations prior to the sync instruction will complete before subsequent load and store operations are allowed to issue.

Store and load operations can occur to/from:

- L2/DRAM
  - Unmarked memory (shared memory)
  - Marked memory (private memory)
- I/O space (for example, the store operation used to perform the buffer_free operation). (I/O space is accessed when the I/O bit is set in the physical address. See the *Advanced Topics* chapter for a ladder diagram showing the timing of the sync and the buffer free operation.)

Note that marked and unmarked memory are only relevant for SE-S applications.

There are two key reasons to use sync:

1. To guarantee that an IOBDMA operation has completed before accessing the data in the scratchpad.
2. To make sure that a write to shared memory is visible to other cores, and to the hardware units. This is essential when accessing shared memory from multiple cores. (See Section 8.1 – "Multicore Programming and Shared Memory (Synchronization)".

The sync variations are (in order of potentially decreasing performance cost):

- sync (synchronize all): Synchronize all store and load operations, including IOBDMAs. (The entire Write Buffer is ejected.)
- syncs (synchronize shared): Synchronize all store and load operations to shared memory and I/O space. Do *not* synchronize store and load operations to private memory.
- syncw (synchronize write): Synchronize all store operations. (The entire Write Buffer is ejected.)
- syncsws (synchronize write shared): Synchronize all store operations to shared memory and I/O space. Do *not* synchronize store operations to private memory.
- synciobdma (synchronize IOBDMA): Synchronize all IOBDMA operations. This instruction will not return until all outstanding IOBDMA commands have completed and the data is ready to be read from the scratchpad (potentially stalling the instruction pipeline).

Performance Note: The relative performance cost of the sync variations depends on the contents of the Write Buffer. For instance, syncw is not always more costly then syncws: if the Write Buffer only contains writes targeting cache lines corresponding to unmarked (shared) pages, the cost is the same.

(Note: In this document, syncw is referred to as "synchronize shared". In the *HRM* it is referred to as "synchronize special". Similarly, this document refers to syncws as "synchronize shared", but the *HRM* refers to it as "synchronize stores special".)

The syncs and syncws commands apply to any cores running SE-S applications.

The Simple Executive API contains convenient macros which can be used from C-code for each sync variation:

- sync: CVMX_SYNC
- syncs : CVMX_SYNCS
- syncw: CVMX_SYNCW
- syncsws: CVMX_SYNCWS
- synciobdma: CVMX_SYNCIOBDMA

Note: For the SDK Linux Kernel, and Linux Applications, including Simple Executive compiled under Linux (SE-UM), all memory is unmarked and CVMX_SYNCWS is redefined to be the same as CVMX_SYNCW.

Table 2: Various sync Instructions Available

Sync Instruction	L2/DRAM				I/O Space		IODBMA	Instruction Summary
	Unmarked (Shared) Memory		Marked (Private) Memory					
	Load	Store	Load	Store	Load	Store		
`sync`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Sync all load and store operations, *and* `synciobdma`. See Note1.
`syncs`	Yes	Yes			Yes	Yes	Yes	Sync all load and store operations to shared memory and I/O space, *and* `synciobdma`. See Note1.
`syncw`		Yes		Yes		Yes		Sync all store operations. See Note1.
`syncws`		Yes				Yes		Sync all store operations to shared memory and I/O space. See Note1. See Note2.
`synciobdma`							Yes	Sync all IODBMAs.

Notes

Note1: An I/O store is complete when it reaches the coherent memory bus. L2/DRAM stores are complete when the stored value is visible to every other core and to all OCTEON I/O units.
Note2: For Linux Kernel, or Linux Applications, including Simple Executive compiled under Linux, these `CVMX_SYNCWS` is defined to be the same as `CVMX_SYNCW`, so both marked and unmarked memory are synchronized if `CVMX_SYNCWS` is executed.

Select the synchronization command which will satisfy the synchronization requirement with the least performance cost. For example, to guarantee all IOBDMAs have completed, `synciobdma`, `syncs`, and `sync` will all satisfy the requirement. The best choice is `synciobdma`, which has the least performance cost.

Note: Writes to I/O space go through the Write Buffer, but are not stored there. The `sync` instructions which synchronize writes to I/O space guarantee that the I/O space store has not only been *issued* to the CMB, but the CMB has confirmed that it has been *committed* to the bus.

Figure 8: Marked and Unmarked memory, store and the `sync` Instruction

Write Buffer and **`sync*`** Instructions

Each core's writes (**`store`**) to L2/DRAM are buffered in the core's Write Buffer.

Memory allocated via the **`cvmx_bootmem*()`** functions (shared memory) is *unmarked* memory. All other memory is *marked*.

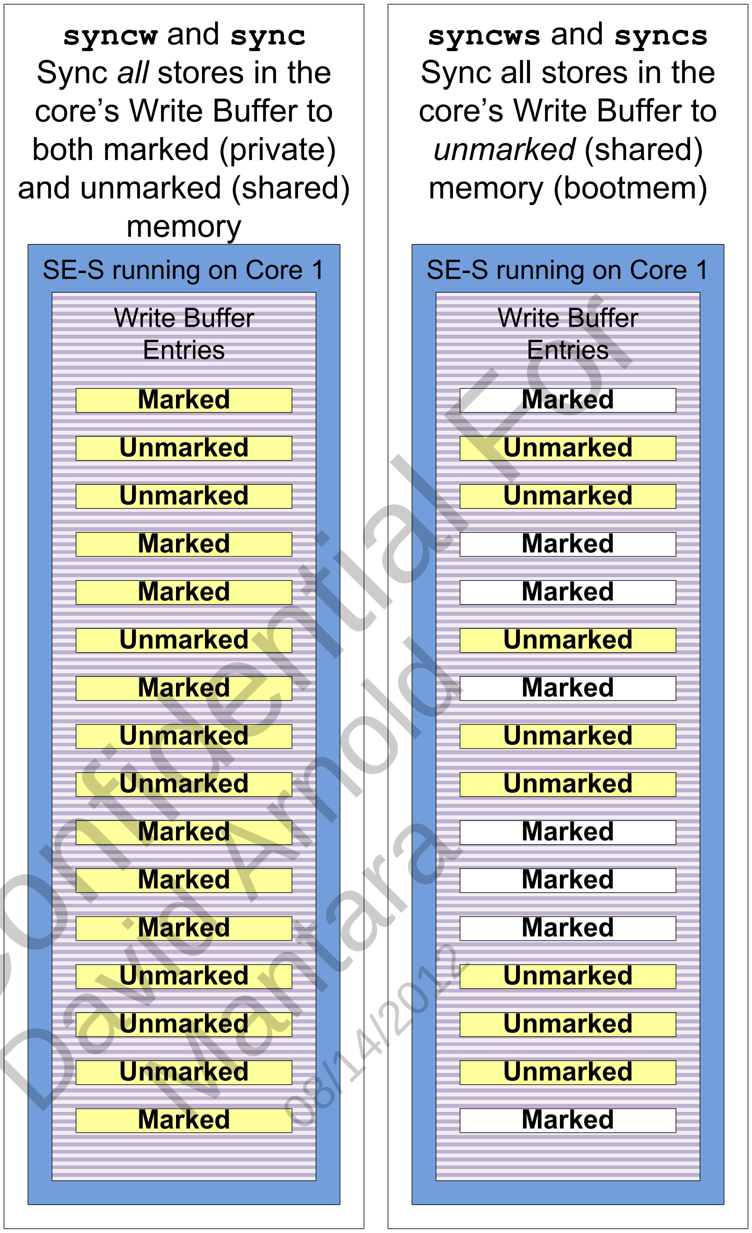
To flush only writes to shared memory, issue either:

- **`syncws`** (**`CVMX_SYNCWS`**)
- **`syncw`** (**`CVMX_SYNCW`**)
- **`syncs`** (**`CVMX_SYNCS`**)
- **`sync`** (**`CVMX_SYNC`**)

The **`sync`** and **`syncs`** instructions will wait for all outstanding IOBDMA operations on the core to complete (potentially stalling the instruction pipeline), so if IOBDMAs completion is not important, then use **`CVMX_SYNCWS`** for maximum performance.

Write buffer entries shown in yellow are affected by the instruction.

Note: For Linux Kernel, or Linux Applications, including Simple Executive compiled under Linux, these **`CVMX_SYNCWS`** is defined to be the same as **`CVMX_SYNCW`**, and both marked and unmarked memory is synchronized.



`syncw` and **`sync`** Sync *all* stores in the core's Write Buffer to both marked (private) and unmarked (shared) memory

`syncws` and **`syncs`** Sync all stores in the core's Write Buffer to *unmarked* (shared) memory (bootmem)

8.1 *Multicore Programming and Shared Memory (Synchronization)*

The following example is taken from the `syncw` instruction description in the *HRM*. The command used in this example is `syncws`, which will only synchronize unmarked stores and stores to I/O space. Note that this illustration applies to SE-S applications using unmarked memory (shared memory allocated via the `cvmx_bootmem*()` functions). When using Linux, all memory is unmarked. The same code can be used for both Linux and SE-S applications because the macro `CVMX_SYNCWS` is defined to be `CVMX_SYNCW`.

Two cores are used in this example, a reader and a writer. There are two data structures in shared memory: `shared_flag`, and `shared_data`. The value of `shared_flag` is set to 1 when `shared_data` is valid. The value of `shared_flag` is set to 0 when `shared_data` is invalid.

On entry to this section of pseudo code, `shared_flag` is `0` and the value is observable by Core B (the L2 block holding a copy of `shared_flag` is 0).

```
In the shared header file:
--------------------------
extern volatile uint64_t shared_flag;
extern volatile data_type_t shared_data;

CORE A (WRITER)
---------------
      // shared_flag == 0 in memory on entry

      shared_data = new value;// Write new value to shared memory
      CVMX_SYNCWS;            // flush shared_data value from
                              // Core A Write Buffer to L2/DRAM now to
                              // make sure it gets there before the
                              // shared_flag update.

      shared_flag = 1;        // Notify Core B shared_data is now valid.
      CVMX_SYNCWS;            // Flush shared_flag value from Core A
                              // Write Buffer to L2/DRAM.  (This step is
                              // not required, but can improve
                              // performance.

CORE B (READER)
---------------
      // shared_flag == 0 in memory on entry

      // loop until shared_flag is set (shared_flag will be set when
      // shared_data is valid)
      while (shared_flag == 0)
      {
         read shared_flag from shared memory
      };

      // shared_data is now valid
      read shared_data from shared memory
```

Without any `CVMX_SYNCWS` instructions, it is unpredictable when the Write Buffer entry containing `shared_flag` and the Write Buffer entry containing `shared_data` will be flushed (in this example, they are assumed to be in two different Write Buffer entries). This not only makes the performance unpredictable, but the *order* of completion of the two writes is not guaranteed: the `shared_flag` might be written *before* the `shared_data`. In this case, the reader could (depending on the exact timing of events) receive an incorrect value for shared_data. (See Figure 11 – "Ladder Diagram: Bad Code – No sync Instructions".)

If only one CVMX_SYNCWS was used after the write to both shared_data and to shared_flag), since the order of completion of the two writes is not guaranteed, the shared_flag might be written *before* the shared_data (assuming they are in different Write Buffer entries). In this case, the reader could (depending on the exact timing of events) receive an incorrect value for shared_data. (See Figure 12 – "Ladder Diagram – Bad Code – One sync Instruction".)

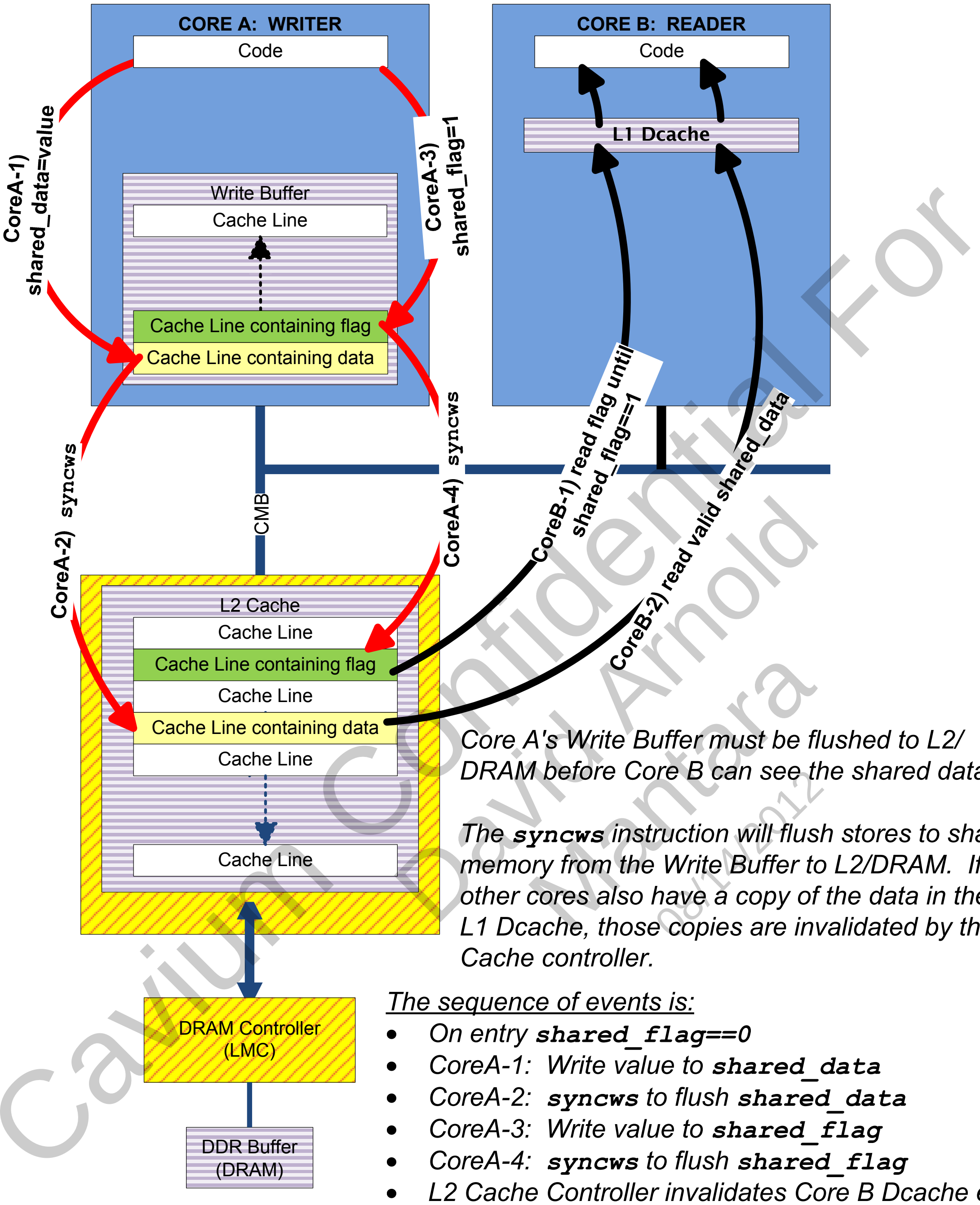
The second CVMX_SYNCWS instruction is not necessary for correctness, but improves performance by flushing the write to shared_flag from the Write Buffer. Without the CVMX_SYNCWS instruction, if Core A is not performing many store operations (which could cause the cache line to be flushed before reuse) then the flush of shared_flag might not occur until hundreds of thousands of cycles after the write to it. Applications should therefore include this second CVMX_SYNCWS instruction after writes to flags or locks. (See Figure 10 – "Ladder Diagram: Good Code – Two sync Instructions".):

- The first CVMX_SYNCWS instruction guarantees the new value of shared_data is visible to Core B.
- The second CVMX_SYNCWS guarantees the new value of shared_flag is visible to Core B.

Figure 9: Multicore Programming and the `syncws` Instruction

Multicore Processing and **`syncws`**

This figure shows example steps a reader and writer must take to reliably share data.



Core A's Write Buffer must be flushed to L2/ DRAM before Core B can see the shared data.

*The **`syncws`** instruction will flush stores to shared memory from the Write Buffer to L2/DRAM. If other cores also have a copy of the data in their L1 Dcache, those copies are invalidated by the L2 Cache controller.*

The sequence of events is:

- *On entry **`shared_flag==0`***
- *CoreA-1: Write value to **`shared_data`***
- *CoreA-2: **`syncws`** to flush **`shared_data`***
- *CoreA-3: Write value to **`shared_flag`***
- *CoreA-4: **`syncws`** to flush **`shared_flag`***
- *L2 Cache Controller invalidates Core B Dcache entry*
- *CoreB-1: read **`shared_flag`** until it equals 1 (valid) (the loop reading **`shared_flag`** can begin any time)*
- *CoreB-2: read valid **`shared_data`***

Figure 10: Ladder Diagram: Good Code – Two sync Instructions

Good Code: Multicore Processing and Shared Memory Write With Two **sync** Instructions

GOOD CODE (two sync instructions):
CORE A (WRITER)

```
-------------------------
// shared_flag==0 in memory on entry

shared_data=new value;// Write new value to
                      // shared memory
CVMX_SYNCWS;          // flush shared_data
                      // value from Core A
                      // Write Buffer to
                      // L2/DRAM now to
                      // make sure it arrives
                      // before the shared_flag
                      // update.
shared_flag=1;        // Notify Core B
                      // shared_data is now
                      // valid.
CMMX_SYNCWS;          // Flush shared_flag
                      // value from Core A
                      // Write Buffer to
                      // L2/DRAM
```

CORE B (READER)

```
------------------------
// shared_flag==0 in memory on entry

// loop until shared_flag is set (shared_flag
// will be set when shared data is valid
do {
   // force C compiler to assume
   // shared_flag could have changed
   COMPILER_BARRIER;
} while (shared_flag == 0);

// shared_data is now valid
read shared_data from shared memory
```

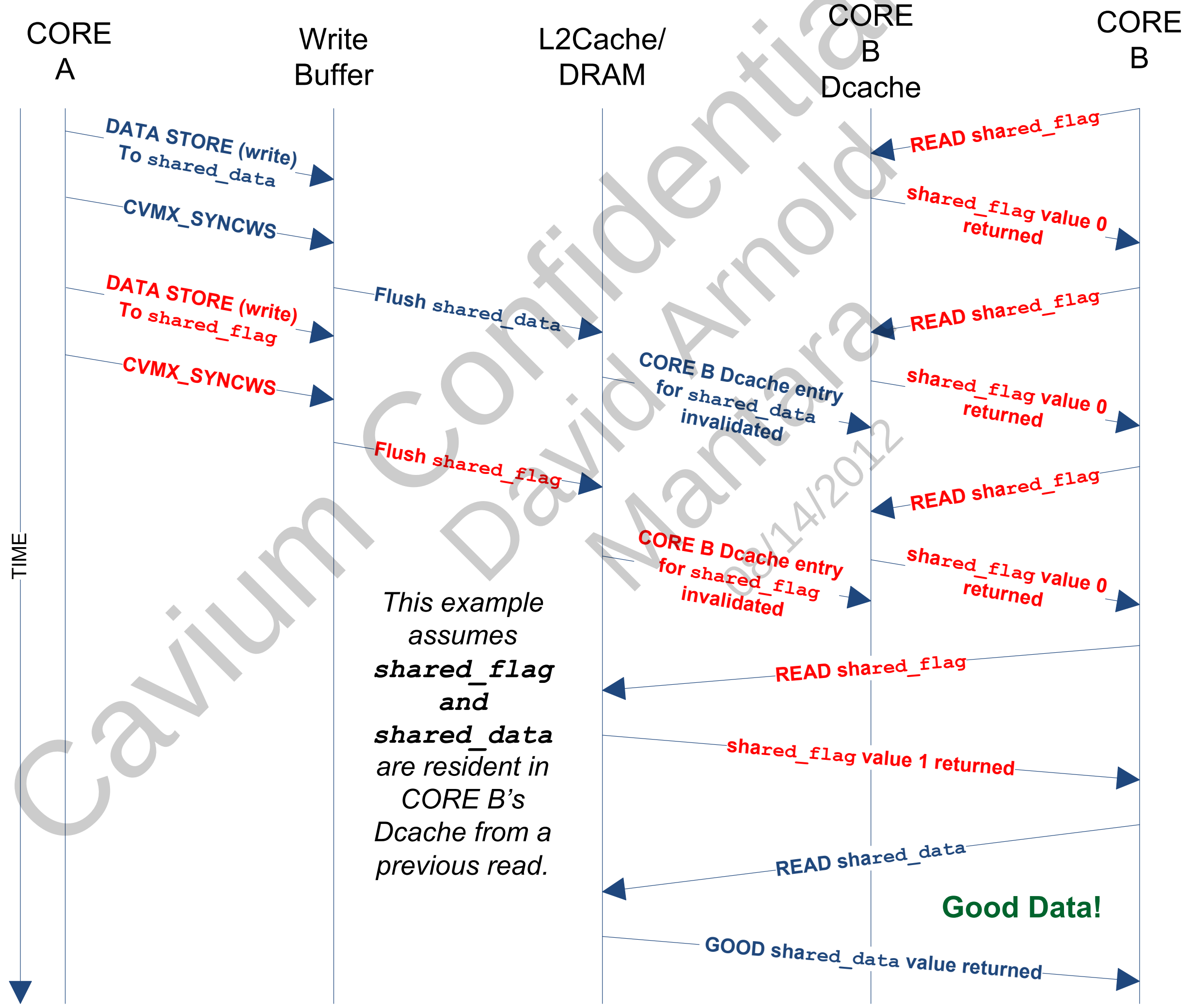

CORE A
Write Buffer
L2Cache/ DRAM
CORE B Dcache
CORE B
DATA STORE (write) To shared_data
CVMX_SYNCWS
DATA STORE (write) To shared_flag
CVMX_SYNCWS
Flush shared_data
Flush shared_flag
READ shared_flag
shared_flag value 0 returned
READ shared_flag
shared_flag value 0 returned
CORE B Dcache entry for shared_data invalidated
READ shared_flag
shared_flag value 0 returned
CORE B Dcache entry for shared_flag invalidated
READ shared_flag
shared_flag value 1 returned
READ shared_data
Good Data!
GOOD shared_data value returned
TIME
This example assumes shared_flag and shared_data are resident in CORE B's Dcache from a previous read.

Figure 11: Ladder Diagram: Bad Code – No `sync` Instructions

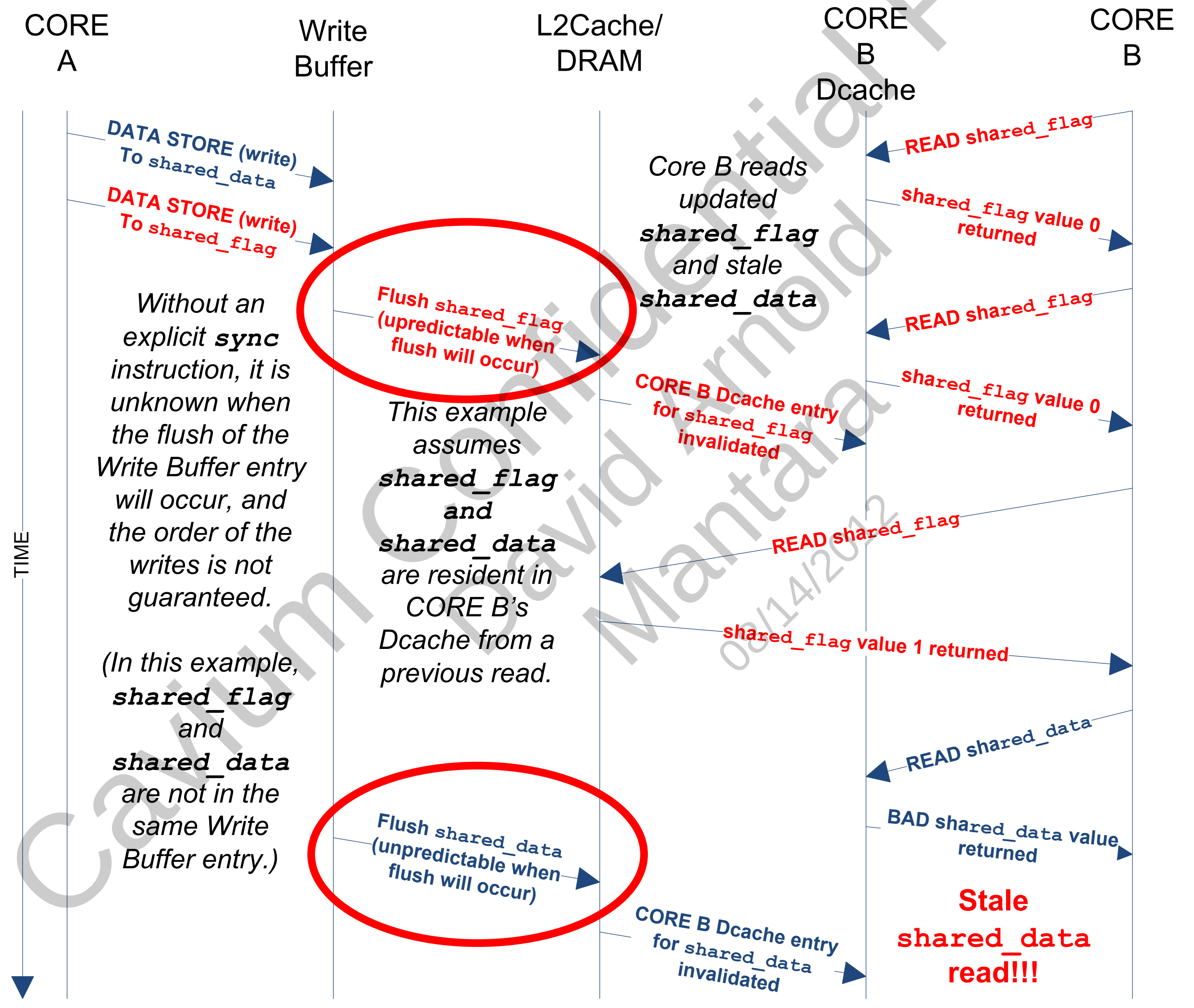
Bad Code: Multicore Processing and Shared Memory Write Without **sync** Instruction

BAD CODE (no **sync** instruction):
CORE A (WRITER)

```
--------------
// shared_flag==0 in memory on entry

shared_data=new value;// Write new value
                      // to shared memory
shared_flag=1;        // Notify Core B
                      // shared_data is
                      // now valid.
```

CORE B (READER)

```
------------------------
// shared_flag==0 in memory on entry

// loop until shared_flag is set (shared_flag
// will be set when shared data is valid
do {
   // force C compiler to assume
   // shared_flag could have changed
   COMPILER_BARRIER;
} while (shared_flag == 0);

// shared_data is now valid
read shared_data from shared memory
```



Core B reads updated ***shared_flag*** *and stale* ***shared_data***

Without an explicit ***sync*** *instruction, it is unknown when the flush of the Write Buffer entry will occur, and the order of the writes is not guaranteed.*

(In this example, ***shared_flag*** *and* ***shared_data*** *are not in the same Write Buffer entry.)*

This example assumes ***shared_flag and shared_data*** *are resident in CORE B's Dcache from a previous read.*

Stale shared_data read!!!

Figure 12: Ladder Diagram – Bad Code – One sync Instruction

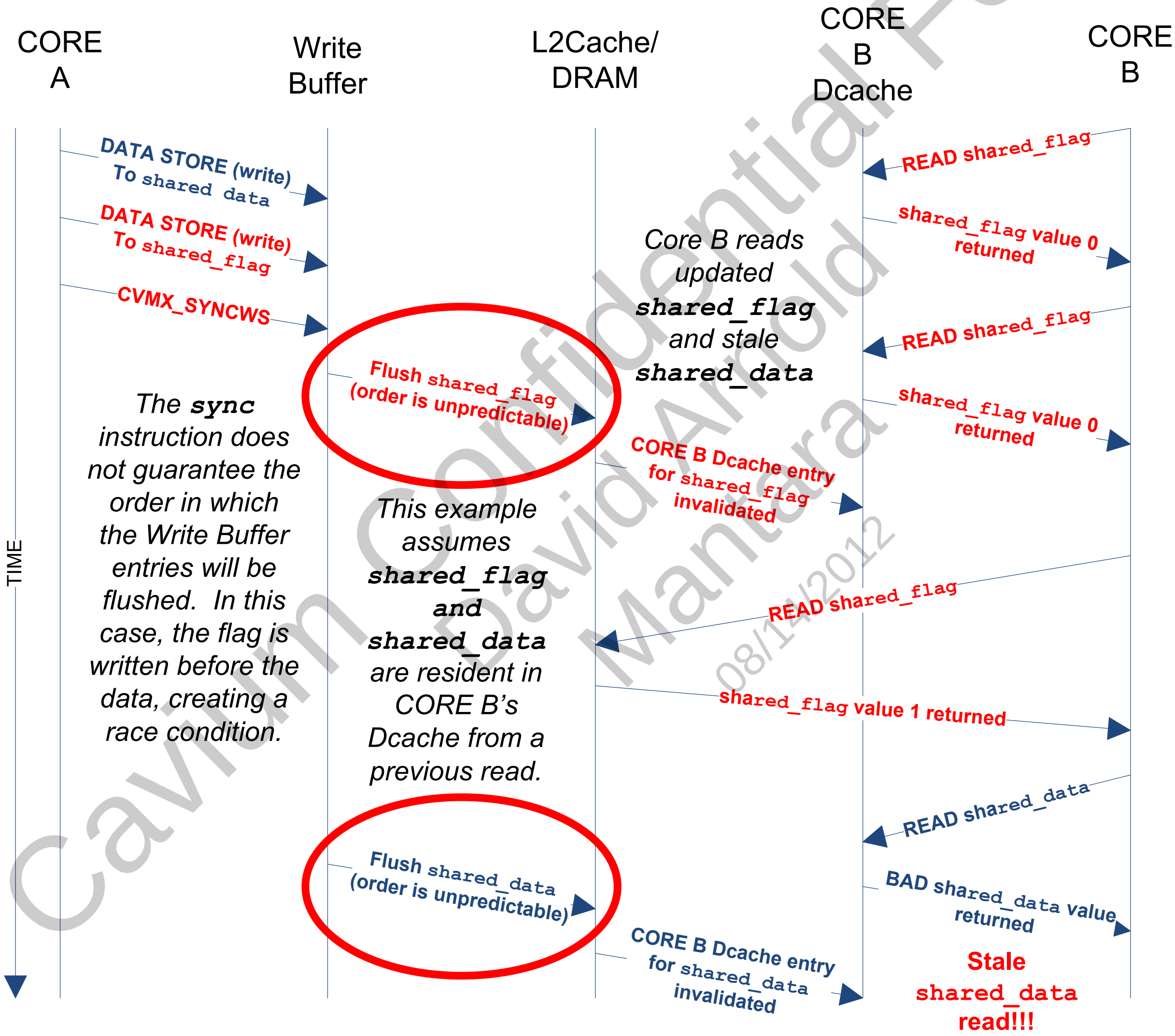
Bad Code: Multicore Processing and Shared Memory Write With Only One sync Instruction

BAD CODE (only one sync instruction):
CORE A (WRITER)

```
--------------
// shared_flag==0 in memory on entry

shared_data=new value; // Write new value to
                       // shared memory
shared_flag=1;         // Notify Core B
                       // shared_data is now
                       // valid.
CMMX_SYNCWS;           // Flush shared_flag and
                       // shared_data value from
                       // Core A’s Write Buffer
                       // to L2/DRAM.
```

CORE B (READER)

```
-------------------------
// shared_flag==0 in memory on entry

// loop until shared_flag is set (shared_flag
// will be set when shared data is valid
do {
   // force C compiler to assume
   // shared_flag could have changed
   COMPILER_BARRIER;
} while (shared_flag == 0);

// shared_data is now valid
read shared_data from shared memory
```


CORE A
Write Buffer
L2Cache/ DRAM
CORE B Dcache
CORE B
TIME
DATA STORE (write) To shared_data
DATA STORE (write) To shared_flag
CVMX_SYNCWS
READ shared_flag
shared_flag value 0 returned
READ shared_flag
shared_flag value 0 returned
Flush shared_flag (order is unpredictable)
CORE B Dcache entry for shared_flag invalidated
READ shared_flag
shared_flag value 1 returned
READ shared_data
BAD shared_data value returned
Flush shared_data (order is unpredictable)
CORE B Dcache entry for shared_data invalidated

*Core B reads updated **shared_flag** and stale **shared_data***

*The **sync** instruction does not guarantee the order in which the Write Buffer entries will be flushed. In this case, the flag is written before the data, creating a race condition.*

*This example assumes **shared_flag and shared_data** are resident in CORE B’s Dcache from a previous read.*

Stale shared_data read!!!

9 How to Measure the Cycles Used by a Section of Code

This section shows how to measure the number of cycles which are consumed by a section of code. This section also shows how to determine whether there is any value of using asynchronous operations in a specific section of time critical code.

The actual value of using asynchronous operations will vary depending on the processor, which core is performing the operation, and the system load. (On some processor models, lower-numbered cores have a smaller delay than higher-numbered cores. Different core priority rules apply on different processors.) Performance tuning is an art. (See the *Packet Processing Math* figure in the *Software Overview* chapter.) (Note that this example will perform quite differently on OCTEON II which has faster FAU access time because accesses go through the L2 cache.)

This example (performed on an unloaded (not busy) system) shows the cycles saved by using asynchronous versus synchronous read of FAU registers, and shows how to determine the number of cycles consumed by a section of code. The function `cvmx_clock_get_count(CVMX_CLOCK_CORE)` will return the current cycle value.

Note: The functions used to read the clock cycle have changed in SDK 2.0 due to the presence of more clocks in CN63XX:

- SDK 2.0: use `cvmx_clock_get_count(CVMX_CLOCK_CORE)`
- Earlier SDKs: `cvmx_get_cycle()`

9.1 Synchronous Operation Timed

```
uint64_t startCycle;
unit64_t stopCycle;
int64_t pktCnt = 8;

startCycle = cvmx_clock_get_count(CVMX_CLOCK_CORE);

// Add 1 to the FAU register FAU_INGRESS_PKT_CNT atomically, and return
// the previous register value
pktCnt = cvmx_fau_fetch_and_add64(FAU_INGRESS_PKT_CNT, 1);

stopCycle = cvmx_clock_get_count(CVMX_CLOCK_CORE);

// Note cvmx_fau_fetch_and_add64() returns the previous register value,
// so add 1 when printing pktCnt
cvmx_dprintf("cvmx_fau_fetch_and_add64: Packet Count = %lld (cycles = %lld)\n",
             (unsigned long long)pktCnt+1,
             (unsigned long long)stopCycle - startCycle);

startCycle = cvmx_clock_get_count(CVMX_CLOCK_CORE);
```

Console message:

```
PP0:~CONSOLE-> cvmx_fau_fetch_and_add64: Packet Count = 9 (cycles = 71)
```

9.2 Asynchronous Operation Timed

The following pseudo code shows how to time an asynchronous operation.

```
uint64_t startCycle;
unit64_t stopCycle;
int64_t pktCnt = 8;


startCycle = cvmx_clock_get_count(CVMX_CLOCK_CORE);


// Write a known pattern to the scratchpad area.  When this value is
// overwritten, the IOBDMA operation has completed.
cvmx_scratch_write64(SCRATCH_INGRESS_PKT_CNT, 0xACEDC0DE);

// Add 1 to the FAU register FAU_INGRESS_PKT_CNT asynchronously, and return
// the previous register value in the scratchpad area SCRATCH_INGRESS_PKK_CNT
cvmx_fau_async_fetch_and_add64(SCRATCH_INGRESS_PKT_CNT,
                               FAU_INGRESS_PKT_CNT, 1);

stopCycle = cvmx_clock_get_count(CVMX_CLOCK_CORE);

// Since this is an asynchronous operation, it will not stall the core, but the
// IOBDMA operation may not have completed yet (written the old FAU register
// value to the scratchpad area).
cvmx_dprintf("scratch_write64 & async_fetch_and_add64: cycles = %lld\n",
              (unsigned long long)stopCycle - startCycle);

// PUT SOME OTHER USEFUL CODE HERE OR THIS WOULD ACTUALLY BE SLOWER
cvmx_dprintf("Put some useful code here!\n\n");

// time delay before operation completes
startCycle = cvmx_clock_get_count(CVMX_CLOCK_CORE);

// Note:  if there is no useful work to do while waiting, use CVMX_SYNCIOBDMA
// instead of the following loop
do // read the scratch register, waiting for the result to arrive
{
    // cvxm_scratch_read64() uses a volatile variable, so the read of pktCnt
    // will occur each time
    pktCnt = cvmx_scratch_read64(SCRATCH_INGRESS_PKT_CNT);
} while (pktCnt == 0xACEDC0DE);

stopCycle = cvmx_clock_get_count(CVMX_CLOCK_CORE);

// Note cvmx_fau_async_fetch_and_add64() returns the previous
// register value, so add 1 when printing pktCnt
cvmx_dprintf("cvmx_scratch_read64: Packet Count = %lld (cycles = %lld)\n",
              (unsigned long long)pktCnt+1,
              (unsigned long long)stopCycle - startCycle);
```

Console messages:

```
PP0:~CONSOLE-> scratch_write64 & async_fetch_and_add64: cycles = 7
PP0:~CONSOLE-> cvmx_scratch_read64: Packet Count = 9 (cycles = 4)
```

When using the asynchronous operation, the cycle count will reduce to approximately 11 cycles if sufficient work is done before retrieving the result from scratchpad (so that the `do` / `while` loop is only executed once).

10 About the `volatile` Type Qualifier

The compiler only knows about the program it is compiling. If, in the compiler's view, a variable being read hasn't changed, there is no need to read it again. If the variable is changed by another program or another core, the compiler doesn't know that. A common example is reading shared memory which is altered by another core, such as a lock, flag, or data. It is important to stay alert to compiler optimizations which can change the code: the code works until optimization is turned on, and then suddenly it stops working.

The C language type qualifier `volatile` is used to define a variable that will be changed by something other than the program. In the case of the code shown in Section8.1 – "Multicore Programming and Shared Memory (Synchronization)", Core B accesses `shared_flag` which is changed by Core A.

Typically, the compiler optimizer will only read the value of `i` once in the following expression:

```
        j = i + i;
```

but if `i` is given the type qualifier `volatile`, it will be read each time it is used:

```
        volatile int i;
```

11 Strict Aliasing

This issue pertains to GCC, not to OCTEON. Aliasing is when the same memory location can be accessed by different symbolic names in a C program. The C-language (edition C99) specifies that (with some exceptions) the pointers need to be of the same type (strict aliasing). The strict aliasing rule does not apply to `char *`.

The GNU compiler supports strict aliasing by default. This can cause older code which was written depending on non-strict aliasing to break when the compiler uses strict aliasing to make certain optimizations. If it is not possible to fix the code, then pass the option '`-fno-strict-aliasing`' to the compiler. Note that warning-free compilation of source code is not a guarantee that there are no problems. The typical warning message is "dereferencing type-punned pointer will break strict-aliasing rules".

Signed and unsigned types can be aliased under the strict aliasing rule, as in:

```
        uint i;
        int j;
```

The following two examples show the result of strict aliasing when types match, and when they do not.

Case1: Types Match:

```
      void foo(long *p1, long *p2)
      {
        *p2 = 4;
        // From the compiler's perspective, p1 and p2 can point to the
        // same memory location, because they are the same type
        *p1 = 5;
        return (*p2 * 2);
      }
```

Case 2: Types Do Not Match:

```
      void foo(long *p1, int *p2)
      {
        *p2 = 4;

        // The C and C++ language definitions allow the compiler to
        // assume that p1 and p2 point to different memory because they
        // have different types

        *p1 = 5;
        return (*p2 * 2);
      }
```

So the compiler can optimize this to:

```
      void foo(long *p1, int *p2)
      {
        *p2 = 4;
        *p1 = 5;
        return 8;
      }
```

12 User-Mode Applications Access to Kernel Segments

The virtual memory maps shown in the *Software Overview* chapter specify that SE-UM applications can access CSRs and physical memory (such as FPA-managed buffers) via *xkphys* segment addresses, and can access *cvmseg* (scratchpad) addresses (in *kseg3*). User-mode application access to kernel segment addresses is only allowed in the specifically permitted address ranges.

These accesses are configured via Linux `make menuconfig` options:

- Access to CSRs: `CONFIG_CAVIUM_OCTEON_USER_IO`
- Access to Physical Memory: `CONFIG_CAVIUM_OCTEON_USER_MEM`
- Access to Scratchpad (*cvmseg*): `CONFIG_CAVIUM_OCTEON_CVMSEG_SIZE` (if size is non-zero, the kernel will set access permissions appropriately for the application)

Both `CONFIG_CAVIUM_OCTEON_USER_IO` and `CONFIG_CAVIUM_OCTEON_USER_MEM` can be configured to allow all, allow none, or conditionally allow user-mode application access. On startup, SE-UM applications request access (via `cvmx_user_app_init()`), which is either granted or denied based on the values of these two variables. See the *Software Overview* chapter for more information on Cavium Networks-specific kernel configuration variables.

> ***Once the kernel configuration variables are set properly, SE-UM applications access the scratchpad via the API functions, including `cvmx_scratch_read*()`, and access CSRs via `cvmx_read_csr()` and `cvmx_write_csr()`. The default API configuration allows SE-S applications to run correctly using the same functions without modifying the API.***

(Note that user-mode applications can use kernel-segment addresses without needing kernel-segment TLB access because these addresses are not translated by the TLB. Permission for user-mode applications to access these addresses is granted via Cavium Networks-specific Coprocessor 0 (CP0) `CvmMemCtl` register fields, bypassing the usual MIPS protection that prevents user-mode processes from accessing kernel segments.)

SE-S applications always run in kernel mode, so access to kernel segments is automatically enabled.

See the *Advanced Topics* chapter for more details.

13 32-bit Application Access to 64-Bit Addresses

32-bit SE-S and SE-UM applications need to access 64-bit CSRs and scratchpad addresses via kernel segment addresses. These accesses are automatically handled by the SDK API. This section provides a brief explanation of the problem and how it is solved.

The problem is that the 32-bit applications have 32-bit pointers, not 64-bit pointers.

Although the C compiler generates 32-bit pointers, on OCTEON processors the 32-bit applications run in a 64-bit environment with 64-bit operations and addressing. SDK API functions (such as `cvmx_scratch_read*()`, and `cvmx_read_csr()`) use inline assembly code to store the needed 64-bit address into a general-purpose register and use 64-bit operations to load and store 64-bit values (`ld` (load doubleword) and `sd` (store doubleword)). (In the case of *cvmseg* accesses, the 32-bit address is automatically sign-extended when it is put into a 64-bit general-purpose register, creating the 64-bit *cvmseg* address.)

Note that the 32-bit pointers are a compiler-level restriction, not a hardware-level restriction.

(In contrast, standard Linux 32-bit applications execute in a 32-bit environment where hardware mechanisms restrict them to 32-bit addressing (a 2 Gbyte address range) and 32-bit operations which limit data sizes to 32 bits ("native machine" instructions such as `lw` (load word) and `sw` (store word)).)

32-bit applications do not use inline assembly code to access physical memory via *xkphys* addresses because these accesses typically are done via a pointer, and pointers are only 32-bits. 32-bit SE-UM applications use the *reserve32* region to access memory allocated via the `cvmx_bootmem*()` functions, such as FPA-managed buffers. SE-S application access to bootmem is discussed in the *Software Overview* chapter.

See the *Advanced Topics* chapter for more details.