Chapter 4

RSP Coprocessor 0

This chapter describes the RSP Coprocessor 0, or system control coprocessor.

The RSP Coprocessor 0 does not perform the same functions or have the same registers as the R4000-series Coprocessor 0. In the RSP, Coprocessor 0 is used to control the DMA (Direct Memory Access) engine, RSP status, RDP status, and RDP I/O.

Register Descriptions

RSP Point of View

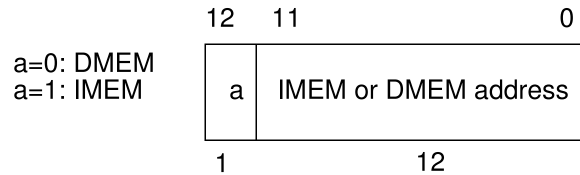
RSP Coprocessor 0 registers are programmed using the `mtc0` and `mtf0` instructions which move data between the SU general purpose registers and the coprocessor 0 registers.

Table 4-1 RSP Coprocessor 0 Registers

Register Number	Name Defined in `rsp.h`	Access Mode	Description
`$c0`	`DMA_CACHE`	RW	I/DMEM address for DMA.
`$c1`	`DMA_DRAM`	RW	DRAM address for DMA.
`$c2`	`DMA_READ_LENGTH`	RW	DMA READ length (DRAM → I/DMEM).
`$c3`	`DMA_WRITE_LENGTH`	RW	DMA WRITE length (DRAM ← I/DMEM).
`$c4`	`SP_STATUS`	RW	RSP Status.
`$c5`	`DMA_FULL`	R	DMA full.
`$c6`	`DMA_BUSY`	R	DMA busy.
`$c7`	`SP_RESERVED`	RW	CPU-RSP Semaphore.
`$c8`	`CMD_START`	RW	RDP command buffer START.
`$c9`	`CMD_END`	RW	RDP command buffer END.
`$c10`	`CMD_CURRENT`	R	RDP command buffer CURRENT.
`$c11`	`CMD_STATUS`	RW	RDP Status.
`$c12`	`CMD_CLOCK`	RW	RDP clock counter.
`$c13`	`CMD_BUSY`	R	RDP command buffer BUSY.
`$c14`	`CMD_PIPE_BUSY`	R	RDP pipe BUSY.
`$c15`	`CMD_TMEM_BUSY`	R	RDP TMEM BUSY.

$c0

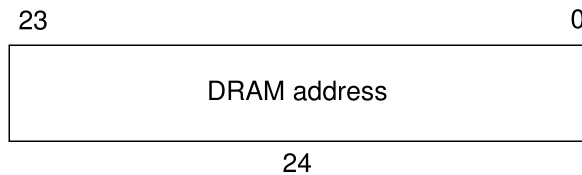
This register holds the RSP IMEM or DMEM address for a DMA transfer.



On power-up, this register is 0x0.

$c1

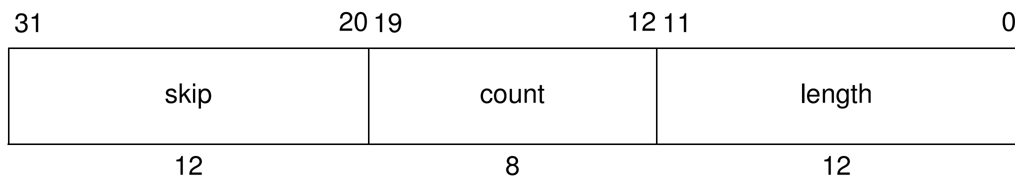
This register holds the DRAM address for a DMA transfer. This is a physical memory address.



On power-up, this register is 0x0.

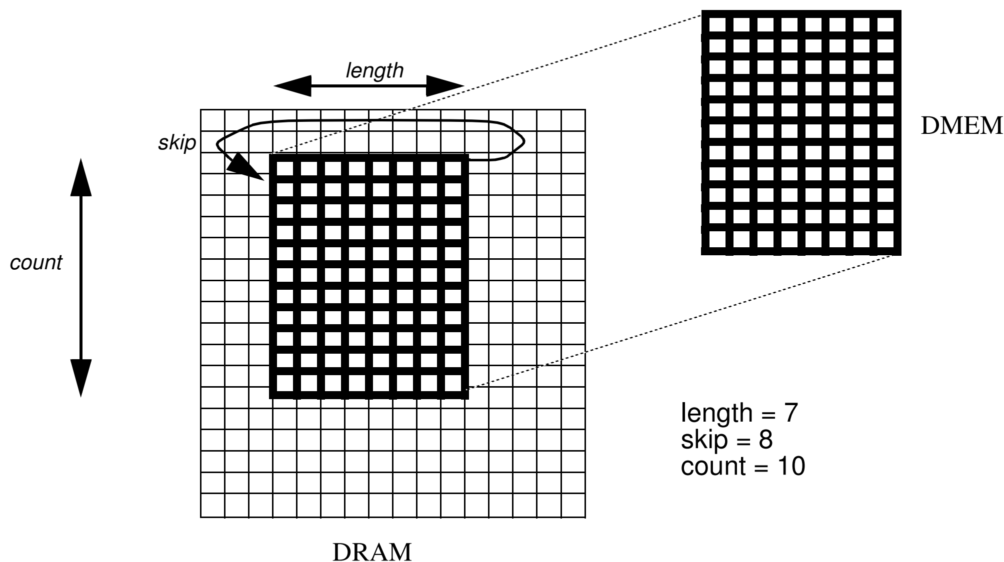
$c2, $c3

These registers hold the DMA transfer length; `$c2` is used for a READ, `$c3` is used for a WRITE.



The three fields of this register are used to encode arbitrary transfers of rectangular areas of DRAM to/from contiguous I/DMEM. *length* is the number of bytes per line to transfer, *count* is the number of lines, and *skip* is the line stride, or skip value between lines. This is illustrated in Figure 4-1:

Figure 4-1 DMA Transfer Length Encoding



Note: DMA *length* and line *count* are encoded as (value - 1), that is a line *count* of 0 means 1 line, a byte *length* of 7 means 8 bytes, etc.

A straightforward linear transfer will have a count of 0 and skip of 0, transferring (length+1) bytes.

The amount of data transferred must be a multiple of 8 bytes (64 bits), hence the lower three bits of *length* are ignored and assumed to be all 1's.

DMA transfer begins when the length register is written.

For more information about DMA transfers, see section "DMA" on page 96.

On power-up, these registers are 0x0.

$c4

This register holds the RSP status.

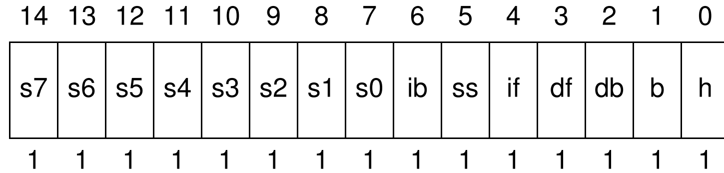


Table 4-2 RSP Status Register

bit	field	Access Mode	Description
0	h	RW	RSP is halted.
1	b	R	RSP has encountered a `break` instruction.
2	db	R	DMA is busy.
3	df	R	DMA is full.
4	if	R	IO is full.
5	ss	RW	RSP is in single-step mode.
6	ib	RW	Interrupt on break.
7	s0	RW	signal 0 is set.
8	s1	RW	signal 1 is set.
9	s2	RW	signal 2 is set.
10	s3	RW	signal 3 is set.
11	s4	RW	signal 4 is set.
12	s5	RW	signal 5 is set.
13	s6	RW	signal 6 is set.
14	s7	RW	signal 7 is set.

The ‘broke’, ‘single-step’, and ‘interrupt on break’ bits are used by the debugger.

The signal bits can be used for user-defined synchronization between the CPU and the RSP.

On power-up, this register contains 0x0001.

When writing the RSP status register, the following bits are used.

Table 4-3 RSP Status Write Bits

bit	Description
0 (0x00000001)	clear HALT.
1 (0x00000002)	set HALT.
2 (0x00000004)	clear BROKE.
3 (0x00000008)	clear RSP interrupt.
4 (0x00000010)	set RSP interrupt.
5 (0x00000020)	clear SINGLE STEP.
6 (0x00000040)	set SINGLE STEP.
7 (0x00000080)	clear INTERRUPT ON BREAK.
8 (0x00000100)	set INTERRUPT ON BREAK.
9 (0x00000200)	clear SIGNAL 0

bit	Description
10 (0x00000400)	set SIGNAL 0.
11 (0x00000800)	clear SIGNAL 1.
12 (0x00001000)	set SIGNAL 1.
13 (0x00002000)	clear SIGNAL 2.
14 (0x00004000)	set SIGNAL 2.
15 (0x00008000)	clear SIGNAL 3.
16 (0x00010000)	set SIGNAL 3.
17 (0x00020000)	clear SIGNAL 4.
18 (0x00040000)	set SIGNAL 4.
19 (0x00080000)	clear SIGNAL 5.
20 (0x00100000)	set SIGNAL 5.
21 (0x00200000)	clear SIGNAL 6.
22 (0x00400000)	set SIGNAL 6.
23 (0x00800000)	clear SIGNAL 7.
24 (0x01000000)	set SIGNAL 7.

$c5

This register maps to bit 3 of the RSP status register, DMA_FULL. It is read only.

On power-up, this register is 0x0.

$c6

This register maps to bit 2 of the RSP status register, DMA_BUSY. It is read only.

On power-up, this register is 0x0.

$c7

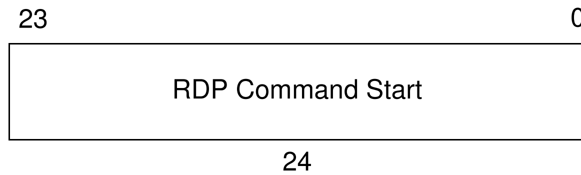
This register is a hardware semaphore for synchronization with the CPU, typically used to share the DMA activity. If this register is 0, the semaphore may be acquired. This register is set on read, so the test and set is atomic. Writing 0 to this register releases the semaphore.

```
GetSema:
          mfc0   $1, $c7
          bne    $1, $0, GetSema
          nop

    # do critical work

ReleaseSema:
          mtc0   $0, $7
```

On power-up, this register is 0x0.

$c8

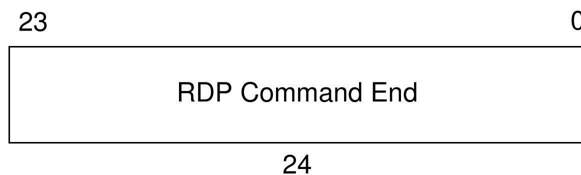
This register holds the RDP command buffer START address. Depending on the state of the RDP STATUS register, this address is interpreted by the RDP as either a 24 bit physical DRAM address, or a 12 bit DMEM address (see `$c11`).



On power-up, this register is undefined.

$c9

This register holds the RDP command buffer END address. Depending on the state of the RDP STATUS register, this address is interpreted by the RDP as either a 24 bit physical DRAM address, or a 12 bit DMEM address (see `$c11`).

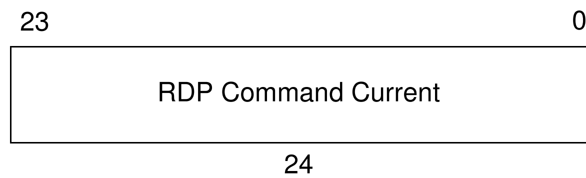


On power-up, this register is undefined.

$c10

This register holds the RDP command buffer CURRENT address. This register is READ ONLY. Depending on the state of the RDP STATUS register, this address is

interpreted by the RDP as either a 24 bit physical DRAM address, or a 12 bit DMEM address (see `$c11`).



On power-up, this register is 0x0.

$c11

This register holds the RDP status.

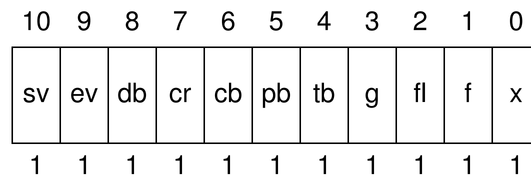


Table 4-4 RDP Status Register

bit	field	Access Mode	Description
0	x	RW	Use XBUS DMEM DMA or DRAM DMA.
1	f	RW	RDP is frozen.
2	fl	RW	RDP is flushed.
3	g	RW	GCLK is alive.
4	tb	R	TMEM is busy.
5	pb	R	RDP PIPELINE is busy.
6	cb	R	RDP COMMAND unit is busy.

bit	field	Access Mode	Description
7	cr	R	RDP COMMAND buffer is ready.
8	db	R	RDP DMA is busy.
9	ev	R	RDP COMMAND END register is valid.
10	sv	R	RDP COMMAND START register is valid.

When bit 0 (XBUS_DMEM_DMA) is set, the RDP command buffer will receive data from DMEM (see `$c8, $c9, $c10`).

On power-up, the GCLK, PIPE_BUSY, and CMD_BUF_READY bits are set, the DMA_BUSY bit is undefined, and all other bits are 0.

When writing the RDP status register, the following bits are used.

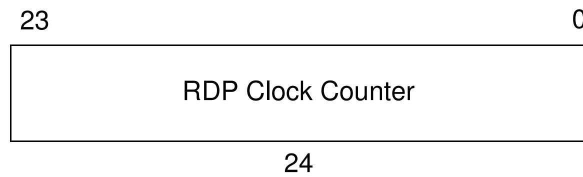
Table 4-5 RSP Status Write Bits (CPU VIEW)

bit	Description
0 (0x0001)	clear XBUS DMEM DMA.
1 (0x0002)	set XBUS DMEM DMA.
2 (0x0004)	clear FREEZE.
3 (0x0008)	set FREEZE.
4 (0x0010)	clear FLUSH.
5 (0x0020)	set FLUSH.
6 (0x0040)	clear TMEM COUNTER.

bit	Description
7 (0x0080)	clear PIPE COUNTER.
8 (0x0100)	clear COMMAND COUNTER.
9 (0x0200)	clear CLOCK COUNTER

$c12

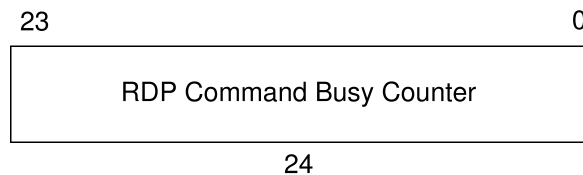
This register holds a clock counter, incremented on each cycle of the RDP clock. This register is READ ONLY.



On power-up, this register is undefined.

$c13

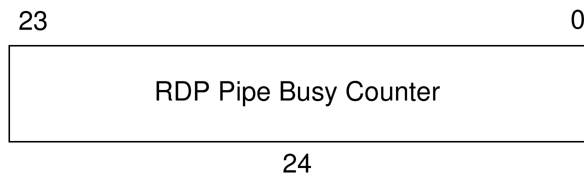
This register holds a RDP command buffer busy counter, incremented on each cycle of the RDP clock while the RDP command buffer is busy. This register is READ ONLY.



On power-up, this register is undefined.

$c14

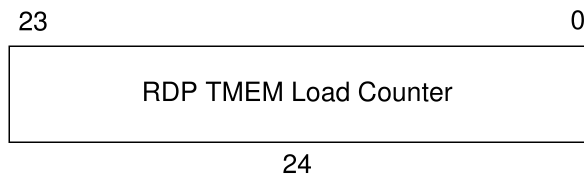
This register holds a RDP pipe busy counter, incremented on each cycle of the RDP clock that the RDP pipeline is busy. This register is READ ONLY.



On power-up, this register is undefined.

$c15

This register holds a RDP TMEM load counter, incremented on each cycle of the RDP clock while the TMEM is loading. This register is READ ONLY.



On power-up, this register is undefined.

CPU Point of View

The RSP Coprocessor 0 registers (and certain other RSP registers) are memory-mapped into the host CPU address space.

Bit patterns for READ and WRITE access are the same as described in the previous section.

Table 4-6 RSP Coprocessor 0 Registers (CPU VIEW)

Register Number	Address	Access Mode	Description
`$c0`	`0x04040000`	RW	I/DMEM address for DMA.
`$c1`	`0x04040004`	RW	DRAM address for DMA.
`$c2`	`0x04040008`	RW	DMA READ length (DRAM → I/DMEM).
`$c3`	`0x0404000c`	RW	DMA WRITE length (DRAM ← I/DMEM).
`$c4`	`0x04040010`	RW	RSP Status.
`$c5`	`0x04040014`	R	DMA full.
`$c6`	`0x04040018`	R	DMA busy.
`$c7`	`0x0404001c`	RW	CPU-RSP Semaphore.
`$c8`	`0x04100000`	RW	RDP command buffer START.
`$c9`	`0x04100004`	RW	RDP command buffer END.
`$c10`	`0x04100008`	R	RDP command buffer CURRENT.
`$c11`	`0x0410000c`	RW	RDP Status.
`$c12`	`0x04100010`	R	RDP clock counter.
`$c13`	`0x04100014`	R	RDP command buffer BUSY.
`$c14`	`0x04100018`	R	RDP pipe BUSY.
`$c15`	`0x0410001c`	R	RDP TMEM BUSY.

Other RSP Addresses

These are also memory-mapped for the CPU.

Table 4-7 Other RSP Addresses (CPU VIEW)

Address	Access Mode	Description
`0x04000000`	RW	RSP DMEM (4096 bytes).
`0x04001000`	RW	RSP IMEM (4096 bytes).
`0x04080000`	RW	RSP Program Counter (PC), 12 bits.

DMA

All data operated on by the RSP must first be DMA'd into DMEM. RSP programs can also use DMA to load microcode into IMEM.

Note: loading microcode on top of the currently executing code at the PC will result in undefined behavior.

Alignment Restrictions

All data sources and destinations for DMA transfers must be aligned to 8 bytes (64 bits), in both DRAM and I/DMEM.

Transfer lengths must be multiples of 8 bytes (64 bits).

Timing

Peak transfer rate is 8 bytes (64 bits) per cycle. There is a DMA setup overhead of 6-12 clocks, so longer transfers are more efficient.

IMEM and DMEM are single-ported memories, so accesses during DMA transfers will impact performance.

DMA Full

The DMA registers are double-buffered, having one pending request and one current active request. The DMA FULL condition means that there is an active request and a pending request, so no more requests can be serviced.

DMA Wait

Waiting for DMA completion is under complete programmer control. When DMA_BUSY is cleared, the transaction is complete.

If there is a pending DMA transaction, this transaction will be serviced before DMA_BUSY is cleared.

DMA Addressing Bits

Since all DMA accesses must be 64-bit aligned, the lower three bits of source and destination addresses are ignored and assumed to be all 0's.

Transfer lengths are encoded as (length - 1), so the lower three bits of the length are ignored and assumed to be all 1's.

The DMA LENGTH registers can be programmed with a line count and line stride, to transfer arbitrary rectangular pieces of memory (such as a portion of an image). See Figure 4-1, "DMA Transfer Length Encoding," on page 84, for more information.

CPU Semaphore

The CPU-RSP semaphore should be used to share DMA resources. Since the CPU could possibly DMA data to/from the RSP while the RSP is running, this semaphore is necessary to share the DMA engine.

> **Note:** The current graphics and audio microcode assume the CPU will not be DMA'ing data to/from the RSP while the RSP is running. This eliminates the need to check the semaphore (on the RSP side), saving a few instructions.

DMA Examples

The following examples illustrate programming RSP DMA transactions:

Figure 4-2 DMA Read/Write Example

```
 ################################################
 # Procedure to do DMA reads/writes.
 # Registers:
 #       $20     mem_addr
 #       $19     dram_addr
 #       $18     dma_len
 #       $17     iswrite?
 #       $11     used as tmp
.name   mem_addr,       $20
.name   dram_addr,      $19
.name   dma_len,        $18
.name   iswrite,        $17
.name   tmp,            $11

DMAproc: # request DMA access: (get semaphore)
                mfc0    tmp, SP_RESERVED
             bne    tmp, zero, DMAproc
                # note delay slot
       DMAFull: # wait for not FULL:
                mfc0    tmp, DMA_FULL
                bne     tmp, zero, DMAFull
                nop
        # set DMA registers:
                mtc0    mem_addr, DMA_CACHE
        # handle writes:
                bgtz    iswrite, DMAWrite
                mtc0    dram_addr, DMA_DRAM
                j       DMADone
                mtc0    dma_len, DMA_READ_LENGTH
        DMAWrite:
                mtc0    dma_len, DMA_WRITE_LENGTH
        DMADone:
                jr      return
                 # clear semaphore, delay slot
                mtc0    zero, SP_RESERVED
.unname mem_addr
.unname dram_addr
.unname dma_len
.unname iswrite
.unname tmp
 #
 ################################################
```

Figure 4-3 DMA Wait Example

```
 ############################################
 # Procedure to do DMA waits.
 #
 # Registers:
 #
 #       $11     used as tmp
 #
.name   tmp,    $11

DMAwait:
        # request DMA access: (get semaphore)
                mfc0    tmp, SP_RESERVED
                bne     tmp, zero, DMAwait
                # note delay slot
        WaitSpin:
                mfc0    tmp, DMA_BUSY
                bne     tmp, zero, WaitSpin
                nop
                jr      return
                 # clear semaphore, delay slot
                mtc0    zero, SP_RESERVED
.unname tmp
 #
 #
 #############################################
```

Controlling the RDP

The RDP has an independent DMA engine which reads commands from DMEM or DRAM into the command buffer. The RDP command buffer registers are programmed to direct the RDP from where to read the command data.

How to Control the RDP Command FIFO

RDP commands are transferred from memory to the command buffer by the RDP's DMA engine.

The RDP command buffer logic examines the CMD_CURRENT and CMD_END registers and will transfer data, 8 bytes (64 bits) at a time, advancing CMD_CURRENT, until CMD_CURRENT = CMD_END.

CMD_START and CMD_END registers are double buffered, so they can be updated asynchronously by the RSP or CPU while the RDP is transferring data. Writing to these registers will set the START_VALID and/or END_VALID bits in the RDP status register, signaling the RDP to use the new pointers once the current transfer is complete.

When a new CMD_START pointer is used, CMD_CURRENT is reset to CMD_START.

Algorithm to program the RDP Command FIFO:

- start with CMD_START and CMD_END set to the same initial value.
- write RDP commands to memory, beginning at CMD_START.
- when an integral number of RDP commands have been stored to memory, advance CMD_END (CMD_END should point to the *next* byte after the RDP command).
- keep advancing CMD_END as subsequent RDP commands are stored to memory.

Examples

The XBUS is a direct memory path between the RSP (and DMEM) and the RDP. This example uses a portion of DMEM as a circular FIFO to send data to the RDP.

This example uses an “open” and “close” interface; the “open” reserves space in the circular buffer, then the data is written, the “close” advances the RDP command buffer registers.

The first code fragment illustrates the initial conditions for the RDP command buffer registers.

Figure 4-4 RDP Initialization Using the XBUS

```
# XBUS initialization
   addi    $4, zero, DPC_SET_XBUS_DMEM_DMA
   addi    outp, zero, 0x1000 # DP init conditions
   mtc0    $4, CMD_STATUS
   mtc0    outp, CMD_START
   mtc0    outp, CMD_END
```

The `OutputOpen` function contains the most complicated part of the algorithm, handling the “wrapping” condition of the circular FIFO. The wrapping condition waits for CMD_CURRENT to advance before re-programming new CMD_START and CMD_END registers.

Figure 4-5 OutputOpen Function Using the XBUS

```
.name   dmemp,   $20
.name   dramp,   $19
.name   outsz,   $18 # caller sets to max size of write
 # open(size)    - wait for size avail in
 #                 ring buffer.
 #               - possibly handle wrap
 #               - wait for 'current' to get
 #                 out of the way
                .ent    OutputOpen
OutputOpen: # check if the packet will fit in the buffer
                addi    dramp, zero, (RSP_OUTPUT_OFFSET
                           + RSP_OUTPUT_SIZE8)
                add     dmemp, outp, outsz
                sub     dramp, dramp, dmemp
                bgez    dramp, CurrentFit
                nop
WrapBuffer: # packet won't fit, wait for current to wrap
                mfc0    dramp, CMD_STATUS
                andi    dramp, dramp, 0x0400
                bne     dramp, zero, WrapBuffer
AdvanceCurrent:  # wait for current to advance
                mfc0    dramp, CMD_CURRENT
                addi    outp, zero, RSP_OUTPUT_OFFSET
                beq     dramp, outp, AdvanceCurrent
                nop
                mtc0    outp, CMD_START  # reset START
CurrentFit:  # done if current_address <= outp
                mfc0    dramp, CMD_CURRENT
                sub     dmemp, outp, dramp
                bgez    dmemp, OpenDone

     # loop if current_address <= (outp + outsz)
                add     dmemp, outp, outsz
                sub     dramp, dmemp, dramp
                bgez    dramp, CurrentFit
                nop
OpenDone:
                jr      return
                nop
                .end    OutputOpen
```

After calling `OutputOpen`, the program writes the RDP commands to DMEM, advancing `outp`. Once the complete RDP command is written to DMEM, `OutputClose` is called.

Figure 4-6 OutputClose Function Using the XBUS

```
####################################################
# OutputClose
####################################################
                .ent    OutputClose
OutputClose:
        #
        # XBUS RDP output
        #
                jr      return
                mtc0    outp, CMD_END
                .end    OutputClose
.unname outsz
.unname dramp
.unname dmemp
```