Advanced Topics

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

This chapter provides advanced information for readers who are adding code to the Simple Executive API, writing a custom API, reading Simple Executive code, or debugging code running on OCTEON. Before reading this chapter, please read the *Essential Topics* chapter which provides an introduction to some of the advanced material in this chapter.

This chapter includes information on:

- Accessing CSRs from Software
- Control and Status Registers (CSRs)
- Physical Address Map and Device ID (DID)
- Potential race conditions when can occur when accessing CSRs or freeing a buffer
- Don't Write Back Engine details
- Scratchpad and IOBDMA details
- "Unprotected" buffer pool details (advanced FPA pool configuration information)
- Accessing selected addresses in 64-Bit kernel mode segments from user mode applications, including 32-bit applications

2 Configuration and Status Registers (CSRs)

This section provides information on accessing Configuration and Status registers (CSRs) from software. CSRs provide additional functionality beyond the API.

CSRs sometimes provide additional functionality beyond what is supported in the current SDK release API. The CSRs shown in this section are the FPA Control and Status registers present in the OCTEON CN58XX processor. Exact CSR names and fields vary on different processor models, but the naming convention used to convert hardware CSR and field names to a software name apply to all processors.

The CSRs are fully documented in the Hardware Reference Manual. Note that there are sometimes multiple registers CSR's with a similar names, but using different numbers in the name to indicate a specific CSR. In this case the name of the CSR appears in the text or SDK software containing an "x" or "X". For example: `FPA_QUE`*x*`_AVAILABLE`. "*x*" can be any value between 0-7, inclusive, corresponding to one of 8 FPA pools.

Information about the system's memory map and accessing CSRs can be found in Table 2 "CN58XX Physical Addresses".

2.1 CSR Name Definitions

CSRs are defined in the Simple Executive include files located in the `executive` directory. If you want to manipulate OCTEON CSRs, include `cvmx-csr.h`. (This include file will include the necessary include files to get all of the CSR definitions.) Each CSR is assigned the appropriate physical address in the include file. The CSR name which appears in the *HRM* will become a name (in all upper case) which is used to access the CSR. This name is pre-pended with "`CVMX`" (for example: `CVMX_FPA_CTL_STATUS`).

When there are multiple CSRs which only vary by the "X", such as the CVMX_FPA_QUEX registers, then the Simple Executive API convention is to create a macro that accepts "X" as a parameter. The macro will use the number to calculate the address of the matching CSR. This allows the code to easily multiple CSRs which perform the same function on different objects. For example, the macro CVMX_FPA_QUEX_PAGE_INDEX() takes a pool number as a parameter: To access the CSR for Pool 2 (`CVMX_FPA_QUE2_PAGE_INDEX`), use the macro `CVMX_FPA_QUEX_PAGE_INDEX(2)`.

For example, the of the FPA CSRs macros are:

```
        CVMX_FPA_CTL_STATUS
        CVMX_FPA_INT_ENB
        CVMX_FPA_INT_SUM
        CVMX_FPA_QUE_ACT
        CVMX_FPA_QUE_EXP
        CVMX_FPA_QUEX_AVAILABLE(offset)
        CVMX_FPA_QUEX_PAGE_INDEX(offset)
        CVMX_FPA_FPFX_MARKS(offset)
        CVMX_FPA_FPFX_SIZE(offset)
```

Similarly, for the SSO, the macro `POW_QOS_THRX()` can be used to the threshold for QoS queue 5 (`POW_QOS_THR5`): `POW_QOS_THRX(5)`.

2.2 CSR Data Structures

To access a *field* inside the CSR, instead of the *entire* CSR, read the CSR into a data structure, then access the field. CSR data structures are given the same name as the CSR, except they are all lower case. The typedefs end in the characters "`_t`". The CSR data structure fields will also have names (also in lower case) matching the Hardware Reference Manual names (see Table 1: "Accessing CSR Fields").

CSR data structures are unions. The CSR can be accessed as a `uint64_t`, by the subfields (`s`), or by the chip-specific name, such as `cn58xx`.

The `s` structure is a subfields which contains all fields that don't conflict between the different OCTEON models. Accessing common fields via the `s` structure streamlines common code. (See Section 2.3- "Accessing CSRs via CSR Definitions and Data Structures".)
The following data structure is shown without the endian considerations or the comments.

Note that the `cvmx_ipd_bp_prt_red_end.s` structure contains the same content as the `cvmx_ipd_bp_prt_red_end_cn30xx`, but the field widths are different. The number of ports supported varies with the OCTEON model, so the width of the fields in the data structures vary. The "`.s`" version has the widest field, so it can safely be used on all chips, as shown in the example code following the data structure.

Data Stucture:

```
/**
 * cvmx_ipd_bp_prt_red_end
 *
 * IPD_BP_PRT_RED_END = IPD Backpressure Port RED Enable
 *
 * When IPD applies backpressure to a PORT and the corresponding bit in this
CSR is set,
 * the RED Unit will drop packets for that port.
 */
typedef union
{
    uint64_t u64;
    struct cvmx_ipd_bp_prt_red_end_s
    {
        uint64_t reserved_40_63          : 24;
        uint64_t prt_enb                 : 40;
    } s;
    struct cvmx_ipd_bp_prt_red_end_cn30xx
    {
        uint64_t reserved_36_63          : 28;
        uint64_t prt_enb                 : 36;
    } cn30xx;
    struct cvmx_ipd_bp_prt_red_end_cn30xx cn31xx;
    struct cvmx_ipd_bp_prt_red_end_cn30xx cn38xx;
    struct cvmx_ipd_bp_prt_red_end_cn30xx cn50xx;
    struct cvmx_ipd_bp_prt_red_end_s     cn52xx;
    struct cvmx_ipd_bp_prt_red_end_s     cn56xx;
    struct cvmx_ipd_bp_prt_red_end_cn30xx cn58xx;
} cvmx_ipd_bp_prt_red_end_t;
```

Example Use (from `cvmx-helper-util.c`) :

```
int cvmx_helper_setup_red(int pass_thresh, int drop_thresh)
{
    cvmx_ipd_portx_bp_page_cnt_t page_cnt;
    cvmx_ipd_bp_prt_red_end_t ipd_bp_prt_red_end;
    cvmx_ipd_red_port_enable_t red_port_enable;
    int queue;
    int interface;
    int port;

    <code omitted>

    ipd_bp_prt_red_end.u64 = 0;
    ipd_bp_prt_red_end.s.prt_enb = 0;
    cvmx_write_csr(CVMX_IPD_BP_PRT_RED_END, ipd_bp_prt_red_end.u64);

    <code omitted>

    return 0;
}
```

In the following CSR data structure, all the OCTEON models share identical CSR fields (shown in big endian format):

```
/**
 * cvmx_fpa_ctl_status
 *
 * FPA_CTL_STATUS = FPA's Control/Status Register
 *
 * The FPA's interrupt enable CSR.
 */
typedef union
{
    uint64_t u64;
    struct cvmx_fpa_ctl_status_s // the subfields
    {
        uint64_t reserved_18_63          : 46;
        uint64_t reset                   : 1;
        uint64_t use_ldt                 : 1;
        uint64_t use_stt                 : 1;
        uint64_t enb                     : 1;
        uint64_t mem1_err                : 7
        uint64_t mem0_err                : 7
    } s;
    struct cvmx_fpa_ctl_status_s         cn3020;
    struct cvmx_fpa_ctl_status_s         cn30xx;
    struct cvmx_fpa_ctl_status_s         cn31xx;
    struct cvmx_fpa_ctl_status_s         cn36xx;
    struct cvmx_fpa_ctl_status_s         cn38xx;
    struct cvmx_fpa_ctl_status_s         cn56xx;
    struct cvmx_fpa_ctl_status_s         cn58xx;
} cvmx_fpa_ctl_status_t;
```

2.3 Accessing CSRs via CSR Definitions and Data Structures

To read a CSR, call the function `cvmx_read_csr()`. Use this function to access the register by providing it with name of the CSR to be accessed (either a CSR name or CSR macro) as a parameter.

To write a CSR, use `cvmx_write_csr()`, providing the function with the name of the CSR or CSR macro. By using this function, the race condition shown in Section 4 – "Race Condition Accessing CSRs" is easily avoided.

To access a field inside the CSR, first read the CSR into the appropriate data structure, and then access the field.

Example 1: Read from a CSR, modify a field, and then write to the CSR

This example enables the FPA, the step are:

- Read the CVMX_FPA_CTL_STATUS CSR,
- Write a "1" to the Enable field (setting the Enable bit)
- Write the new value to the CSR

```
cvmx_fpa_ctl_status_t status;

status.u64 = cvmx_read_csr(CVMX_FPA_CTL_STATUS);
status.s.enb = 1;
cvmx_write_csr(CVMX_FPA_CTL_STATUS, status.u64);
```

Example 2: Read from a CSR using the CSR macro, which requires a pool number

In the case of CVMX_FPA_QUE*X*_AVAILABLE, the pool number is (also) provided as an argument to the CSR name macro. This number is then used in calculating the address of the appropriate FPA_QUE*x*_AVAILABLE address for this pool. For example:

```
cvmx_fpa_quex_available_t queue_size_CSR;

// Ask FPA the number of buffers available
printf("\nReading the FPA CSR to see how many buffers"
       " are available.\n");
queue_size_register.u64 =
                           cvmx_read_csr(CVMX_FPA_QUEX_AVAILABLE(MY_POOL));

// que_siz, a bit field, is declared uint64_t, but is modified by the
// compiler to be a unsigned int, thus is printed %u instead of %lu
printf("The number of buffers available in MY_POOL (pool# %u) = %u\n",
        MY_POOL, queue_size_register.s.que_siz);
```

The following table shows CSR field access for the FPA as of SDK 1.9. Note that, for the FPA unit, the subfields works well to access CSR fields.

Table 1: Accessing CSR Fields Using Subfields

CSR	Field Name	Access from Simple Executive typedef (union)For example: `cvmx_fpa_available_t avail;`	Field (N is one of pool 0-7) For example: `avail.s.que_siz`
FPA_CTL_STATUS	ENB	cvmx_fpa_ctl_status_t	s.enb
FPA_FPF*n*_SIZE	FPF_SIZ	cvmx_fpa_fpf0_size_t	s.fpf_siz
FPA_FPF*n*_MARKS	FPF_RD	cvmx_fpa_fpf_marks_t	s.fpf_rd
FPA_FPF*n*_MARKS	FPF_WR	cvmx_fpa_fpf_marks_t	s.fpf_wr
FPA_INT_ENB	FED0_SBE	cvmx_fpa_int_enb_t	s.fed0_sbe
FPA_INT_ENB	FED0_DBE	cvmx_fpa_int_enb_t	s.fed0_dbe
FPA_INT_ENB	FED1_SBE	cvmx_fpa_int_enb_t	s.fed1_sbe
FPA_INT_ENB	FED1_DBE	cvmx_fpa_int_enb_t	s.fed1_dbe
FPA_INT_ENB	Q*n*_UND	cvmx_fpa_int_enb_t	s.q*N*_und
FPA_INT_ENB	Q*n*_COFF	cvmx_fpa_int_enb_t	s.q*N*_coff
FPA_INT_ENB	Q*n*_PERR	cvmx_fpa_int_enb_t	s.q*N*_perr
FPA_INT_SUM	FED0_SBE	cvmx_fpa_int_sum_t	s.fed0_sbe
FPA_INT_SUM	FED0_DBE	cvmx_fpa_int_sum_t	s.fed0_dbe
FPA_INT_SUM	FED1_SBE	cvmx_fpa_int_sum_t	s.fed1_sbe
FPA_INT_SUM	FED1_DBE	cvmx_fpa_int_sum_t	s.fed1_dbe
FPA_INT_SUM	Q*n*_UND	cvmx_fpa_int_sum_t	s.q*N*_und
FPA_INT_SUM	Q*n*_COFF	cvmx_fpa_int_sum_t	s.q*N*_coff
FPA_INT_SUM	Q*n*_PERR	cvmx_fpa_int_sum_t	s.q*N*_perr
FPA_QUE*n*_PAGES_AVAILABLE	QUE_SIZ	cvmx_fpa_quex_available_t	s.que_siz
FPA_QUE*n*_PAGE_INDEX	PG_NUM	cvmx_fpa_quex_page_index_t	s.pg_num
FPA_QUE_EXP	EXP_INDX	cvmx_fpa_que_exp_t	s.exp_indx
FPA_QUE_EXP	EXP_QUE	cvmx_fpa_que_exp_t	s.exp_que
FPA_QUE_ACT	ACT_INDX	cvmx_fpa_que_act_t	s.act_indx
FPA_QUE_ACT	ACT_QUE	cvmx_fpa_que_act_t	s.act_que

2.3.1 Accessing CSR Fields Via OCTEON Model-Specific Typedefs

The `cvmx_mio_fus_dat3` data structure is a good example of the difference between the subfields structure and the chip-specific structure: `zip_info` (in CN63XX) and `zip_crip` conflict, so the chip-specific data structure must be used to access those fields. No other fields conflict, so all others are in the subfields.

The `cvmx_mio_fus_dat3` data structure (shown in big endian only):

```
/**
 * cvmx_mio_fus_dat3
 */
union cvmx_mio_fus_dat3
{
      uint64_t u64;
      struct cvmx_mio_fus_dat3_s // the subfields
      {
      uint64_t reserved_58_63      :  6;
      uint64_t pll_ctl             : 10;
      uint64_t dfa_info_dte        :  3;
      uint64_t dfa_info_clm        :  4;
      uint64_t reserved_40_40      :  1;
      uint64_t ema                 :  2;
      uint64_t efus_lck_rsv        :  1;
      uint64_t efus_lck_man        :  1;
      uint64_t pll_half_dis        :  1;
      uint64_t l2c_crip            :  3;
      uint64_t pll_div4            :  1;
      uint64_t reserved_29_30      : 2; // use neither of the conflicted names
      uint64_t bar2_en             : 1;
      uint64_t efus_lck            : 1;
      uint64_t efus_ign            : 1;
      uint64_t nozip               : 1;
      uint64_t nodfa_dte           : 1;
      uint64_t icache              : 24;
      } s;
      struct cvmx_mio_fus_dat3_cn30xx
      {
      uint64_t reserved_32_63                : 32;
      uint64_t pll_div4                      : 1;
      uint64_t reserved_29_30                : 2; // no name conflict
      uint64_t bar2_en                       : 1;
      uint64_t efus_lck                      : 1;
      uint64_t efus_ign                      : 1;
      uint64_t nozip                         : 1;
      uint64_t nodfa_dte                     : 1;
      uint64_t icache                        : 24;
      } cn30xx;
      struct cvmx_mio_fus_dat3_cn31xx
      {
      uint64_t reserved_32_63                : 32;
      uint64_t pll_div4                      : 1;
      uint64_t zip_crip                      : 2; // conflicts with zip_info
      uint64_t bar2_en                       : 1;
      uint64_t efus_lck                      : 1;
      uint64_t efus_ign                      : 1;
      uint64_t nozip                         : 1;
      uint64_t nodfa_dte                     : 1;
      uint64_t icache                        : 24;
      } cn31xx;
      struct cvmx_mio_fus_dat3_cn38xx
      {
      uint64_t reserved_31_63                 : 33;
      uint64_t zip_crip                       : 2; // conflicts with zip_info
      uint64_t bar2_en                        : 1;
      uint64_t efus_lck                       : 1;
      uint64_t efus_ign                       : 1;
      uint64_t nozip                          : 1;
      uint64_t nodfa_dte                      : 1;
      uint64_t icache                         : 24;
      } cn38xx;
      struct cvmx_mio_fus_dat3_cn38xx        cn50xx;
      struct cvmx_mio_fus_dat3_cn38xx        cn52xx;
      struct cvmx_mio_fus_dat3_cn38xx        cn56xx;
      struct cvmx_mio_fus_dat3_cn38xx        cn58xx;
      struct cvmx_mio_fus_dat3_cn63xx
      {
      uint64_t reserved_58_63                 : 6;
      uint64_t pll_ctl                        : 10;
      uint64_t dfa_info_dte                   : 3;
      uint64_t dfa_info_clm                   : 4;
      uint64_t reserved_40_40                 : 1;
      uint64_t ema                            : 2;
      uint64_t efus_lck_rsv                   : 1;
      uint64_t efus_lck_man                   : 1;
      uint64_t pll_half_dis                   : 1;
      uint64_t l2c_crip                       : 3;
      uint64_t reserved_31_31                 : 1;
      uint64_t zip_info                       : 2; // conflicts with zip_crip
      uint64_t bar2_en                        : 1;
      uint64_t efus_lck                       : 1;
      uint64_t efus_ign                       : 1;
      uint64_t nozip                          : 1;
      uint64_t nodfa_dte                      : 1;
      uint64_t reserved_0_23                  : 24;
      } cn63xx;
};
typedef union cvmx_mio_fus_dat3 cvmx_mio_fus_dat3_t;
```

The use of the CSR in the `octeon_model_get_string_buffer()` function illustrates how common fields are accessed as compared with chip-specific fields. In this example the `cvmx_mio_fus_dat2` union (a different data structure than the one shown above) is alternately accessed either via the `.s` or the chip-specific data structure name. (Note that all early-access version (PASS_*N*) references have been removed from the data structures shown in this chapter. See Section 10 – "Pass_*N* Specific Code").

```
if (fus_dat3.s.nodfa_dte)
    {
        if (fus_dat2.s.nocrypto) // use subfields data structure
            suffix = "CP";
        else
            suffix = "SCP";
    }
<code omitted>
case 4: /* CN57XX, CN56XX, CN55XX, CN54XX */
            if (fus_dat2.cn56xx.raid_en) // use chip-specific data structure
            {
                if (fus3.cn56xx.crip_1024k)
                    family = "55";
                else
                    family = "57";
                if (fus_dat2.cn56xx.nocrypto)
                    suffix = "SP";
                else
                    suffix = "SSP";
            }
<code omitted>
```

2.4 Types of Control and Status Registers (CSRs)

There are two different types of Control and Status Registers (CSRs): those on the I/O Bus and those on the Register Slave Logic (RSL) bus. The two types of CSRs are clearly marked in Table 2 – "CN58XX Physical Addresses".

2.4.1 CSRs on the Register Slave Logic (RSL) Bus

RSL stands for "Register Slave Logic". It's a low-bandwidth internal bus bridged from a block inside the IOB known as RML ("Register Master Logic"). There's an RSL sub-block inside each block that has RSL CSRs. Because it is a low-bandwidth bus, it is also a slow bus, meaning that the actual CSR update due to a write can occur a "long time" after the core has issued the store instruction. When customizing software, after writing to CSRs on the RSL bus, take the special steps which are explained in Section 4 – "Race Condition Accessing CSRs". Note that the RSL bus is not shown in the block diagrams, but it connects the RML block inside the I/O Bridge to the hardware units which are also connected to the I/O Bus.

2.4.2 CSRs on the I/O Bus

CSRs may also be on the I/O Bus. These CSRs are connected directly to the I/O Bus, and provide much faster access than RSL CSRs.

The I/O Bus is a non-coherent bus: the ordering and transaction completion rules are much looser than they are for the Coherent Memory Bus (CMB).

3 Physical Address Map and Device ID (DID)

This section includes a physical address map taken from the CN58XX *HRM*. This address map shows the RSL (Register Slave Logic) and I/O Bus CSRs (Configuration and Status Registers), and the Major DID and Sub-DID. This information is useful when looking into the details of the API.

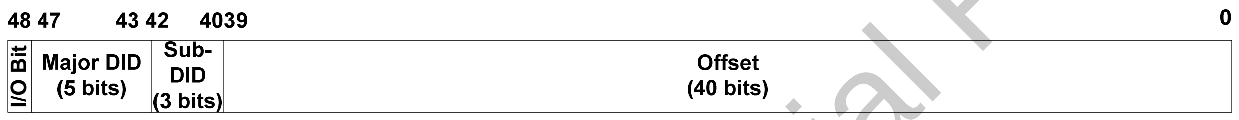
In particular the DID definitions are not usually visible to the user, but are included here because they are referenced in IOBDMA operations, as seen in Section 7.2.1 – "Starting the IOBDMA Operation".

Caveat: The CN58XX physical address map is included here(Table 2 – "CN58XX Physical Addresses") to help explain about RSL and I/O Bus CSRs and DIDs, and is not intended to be the actual reference used when writing code. For precise and up-to-date information see the current HRM for the desired OCTEON model. (Search on the string "physical addresses" in the HRM to locate the table.)

Each physical address is 49 bits long and consists of an I/O bit, Major DID, sub-DID, and offset, as shown in the next figure.

Figure 1: Physical Address Format

Physical Address Format



Physical addresses are 49 bits long (PABITS) per the MIPS specification, and consist of the following components:

- I/O Bit: Bit <48> of the physical address. If this bit is set to 1, the address is on the I/O bus If this bit is set to 0, the address is on the CMB (L2/DRAM).
- Major DID: Bits <47:43> of the physical address. These bits direct the request to the correct hardware block.
- Sub-DID: Bits <42:40> of the physical address. These bits direct the request within the hardware block selected by the Major DID.
- Offset: Bits <39:0>. Either CMB address or I/O Bus device address.

The following table show an example Physical Addresses Map, with the unit's Major DID and Sub-Did shown in the columns on the right. Acronyms used in this table and similar tables may be found in the *Glossary* chapter.

Table 2: CN58XX Physical Addresses

HW Unit	Physical Addresses		Memory or BUS - Only DRAM is Cached	Major DID	Sub - DID
	FROM	TO			
DR0	0x0 0000 0000 0000	0x0 0000 0FFF FFFF	DR0 DRAM (first 256 MB of DRAM) - CACHED		
BOOT	0x0 0000 1000 0000	0x0 0000 1FFF FFFF	Boot Bus (Uncached). Converted to 0x1 0000 1000 0000 - x1 0000 0FFF FFFF. See Note 1.	0x0	0x0
DR2	0x0 0000 2000 0000	0x0 0003 FFFF FFFF	DR2 DRAM (add DRAM memory above the first 512 MB) - CACHED		
DR1	0x0 0004 1000 0000	0x0 0004 1FFF FFFF	DR1 DRAM (second 256 MB of DRAM) - CACHED		
BOOT	0x1 0000 0000 0000	0x1 0000 FFFF FFFF	Boot Bus (Uncached) (converted from 0x0 0000 1000 0000 - 0x0 0000 1FFF FFFF). See Note 2.	0x0	0
CSR	0x1 0700 0000 0000	0x1 0700 0000 08FF	CIU and GPIO I/O Bus type CSRs	0x3	0
CSR	0x1 1800 0000 0000	0x1 1800 0000 1FFF	MIO BOOT, LED, FUS, TWSI, UART0, UART1, SMI RSL type CSRs	0x3	0
CSR	0x1 1800 0800 0000	0x1 1800 0800 1FFF	GMX0 RSL type CSRs	0x3	0
CSR	0x1 1800 1000 0000	0x1 1800 1000 1FFF	GMX1 RSL type CSRs	0x3	0
CSR	0x1 1800 2000 0000	0x1 1800 2000 001F	KEY RSL type CSRs	0x3	0
CSR	0x1 1800 2800 0000	0x1 1800 2800 01FF	FPA RSL type CSRs	0x3	0
CSR	0x1 1800 3000 0000	0x1 1800 3000 07FF	DFA RSL type CSRs	0x3	0
CSR	0x1 1800 3800 0000	0x1 1800 3800 00FF	ZIP RSL type CSRs	0x3	0
CSR	0x1 1800 4000 0000	0x1 1800 4000 000F	RNM RSL type CSRs	0x3	0
CSR	0x1 1800 5000 0000	0x1 1800 5000 1FFF	PKO RSL type CSRs	0x3	0
CSR	0x1 1800 5800 0000	0x1 1800 5800 1FFF	TIM RSL type CSRs	0x3	0
CSR	0x1 1800 8000 0000	0x1 1800 8000 07FF	L2C RSL type CSRs	0x3	0
CSR	0x1 1800 8800 0000	0x1 1800 8800 007F	LMC RSL type CSRs	0x3	0
CSR	0x1 1800 9000 0000	0x1 1800 9000 07FF	SPX0, SRX0, and STX0 RSL type CSRs	0x3	0
CSR	0x1 1800 9800 0000	0x1 1800 9800 07FF	SPX1, SRX1, and STX1 RSL type CSRs	0x3	0
CSR	0x1 1800 A000 0000	0x1 1800 A000 1FFF	PIP RSL type CSRs	0x3	0
CSR	0x1 1800 A800 0000	0x1 1800 A800 00FF	TRA RSL type CSRs	0x3	0
CSR	0x1 1800 B000 0000	0x1 1800 B000 03FF	ASX0 RSL type CSRs	0x3	0
CSR	0x1 1800 B800 0000	0x1 1800 B800 03FF	ASX1 RSL type CSRs	0x3	0
CSR	0x1 1800 F000 0000	0x1 1800 F000 07FF	IOB RSL type CSRs	0x3	0
PCI	0x1 1900 0000 0000	0x1 190F FFFF FFFF	PCI Bus Config/IACK/Special space (sub-DID 1)	0x3	1
PCI	0x1 1A00 0000 0000	0x1 1A0F FFFF FFFF	PCI Bus IO space (sub-DID 2)	0x3	2
PCI	0x1 1B00 0000 0000	0x1 1B0F FFFF FFFF	PCI Bus Memory space (sub-DID 3)	0x3	3
PCI	0x1 1C00 0000 0000	0x1 1C0F FFFF FFFF	PCI Bus Memory space (sub-DID 4)	0x3	4
PCI	0x1 1D00 0000 0000	0x1 1D0F FFFF FFFF	PCI Bus Memory space (sub-DID 5)	0x3	5
PCI	0x1 1E00 0000 0000	0x1 1E0F FFFF FFFF	PCI Bus Memory space (sub-DID 6)	0x3	6
PCI	0x1 1F00 0000 0000	0x1 1F0F FFFF FFFF	NPI I/O Bus type CSRs, doorbells (sub-DID 7)	0x3	7
KEY	0x1 2000 0000 0000	0x1 2000 0000 1FFF	KEY Memory operation		
FPA	0x1 2800 0000 0000	0x1 280F FFFF FFFF	FPA Pool 0 Allocate/Free operations	0x5	0
FPA	0x1 2900 0000 0000	0x1 290F FFFF FFFF	FPA Pool 1 Allocate/Free operations	0x5	1
FPA	0x1 2A00 0000 0000	0x1 2A0F FFFF FFFF	FPA Pool 2 Allocate/Free operations	0x5	2
FPA	0x1 2B00 0000 0000	0x1 2B0F FFFF FFFF	FPA Pool 3 Allocate/Free operations	0x5	3
FPA	0x1 2C00 0000 0000	0x1 2C0F FFFF FFFF	FPA Pool 4 Allocate/Free operations	0x5	4
FPA	0x1 2D00 0000 0000	0x1 2D0F FFFF FFFF	FPA Pool 5 Allocate/Free operations	0x5	5
FPA	0x1 2E00 0000 0000	0x1 2E0F FFFF FFFF	FPA Pool 6 Allocate/Free operations	0x5	6
FPA	0x1 2F00 0000 0000	0x1 2F0F FFFF FFFF	FPA Pool 7 Allocate/Free operations	0x5	7
DFA	0x1 3700 0000 0000	0x1 3707 FFFF FFFF	DFA I/O Bus type CSRs and operations	0x6	7
ZIP	0x1 3800 0000 0000	0x1 3800 0000 0007	ZIP doorbell store operations	0x7	0
RNG	0x1 4000 0000 0000	0x1 4000 0000 07FF	RNG Load/IOBDMA operations	0x8	0

HW Unit	Physical Addresses		Memory or BUS - Only DRAM is Cached	Major DID	Sub - DID
CSR	0x1 4F00 0000 0000	0x1 4F00 0000 07FF	IPD I/O Bus type CSRs	▨	▨
PKO	0x1 5200 0000 0000	0x1 5200 0003 FFFF	PKO doorbell store operations	0xA	2
SSO	0x1 6000 0000 0000	0x1 600F FFFF FFFF	SSO getwork loads/IODBMAs, store work operations	0xC	0
SSO	0x1 6100 0000 0000	0x1 610F FFFF FFFF	SSO status loads, store work operations (sub-DID 1)	0xC	1
SSO	0x1 6200 0000 0000	0x1 6200 0000 FFFF	SSO memory loads (sub-DID 2)	0xC	2
SSO	0x1 6300 0000 0000	0x1 630F FFFF FFFF	SSO index loads, store operations (sub-DID 3)	0xC	3
SSO	0x1 6300 0000 0000	0x1 6300 0000 0007	SSO NullRd loads (sub-DID 4)	0xC	4
CSR	0x1 6700 0000 0000	0x1 6700 0000 03FF	SSO I/O Bus type CSRs	▨	▨
FAU	0x1 F000 0000 0000	0x1 F00F FFFF FFFF	FAU Operations	0x1E	0

Notes

Note1: The first one is primarily for booting (0x0 0000 1000 0000 - 0x0 0000 1FFF FFFF), and is required by the MIPS specification, which says that execution shall start at `0xBFC00000` coming out of reset. But it's teeny. The rest of the boot bus is reachable via the I/O space (0x1 0000 1000 0000 - x1 0000 0FFF FFFF). It's still uncached.

The following table shows the conversion between DID and physical address bits (address bits are shown in binary and in hex).

Table 3: Converting a DID and Sub-DID into a Physical Address (CN58XX Example)

HARDWARE UNIT	MAJOR DID (Decimal)	MAJOR DID (HEX)	MAJOR DID (5 bits: <47:43>) (binary)	SUB-DID (3 bits: <42:40>)	I/O Bit (bit 48)	Major-DID	Major-DID	Major-DID	Major-DID	Major-DID	Sub-DID (bit 42)	Sub-DID (bit 41)	Sub-DID (bit 40)	High 9 bits of physical Address ("X" means 0 or 1)	Device Addressed
Boot Bus	**0**	**0x0**	00000	0	0,1	0	0	0	0	0	0	0	0	`0xX_00XX`	Boot bus uncached I/O space
PCI / PCI-X	**3**	**0x3**	00011	0	1	0	0	0	1	1	0	0	0	`0x1_18XX`	RML CSRs
PCI / PCI-X	**3**	**0x3**	00011	1	1	0	0	0	1	1	0	0	1	`0x1_19XX`	PCI Bus config/IACK/Special Space
PCI / PCI-X	**3**	**0x3**	00011	2	1	0	0	0	1	1	0	1	0	`0x1_1AXX`	PCI Bus I/O space
PCI / PCI-X	**3**	**0x3**	00011	3	1	0	0	0	1	1	0	1	1	`0x1_1BXX`	PCI Bus Memory Space
PCI / PCI-X	**3**	**0x3**	00011	4	1	0	0	0	1	1	1	0	0	`0x1_1CXX`	PCI Bus Memory Space
PCI / PCI-X	**3**	**0x3**	00011	5	1	0	0	0	1	1	1	0	1	`0x1_1DXX`	PCI Bus Memory Space
PCI / PCI-X	**3**	**0x3**	00011	6	1	0	0	0	1	1	1	1	0	`0x1_1EXX`	PCI Bus Memory Space
PCI / PCI-X	**3**	**0x3**	00011	7	1	0	0	0	1	1	1	1	1	`0x1_1EXX`	NPI I/O Bus type CSRs, doorbells
FPA	**5**	**0x5**	00101	0	1	0	0	1	0	1	0	0	0	`0x1_28XX`	FPA Pool 0
FPA	**5**	**0x5**	00101	1	1	0	0	1	0	1	0	0	1	`0x1_29XX`	FPA Pool 1

HARDWARE UNIT	MAJOR DID (Decimal)	MAJOR DID (HEX)	MAJOR DID (5 bits: <47:43>) (binary)	SUB-DID (3 bits: <42:40>)	I/O Bit (bit 48)	Major-DID	Major-DID	Major-DID	Major-DID	Major-DID	Sub-DID (bit 42)	Sub-DID (bit 41)	Sub-DID (bit 40)	High 9 bits of physical Address ("X" means 0 or 1)	Device Addressed
FPA	**5**	**0x5**	00101	2	1	0	0	1	0	1	0	1	0	`0x1_2AXX`	FPA Pool 2
FPA	**5**	**0x5**	00101	3	1	0	0	1	0	1	0	1	1	`0x1_2BXX`	FPA Pool 3
FPA	**5**	**0x5**	00101	4	1	0	0	1	0	1	1	0	0	`0x1_2CXX`	FPA Pool 4
FPA	**5**	**0x5**	00101	5	1	0	0	1	0	1	1	0	1	`0x1_2DXX`	FPA Pool 5
FPA	**5**	**0x5**	00101	6	1	0	0	1	0	1	1	1	0	`0x1_2EXX`	FPA Pool 6
FPA	**5**	**0x5**	00101	7	1	0	0	1	0	1	1	1	1	`0x1_2FXX`	FPA Pool 7
DFA	**6**	**0x6**	00110	7	1	0	0	1	1	0	1	1	1	`0x1_37XX`	DFA I/O Bus type CSRs and operations.
ZIP	**7**	**0x7**	00111	0	1	0	0	1	1	1	0	0	0	`0x1_38XX`	ZIP doorbell store operations.
RNG / RNM	**8**	**0x8**	01000	0	1	0	1	0	0	0	0	0	0	`0x1_40XX`	RNG Load/IOBDMA operations
PKO	**10**	**0xA**	01010	2	1	0	1	0	1	0	0	1	0	`0x1_52XX`	PKO doorbell store operations
SSO (POW)	**12**	**0xC**	01100	0	1	0	1	1	0	0	0	0	0	`0x1_60XX`	getwork loads/IOBDMAs, store work operations
SSO (POW)	**12**	**0xC**	01100	1	1	0	1	1	0	0	0	0	1	`0x1_61XX`	status loads, store work operations
SSO (POW)	**12**	**0xC**	01100	2	1	0	1	1	0	0	0	1	0	`0x1_62XX`	memory loads
SSO (POW)	**12**	**0xC**	01100	3	1	0	1	1	0	0	0	1	1	`0x1_63XX`	Index loads, store operations
SSO (POW)	**12**	**0xC**	01100	4	1	0	1	1	0	0	1	0	0	`0x1_64XX`	NullRd loads
FAU	**30**	**0x1E**	11110	0	1	1	1	1	1	0	0	0	0	`0x1_F0XX`	FAU Accesses

4 Race Condition Accessing CSRs

If the application does not use the `cvmx_csr_write()` function provided in the SDK, it is important to be aware of this race condition and design code accordingly.

Race conditions can happen if a RSL CSR write (for example, to configure the FPA) has not completed before the unit is used (for example, populating a FPA pool).

To ensure an RSL CSR write has completed, read an RSL CSR to guarantee that the write has completed, otherwise a race condition may occur. (The `cvmx_write_csr()` function performs the fast and harmless `CVMX_MIO_BOOT_BIST_STAT` RSL CSR read to ensure the CSR write has completed.)

This section contains the technical details of the race condition.

This problem is especially applicable to the FPA unit: the CSR initialization must complete before the FPA can be used. After CSR initialization, the allocated memory buffers are freed to the FPA to populate the FPA buffer pool. The CSR initialization write travels down a different bus (RSL bus) than the free buffer write (I/O Bus). If the CSR write has not completed before the buffers are freed, a race condition may occur. In particular, if the buffers are freed to the FPA before the FPA is enabled, the FPA will be incorrectly initialized.

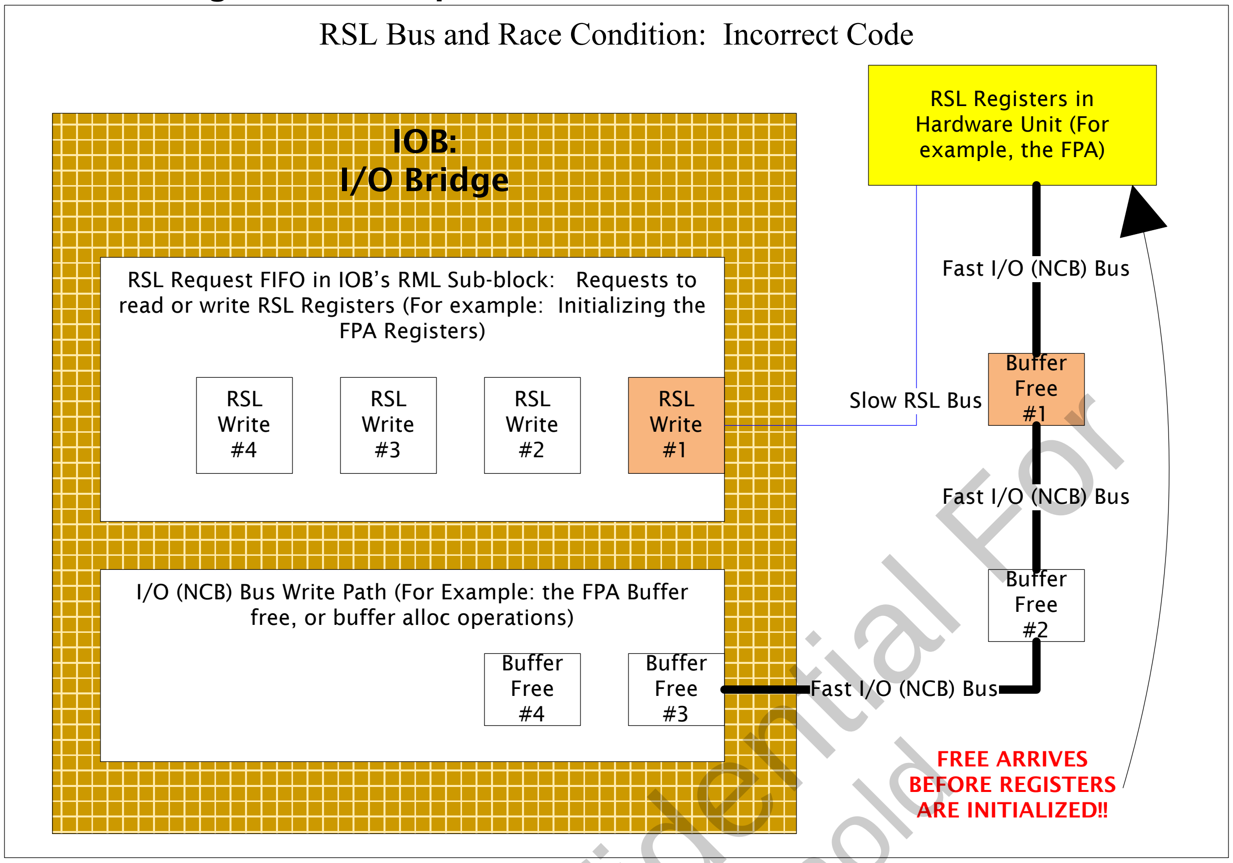
4.1 Technical Details of the Race Condition

Details:

1. Most configuration CSRs are RSL type CSRs.
2. The I/O Bridge controls access to both the RSL Bus and the I/O Bus.
3. Some CSR reads and writes, such as unit initialization, go over the RSL Bus.
4. The requests to allocate and free FPA buffers go over the I/O Bus.
5. The RSL Bus is much slower than the I/O Bus.
6. This can create a race condition which is particularly problematic for the FPA.
7. Because of this, after initializing the FPA RSL CSRs, verify that the RSL CSR write has completed before freeing any buffers to the FPA.
8. To verify that all the RSL CSR writes have completed, after sending all the "writes" to initialize the RSL CSRs, send a "read" to read one of the RSL CSRs. (Note: the API function `cvmx_write_csr()` will handle this correctly.)
9. Otherwise the `buffer_free` transaction used to populate the buffer pool may arrive at the FPA before the CSRs are initialized, including the CSR write to enable the FPA. The data in the FPA could then be incorrect.

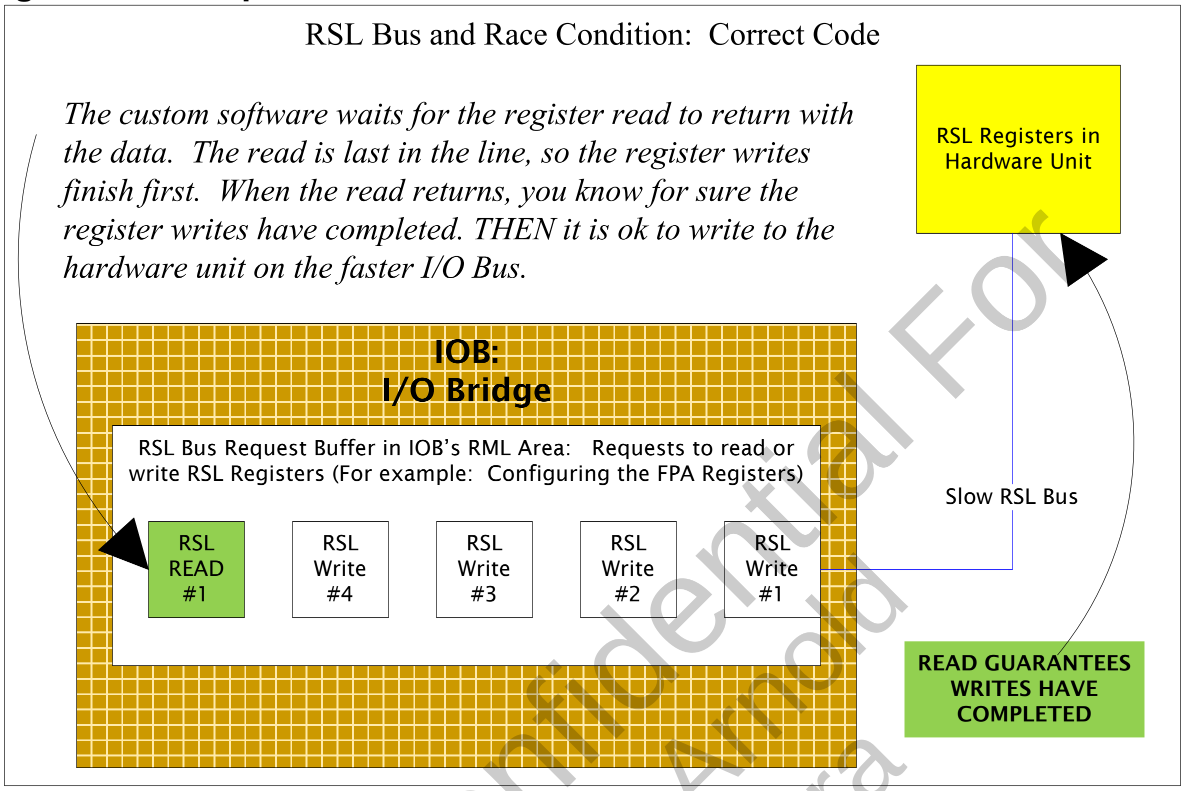
In the figure below, the `buffer_free` transaction arrives at the FPA before the CSRs are initialized. In this incorrect code, the software did not wait for the initialization CSR write transaction to complete before populating the FPA pool by freeing buffers to the pool.

Figure 2: Example: RSL Bus vs. I/O Bus Race Condition


RSL Bus and Race Condition: Incorrect Code
IOB:
I/O Bridge
RSL Request FIFO in IOB’s RML Sub-block: Requests to read or write RSL Registers (For example: Initializing the FPA Registers)
RSL Write #4
RSL Write #3
RSL Write #2
RSL Write #1
I/O (NCB) Bus Write Path (For Example: the FPA Buffer free, or buffer alloc operations)
Buffer Free #4
Buffer Free #3
RSL Registers in Hardware Unit (For example, the FPA)
Fast I/O (NCB) Bus
Slow RSL Bus
Buffer Free #1
Fast I/O (NCB) Bus
Buffer Free #2
Fast I/O (NCB) Bus
FREE ARRIVES BEFORE REGISTERS ARE INITIALIZED!!

In correct code, all the initialization CSRs are written *before* freeing any buffers to the pools. The figure below shows the read request following all the write requests. When the read request returns, all the prior writes have completed. THEN it is okay to populate the pools by freeing the buffers to the FPA.

Figure 3: Example: How to Avoid the RSL Bus vs. I/O Bus Race Condition


RSL Bus and Race Condition: Correct Code
The custom software waits for the register read to return with the data. The read is last in the line, so the register writes finish first. When the read returns, you know for sure the register writes have completed. THEN it is ok to write to the hardware unit on the faster I/O Bus.
RSL Registers in Hardware Unit
IOB: I/O Bridge
RSL Bus Request Buffer in IOB's RML Area: Requests to read or write RSL Registers (For example: Configuring the FPA Registers)
RSL READ #1
RSL Write #4
RSL Write #3
RSL Write #2
RSL Write #1
Slow RSL Bus
READ GUARANTEES WRITES HAVE COMPLETED

4.2 Avoiding the Race Condition

To avoid the race condition, either read a RSL CSR after writing a RSL CSR, or use the SDK function `cvmx_write_csr()` (`executive/cvmx-access-native.h`):

```
static inline void cvmx_write_csr(uint64_t csr_addr, uint64_t val)
{
    cvmx_write64_uint64(csr_addr, val);

    /* Perform an immediate read after every write to an RSL CSR to force
        the write to complete.  It doesn't matter what RSL read we do, so we
        choose CVMX_MIO_BOOT_BIST_STAT because it is fast and harmless */
    if ((csr_addr >> 40) == (0x800118)) // See the note in the text below
        cvmx_read64_uint64(CVMX_MIO_BOOT_BIST_STAT);
}
```

Note that this code is checking to see if the write is to an RSL CSR. The RSL CSRs contain the physical address pattern **0x1 1800** 0000 0000 (bits <48:0>) (see Table 2 – "CN58XX Physical Addresses"). When addressing an RSL CSR using an *xphys* virtual address, the bits <63:49> are `0x800`, resulting in an address of `0`**`x8001 18`**`00 0000 0000`. Thus the test to see if the address is an RSL CSR address is to look for the distinctive `0x800118` pattern in address bits <63:40>. (Note that the I/O space is selected if bit 48 of the physical address is "1". Physical memory is selected if bit 48 of the physical address is "0".) See the *Software Overview* chapter for more information on virtual and physical memory addressing.

5 Race Condition if `buffer_free` before Buffer Writes Complete

A `syncws` (`CVMX_SYNCWS`) or `syncw` (`CVMX_SYNCW`) instruction should be issued before freeing an FPA buffer or a race condition may occur. This section contains technical details about the race condition which can occur if the `syncw/syncws` is omitted. The API function `cvmx_fpa_free()` automatically manages this potential race condition.

This condition occurred in the following real-world example, and is why the `cvmx_fpa_free()` function automatically handles this potential race condition:

1. Create a new packet with a bunch of writes
2. Attempt to allocate a PKO command buffer, it fails
3. Free the Packet Data buffer since we can't send it
4. IPD allocates the same Packet Data buffer
5. Original Packets write complete

5.1 Technical Details about the Race Condition

When software writes to a FPA-managed buffer (i.e. a store transaction to a L2/DRAM location), the store transaction goes initially into the core's Write Buffer, where it may be temporarily delayed before being flushed to the CMB on its way to L2/DRAM.

When software frees a buffer, the `buffer_free` transaction is a store to the I/O Bus. The I/O bus store goes initially into the core's Write Buffer, but it is *not* delayed there. It goes immediately out to the I/O Bus.

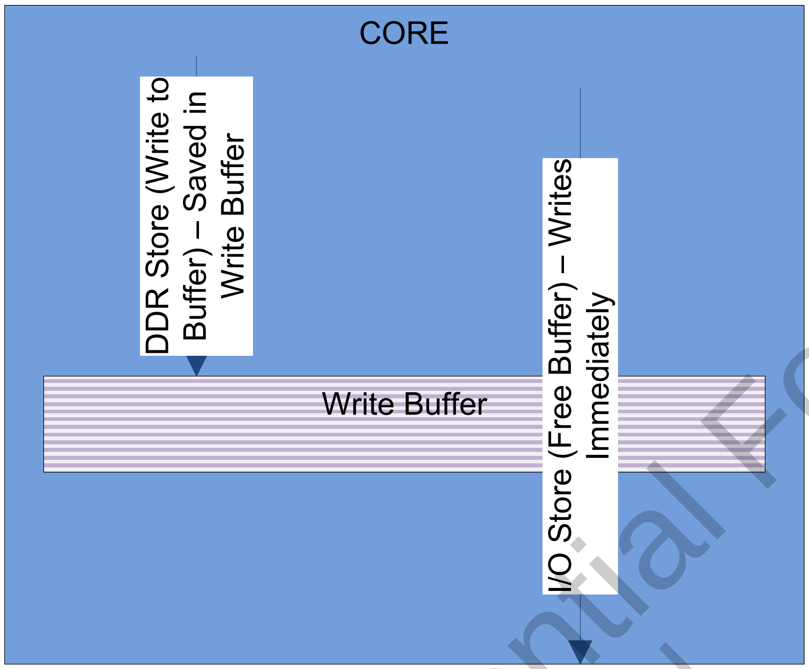
The difference in how these two store transactions are handled by the Write Buffer logic will cause the `buffer_free` transaction to be committed to the CMB before the store transaction to the buffer has been flushed to L2/DRAM.

The buffer can then potentially be re-allocated, and the *new* owner can write to it. Then, later, the stale store completes, overwriting the new owner's data.

This specific race condition problem only applies to store transactions from the cores. Other hardware units, such as the IPD (which writes to Packet Data buffers), do not use a Write Buffer to buffer store transactions.

Figure 4: Write to L2/DRAM versus Write to I/O Space (CSRs)

Write to L2/DRAM versus Write to I/O Space (CSRs)

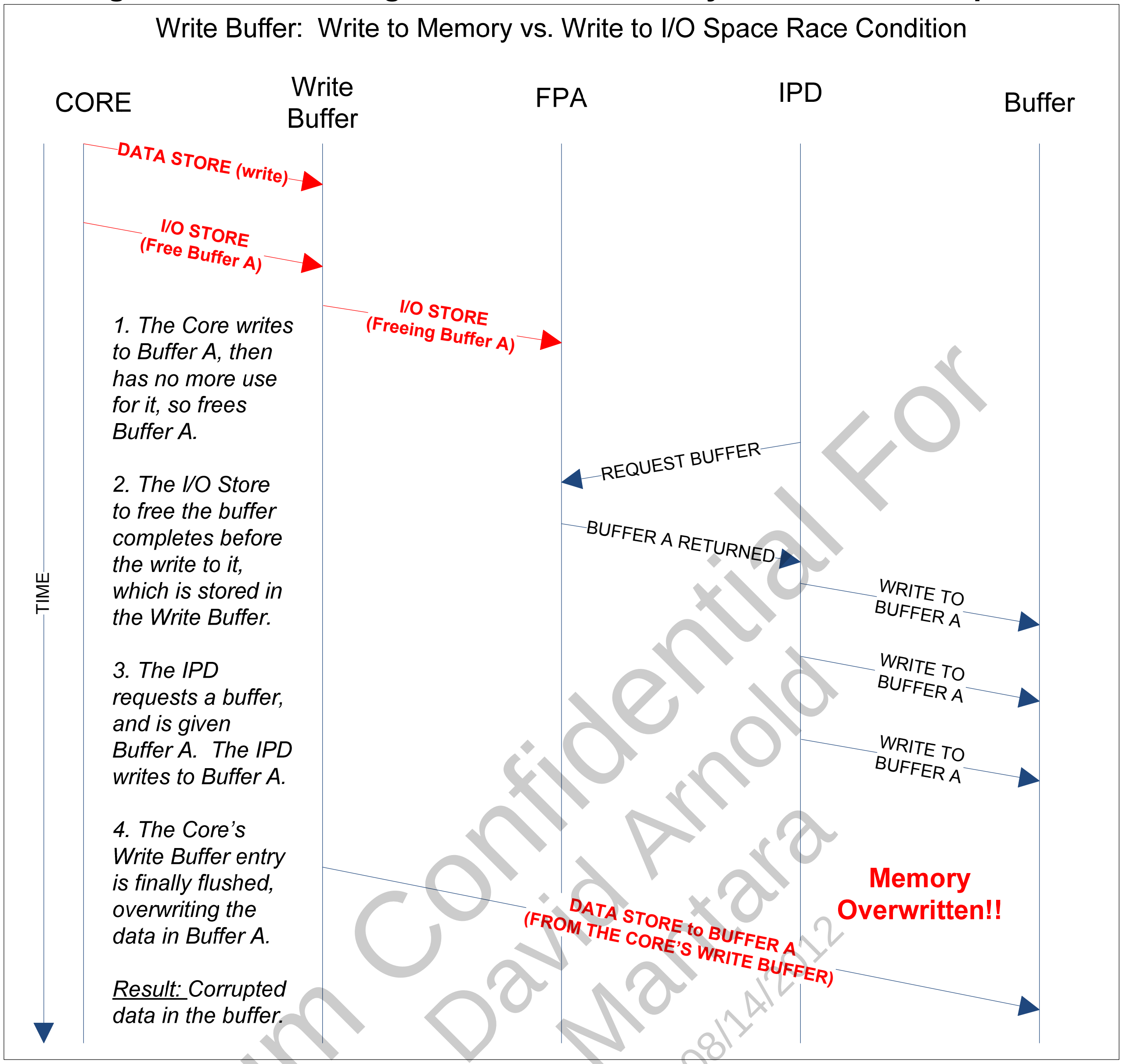

CORE
DDR Store (Write to Buffer) – Saved in Write Buffer
Write Buffer
I/O Store (Free Buffer) – Writes Immediately

Writes to DRAM *are* buffered in the core's Write Buffer. Writes to I/O Space, such as the write to free the buffer are *not* stored in the core's Write Buffer.

The **`write`** operaton to a buffer is a write to L2/DRAM, and is buffered in the core's Write Buffer. The **`buffer_free`** operation is a write transaction to I/O Space. Thus the **`write`** transaction to free the buffer will complete before the **`write`** to the buffer.

The buffer can then be re-allocated, the new owner can **`write`** to it, then the stale **`write`** completes, overwriting the new owner's data.

Figure 5: Ladder Diagram: Write to Memory vs. Write to I/O Space


Write Buffer: Write to Memory vs. Write to I/O Space Race Condition
CORE
Write Buffer
FPA
IPD
Buffer
TIME
DATA STORE (write)
I/O STORE (Free Buffer A)
I/O STORE (Freeing Buffer A)
REQUEST BUFFER
BUFFER A RETURNED
WRITE TO BUFFER A
WRITE TO BUFFER A
WRITE TO BUFFER A
DATA STORE to BUFFER A (FROM THE CORE'S WRITE BUFFER)
Memory Overwritten!!

1. The Core writes to Buffer A, then has no more use for it, so frees Buffer A.

2. The I/O Store to free the buffer completes before the write to it, which is stored in the Write Buffer.

3. The IPD requests a buffer, and is given Buffer A. The IPD writes to Buffer A.

4. The Core's Write Buffer entry is finally flushed, overwriting the data in Buffer A.

Result: Corrupted data in the buffer.

5.2 Avoiding the Race Condition

To avoid this race condition, either use the SDK command `cvmx_fpa_free()` instead of the `cvmx_fpa_free_nosync()` command, or the `syncw/syncws` instruction before freeing the buffer.

The CVMX_SYNCWS used in the SDK function cvmx_fpa_free() is shown in the following code:

```
static inline void cvmx_fpa_free(void *ptr, uint64_t pool, uint64_t
num_cache_lines)
{
    cvmx_addr_t newptr;

    newptr.u64 = cvmx_ptr_to_phys(ptr);
    newptr.sfilldidspace.didspace =
        CVMX_ADDR_DIDSPACE(CVMX_FULL_DID(CVMX_OCT_DID_FPA,pool));

    /* Make sure that any previous stores to memory go out before we     */
    /* free this buffer. This also serves as a memory barrier to prevent */
    /* GCC from reordering operations to after the free.                 */
    CVMX_SYNCWS;

    /* num_cache_lines is the number of DWB cache lines                  */
    cvmx_write_io(newptr.u64, num_cache_lines);
}
```

6 Don't Write Back (DWB) Operations

Most users won't need the information in this section: they will meet their performance needs with the default values set in the API. The information in this section is for users who need optimum performance and are tuning their system for a specific application.

The Don't Write Back (DWB) operation is used to clear the dirty bit in a L2 cache line, so that the cache line contents are not flushed to DRAM. (See the *Software Overview* chapter in the "Caching" section for information about caching.)

6.1 DWB Operation Effects

When a buffer is accessed, its contents are loaded from DRAM to the shared L2 cache (if it is not already in L2 cache) and then into the core-local L1 Dcache. A write to the buffer causes the corresponding L2 cache line to be marked *dirty* (the *dirty bit* is set). The dirty bit tells the L2 cache controller to write the cache line back to DRAM before the cache line is reused (which can be at any time, depending on memory access requests from other cores and accelerators).

The Don't Write Back (DWB) operation tells the L2 Cache Controller to clear the dirty bit in the corresponding L2 cache line, so that the data in the cache line is not written back to DRAM. For example, the DWB operation is used to discard the data in Packet Data buffers after the packet has been transmitted because it is okay to discard data which is no longer needed. This can improve performance by avoiding unnecessary operations.

The DWB command is most often used when freeing a FPA-managed buffer (such as a Packet Data buffer). Hardware units such as the PKO can be configured to automatically issue the DWB commands when freeing FPA-managed buffers.

DWB command can be initiated using:

1. The `buffer_free` operation, which includes the number of 128-byte cache lines to not write back). The `buffer_free` operation can be issued by:
   - software running on the core (via `cvmx_fpa_free()`)
   - the DFA hardware unit (for Command buffers)
   - the PKO hardware unit (for Packet Data buffers and PKO Command buffers)
   - the TIMER hardware unit (for Timer Chunk buffers)
   - the PCI/PCIe DMA Engines (for Command buffers)
   - the RAID hardware unit (for Command buffers)
   - the ZIP hardware unit (for Command buffers)
2. The `pref 29` instruction (`CVMX_DONT_WRITE_BACK`), issued by a core

6.2 The `buffer_free` Operation and DWB

The I/O Bridge contains a DWB Engine which intercepts `buffer_free` operations destined for the FPA. If the `buffer_free` operation specifies a non-zero number of cache lines to not write back, then the DWB Engine issues these commands to the L2 cache controller. The L2 cache controller clears the dirty bit in the selected cache line. After issuing the DWB commands, the DWB Engine forwards the `buffer_free` operation to the FPA. (See the *HRM* (search on "Don't-Write-Back Engine" (note the hyphens)) for details on the I/O Bridge's DWB Engine.)

Note that specifying the number of cached lines to DWB does not guarantee that the dirty bits for those cache lines will be cleared:

- The DWB Engine can only buffer a limited number of `buffer_free` operations. If there is no more space inside the DWB engine, the `buffer_free` operation is sent to the FPA without the DWB commands being issued.
- The exact order of the L2 cache-line write and the "clear dirty bit" operations will depend on system timing. If the DWB command arrives after the cache line is flushed, the DWB command will be discarded (hardware will ensure that the DWB command will not affect the cache line after the cache line is flushed, even if it is reused).

See Figure 7 – "Ladder Diagram: `buffer_free` and DWB" for a ladder diagram showing DWB operations.

6.2.1 The `buffer_free` Operation Issued by Cores

Software can use the `cvmx_fpa_free()` function to free a buffer to the specified FPA pool. The third argument is the number of cache lines in the buffer to not write back. In the example below DWB is set to one cache line:

```
cvmx_fpa_free(buf, CVMX_FPA_PACKET_POOL, 1);
```

Performance Tips:

- Most applications should not use DWB from software running on the core to avoid extra DWB transactions on the bus, but a small percentage gain performance by using it. To avoid using DWB set the DWB argument to `cvmx_fpa_free()` to zero.
- If buffers are not modified, do not specify DWB for any cache lines: there is no need to send DWB commands to clear a dirty bit which is not set.
- Various configurations can be tested to determine the best configuration for the application by conducting performance testing with the anticipated traffic load.

6.2.2 The `buffer_free` Operation Issued by Other Hardware Units

Hardware Units can be configured (via a CSR field) to specify the number of cache lines to DWB when executing the `buffer_free` operation. Simple Executive software configures the DWB variable for each hardware unit to a default value which provide the best system performance for most applications.

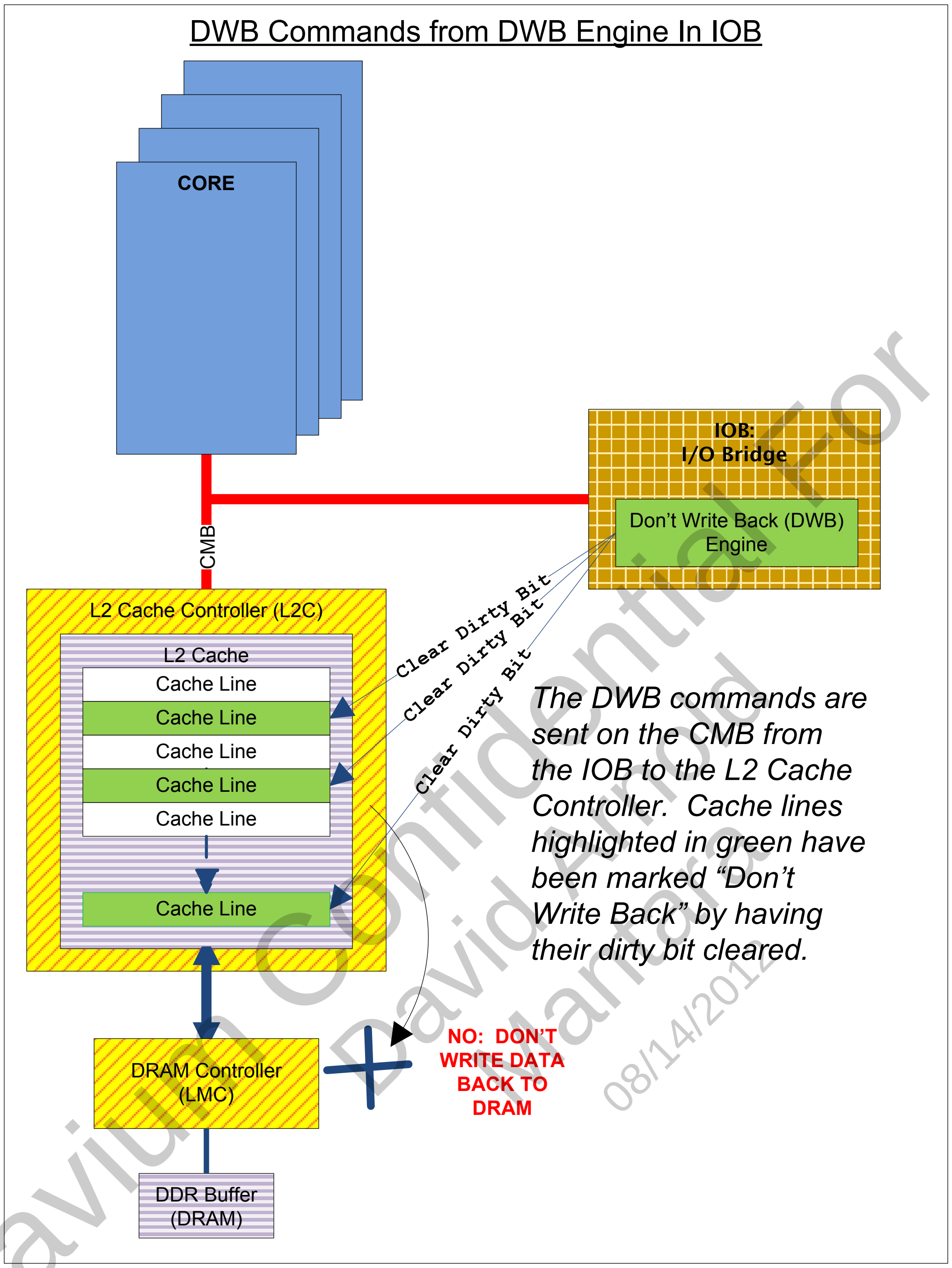
For example, the PKO automatically frees PKO Command Buffers when it has processed all the commands contained in the buffer, and may optionally free the Packet Data Buffers when they are no longer needed. Whether the PKO specifies DWB when freeing these buffers is controlled by the PKO CSR field `PKO_REG_FLAGS[ENA_DWB]`. When this field is set to 1, the PKO will issue DWB command when freeing the buffers. Simple Executive sets this field to 1 by default:

```
void cvmx_pko_enable(void)
{
    <code omitted>
    flags.s.ena_dwb = 1;
    <code omitted>
}
```

Performance Tips:

- Enabling the hardware units other than the cores to use DWB is strongly recommended: it ensures that DRAM bandwidth is not wasted writing discarded data from L2 cache to memory, and frees the L2 cache line for immediate reuse. For the most commonly used buffer types, the API will set the DWB to recommended values. Modifying the values from the default may impact system performance and requires the application developer to have extensive experience programming OCTEON applications.
- Various configurations can be tested to determine the best configuration for the application by conducting performance testing with the anticipated traffic load.

Figure 6: Don’t Write Back Commands from IOB


DWB Commands from DWB Engine In IOB
CORE
CMB
IOB:
I/O Bridge
Don’t Write Back (DWB) Engine
L2 Cache Controller (L2C)
L2 Cache
Cache Line
Cache Line
Cache Line
Cache Line
Cache Line
Cache Line
Clear Dirty Bit
Clear Dirty Bit
Clear Dirty Bit
The DWB commands are sent on the CMB from the IOB to the L2 Cache Controller. Cache lines highlighted in green have been marked “Don’t Write Back” by having their dirty bit cleared.
NO: DON’T WRITE DATA BACK TO DRAM
DRAM Controller (LMC)
DDR Buffer (DRAM)

6.3 The pref Instruction and DWB

A core can also issue a DWB command to the L2 Cache Controller by executing the Cavium Networks-specific `pref 29` (prefetch) instruction. This instruction does not perform the normal prefetch action of reading data from DRAM to cache, but instead causes the L2 Cache Controller to clear the dirty bit for the targeted cache block (only one cache block). The Simple Executive define `CMVX_DONT_WRITE_BACK` is used to issue this instruction.

Performance Tips:

- This instruction generally does *not* improve system performance because it adds transactions to the CMB, unnecessarily offloading the bus from L2 to DRAM.
- When using the `pref` instruction to issue DWB commands from software, only issue DWB instructions for cache lines which are dirty. (If the cache line has not been written to (is not dirty), then no DWB instruction is needed to clear the dirty bit.) For instance, if the buffer is four cache lines in size (512 bytes), but only the first two cache lines (256 bytes) are dirty, then only two `pref 29` instructions are needed, not four. This may improve system performance by limiting the number of DWB commands sent.
- If buffers are not modified, do not specify DWB for any cache lines: there is no need to send DWB commands to clear a dirty bit which is not set.
- Various configurations can be tested to determine the best configuration for the application by conducting performance testing with the anticipated traffic load.

For details on the Cavium Networks-specific `pref` (prefetch) instruction, see the *HRM* in the section on "CPU Load, Store, Memory, and Control Instructions". Also see the *Essential Topics* chapter for an example of using a similar instruction, the `pref 0` (`CVMX_PREFETCH`) instruction, to load data from DRAM to both L1 Dcache and L2 cache. The `pref` instruction will convert the virtual address into a physical address (the instruction description in the HRM does into detail).

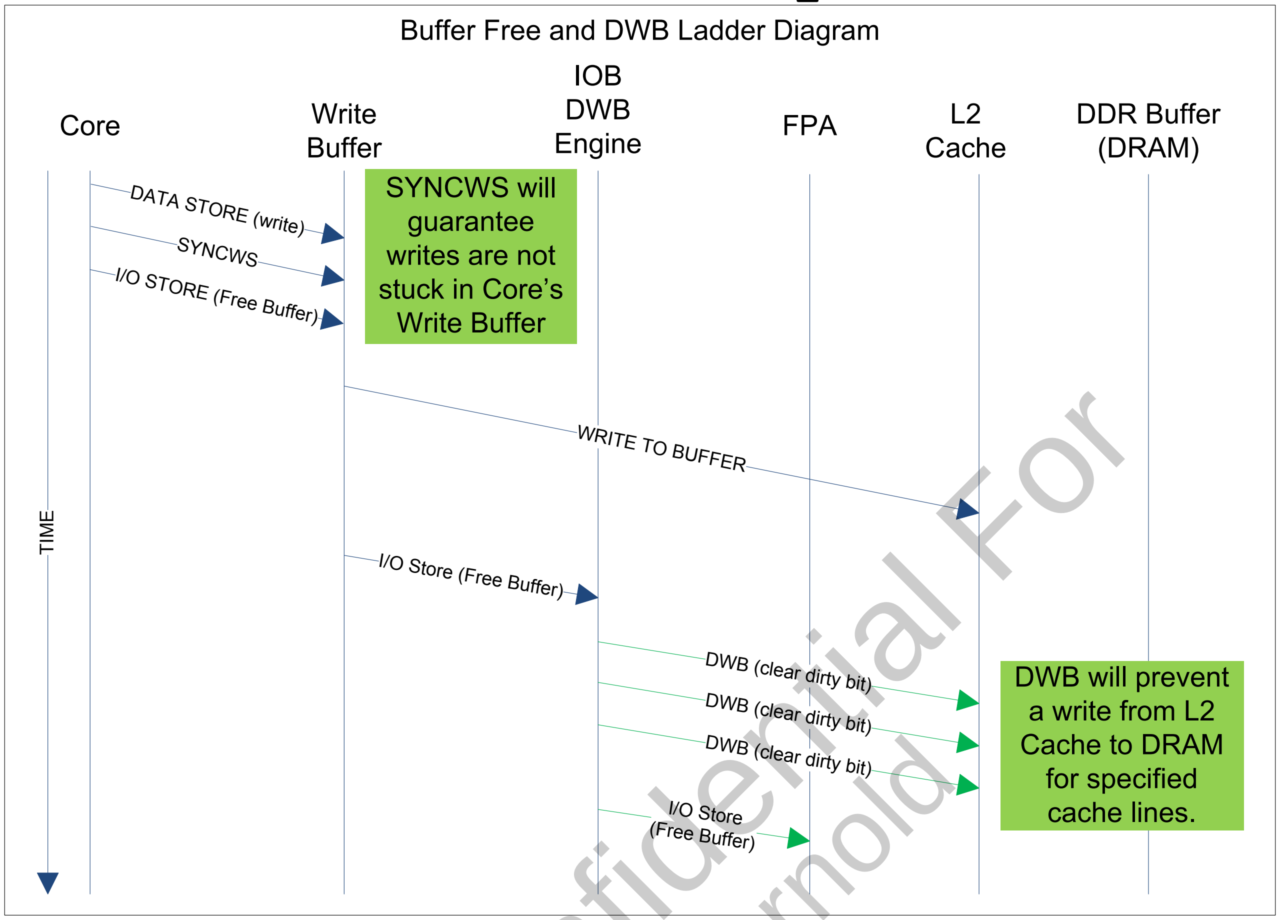
6.4 DWB and the `syncws` Instruction

The following ladder diagram shows how DWB can be used to prevent writes from being flushed to DRAM:

- The core writes to the buffer
- The core issues a `syncws` instruction causing writes which are buffered in the Write Buffer to flush to L2 cache
- The core performs the `buffer_free` operation, specifying the number of cache lines to DWB
- The I/O Bridges DWB Engine intercepts the `buffer_free` operation, and issues DWB commands to the L2 Cache Controller, then forwards the `buffer_free` operation to the FPA
- The DWB commands clear each L2 cache line's dirty bit so that the buffer contents are not written from L2 cache to DRAM

(Note that issuing DWB commands does not guarantee that the cache line will not be flushed. The exact system timing may result in the cache line being flushed before the DWB commands arrive.)

Figure 7: Ladder Diagram: `buffer_free` and DWB



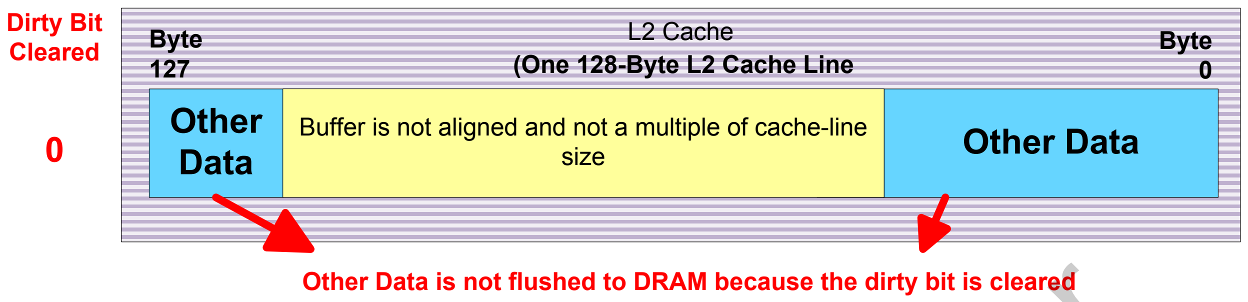
6.5 DWB and Buffer Alignment and Size

This issue does not usually apply because users use the SDK APIs to allocate buffers. This information is provided for users who are creating a custom API.

If the SDK is not used to allocate buffers, and the user did not take care to allocate a buffer which is aligned on cache-line size and also a multiple of cache-line size, then problems can occur when DWB is used. DWB used with the `buffer_free` operation is specified in 128-byte cache lines, and will clear the dirty bit on an entire cache line in L2 cache. If non-buffer data (other data) is sharing the cache line with the buffer, and the other data is dirty, the other data will not be flushed to DRAM because the dirty bit for the entire cache line was cleared by the DWB operation.

Figure 8: DWB and Buffer Alignment and Size

DWB is one of the Reasons Why The Buffer *Must* be a Aligned on the 128-Byte Boundary, and a Multiple of Cache-Line Size


Dirty Bit Cleared
0
Byte 127
L2 Cache
(One 128-Byte L2 Cache Line
Byte 0
Other Data
Buffer is not aligned and not a multiple of cache-line size
Other Data
Other Data is not flushed to DRAM because the dirty bit is cleared

DWB instructions affect an entire cache line, so buffers must be cache-line aligned and multiples of cache line size. Otherwise, other data might share the cache line.

The other data (data not in the buffer) which shares the cache line will not be flushed to DRAM if the dirty bit for the entire cache line is cleared, leading to data corruption.

The *FPA* chapter and *PIP/IPD* chapters provide other reasons why the buffer must be 128-byte aligned and a multiple of cache line size.

7 Scratchpad and IOBDMA Details

See the *Essential Topics* chapter for an introduction to the scratchpad and IOBDMA. This section provides more information for readers who need technical details. (The Cavium Networks-specific `CVMSEG` segment, *scratchpad* memory, and IOBDMA operations were first introduced in the *Software Overview* chapter, with an emphasis on their place in the virtual memory map.)

The Simple Executive function `cvmx_send_single()` will initiate an IOBDMA operation. The I/O Bridge will DMA the result of the IOBDMA operation from the selected hardware source (such as the FPA unit) to the selected core-specific scratchpad area (which was allocated via the configuration utility). (Note that specifying an incorrect target scratchpad area address in the command may corrupt Dcache.)

IOBDMA operations must always target a scratchpad address which is 64-bit (8 byte) aligned, so the user can only define 16 scratchpad areas used for IOBDMA per cache line.

The scratchpad is byte addressable by the cores (for either read or write).

See Section 11 – "Access to CSRs and Memory via *xkphys*, and `CVMSEG`" for more information.

7.1 *Scratchpad Access:* `CVMSEG LM` *and* `CVMSEG IO`

Scratchpad memory is accessed via the memory segment *cvmseg*.

The special *cvmseg* memory consists of two segments:

- `CVMSEG LM` ("Local Memory", the scratchpad)
- `CVMSEG IO` (a store instruction to this address initiates an IOBDMA operation)

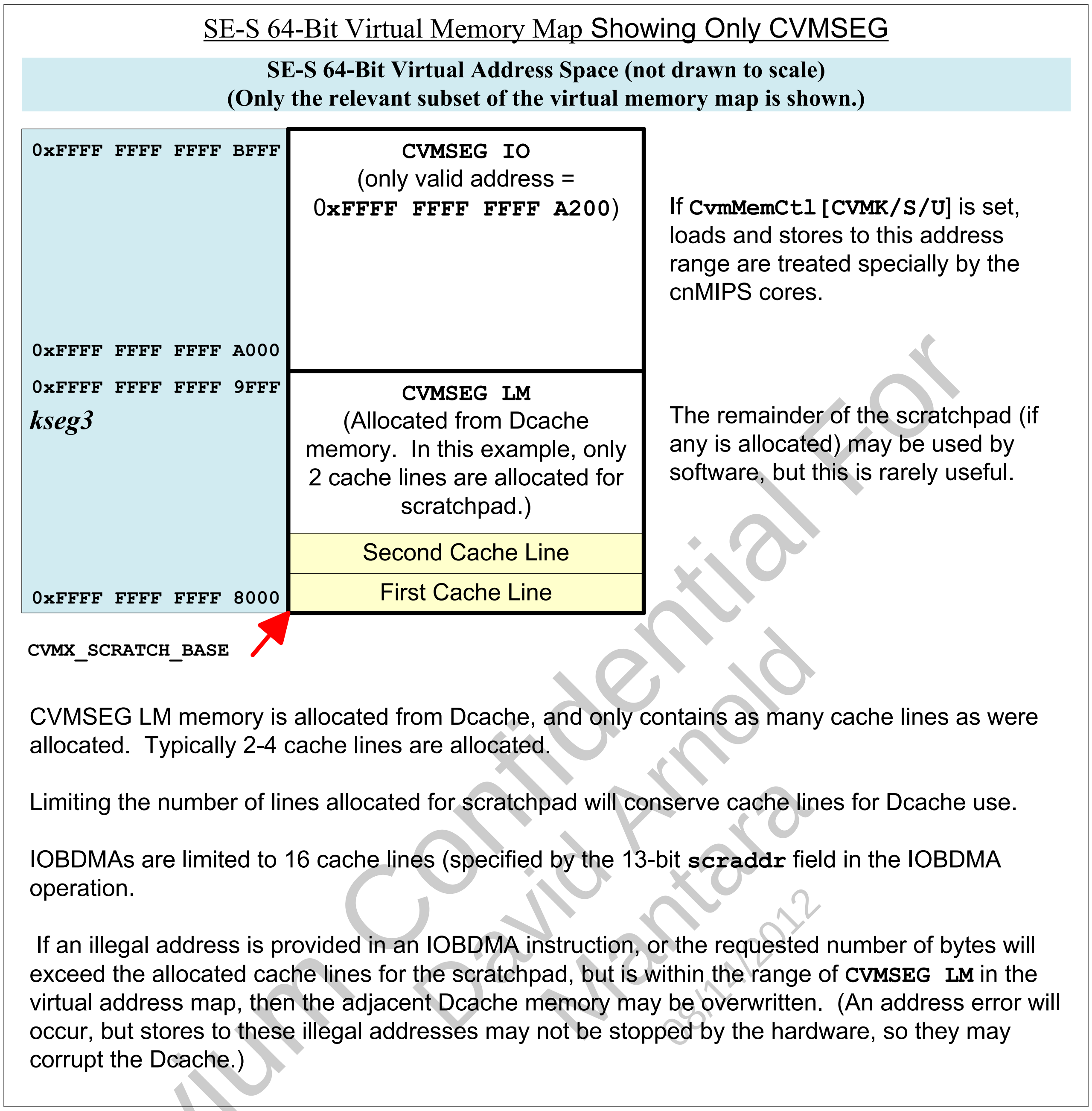
`CVMSEG IO` has only one valid address: `0xFFFF FFFF FFFF A200`. A store instruction to this address starts an IOBDMA operation.

In the SDK, the function `cvmx_send_single()` (`cvmx-access-native.h`) is used to initiate an IOBDMA operation:

```
static inline void cvmx_send_single(uint64_t data)
{
    // set CVMX_IOBDMA_SENDSINGLE to the address of CVMSEG IO
    const uint64_t CVMX_IOBDMA_SENDSINGLE = 0xffffffffffffa200ull;

    // write the IOBDMA command to CVMSEG IO
    cvmx_write64(CVMX_IOBDMA_SENDSINGLE, data);
}
```

Figure 9: `CVMSEG` and 64-bit Virtual Memory Map


SE-S 64-Bit Virtual Memory Map Showing Only CVMSEG
SE-S 64-Bit Virtual Address Space (not drawn to scale)
(Only the relevant subset of the virtual memory map is shown.)
0xFFFF FFFF FFFF BFFF
0xFFFF FFFF FFFF A000
0xFFFF FFFF FFFF 9FFF
kseg3
0xFFFF FFFF FFFF 8000
CVMSEG IO
(only valid address = 0xFFFF FFFF FFFF A200)
CVMSEG LM
(Allocated from Dcache memory. In this example, only 2 cache lines are allocated for scratchpad.)
Second Cache Line
First Cache Line
CVMX_SCRATCH_BASE
If CvmMemCtl[CVMK/S/U] is set, loads and stores to this address range are treated specially by the cnMIPS cores.
The remainder of the scratchpad (if any is allocated) may be used by software, but this is rarely useful.

CVMSEG LM memory is allocated from Dcache, and only contains as many cache lines as were allocated. Typically 2-4 cache lines are allocated.

Limiting the number of lines allocated for scratchpad will conserve cache lines for Dcache use.

IOBDMAs are limited to 16 cache lines (specified by the 13-bit **`scraddr`** field in the IOBDMA operation.

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.)

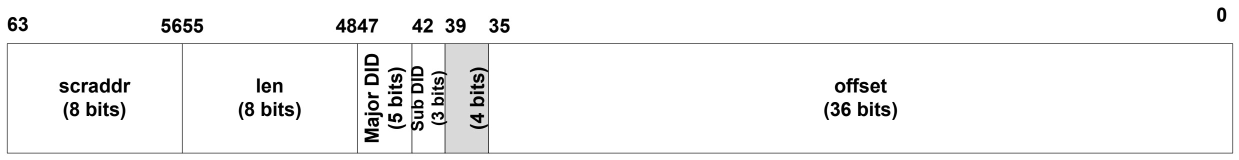
Legal `CVMSEG LM` addresses start at virtual address `0xFFFF FFFF FFFF 8000` and may increase up to `0xFFFF FFFF FFFF9AF`, depending on the number of cache lines allocated in `CvmMemCtl[LMEMSZ]`. References above the range allocated by `CvmMemCtl[LMEMSZ]` (but at `0xFFFF FFFF FFFF 9FFF` or below) cause an address error, but stores to these illegal addresses may not be stopped by the hardware, so can cause Dcache corruption.

See Section 11 – "Access to CSRs and Memory via *xkphys*, and `CVMSEG`" for more information.

The IOBDMA instruction includes the `CVMSEG LM` offset (`scraddr`) where the result of the IOBDMA operation should be stored. The format of the result is shown below.

Figure 10: IOBDMA Operation (Store Data) Format

IOBDMA Operation: Store Data Format

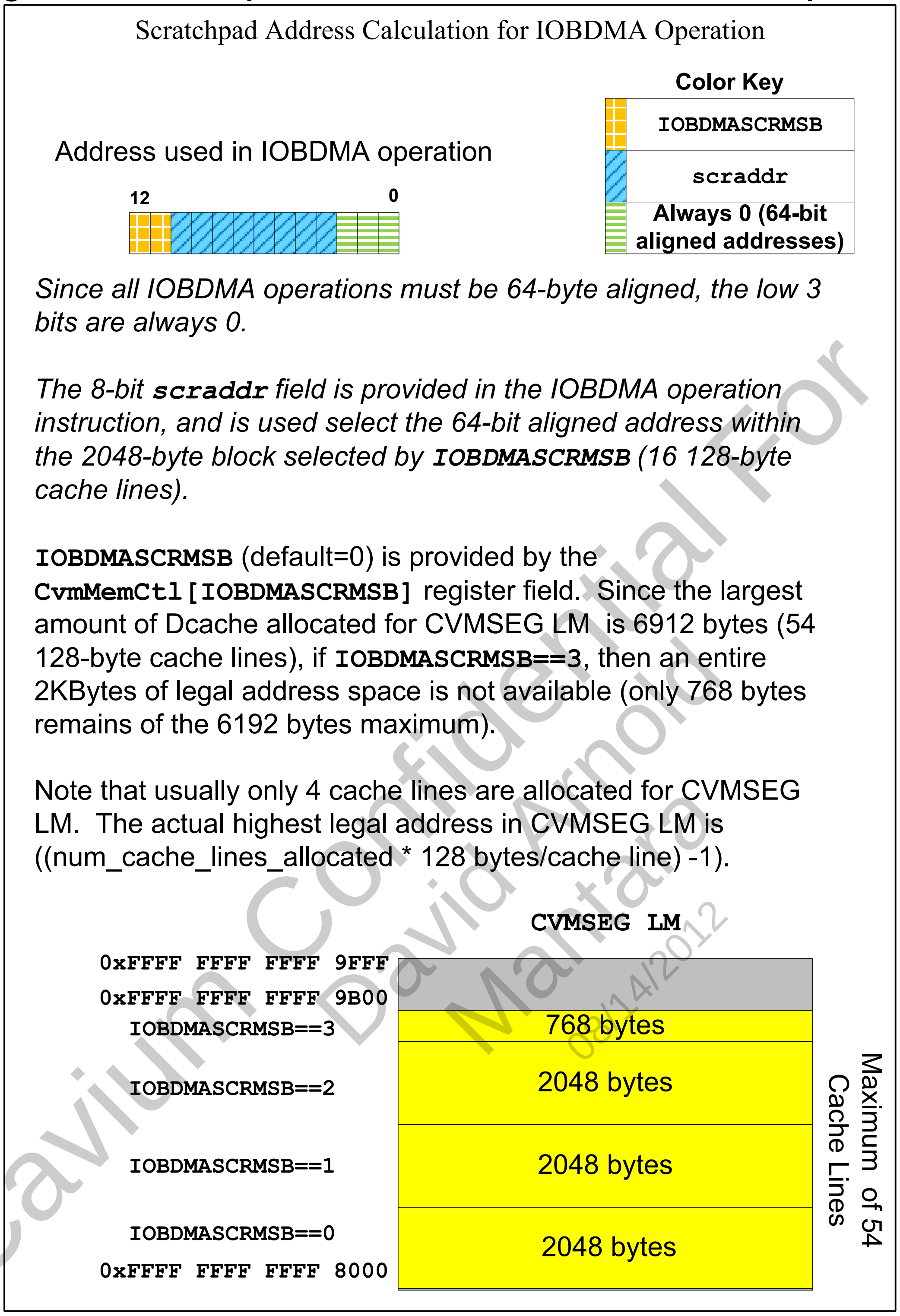


scraddr: bits 3-10 of the scratchpad address where the core puts the result of the IOBDMA operation
len: the number of 64-bits words in the result. The results are placed sequentially in CVMSEG LM from the starting address.
Major DID: (Major Device ID) – Directs the request to the correct hardware block (as in physical addresses)
Sub DID: Directs the request within the hardware block selected by Major DID
offset: Interpreted by the hardware on the I/O bus in the same way as a normal physical address.

Hardware Constraints:

- **len** must not be 0
- If **len** is 2 or 3, then **scraddr<1:0>** must not be 3
- If **len** is >= 4, then **scraddr<0>** must be 0

Figure 11: Scratchpad Address Calculation for IOBDMA Operation



Since all IOBDMA operations must be 64-byte aligned, the low 3 bits are always 0.

The 8-bit ***`scraddr`*** *field is provided in the IOBDMA operation instruction, and is used select the 64-bit aligned address within the 2048-byte block selected by* ***`IOBDMASCRMSB`*** *(16 128-byte cache lines).*

`IOBDMASCRMSB` (default=0) is provided by the **`CvmMemCtl[IOBDMASCRMSB]`** register field. Since the largest amount of Dcache allocated for CVMSEG LM is 6912 bytes (54 128-byte cache lines), if **`IOBDMASCRMSB==3`**, then an entire 2KBytes of legal address space is not available (only 768 bytes remains of the 6192 bytes maximum).

Note that usually only 4 cache lines are allocated for CVMSEG LM. The actual highest legal address in CVMSEG LM is ((num_cache_lines_allocated * 128 bytes/cache line) -1).

For example: `cvmx_fpa_alloc_async()` will start an IOBDMA operation which will instruct the FPA unit to allocate an available buffer, and store the buffer's physical address in the `CVMSEG LM` (scratchpad) memory.

Note: If an illegal address is provided in an IOBDMA instruction, or the requested number of bytes will exceed the allocated cache lines in* `CVMSEG LM`*, but within the range shown 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.)

7.2 Example IOBDMA Details

This is an example of an asynchronous `get_buffer` operation which will result in an IOBDMA from the FPA to the scratchpad.

7.2.1 Starting the IOBDMA Operation

In this example, the user uses the `cvmx_fpa_async_alloc()` function to start the IOBDMA operation. This function hides the details from the user. Note that Device IDs (or DIDs) are discussed in the *HRM*, and are most easily found in the hardware unit-specific chapter where the IOBDMA operation data structure is presented.

```
cvmx_fpa_async_alloc(CVMX_SCR_SCRATCH, CVMX_MY_FIRST_POOL);

static inline void cvmx_fpa_async_alloc(uint64_t scr_addr, uint64_t pool)
{
   cvmx_fpa_iobdma_data_t data;

   /* Hardware only uses 64 bit aligned locations, so convert from byte address
   ** to 64-bit index
   */
   data.s.scraddr = scr_addr >> 3;
   data.s.len = 1;
   data.s.did = CVMX_FULL_DID(CVMX_OCT_DID_FPA, pool);
   data.s.addr = 0;
   cvmx_send_single(data.u64);
}
```

In the SDK, a data structure is defined to hold the data used in this particular IOBDMA operation. (This software data structures matches the information shown in Figure 10 – "IOBDMA Operation (Store Data) Format".)

```
typedef union
{
    uint64_t        u64;
    struct {
        uint64_t scraddr : 8; // Offset of scratchpad to write data to
        uint64_t len     : 8; // The number of 64-bit words of data to write
        uint64_t did     : 8; // Source Address (Major DID << 3) | Sub-DID)
        uint64_t addr    :40; // The remainder of the physical address
    } s;
} cvmx_fpa_iobdma_data_t;
```

The user must ensure that a user-defined scratchpad area has been configured and is in `cvmx-config.h` before using the `cvmx_fpa_async_alloc()` function. The target scratchpad area name is passed as an argument to the function. Note that the default configuration provides 4 128-byte cache lines for scratchpad. These cache lines can be configured to contain data which smaller than a whole cache line, for example two cache lines can be configured to contain four 64-bit scratchpad areas. See Figure 12: "Example of Two Scratchpad Areas". (Configuration information is in the *Configuration* chapter.)

In `cvmx_fpa_async_alloc()`, set `len` to 1 (one 64-bit word). (Note: requesting more than one buffer can cause problems. If the entire request cannot be satisfied, no buffers are returned (all addresses returned are zeros). We recommend only requesting one buffer at a time.)

The `did` is set to the Device ID for the appropriate pool. For example, the Source Device for pool 0 is Major DID=5, Sub-DID = 0. See Table 3 – "Converting a DID and Sub-DID into a Physical Address (CN58XX Example)".

Table 4: Major Device ID (DID) and Sub-DID used in FPA IOBDMA

Hardware Unit	Major DID (Decimal)	Sub-DID	Note
FPA	5	Pool number (0-7)	Sub-DID 0 addresses Pool 0. Sub-DID 1 addresses Pool 1, etc.

7.2.2 Verifying the IOBDMA Operation is Complete

To verify the IOBDMA operation is complete, either `CVMX_SYNCIOBDMA` can be used, or polling can be used.

7.2.2.1 Using `CVMX_SYNCIOBDMA`

Users can use `CVMX_SYNCIOBDMA` instruction to force completion of the IOBDMA operation. The `CVMX_SYNCIOBDMA` instruction will stall the core until all prior IOBDMA operations for the core have completed. (See the *Essential Topics* chapter for information about `CVMX_SYNCIOBDMA`.)

7.2.2.2 Using Polling

In the case of the asynchronous `get_work` operation, users can initialize the scratchpad area to 0, and use polling to check if the IOBDMA operation has completed (is complete if non-zero value is present). (An example of this is shown in the asynchronous `get_work` example in the *Essential Topics* chapter.) In the case of the asynchronous FAU operation, the sequence "ACEDC0DE" was used in the *Essential Topics* chapter (again assuming this will never be a legal value.) Note that initializing the scratchpad area to zero won't work for the asynchronous `get_work` operation, where a return value of zero means "not enough buffers are available to satisfy the request".

7.2.3 Reading the Scratchpad

When the IOBDMA completes, it writes `len` number of buffer addresses to the scratchpad (or NULLs if insufficient free buffers exist to satisfy the request). Later, the buffer address(es) may be retrieved from the scratchpad address. The SDK API functions to read and write from scratchpad addresses are defined in cvmx-scratch.h. Software access to the scratchpad is byte-addressable. The following code retrieves the buffer returned by the IOBDMA command:

```
CVMX_SYNCIOBDMA;  // guarantee that IOBDMA command has completed
newbuf = (void *)cvmx_scratch_read64(CVMX_SCR_SCRATCH);
```

Note that if len is larger than the number of available addresses in the selected pool, then all (len) addresses returned for the IOBDMA operation are NULL, indicating that the pool does not have enough addresses to satisfy the request.
To read the scratchpad, call cvmx_scratch_read64(offset)(defined in cvmx-scratch.h). The offset is converted from a byte offset to a virtual address inside the function.

The code calls cvmx_scratch_read64() to read 64 bytes from the specified scratchpad. This function hides the details from the user:

```
newbuf = (void *)cvmx_scratch_read64(CVMX_SCR_SCRATCH);
```

<u>Details (SDK 1.9):</u>
In hal.c:

```
#define CASTPTR(type, v)  ((type *)(long)(v))
```

In cvmx-scratch.h:

```
#define CVMX_SCRATCH_BASE       (-32768L)   // 0xffffffffffff8000
```

In cvmx_config.h, the scratchpads are defined as an offset from the CVMX_SCRATCH_BASE:

```
#define CVMX_SCR_SCRATCH             0       // IOBDMA must be 64-bit
aligned
#define CVMX_SCR_REG_AVAILABLE_BASE 16

/**
 * Reads a 64 bit value from the processor local scratchpad memory.
 *
 * @param address byte address to read from
 *
 * @return value read
 */
static inline uint64_t cvmx_scratch_read64(uint64_t address)
{
    return *CASTPTR(volatile uint64_t, CVMX_SCRATCH_BASE + address);
}
```

Figure 12: Example of Two Scratchpad Areas

Two Scratchpad Areas Allocated from Four Available Scratchpad Cache Lines

This configuration can be seen in the **`traffic-gen`** *example:*

Two scratchpad areas are defined, one in **`executive-config.h`** (size is shown in bytes):

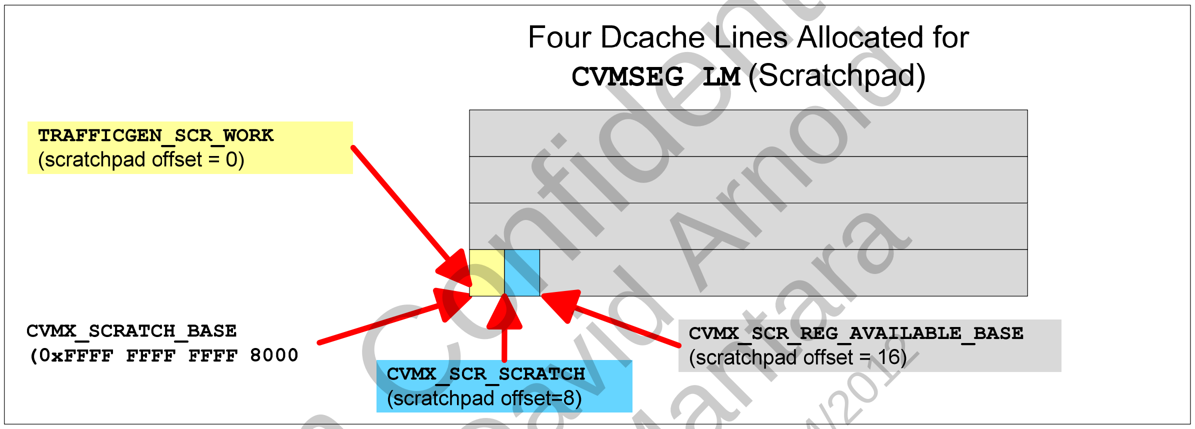
```
scratch TRAFFICGEN_SCR_WORK
        size        = 8 // size of each element
        iobdma      = true
        permanent   = true
        description = "Async get work";
```

(**`count`** == 1, by default)
(Note that updating **`executive-config.h`** with a newer SDK version would be easier if this addition had been made to a custom **`*-config.h`** file instead.)

...and the other (default) in in **`cvmx-resources.config`** (size is shown in bytes):

```
scratch CVMX_SCR_SCRATCH
        size        = 8 // size of each element
        iobdma      = true
        permanent   = false
        description = "Generic scratch iobdma area";
```

(**`count`** == 1, by default)
Of the four scratchpad cache lines allocated in **`cvmx-user-app-init()`**, two 8-byte scratchpad areas are allocated out of the first 128-byte cache line. The other cache lines are shown as available, but not configured.



In **`cvmx_config.h`**, the scratchpad area names are defined as an offset from **`CVMX_SCRATCH_BASE`**:

```
#define TRAFFICGEN_SCR_WORK      (0) /**< Async get work */
#define CVMX_SCR_SCRATCH         (8) /**< Generic scratch iobdma area */
#define CVMX_SCR_REG_AVAIL_BASE (16)/**< First location available
after cvmx-config.h allocated region. */
```

`CVMX_SCR_BASE` is defined in **`cvmx-scratch.h`** to be the start of the **`CVMSEG LM`** segment:

```
#define CVMX_SCRATCH_BASE       (-32768L)    // 0xffffffffffff8000
```

Code uses these scratchpad area names to initiate IOBDMA operations, and to read the operation results, for example (from the **`fpa_simplified`** example):

```
cvmx_fpa_async_alloc(CVMX_SCR_SCRATCH, CVMX_MY_FIRST_POOL);

CVMX_SYNCIOBDMA;  // guarantee that IOBDMA command has completed
newbuf = (void *)cvmx_scratch_read64(CVMX_SCR_SCRATCH);
```

7.3 Modifying the Allocated Scratchpad Size

For Linux and SE-UM applications, the scratchpad size is configured via the `make menuconfig` option `CONFIG_CAVIUM_OCTEON_CVMSEG_SIZE` (see the *Software Overview* chapter for details). This section applies to SE-S applications.

The Cavium Networks-specific `CVMSEG` segment is configured when `cvmx_user_app_init()` is called. Users cannot modify this configuration using Simple Executive configuration variables. To modify the configuration, edit the code shown below.

When using the default Simple Executive configuration, a minimum of at least 1 cache line is required for scratchpad: 8 bytes of this cache line are used as a generic IOBDMA area (`CVMX_SCR_SCRATCH`). If this scratchpad area is not used, `CVMSEG` can be configured to 0 cache lines. By default, Simple Executive allocates 4 cache lines from Dcache for the scratchpad (512 bytes), even though only 1 cache line is used.

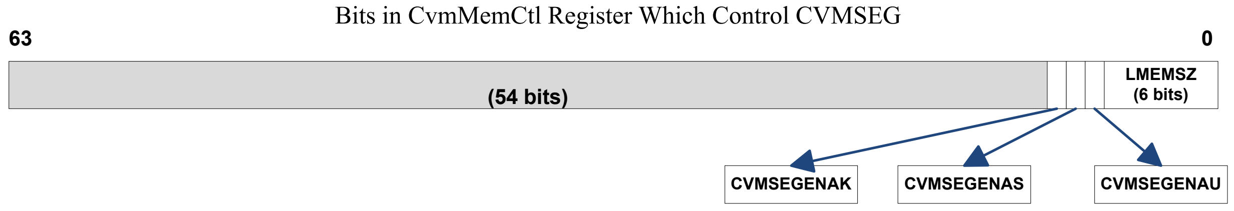
In `cvmx_user_app_init()` (located in `executive/cvmx-app-init.c`), the CSR is first read into a local variable (`tmp`) using the `CVMX_MF_CVM_MEM_CTL` macro, the local value modified, then the modified value is written to the CSR using the `CVMX_MT_CVM_MEM_CTL` macro:

```
    // Set up 4 cache lines of local memory, make available from
    // Kernel space
    // Bit 8 = CVMSEGNAK  (access CVMSEG from kernel/debug mode)
    // Bit 7 = CVMSEGENAS (access CVMSEG from supervisor mode)
    // Bit 6 = CVMSEGENAU (access CVMSEG from user mode)
    // Bits 0-5 = LMEMSZ - size of local memory in 128-byte cache blocks
    CVMX_MF_CVM_MEM_CTL(tmp); // read CvmMemCtl register into tmp
    tmp &= ~0x1ffull;         // clear low 9 bits in CvmMemCtl value
    tmp |= 0x104ull;          // CVMSEGNAK=1, size=4 cache lines, all others=0
    CVMX_MT_CVM_MEM_CTL(tmp);
    CVMX_DCACHE_INVALIDATE;   // invalidate existing Dcache entries
```

Note: For safety, always follow any change of `CVMSEG LM` size with `CVMX_DCACHE_INVALIDATE` to invalidate the Dcache. This will invalidate any data which is stored in the portion of Dcache now reserved for scratchpad.

The following figure shows the relevant portion of the `CvmMemCtl` register. For more information, see the *HRM*.

Figure 13: Modifying `CVMSEG` Configuration



Bits shown in gray are not part of CVMSEG configuration.
CVMSEGENAK: If set, CVMSEG is available for load/store operations in kernel/debug mode (CVMSEG ENAble Kernel).
CVMSEGENAS: If set, CVMSEG is available for load/store operations in supervisor mode (CVMSEG ENAble Supervisor).
CVMSEGENAU: If set, CVMSEG is available for load/store operations in user mode (CVMSEG ENAble User).
LMEMSZ: The size of local memory (scratchpad) in cache blocks (0-54)

In **`cvmx_user_app_init()`**, the CVMSEG control fields are set to 0x104, enabling access for kernel-mode and setting LMEMSZ to 4 cache lines:

8								0
1	0	0	0	0	0	1	0	0

We do not recommend setting LMEMSZ to greater than four because the memory for scratchpad comes from Dcache memory, diminishing the memory used for Dcache operations.

8 Asynchronous CSR Read

It is possible to asynchronously read the some CSRs via the `cvmx_read_csr_async()` function (which uses IOBDMA to the scratchpad).

The asynchronous read can be used to hide I/O load latencies by allowing the software to continue to execute instructions while the value of a CSR is being asynchronously transferred to the core’s scratchpad memory.

If this level of performance tuning is essential for the application, then carefully test to see whether asynchronous access to the target CSR works and actually improves performance. In general, the time is better spent on easier optimizations which will have a larger return on the investment.

9 “Unprotected” Buffer Pools

This section is provided to explain the dangers of using unprotected pools, which can be configured via the Simple Executive configuration. Users may then decide whether unprotected pools are necessary in the application.

We recommend that protected always be set to 1 (TRUE). The “unprotected pools” feature should not be used. Information about unprotected pools is provided in this section to illustrate the type of difficult-to-debug problems which can occur if this configuration keyword is not set appropriately.

If the "protected" field is set to "false" (0), AND two or more pools have the *same* buffer size, then multiple #define macros can refer to the same pool. This is sometimes used if there are more than 8 different uses for FPA pools. In the example below, both `CVMX_MY_POOL` and `CVMX_MY_THIRD_POOL` will refer to pool number three.

For example, given `my-config.h with` the following contents:

```
cvmxconfig
        {
                fpa CVMX_MY_POOL     // shared with CVMX_MY_THIRD_POOL
                    size         = 2  // must be identical with shared pool
                    protected    = 0  // not protected
                    description = "MY CUSTOM POOL";
                fpa CVMX_MY_SECOND_POOL
                    size         = 4
                    protected    = 1  // protected
                    description = "MY SECOND CUSTOM POOL";
                fpa CVMX_MY_THIRD_POOL  // shared with CVMX_MY_POOL
                    size         = 2     // must be identical with shared pool
                    protected    = 0     // not protected
                    description = "MY THIRD CUSTOM POOL";
        }
```

The `cvmx-config.h` file will contain:

```
/************************* FPA allocation *********************************/
/* Pool sizes in bytes, must be multiple of a cache line */
#define CVMX_FPA_POOL_0_SIZE (16 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_1_SIZE (10 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_2_SIZE (8 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_3_SIZE (2 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_4_SIZE (4 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_5_SIZE (0 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_6_SIZE (0 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_7_SIZE (0 * CVMX_CACHE_LINE_SIZE)

/* Pools in use */
#define CVMX_FPA_PACKET_POOL                (0) /**< Packet buffers*/
#define CVMX_FPA_PACKET_POOL_SIZE           CVMX_FPA_POOL_0_SIZE
#define CVMX_FPA_WQE_POOL                   (1) /**< Work queue entrys */
#define CVMX_FPA_WQE_POOL_SIZE              CVMX_FPA_POOL_1_SIZE
#define CVMX_FPA_OUTPUT_BUFFER_POOL         (2) /**< PKO queue command
buffers*/
#define CVMX_FPA_OUTPUT_BUFFER_POOL_SIZE    CVMX_FPA_POOL_2_SIZE
#define CVMX_MY_POOL                        (3) /**< MY CUSTOM POOL*/
#define CVMX_MY_POOL_SIZE                   CVMX_FPA_POOL_3_SIZE
#define CVMX_MY_THIRD_POOL                  (3) /**< MY THIRD CUSTOM POOL */
#define CVMX_MY_THIRD_POOL_SIZE             CVMX_FPA_POOL_3_SIZE
#define CVMX_MY_SECOND_POOL                 (4) /**< MY SECOND CUSTOM POOL */
#define CVMX_MY_SECOND_POOL_SIZE            CVMX_FPA_POOL_4_SIZE
```

Note: If protected = false, and there more than one pool is defined with the same buffer size, these pools will be shared. If this is not planned, then difficult-to-debug errors may occur.

Another error can occur if it is intended that the pools be shared, but different buffer sizes were specified during configuration. An example of this type of error is shown in the next section.

Example of Configuration Error:

Note: The pool will not be shared unless the buffer sizes are identical. If they are not identical, the config utility (`host/bin/cvmx-config`) will quietly create different #defines for the pools: they will not be shared.

```
cvmxconfig
        {
                fpa CVMX_MY_POOL       // shared with CVMX_MY_THIRD_POOL
                    size        = 6    // ERROR:  size does not match!
                    protected   = 0    // not protected
                    description = "MY CUSTOM POOL";
                fpa CVMX_MY_SECOND_POOL
                    size        = 4
                    protected   = 1     // protected
                    description = "MY SECOND CUSTOM POOL";
                fpa CVMX_MY_THIRD_POOL  // shared with CVMX_MY_POOL
                    size        = 2     // ERROR:  size does not match!
                    protected   = 0     // not protected
                    description = "MY THIRD CUSTOM POOL";
        }
```

The `cvmx-config.h` file will contain:

```
/************************* FPA allocation *********************************/
/* Pool sizes in bytes, must be multiple of a cache line */
#define CVMX_FPA_POOL_0_SIZE (16 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_1_SIZE (10 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_2_SIZE (8 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_3_SIZE (6 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_4_SIZE (4 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_5_SIZE (2 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_6_SIZE (0 * CVMX_CACHE_LINE_SIZE)
#define CVMX_FPA_POOL_7_SIZE (0 * CVMX_CACHE_LINE_SIZE)

/* Pools in use */
#define CVMX_FPA_PACKET_POOL                 (0)  /**< Packet buffers*/
#define CVMX_FPA_PACKET_POOL_SIZE            CVMX_FPA_POOL_0_SIZE
#define CVMX_FPA_WQE_POOL                    (1)  /**< Work queue entrys */
#define CVMX_FPA_WQE_POOL_SIZE               CVMX_FPA_POOL_1_SIZE
#define CVMX_FPA_OUTPUT_BUFFER_POOL          (2)  /**< PKO queue command
buffers*/
#define CVMX_FPA_OUTPUT_BUFFER_POOL_SIZE     CVMX_FPA_POOL_2_SIZE
#define CVMX_MY_POOL                         (3)  /**< MY CUSTOM POOL*/
#define CVMX_MY_POOL_SIZE                    CVMX_FPA_POOL_3_SIZE
#define CVMX_MY_SECOND_POOL                  (4)  /**< MY SECOND CUSTOM POOL */
#define CVMX_MY_SECOND_POOL_SIZE             CVMX_FPA_POOL_4_SIZE
#define CVMX_MY_THIRD_POOL                   (5)  /**< MY THIRD CUSTOM POOL */
#define CVMX_MY_THIRD_POOL_SIZE              CVMX_FPA_POOL_5_SIZE
```

If it is absolutely needed to refer to the same pool by different names, for ease in debugging it is better to simply use a simple #define instead of configured-in sharing:

```
#define NAME_1   NAME2
```

10 Pass_*N* Specific Code in the SDK

The `OCTEON_MODEL` string (passed to the `env_setup` script) can contain a `PASS` substring. Setting the `OCTEON_MODEL` to include a `PASS` substring can cause conditional compilation of additional code and/or additional run-time conditional code needed to work around issues with some versions of some chips

Application designers should aim to write code that will run on the production-released OCTEON, and only add PASS_*N*-specific code where they are aware of a specific need to support systems built with earlier versions of their OCTEON device.

11 Access to CSRs and Memory via *xkphys*, and `CVMSEG`

If you are using the OCTEON SDK, these details are handled directly and it is not expected there needs to be any modification other than setting Linux kernel configuration variables as needed:

For SE-UM applications, 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)

This section provides more detail on the access issues and how they are resolved.

Note that this section provides basic information on this topic to help users understand the underlying issues. This section is not intended to provide complete information on this topic. For complete information, refer to the Hardware Reference Manual.

The *Software Overview* chapter presented the virtual memory maps: Linux kernel mode, SE-UM 64-bit, SE-UM 32-bit, SE-S 64-bit, and SE-S 32-bit. This section provides more information on how CSRs and memory can be accessed via the *xkphys* address segment, and also how the Cavium Networks-specific `CVMSEG` is accessed. In particular, how user-mode applications access the kernel segment *xkphys* and *cvmseg* addresses, and how 32-bit SE-S and SE-UM applications can access CSRs via 64-bit *xkphys* addresses. As an example, to enable 32-bit SE-UM applications to read and write CSRs in the kernel *xkphys* segment, they must have access to CSR addresses in the kernel *xkphys* segment, 64-bit addressing, and 64-bit operations.

Note that *xkphys* addresses and *cvmseg* virtual addresses are not translated through 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.

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.

11.1 Accessing Kernel Address Ranges from User-Mode Applications

Cavium Networks provides the following CP0 `CvmMemCtl` register fields to:

- Permit SE-UM applications to access specific address ranges that lie within the privileged *xkphys* kernel segment (*xkphys* accesses to I/O space where CSRs are located, and *xkphys* accesses to physical memory)
- Allow kernel-mode, supervisor-mode, or user-mode processes to access *kseg3* addresses in the *cvmseg* address range, and treat these accesses as specially as accesses to the scratchpad.

Table 5: Configuring CSR, *xkphys* Memory, and `CVMSEG` Access

`CvmMemCtl[Field]`	Description
XKIOENAU	Allow user-mode program to access CSRs (I/O Space in *xkphys*:) If set (and `Status[UX]` is set), user-level loads/stores can use *xkphys* addresses with VA<48>==1 (provides access to CSRs in I/O space). See Note1.
XKMEMENAU	Allow user-mode program to access memory (via *xkphys* segment addresses): If set (and `Status[UX]` is set), user-level loads/stores can use *xkphys* addresses with VA<48>==0 (provides access to physical memory). See Note2.
CVMSEGENA[K/S/U]	Access `CVMSEG` (K=kernel, U=user, S=supervisor). See Note3.
Notes	
Note1: For SE-UM applications, set via the Linux `make menuconfig` option: `CONFIG_CAVIUM_OCTEON_USER_IO`. SE-S applications do not need to set this bit because they run in kernel mode.	
Note2: For SE-UM applications, set via the Linux `make menuconfig` option: `CONFIG_CAVIUM_OCTEON_USER_MEM`. SE-S applications do not need to set this bit because they run in kernel mode.	
Note3: For SE-UM applications, set via the Linux `make menuconfig` option: `CONFIG_CAVIUM_OCTEON_CVMSEG_SIZE`. If the CVMSEG size is non-zero, `CVMSEGENA[K/S/U]` is set. SE-S applications set this bit when they set the size of `CVMSEG LM`.	

The Linux `make menuconfig` options shown in the following table were introduced with SDK 1.9. These options can be set to allow, per-process, or disable. Whether a SE-UM process can get access to the *xkphys* addresses depends on the values of both variables, as shown in the following table:

Table 6: Configuring SE-UM Access to *xkphys* Addresses

		CONFIG_CAVIUM_OCTEON_USER_IO		
		allow	**per-process**	**disable**
CONFIG_CAVIUM_OCTEON_USER_MEM	**allow**	I/O - Y Mem - Y	I/O - C Mem - Y	I/O - N Mem -Y
	per-process	I/O - Y Mem - C	I/O - C Mem - C	I/O - N Mem - C
	disable	I/O - Y Mem - N	I/O - C Mem Access - N	I/O - N Mem - N
NOTES				
Y - Access unconditionally allowed C - Access can be enabled on a per-process basis. N - Access unconditionally denied.				
Access is enabled with the `sysmips()` system call as shown in the `cvmx_linux_enable_xkphys_access()` function. The third argument to `sysmips()` specifies the access: bit-0 controls `OCTEON_USER_MEM` bit-1 controls `OCTEON_USER_IO`				
The `cvmx_linux_enable_xkphys_access()` function tries to enable both `OCTEON_USER_MEM` and `OCTEON_USER_IO`. If either fails due to the restrictions shown in the table above, neither will be enabled.				

11.2 Running in a 64-Bit Environment

All processes running on OCTEON run in a 64-bit environment, including 32-bit processes (see Section 11.2.1 – "32-Bit Applications in 64-Bit Environment"). This 64-bit environment is created by setting standard MIPS register fields which enable 64-bit operations, and 64-bit addressing, as shown in the following table. All of these fields (PX, [UX or KX]) are set for all processes running on OCTEON, providing them with a 64-bit environment (64-bit operations and 64-bit addressing).

Table 7: Configuring 64-Bit Operations and Addressing

Register[Field]	Description
Status[PX]	Enables 64-bit operations in user mode without enabling 64-bit addressing.
Status[UX]	Enables 64-bit addressing for user segments (required for 64-bit addressing). Allows user-mode processes to execute instructions which perform 64-bit operations. The XTLB refill vector is used for references to user segments. This bit is used for user-mode processes.
Status[KX]	Enables 64-bit addressing for kernel segments (required for 64-bit addressing). The XTLB refill vector is used for references to kernel segments. This bit is used for kernel-mode processes.

See MIPS® Architecture For Programmers, Volume III: The MIPS64® and microMIPS64™ Privileged Resource Architecture, available at http://www.mips.com for more information on these fields.

11.2.1 32-Bit Applications in 64-Bit Environment

Both Status[UX] (64-bit addressing via XTLB) and Status[PX] (64-bit operations such as sd (store double word) are set for 32-bit applications, creating a 64-bit environment. The creation of a 64-bit environment does not turn a 32-bit application into a 64-bit application. The 64-bit environment is a hardware-level concept. The 32-bit application is a compiler-level concept: the compiler creates a 32-bit pointer for 32-bit applications, and 64-bit pointer for 64-bit applications.

Thus, the significant difference between 64-bit applications and 32-bit applications running on OCTEON processors is in how the compiler handles *pointers*. 64-bit addresses cannot be accessed via pointers from 32-bit applications at the C-code level because the pointer data type is only 32-bits.

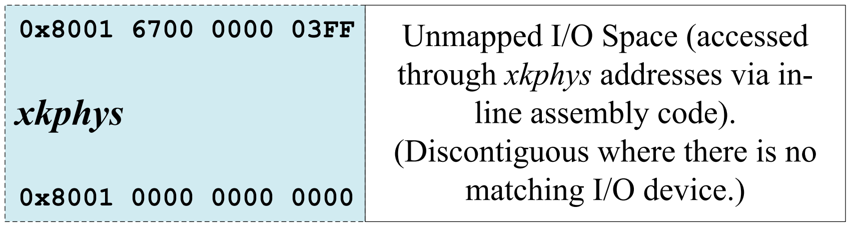
Because the Status[UX] and Status[PX] fields are set, 64-bit addresses can be used as long as they are not stored in a pointer data type. They can be accessed by 64-bit operation such as sd or ld. For a description of using the 64-bit environment to work-around the 32-bit pointer limitation, see Section 11.3.2 – "32-Bit SE-S or SE-UM Applications" and Section11.5.1 – "32-Bit Application Access to Scratchpad via".

11.3 CSR Access

Access to CSRs is via *xkphys* addresses.

Figure 14: Accessing CSRs via the *xkphys* Segment

CSR Access


0x8001 6700 0000 03FF
xkphys
0x8001 0000 0000 0000
Unmapped I/O Space (accessed through xkphys addresses via in-line assembly code). (Discontiguous where there is no matching I/O device.)

Permission to Access CSRs via *xkphys*:
Linux SE-UM applications which run in user mode gain access to *xkphys* addresses via the **`make menuconfig`** option **`CONFIG_CAVIUM_OCTEON_USER_IO`**.

32-Bit Application Access to CSRs via *xkphys*:
32-Bit access to 64-bit addresses is via inline assembly code.

Table 8: Accessing CSRs via the *xkphys* Segment

Application Type	Runs in	CSR Access in *xkphys* segment
64-bit SE-S	Kernel Mode	Access is automatically enabled: process runs in kernel mode.
64-bit SE-UM	User Mode	Set `Status[XKIOENAU]` to enable access to CSRs in *xkphys*. (For SE-UM applications, set via the Linux `make menuconfig` option: `CONFIG_CAVIUM_OCTEON_USER_IO`.)
32-bit SE-S	Kernel Mode	Access is automatically enabled: process runs in kernel mode. To solve 32-bit process, 64-bit address problem: Applications use `cvmx_csr*()` API functions. These functions use inline assembly to create 64-bit addresses used in load and store to CSRs because 32-bit pointers cannot hold the 64-bit *xkphys* addresses.
32-bit SE-UM	User Mode	Set `Status[XKIOENAU]` to enable access to CSRs in *xkphys*. (For SE-UM applications, set via the Linux `make menuconfig` option: `CONFIG_CAVIUM_OCTEON_USER_IO`.) To solve 32-bit process, 64-bit address problem: Applications use `cvmx_csr*()` API functions. These functions use inline assembly to create 64-bit addresses used in load and store to CSRs because 32-bit pointers cannot hold the 64-bit *xkphys* addresses.

11.3.1 User-Mode Application Access to *xkphys* CSR Addresses

Access to kernel segments such as *xkphys* is not normally allowed when the processor is operating in user mode. SE-UM application access can be configured as shown in Table 5 – "Configuring CSR, *xkphys* Memory, and `CVMSEG` Access" and Table 6- "Configuring SE-UM Access to *xkphys* Addresses". Note that *xkphys* segment accesses bypass the TLB.

11.3.2 32-Bit SE-S or SE-UM Applications Access to CSRs

How do 32-bit SE-S or SE-UM applications access CSRs, which are outside of the 32-bit address space?

32-bit SE-S and SE-UM applications run in a 64-bit environment with 64-bit registers, addresses, and operations. (The `Status[PX]` bit and `Status[UX | KX]` fields are set, enabling 64-bit operations and addresses.)

Although the compiler stores pointers in 32-bits, in assembly language numbers can be stored in 64 bits. CSRs are generally read or written (load or store), but are not accessed by a program through C-language *pointers*. Instead of using pointers, the API accesses CSRs using inline assembly language code to generate the instruction sequence that creates a 64-bit CSR address in a 64-bit general-purpose register and then issues the 64-bit load (`ld`) (load doubleword) instead of the 32-bit load (`lw`) (load word) used by the 32-bit application.

11.4 Memory Access (Accessing FPA-Managed Buffers)

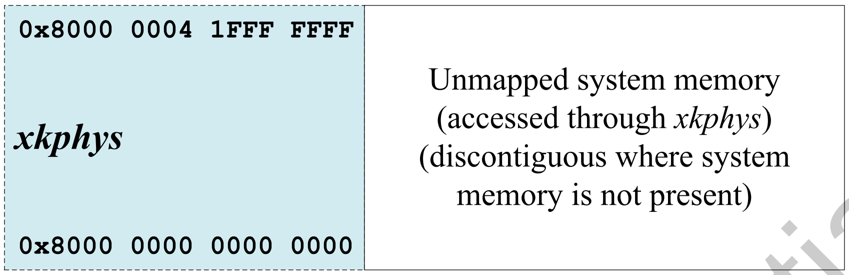
Physical memory (including FPA-managed buffers) can be accessed via TLB-mapped virtual addresses, or for 64-bit applications, access can be via addresses in the *xkphys* segment.

11.4.1 64-Bit Application Memory Access

For the Linux kernel, 64-bit SE-S and 64-bit SE-UM applications, access to physical memory (including FPA-managed buffers) is via *xkphys* addresses.

Figure 15: Accessing memory via *xkphys* -Linux kernel, 64-bit (SE-S, SE-UM)

Memory Access via *xkphys* Addresses


0x8000 0004 1FFF FFFF
xkphys
0x8000 0000 0000 0000
Unmapped system memory (accessed through xkphys) (discontiguous where system memory is not present)

Permission to Access to Physical Memory via *xkphys*:
Linux SE-UM applications which run in user mode gain access to physical memory in *xkphys* via the **`make menuconfig`** option **`CONFIG_CAVIUM_OCTEON_USER_MEM`**.

32-Bit Application Access to CSRs via *xkphys*:
32-32-bit applications do not usually access physical memory via *xkphys* addresses because programs usually use pointers to access memory, and 32-bit pointers are not large enough to hold *xkphys* addresses.

Table 9: Accessing Memory via the *xkphys* Segment

Application Type	Runs in	Memory access in *xkphys* segment
64-bit SE-S	Kernel Mode	Access is automatically enabled: process runs in kernel mode.
64-bit SE-UM	User Mode	Set `Status[XKMEMENAU]` to enable access to physical memory via *xkphys*addresses. (For SE-UM applications, set via the Linux `make menuconfig` option: `CONFIG_CAVIUM_OCTEON_USER_MEM`.)
32-bit SE-S	Kernel Mode	32-bit SE-S applications do not access memory via *xkphys* addresses because programs usually access memory via pointers. See the Software Overview chapter for the virtual memory map and details on memory access.
32-bit SE-UM	User Mode	32-bit SE-UM applications do not access memory via *xkphys* addresses because programs usually access memory via pointers. See the Software Overview chapter for the virtual memory map and details on memory access.

11.4.1.1 User-Mode Application Access to *xkphys* Memory Addresses

Access to kernel segments such as *xkphys* is not normally allowed when the processor is operating in user mode. SE-UM application access can be configured as shown in Table 5 – "Configuring CSR, *xkphys* Memory, and `CVMSEG` Access" and Table 6- "Configuring SE-UM Access to *xkphys* Addresses". Note that *xkphys* segment accesses bypass the TLB.

11.4.2 32-Bit Application Memory Access

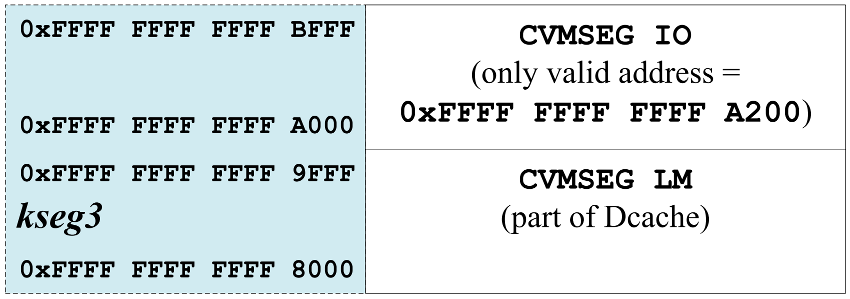
Because physical memory access is usually via pointers, the simple inline functions used to load and store to CSR addresses are not appropriate. See the *Software Overview* chapter for details on how 32-bit SE-S and SE-UM applications access physical memory (including FPA-managed buffers).

11.5 *Accessing the Scratchpad via cvmseg Addresses*

The figure below shows the `CVMSEG` virtual address space.

Figure 16: Accessing the Scratchpad via *cvmseg*

Access to **CVMSEG** Virtual Address Space



When access is enabled, loads and stores to CVMSEG addresses are treated specially by the cnMIPS cores. For example:

- A store to **`0xFFFF FFFF FFFF A200`** in **`CVMSEG IO`** will start an IOBDMA operation
- A load from the **`CVMSEG LM`** address range will result in a read of scratchpad memory

Permission to Access **`CVMSEG`**:
Linux SE-UM applications which run in user mode gain access to *cvmseg* addresses when the **`make menuconfig`** option **`CONFIG_CAVIUM_OCTEON_CVMSEG_SIZE`** is non-zero.

32-Bit Application Access to **`CVMSEG`**:
32-Bit access to 64-bit addresses is via inline assembly code.

If the CP0 register `CvmMemCtl[CVMSEGENA[K/S/U]]` is set, the cores treat loads and stores to the `CVMSEG` address range specially: a store to `0xFFFF FFFF FFFF A200` is really needed to start an IOBDMA operation, not a store to `0xFFFF A200`. (`K`=access for kernel-mode processes, `S`=access for supervisor-mode processes, and `U`=access for user-mode processes.)

Table 10: Accessing CVMSEG

Application Type	Runs in	CVMSEG Access in *cvmseg* (*kseg3*) segment
64-bit SE-S	Kernel Mode	Set CvmMemCtl[CVMSEGENAK]
64-bit SE-UM	User Mode	Set CvmMemCtl[CVMSEGENAU]. (For SE-UM applications, set via the Linux make menuconfig option: CONFIG_CAVIUM_OCTEON_CVMSEG_SIZE.)
32-bit SE-S	Kernel Mode	Set CvmMemCtl[CVMSEGENAK] Applications use cvmx_scratch*() API functions to access CVMSEG. These functions use sign-extension of 32-bit CVMSEG addresses to create the 64-bit address used in load and store because 32-bit pointers cannot hold the 64-bit CVMSEG addresses.
32-bit SE-UM	User Mode	Set CvmMemCtl[CVMSEGENAU]. (For SE-UM applications, set via the Linux make menuconfig option: CONFIG_CAVIUM_OCTEON_CVMSEG_SIZE.) Applications use cvmx_scratch*() API functions to access CVMSEG. These functions use sign-extension of 32-bit CVMSEG addresses to create the 64-bit address used in load and store because 32-bit pointers cannot hold the 64-bit CVMSEG addresses.

For SE-S applications, CVMSEG size and access are configured by the Simple Executive. See Section 7.3 – "Modifying the Allocated Scratchpad Size" for more information.

11.5.1 32-Bit Application Access to Scratchpad via *cvmseg*

How do 32-bit applications access CVMSEG, which are outside of the 32-bit address space? This access is managed for the user by the API functions, such as the cvmx_scratch_read*() functions.

32-bit applications run in a 64-bit environment with 64-bit registers, addresses, and operations. (The Status[PX] bit and Status[UX | KX] fields are set, enabling 64-bit operations and addresses.)

Instead of using pointers, the API accesses CVMSEG addresses via 64-bit numbers stored in 64-bit general-purpose registers. The 64-bit address stored in the 64-bit general-purpose register is then used in the load/store operation.

The creation of the 64-bit CVMSEG address from a 32-bit address is automatic: the load/store unit sign-extends bit 31 of the address register to a 64-bit length (sign extension is done for 32-bit values regardless of the mode of the processor). So, because bit 31 is set for 32-bit CVMSEG references, all of bits <63:32> are set, generating the required 64-bit `CVMSEG` address. For example, `0xFFFF XXXX` becomes `0xFFFF FFFF FFFF XXXX`.