HARDWARE AND SOFTWARE BOOT INITALIZATION OF THE IDT79R4000

APPLICATION NOTE AN-119

By Andrew Ng

INTRODUCTION

This application note is aimed at engineers that are bringing up or debugging an R4000 system prototype for the first time. Various debug techniques, pitfalls, and diagnostics are discussed that are based on similar experiences of other engineers here at IDT. The discussions will be a mixture of both hardware and assembly code software, since both hardware and software skills and techniques are required to initialize the part. In places, the R4000 User's Manual [2] will need to be referred to for more detail. The topics will proceed in a chronological order that begins with power on, continuing through the Reset sequence and finishing with some simple diagnostics — similar to the order that one might take when actually debugging a prototype board.

The first section details the hardware Reset sequencing, which includes managing the various Reset control lines and loading the R4000s serial configuration register. After the Reset sequence, the R4000 issues its initial instruction fetch. Logic analyzer connections are discussed so that the instruction fetch and other System Interface reads and writes can be verified. Then, the first few lines of boot assembly code, which determine some of the software programmable configuration options that the R4000 can do, are discussed. Some example assembly code for the initial testing of uncached read and write cycles to memory and I/O is given. Finally, in the last section, initialization of the caches is discussed so that block reads and writes can be executed and debugged. After reaching this stage of the debug, the chances of an operating system kernel booting up with a prompt are fairly good.

HARDWARE RESET SEQUENCE

In Figure 1, the R4000 Reset Interface requires the generation of several control signals, including VCCOk, $\overline{\text{ColdReset}}$, and $\overline{\text{Reset}}$. Primarily, these signals distinguish between power-on Resets, power-on-cold resets and power-on-warm resets, and to allow sufficient time for the PLL (Phase Locked Loop) circuitry to stabilize. Only the power-on reset is discussed in detail, since the cold and warm resets controls are a subset of the power-on case.

The first requirement is that VCCOk, which indicates that the supply voltage has reached at least 4.75V for 100ms or more, be de-asserted. The 100 ms de-assertion time is typically accomplished by using a power management chip which delays a power-up signal until a fixed time period or RC (Resistor/Capacitor) constant has elapsed. The power-up signal can be double-registered so that it is synchronized for the assertion of VCCOk. $\overline{\text{ColdReset}}$ and $\overline{\text{Reset}}$ must be de-asserted sometime before VCCOk is asserted. De-asserting VCCOk holds both the ModeClock and the output clocks, such as MasterOut, HIGH. (Although the ModeClock is guaranteed to be HIGH, the value of MasterOut is not guaranteed, technically, until after the PLL synchronizes). If MasterOut is used to clock the reset circuitry state machine, $\overline{\text{ColdReset}}$ and $\overline{\text{Reset}}$ must be de-asserted asynchronously from the output clocks. Technically, $\overline{\text{ColdReset}}$ and $\overline{\text{Reset}}$ are sampled synchronously when asserting and de-asserting. Therefore, while using the input clock, MasterIn to clock the reset circuitry state machine may make more sense than using MasterOut.

In Figure 2, 128 Master Clocks (either MasterIn or MasterOut) after VCCOk is asserted, the ModeClock will begin toggling by first going LOW and then 128 Master Clocks after that going HIGH for the first time. Thus the ModeClock period is 256 Master Clocks. On the first rising edge of the ModeClock, the R4000 starts accepting serial data on the ModeIn pin. Many systems use an Nx1 bit serial PROM for this function. Because the setting of the mode bits can be somewhat experimental when first bringing up a system, one might choose a reprogrammable serial bit EEPROM, or, perhaps, use a signal generator. Most serial bit PROMs have a built-in address incrementor/counter which requires a Clock input pin and a Reset input pin, in addition to the Data output pin. Thus, the serial PROM has an internal counter to generate the address for the mode bit data. When using a signal generator, one should consider designing in an inverter to invert the ModeClock, so the pattern generator can synchronize on the first falling edge of ModeClock, and, thus drive valid data in time for the first ModeClock rising edge. Using the inverted the ModeClock also provides ample hold time.

Sometime after the mode bits have been read, the R4000 will begin driving the output clocks. From the point where VCCOk is asserted, the R4000 needs to see a minimum of 64K Master Clocks (either MasterIn or MasterOut, which is just enough to read all the mode bits). A time of at least 100ms is more realistic before $\overline{\text{ColdReset}}$ can be de-asserted, so internal syncing of the PLL can be completed and fully stabilized throughout the system. Several ways exist to count out this period (a 50MHz MasterIn clock is assumed). One is to use a 24-bit counter based off the MasterIn clock. Another is to use a RC circuit to generate a 100 ms delay from VCCOk and then synchronize the resulting $\overline{\text{ColdReset}}$ signal by double-registering it. Another is to use a 16-bit counter based off the ModeClock, which, although not specified, continues to toggle, even after the mode bits have been read in. A fourth method can use some serial PROMs, which have a count/done pin that asserts LOW after all the bits have been read. If the number of bits is greater than 32K, then an adequate delay can be generated.

After $\overline{\text{ColdReset}}$ is de-asserted, then $\overline{\text{Reset}}$ must be de-asserted after a minimum of 64 Master Clocks have occurred. This requires a 6-bit counter, since $\overline{\text{Reset}}$ must be de-asserted synchronously.

The sequence for cold Resets is the same as power-up Resets, except that VCCOk needs only to be de-asserted for 64 Master Clocks, instead of 100ms. The sequence for warm Resets requires only that the $\overline{\text{Reset}}$ has asserted for 64 Master Clocks.

After the reset sequence, the R4000 will assert $\overline{\text{ValidOut}}$ along with an uncached read of the first instruction. The first instruction fetch will be discussed in more detail after the following section, which continues to specify the boot Reset configuration serial bits.

SERIAL BOOT MODE PROM — SPECIFIC CASES

The R4000 requires that 256 bits be serially loaded into its initialization logic on its ModeIn input pin for the first 256 ModeClocks. Of the 256 bits, only the first 64 are defined. Although specific systems will have specific values, an example of some “workable” values that can be used as a start for debugging are listed in Table 1 in binary. The rest are reserved to 0. Most of the bits are described by the R4000

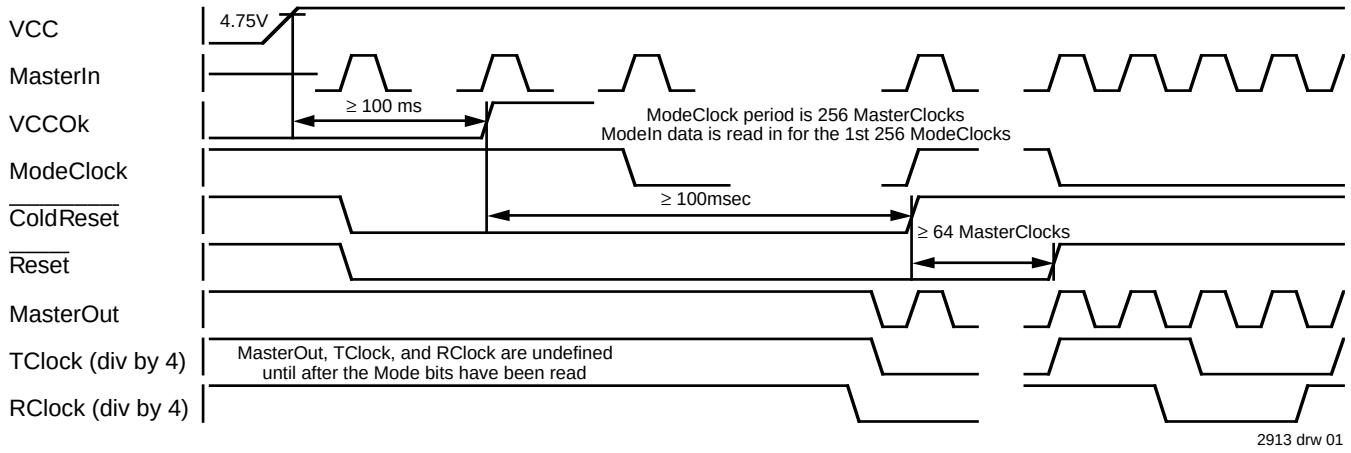


Figure 1. R4000 Reset Timing

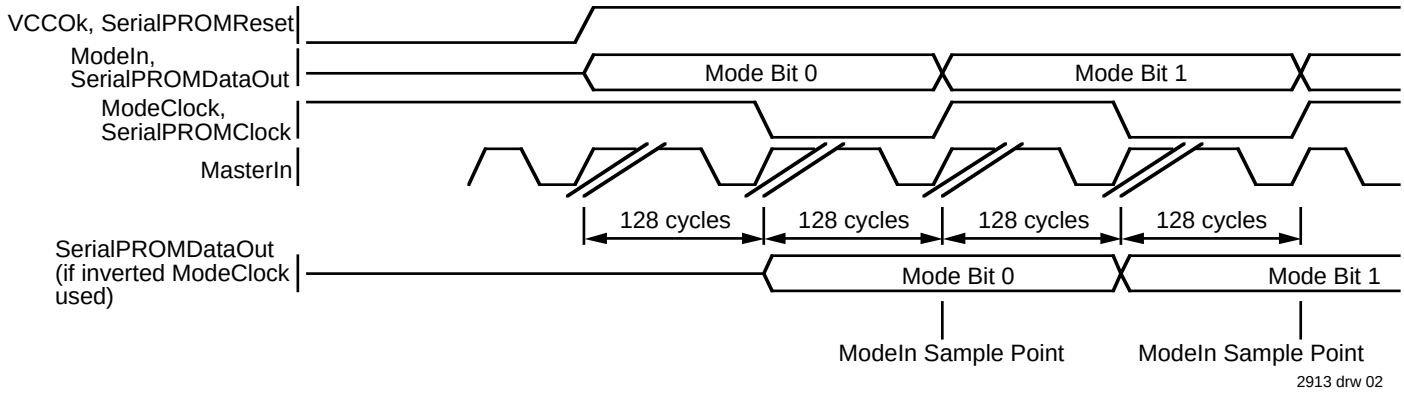


Figure 2. R4000 Serial Initialization Timing

Users Manual [2]. However, the values to choose for some bits can be confusing during initial debug. An example is the PLLOn configuration bit. This bit is intended only for chip testing and should be left asserted. The symptom is that the MasterOut and other clock outputs will not toggle. The implication is that the lowest MasterIn clock speed that can be used is 25MHz (for a 50MHz part). However, the SClock divisor configuration bits called SysCkRatio can be programmed to divide by 2, 3, or 4 which can reduce the System Interface frequency to 6.25MHz. One of the most common and perplexing hindrances in finding problems, at 50MHz, is having a noisy clock line to one of the state machines. This noise can clock a signal twice, or perhaps not at all. Therefore, reducing the System Interface frequency during the initial stages of testing is highly recommended.

TABLE 1. EXAMPLE OF SERIAL BOOT PROM VALUES.

Mode Setting	Value	Comments
BlkOrder	1	1 for sub-block ordering if PC, 0 for sequential ordering if SC/MC
EIBParMode	0	ECC
EndBIt	0	Little Endian ordering
DShMdId	0	dirty shared mode enabled
NoSCMode	0	present (depends on package type)
SysPort	00	64 bits
SC64BitMd	0	128 bits
EISpltMd	0	Secondary cache unified
SCBlkSz	11	Secondary block size of 32 words (depends on system)
XmitDatPat	0000	Xmit Data Pattern DD (depends on system)
SysClkRatio	010	system interface bus divided by 4 (see text)
reserved	0	
TimIntDis	0	timer interrupt connection enabled
PotUpdDis	0	potential updates disabled
TWrSUp	0011	(SC write de-assertion delay, depends on SC timing, minimum shown)
TWr2Dly	01	(SC write assertion delay 2, depends on SC timing, minimum shown)
TWr1Dly	01	(SC write assertion delay 1, depends on SC timing, minimum shown)
TWrRc	0	(SC write recovery time, depends on SC timing, minimum shown)
TDis	010	(SC disable time, depends on SC timing, minimum shown)
TRd2Cyc	0011	(SC read cycle time 2, depends on SC timing, minimum shown)
TRd1Cyc	0100	(SC read cycle time 1, depends on SC timing, minimum shown)
reserved	0000	
Pkg179	0	Large Package (depends on package type)
CycDivisor	0011	power down clock divisor
Drv	100	1 clock Drive delay
InitP	0001	pull down di/dt (msb is opposite most fields)
InitN	1000	pull up di/dt
EnblDPLLR	0	disable di/dt mechanism during cold Reset
EnblDPLL	0	disable di/dt mechanism
DsblPLL	0	Enable PLLs (see text)
SRTristate	1	tri-state when Reset or ColdReset is asserted
Bits65:255	0	rest of the bits are reserved

2913 tbl 01

During debug, other serial boot configuration bits that may be of use are the SCBlkSize, which configure the secondary cache line size, if present, to 4, 8, 16, or 32 words. This will control the maximum size of block reads and writes for secondary cache systems. Also, the XmitDatPat bits configure the system interface data rate with various patterns such as D, DDx, DDxx, etc. Another design consideration is if the secondary cache is not used, then sub-block ordering, as programmed with the BlkOrder bit, is mandatory.

BASIC LOGIC ANALYZER CONNECTIONS

After the serial configuration register is read, the majority of the debug effort centers around memory bus cycles on the System Interface. For this reason it is recommended that most of the System Interface be accessible from a Logic Analyzer. This includes the information on the SysAD(63:0) bus.

Two items should be considered when attaching the SysAD bus to a logic analyzer. The first is the latching control circuitry of the SysAD bus as shown in Figure 3. To demultiplex it into separate MemAddr(35:0) and MemData(63:0) busses is usually straightforward, but the multi-level write buffering of SysAD into the MemAddr and MemData is not. Thus, if there are enough pod connections, one should hook up MemData, MemAddr, and SysAD. However, the second consideration is that there usually are not enough pods or probes to do this. Therefore, in a compromise, attaching SysAD is probably more useful than attaching MemAddr, since MemAddr is usually a single level deep register, latch, or buffer. However, it is essential to look at the least significant MemAddr lines to verify that the address can be incremented within a block correctly, especially if sub-block ordering is used. Also, using MemAddr instead of SysAD only requires 36 probes, and possibly less, if not all the physical address lines are used. A

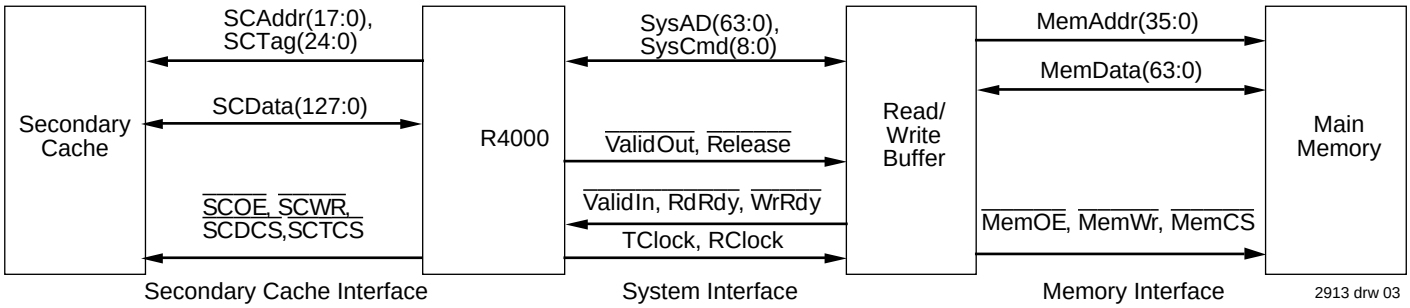


Figure 3. Typical R4000 System

make-shift solution is to hookup only 32 data lines at a time, either MemData(63:32) or MemData(31:0). The upper/lower halves can be swapped as needed, since, during the initial debug, the function of the lines is more important than examining the sequential flow of instructions and data.

It is also essential to bring out the entire SysCmd(8:0) bus. This bus acts as the control and status lines, and determines whether the transaction is a read or write, etc. Along with SysCmd bus, ValidOut, ValidIn, and Release are essential, since they indicate when the SysAD and SysCmd busses are valid, and when they can be driven by the memory system. RdRdy and WrRdy and read/write buffer control lines such as MemOE and MemWr are also sometimes needed.

If a state analyzer is being used, one should consider attaching the RClock output, which leads the TClock, that is usually used by 25% (of the TClock period), as the state clock to trigger the logic analyzer, so sufficient hold time is provided (at the expense of having less setup time). Otherwise one of the other output clocks, either the TClock or the MasterOut, should be attached.

Thus, the minimum number of logic analyzer probes needed is 64+9+3+1=77. A typical number would be 64+32+9+3+1=109 and could be as many as 64+36+64+9+3+1=177. Additional pods will be needed to test for specific cases, such as the control lines during Reset, the ECC bits during fault checking, etc.

If the secondary cache is present, one should be prepared to examine its interface. However, because of the enormous number of lines (128 data, up to 36 address and tag, and 4 control lines) and the relative straightforwardness of the functional design, the secondary cache will probably only need to be on the logic analyzer temporarily. The secondary cache lines may require oscilloscope probing to verify the electrical signal transmission line design. To help follow the processor flow, leaving the control lines SCOE and one of the SCWr lines connected to the logic analyzer at all times can be helpful.

MINIMAL SOFTWARE BOOT CODE

After Reset, the R4000 will be executing instructions out of uncached memory kernel segment 1 space at virtual address ‘h bfc0 0000, which is hard mapped to physical address ‘h 01fc0 0000. ValidOut will assert LOW, and the SysCmd(8:0) bus will indicate an uncached read of 1 word, ‘b 10011011, and, on a little endian machine, will expect data on SysAD(31:0) at the same time ValidIn is asserted. Big endian machines will expect data on SysAD(63:32). During uncached reads of addresses divisible by 8, (number of bytes per double word), SysAD(63:32) will be ignored on little endian machines. Big endian machines will ignore SysAD(31:0). The second instruction fetch will be similar, except it will be at physical address ‘h 01fc0 0004, and a little endian machine will expect the data to be put on SysAD(63:32), with ValidIn asserted, while SysAD(31:0) is ignored. Likewise, big endian machines will expect the data to be put on SysAD(31:0), while SysAD(63:32) is ignored. The minimal boot code discussed here will get the part initialized and allow various types of memory accesses to take place. This includes initializing the caches so that block reads and writes can be tested.

One common cause of no system commands being generated (ValidOut never asserts), is the GroupStall input pin (if present for the particular R4000 version/type) has to be deasserted.

The next section will discuss the very first operation software should do, namely, initializing the software configuration registers. After initializing the registers, the software can execute various kinds of reads and writes to uncached memory space in order to test the ROM, I/O and RAM chip selects, byte enables, and wait-state timing.

Configuration Registers $14 and $16

The first operation that the boot code software needs to perform is to initialize the software configurable registers. This includes the Status Register and Configuration Register. Most of the registers do not have default values on Reset, and must be programmed before being used. The Status Register, Configuration Register, and the WatchLo Register have effects on loads, stores, and processor operations that must immediately be programmed into a known state. An example of programming Status Register $14 and Configuration Register $16 settings is shown in Listing 1. The other general purpose and coprocessor registers, including the Timer and Compare registers must be initialized before they are used.

```
set noreorder
li              v0,0x30410000   # load constant with CP1, CP0 usable,
                                #  BEV, DE set, IE (interrupts) disabled
mtc0    v0,$14          # move it to the Status Reg
mtc0    zero,$18        # clear R and W trap enable masks in the WatchLo Reg
nop                     # an operation is needed between a mtc0 and mfc0 instruction
mfc0    v1,$16          # get Configuration Reg
nop                     # delay two operations before v1 can be used
li              a0,0xa0000160   # load address constant
sw              v1,0(a0)        # dump Configuration Reg to external memory
li              v0,0x00000033   # load constant with IB, DB set to 32 byte p-cache line widths
                                #    and Kseg0 to be non-coherent cachable
mtc0    v0,$16          # move it back to the Configuration Reg (only bits 5:0 writable)
```

Listing 1. Software for Reading and Writing the Configuration Registers

Figure 4 shows the register fields. Refer to the User's Manual [2] for more detail.

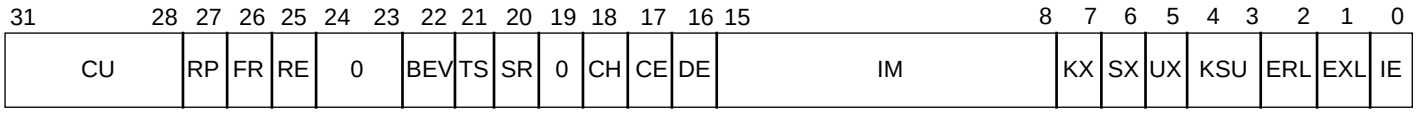


Figure 4. Status Register $14

Two suggestions on programming these fields during initial debugging are to set the BEV bit and the DE bit. Setting BEV, bit 22, the Diagnostic Status Field of the Status Register $14 will send any exceptions to the uncached kernel segment 1 bootstrap exception vector base virtual address 'h bfc0 0200, instead of to the cachable mapped user segment 'h 8000 0000, which requires that the cache and TLB be initialized first. An exception handler for initial diagnostics, such as the (unoptimized) one in Listing 2, can put code at physical address 'h 01fc0 0200 and offsets 'h 0000, 'h 0080, 'h 0080, 'h 0100, and 'h 0180, i.e., physical addresses, 'h 01fc0 0200, 'h 01fc0 0280, 'h 01fc0 0300, 'h 01fc0 0380. The exception handler should at least dump out the cause register $13, the exception vector, $14, and the cache error register $27, and the error exception program counter, $30. If the registers can't be displayed with a UART, they should at least be written out to uncached memory so they can be observed on a logic analyzer. In contrast to the R3000 RFE instruction, the R4000 uses an ERET instruction to return back to the code.

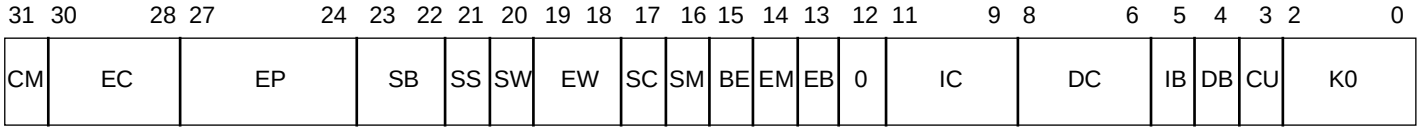
```
li      a0,0xa0000000 # load address constant

mfc0    v1,$13          # get Cause Reg
nop                     # two non-v1 operations needed
nop                     #
sw      v1,0x130(a0)    # dump to memory

mfc0    v1,$14          # get EPC
nop                     # two non-v1 operations needed
nop                     #
sw      v1,0x140(a0)    # dump to memory

mfc0    v1,$27          # get CacheErr Reg
nop                     # two non-v1 operations needed
nop                     #
sw      v1,0x270(a0)    # dump to memory

mfc0    v1,$30          # get ErrorEPC Reg
nop                     # two non-v1 operations needed
nop                     #
sw      v1,0x300(a0)    # dump to memory

mfc0    v1,$12          # get Status Reg
nop                     # two non-v1 operations needed
nop                     #
sw      v1,0x120(a0)    # dump to memory

eret                    # return from exception
```

Listing 2. Software for the Exception Handler

The DE bit is bit 16 in the Diagnostic Status Field of the Status Register $14, when set specifies that cache parity and ECC errors don't cause exceptions. This is somewhat necessary when initializing the cache, otherwise a lot of unnecessarily confusing jumps to the exception handler will probably occur as the cache locations are first initialized, since the tag and data parity haven't been initialized yet.

The WatchLo Register must have its trap on a read/load mask (bit 1) and trap on a write/store mask (bit 0) disabled before loads or stores are attempted, so an inadvertent trap from an address match is not taken. Thus, similar to the example in Listing 1, the WatchLo register can be cleared.

Reading the Configuration Register and dumping its contents out to uncached space allows one to see if various bits in the boot serial PROM were programmed correctly. Figure 4 shows the Configuration Register fields. For instance, the System clock ratio and the transmit data pattern can be checked. The only writable bits are the lower 6, which are also uninitialized on Reset, and, therefore, should be written to by software as soon as possible. Of the writable bits, IB and DB are used to program the primary Instruction Cache line size and the primary Data cache line size to either 4 words or 8 words. The primary cache line sizes must be smaller than or equal to the secondary cache line size. Note that if there is no secondary cache, it is possible to program the data and instruction caches to different line sizes, which is the one case where different block sizes will be presented to the system interface.



2913 drw 05

Figure 4. Configuration Register $16

PRIMARY DATA CACHE INITIALIZATION

In general, it is much simpler to test the data cache than it is the instruction cache. Several reasons exist for this. First, if the data cache read fails, the program can still continue, where as an instruction cache failure may or may not continue and could cause the program to get lost. Second, it is simpler to initialize the data cache since it can be written directly with stores. Finally, forcing cache miss writebacks is more straightforward, since it just requires writing to different addresses as opposed to jumping back and forth in code. As shown in Listing 3, when initializing the caches, the Cache opcode is used heavily. The algorithm in Listing 3 is not the most efficient. However, from a debugging point of view, it does not do any unnecessary System Interface block reads or writes. The idea is to, first, invalidate the tags, and then fill the data slots with any data so that ECC/parity can be set correctly. The base virtual address, 'h 8000 0000, is used because it is in the unmapped cachable kernel segment 0, which does not require the TLB. Note that if an R3000 compiler is being used, which can't generate the R4000 Cache opcode, then a data statement using the ".word" directive can be inserted into the program with the data for the hand assembled hex machine instruction.

In a similar manner, by substituting the appropriate cache instructions, and by adjusting for the cache line size, the secondary data cache can be initialized.

```
    set noreorder                          /* turn off assembly rescheduler (no reordering optimization) */
    li          a0,0x80000000              /* primary data cache start pointer */
    li          a1,0x80002000-0x20         /* 8K last location - 32 */
    mtc0   zero,$28                        /* set TagLo CP0 Reg to 0 */
#ifndef R3000asm
1:          cache       2*4+1,0x00(a0)     /* Index Store Tag, invalidate cache line (prevent writebacks) */
            cache       3*4+1,0x00(a0)     /* Create Dirty Exclusive (prevent block reads) */
#else
1:          .word       0xbc890000         /* use if using R3000 assembler */
            .word       0xbc8d0000
#endif
            nop
            nop
            sw          zero,0x00(a0)      /* fill data slots with good ECC/parity (8 word cache lin          e) */
            sw          zero,0x04(a0)
            sw          zero,0x08(a0)
            sw          zero,0x0c(a0)
            sw          zero,0x10(a0)
            sw          zero,0x14(a0)
            sw          zero,0x18(a0)
            sw          zero,0x1c(a0)
            nop
            nop
#ifndef R3000asm
            cache       2*4+1,0x00(a0)     /* Index Store Tag, invalidate cache line */
#else
            .word       0xbc890000
#endif
            blt         a0,a1,1b           /* if count is less than last */
                                           /* then jump Back to last label called “1”. */
            addu        a0,0x20            /* branch delay slot, increment addr pointer */
```

Listing 3. Primary Data Cache Initialization Software

PRIMARY INSTRUCTION CACHE INITIALIZATION

As shown in Listing 4, the instruction cache is initialized a little differently than the data cache. First, their data slots need to be filled from main memory, using the Fill Cache operation, so the ECC/parity for the data can be set correctly. Then, their tags are invalidated and tag ECC/parity set. As with the data cache, the base virtual address ‘h 8000 0000 is used because it automatically maps to a physical address without requiring the use of the TLB.

The secondary instruction cache can be initialized in a similar manner to the primary cache. The initialization can be accomplished by using the cache fill instruction over the entire secondary cache address space, adjusting for the cache line size, and by substituting the appropriate cache instructions. Note, that if the secondary cache has a unified instruction and data memory, then the cache only needs to initialized once.

```
                .set noreorder                  /* turn off assembly rescheduler */
                                                /*  (no reordering optimization) */

                li      a0,0x80000000           /* primary data cache start pointer */
                li      a1,0x80002000-0x20      /* 8K last location - 32 */
                mtc0    zero,$28                /* set TagLo CP0 Reg to 0 */

1:
#ifndef block_reads_are_being_tested_later
                cache   5*4+0,0x00(a0)          /* fill i-word data slots (8 word cache line size) */
                                                /* 0xbc940000 */
                                                /* note that the fill operation requires that block */
                                                /* reads are working.  Thus during initial debug */
#endif                                          /* one may want to delete the fill operation */

                cache   2*4+0,0x00(a0)          /* index store tag */ /* 0xbc880000 */
                blt     a0,a1,1b                /* if count is less than last */
                                                /* then jump Back to last label called “1”. */
                addu    a0,0x20                 /* branch delay slot, increment addr pointer */
```

Listing 4. Primary Instruction Cache Initialization Software

MINIMAL TEST CODE (BLOCK READS AND WRITES)

Cachable data loads will read from the internal primary cache or the secondary cache, unless the cache line location is invalid or has a non-matching tag. Such cache misses will generate block reads to the external system interface.

The block reads are tested by doing a cached read, which misses in the cache. It is easier to look at cache locations that are initialized as invalid, so writebacks do not occur.

The data cache uses a writeback protocol. So, when writing to a cached location, the data is stored only to the cache, and a dirty bit is set. Main memory is updated later, when the cache line, where the data was stored, is replaced for a cache miss. Because the cache is direct mapped, a cache miss can be created by writing or reading to locations that are modulo cache block size apart, i.e., every 8K apart.

Code in Listing 5 shows a method that may be needed early-on, which is to test writebacks without doing a block read first.

After block reads and writes are tested individually, data writes to cache block offsets of 8K, as in Listing 6, will force a writeback. On the R4000PC without secondary cache, this will be two separate System Interface transactions. However, on R4000s with secondary cache, the write address and data will be issued immediately following the read address, such that the write address and data will come between the read address and when data is returned by the system. In a typical system, the write address and data is FIFO buffered such that after the read is handled, the system issues the write to main memory.

```
/* assume that cache has just been flushed (invalidated) */

li      a0,0x80000000       /* start addr pointer */
cache   3*4+1, 0x00(a0)     /* Create Dirty Exclusive, otherwise a block read
                               will occur on the first store so that the entire cache
                               line is filled */ /* 0xbc8d0000 */
nop
addiu   a1,a0,0x00          /* store incrementing pattern, i.e., 0x0, 0x4, 0x8, 0xC */
sw      a1,0x00(a0)         /* into cache */
addiu   a1,a0,0x04
sw      a1,0x04(a0)
addiu   a1,a0,0x04
sw      a1,0x08(a0)
addiu   a1,a0,0x04
sw      a1,0x0c(a0)
addiu   a1,a0,0x04
sw      a1,0x10(a0)
addi    a1,a0,0x04
sw      a1,0x14(a0)
addiu   a1,a0,0x04
sw      a1,0x18(a0)
addiu   a1,a0,0x04
sw      a1,0x1c(a0)
nop                         /* 2 operations required between store and cache */
nop
cache   0*0+1,0x00(a0)      /* index write back invalidate */ /* 0xbc810000 */
```

Listing 5. Block Write Code with No Block Read

```
li      a0,0x80000000       /* load start addr pointer */
li      a1, zero            /* load data */
sw      a1,0x0000(a0)       /* read from 0000 and possible writeback to xxxx */
li      a1,0x2000           /* load data */
sw      a1,0x2000(a0)       /* read from 2000 and definite writeback to 0000 */
```

Listing 6. Block Read with Writeback

TESTING ALL THE PHYSICAL ADDRESS LINES

The R4000 has 36 of the physical address lines implemented. Although unspecified, one can customarily expect SysAD(63:36) to be 0 during any address phase. Only the bottom 30 out of 36 physical address bits can be tested within the unmapped fixed kernel space provided with 32-bit virtual addressing. One way to test address bits 35:32 is to go into 64-bit virtual addressing by setting the KX (bit 7) in the Status Register $14 and then using the 64-bit kernel space called xkphys. Virtual addresses ‘h 9000 0000 0000 0000 to ‘h 97ff ffff ffff ffff are uncached and automatically mapped such that physical address bits 35:0 are the same as virtual address bits 35:0.

A second, but more tedious way to test address bits 35:30, is to use the mapped space via the Translation Lookaside Buffer (TLB) which converts the software program’s virtual address into the hardware’s physical address. Although initialization of the TLB is beyond the scope of this application note, one tip includes initializing all 48 entries, not just the ones going to be used. This is because the unused entries may happen to power up with a matching virtual address. Should two or more TLB entries match, a TLB shutdown may occur and the CPU does not know which one to choose. In addition, initialize the TLB virtual pages to an unmappable unmapped virtual address space such as ‘h 0x8000 0000 as well as setting the entry’s Valid bit to invalid. This is because the TLB shutdown logic, when two or more entries match, does not take into account the valid bit. Since ‘h 8000 0000 is automatically mapped to a physical address space, and does not go through the TLB, those entries cannot accidently cause a shutdown.

SUMMARY

Bringing up the hardware requires a mixture of hardware and software. The part must be Reset, serial configuration registers loaded and software configuration registers written. A mixture of single doubleword reads, writes, block reads, and block writes can be checked. Reaching this stage is usually sufficient to continue with more intensive diagnostics and operating systems. Continued diagnostics may include interrupt line checks, memory checks, and I/O initialization.

FOR FURTHER INFORMATION

[1] MIPS R4000 Microprocessor Introduction, Integrated Device Technology, Inc., MAN-RISC-10091, Santa Clara, CA, 1991. — Gives a brief general overview of the architecture and features.

[2] MIPS R4000 User's Manual, Integrated Device Technology, Inc., MAN-RISC-00091, Santa Clara, CA, 1991. — Describes the H/W features and functionality of the device as well the bus interface. Also describes the R4000 instruction set architecture from a systems and assembly level programming perspective.

[3] IDT79R4000 Family Data Sheet, Integrated Device Technology, Inc., Oct. 1991. — Contains the Data Sheet with packaging, pinout, AC/DC electrical specifications and thermal parameters.

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit www.renesas.com/contact-us/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.