PowerPC^{TM} Endian Switch Code

Gary Y Tsao
Power Personal Systems Division
IBM Corporation
Austin, Texas 78758
e-mail: tsao@futserv.austin.ibm.com
May 20, 1994

Abstract

PowerPC^{TM} Reference Platform systems may need to perform endian switch to run operating systems in proper endian mode. This report summarizes various considerations when performing endian changing just after boot time for the current set of PowerPC processors and different bus bridges/memory controllers. For a general guideline, endian changing assembly codes are provided in this report.

1. Introduction

When a PowerPC^{TM} system comes up after power-on-reset, big-endian is in effect. When ROS firmware is ready to load the little-endian operating system, the ROS or a pre-loader has to perform the endian changing. The code here shows how to do the endian switch at boot time. Discussion of dynamically switching between cross endian applications is beyond the scope of this report.

The endian changing actually involves two separate actions, i.e. the Power PC endian switch and the system endian switch. In this report we will discuss two ways to perform the endian changing for a given implementation. In addition, the means to do endian switch between Power PC 601^{TM} and other Power PC processors is different. We will show two different methods to do the endian changing for different processors.

The entry conditions to the codes will be discussed. Some typical questions include: cache inhibit? interrupt disabled? address translation disabled? and processor state?

The PowerPC system planar boards need one or more bus bridge/memory controller to link system components. As pointed out in Appendix B of PowerPC Reference Platform Specification, there are at least three different implementations. The assembly codes provided here will cover the B.2.1 Bi-Endian Memory and I/O (Reference Implementation) and B.2.2 Bi-Endian I/O design. B.5 Full Bi-Endian Processor will not be covered here because the job of endian switching is much simpler for this case and a PowerPC processor of this kind is unavailable yet. Different bus bridges/memory controllers put the logic of "byte lane reverse" and "address modification (munge)" in different places. As we will discuss later, for some designs this creates problem for code loaded into the storage before the endian change is completed.

2. Entry Conditions

Some early endian test programs [1] were run with caches disabled. This forces programs to execute only from the main memory. This approach by-passes the cache pollution problem, which arises when I-cache instructions are not in proper endian order. If we disable caches we have to disable both L1 and possibly L2 caches. Usually, the way to disable the L2 cache is implementation dependent. The PowerPC 601 internal cache can be disabled by either mapping to memory-forced I/O controller space [2] or by Block Address Translation (BAT) mechanism with I bit set. Other PowerPC processors' cache can be disabled by either setting bits in the HID0 register or BAT mechanism with I bit set.

The cache pollution problem can also be solved with caches enabled. First, the D-cache can be flushed to ensure any data from firmware is pushed to memory. The I-cache can be invalidated. This is particularly important for the PowerPC 603^{TM} processor because the I-cache is not snooped. Finally, by using palindromic instructions which are stored exactly the same in big-endian or little-endian order (e.g. addi r0,r1,0x138 - 0x38010138), the concern of whether instructions are already in I-cache or the unlimited speculative prefetch of some PowerPC's (604 or $620)^{TM}$ becomes a non-issue. The palindromic instructions buffer the effect of cache line and prefetch uncertainty such that instructions are fetched and executed correctly regardless of when the processor or sytem endian modes are changed. With above considerations in mind, the decision for the code design is to leave caches enabled. These codes will run with caches disabled also.

The processor must be in supervisor state to execute any privileged instructions in this code. Address translation is enabled for both instruction and data. In the code, we assume the instruction and data are already mapped properly. So, the code can be translated and executed and load or store to memory or I/O port. This is true for Reference Implementation when control is passed from the ROS firmware. The interrupts are disabled upon entering the code. This is because the endian mode is unpredictable in the middle of endian switching and if interrupts are enabled, the interrupt handlers might not work. No storage access interrupts such as page fault is allowed because interrupts are disabled. When endian change is completed, the interrupt handlers must be initialized in the proper endian order before the interrupts are enabled again. Load or store to I/O space other than endian port should be avoided during the transition of endianess.

3. The Order To Switch Endian

There are two ways to switch endian. One is changing PowerPC first. The other is changing system endian first. Since the endian switching for PowerPC and system is not atomic, each method has different transition states. Different considerations would be applied to deifferent methods.

3.1 First Sequence

This sequence of endian switching is changing PowerPC first and the system endianess later. The PowerPC and system endian switching are not atomic. The system will behave very peculiarly when PowerPC little-endian is effected but system is still in big-endian mode. It is during this period that the confusion of endian changing occurs.

When PowerPC little-endian is in effect but before system's little-endian is in effect, the address munge [3- Appendix D] becomes effective. If we need to access system endian port (byte) address, say 0x80000092, the program needs to issue 0x80000095 instead, to compensate (unmunge) for the effect of PowerPC address modification. Also, in little-endian mode, instructions are fetched in big-endian order; however, the instructions are swapped within a double word before being passed to the instruction queue. So, in this state, the instructions are executed in reverse order within a double word. To illustrate this as a simplified example, assume following instruction sequence is loaded into storage in big-endian mode:

A

B --> PowerPC Switches Endian mode

C

D

E

F

Assume address of instruction A is double-word aligned. The PowerPC switches to little-endian mode at instruction B. After instruction B, the instruction execution sequence is as follows:

A

B

D

C

F

E

In general, instruction C initiates the system endian changing; usually a "store" instruction to an I/O port. If we put a no-op type instruction (e.g. previously discussed palindromic instruction) for D, instruction C will be executed correctly. The latency, before system endian is in effect, is not a constant. We don’t know exactly when the system is changed to little-endian mode. After instruction C, we put some palindromic instructions to wait for system endian switch to become effective. Palindromic instructions serve two purposes: as a delay timer and more importantly to guarantee that PowerPC executes valid instructions before or after the system is changed to little-endian mode. The number of palindromic instructions should cover the latency of the system endian switch plus a cache line of instructions after system endian changing. The latter guarantees that the instructions prefetched into I-cache are always valid even for the "aggressive prefetch" of 604 or 620. After these palindromic instructions, the first real little-endian instruction then follows. For a 66 MHz 601 of Reference Implementation, we found that between 22 to 24 palindromic instructions is adequate. Proportionally more palindromic instructions are required for higher speed PowerPC processor.

3.2 Second Sequence

This sequence of endian switching is changing system endian port first and PowerPC later. For B.2.2 implementation, this sequence is almost the same as the first sequence and there is no side effect with which to be concerned. For B.2.1 implementation, this sequence (as described in the endian changing code later) works only if the latency of system endian changing is more than 6 cycles. This is true for the Reference Implementation. This 6-cycle latency (see figure 4a, from stb to rfi) is required to insure that the PowerPC endian changing won’t be affected by system endian changing. If system endian changing is instantaneous, the code sequence would be complicated and this sequence is not recommended. In this approach, an address sent to the system enidan port is not munged because PowerPC is still in big-endian mode. Using palindromic instructions in this approach is similar to the first sequence as described above.

4. Impact of Bus Bridge/Memory Controller Design

The image of the endian changing code on the medium (disk) is depicted in Figure 1.

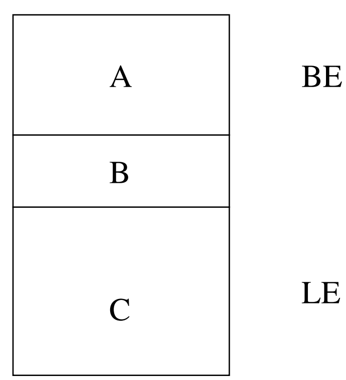

Figure 1

Part A is the big-endian portion of the program. Part C is the little-endian portion of the program. Part B is the endian changing portion including the palindromic instructions. The endian-changing program is loaded into memory in big-endian mode, while the little-endian instructions are stored into memory exactly in little-endian order (B.2.1).

Let’s look at the impact on the Bus Bridge/Memory Controller (BBMC) design as outlined in PowerPC Reference Platform Appendix B. The main consideration of BBMC is where the "byte lane reverse" and "munge" (BLRM) logic are located. For B.2.1 Implementation, BLRM is located between PowerPC and memory or I/O. When the system endian is changed to little endian, the way the little-endian instructions are fetched and executed is as follows. When the little-endian instructions are loaded into cache from memory, the byte-lane-reverse logic reverses the instruction pairs within a double word and changes the byte order of each instruction to a big-endian order. When PowerPC fetches the instructions from the cache, it reverses the instruction pair again within a double word from the cache and stores them in the instruction queue for execution. For this implementation, the little-endian code is loaded and executed correctly by following the BLRM design. For this type of implementation, the cache has the intermediate form which is not a true big or little endian. Let's call it "PowerPC Endian".

For B.2.2 implementation, BLRM is located between I/O and PowerPC or memory. Normally in little-endian mode, instructions are loaded into memory in "PowerPC Endian" form by BLRM. Because they were loaded while the system is in big-endian mode, they are in memory in little-endian order. When the little-endian instructions are loaded into cache from memory, before the system endian mode switched, there is no byte-lane-reverse logic to reverse the instruction pairs. As the result, instructions in cache are stored exactly in original little-endian order. When PowerPC fetches the instructions from the cache, it reverses the instruction pairs from the cache and stores them in instruction queue and code fails! The problem arises because the BLRM logic is not applied to the instructions before storing into the cache. The problem can be fixed by converting portion C to "PowerPC Endian" form which is different form the little-endian form. Figure 1a illustrates the different forms of these endianess. Note that there are two instructions: "63de0d40" and "67de0003".

0	1	2	3	4	5	6	7	
63	de	0d	40	67	de	00	03	Big Endian

0	1	2	3	4	5	6	7	
67	de	00	03	63	de	0d	40	PowerPC Endian

0	1	2	3	4	5	6	7	
40	0d	de	63	03	00	de	67	Little Endian

Figure 1a

For a given operating sytem, there are several ways to have a single binary image on the medium support different BBMCs. Figure 2 illustrates the possibilities. If A+B+C is the only image on the medium, Portion B code has to examine what BBMC is on the planar. If required, it will transform the Portion C of pure little endian code into the proper PowerPC Endian order and creates D portion in memory before performing the endian switching. When doing so Portion B code has to know the length of Portion C which can be obtained by putting labels at the beginning and ending of Portion C code

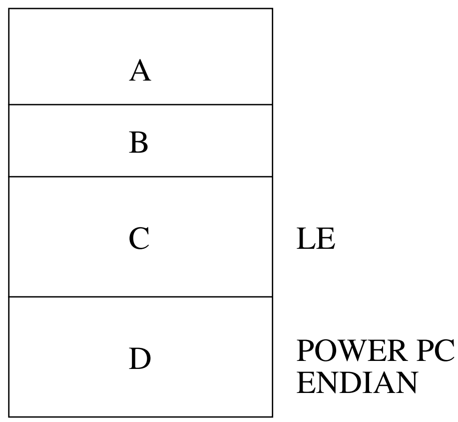

Figure 2

Alternatively, the medium can contain A+B+C+D and all of them are loaded into memory. When the program is executed, Portion B code now has to decide where to branch to the proper BBMC code (C or D). Part C or D has the proper format to execute little-endian portion of the program (usually load the operating system). The BBMC information can be obtained from the Residual Data Structures which are passed by the ROS firmware.

5. Endian Switch Mechanisms of PowerPC

601 uses LM bit of HID0 register to control the endianess of the processor [2 - 2.4.5]. Reference [4] gives detailed analysis of the code sequence to accomplish the transition of endian changing. In summary, the code sequence of endian changing is: . . . , sync, sync, sync, mtspr (set LM bit of HID0), sync, sync, sync, . . .

Other PowerPC processors (603,604 and 620) use two bits - LE and ILE in the MSR register to control the endianess of current processor and exception environments. It is recommended to use "rfi" instruction to effect the endian switching of these processors. "rfi" guarantees context synchronizing [3]. The code sequence of endian changing is: set SRR1 LE bit, set MSR ILE bit, load return address into SRR0 and execute rfi instruction.

6. 601 Endian Switch Code

Figure 3 shows the endian switching code for 601 of the first sequence. Figure 3a shows the endian changing code for the second sequence (only shows the difference from the first sequence). These codes were tested on a 66 MHz Reference Implementation. Note the endian port address is munged for the code of first sequence.

```
# 601 endian changing - first sequence
# Entry:cache enabled,interrupt disabled,sr8 is mapped
# into memory force segment for I/O, translation is on,
# Supervisor mode,begin with 16B aligned,no page fault.
#

# flush d cache --from PPC NT code. good on LRU
        addi    r20,0,0x400     # number cache sectors
        mtspr   CTR,r20         # move to count reg.
        addi    r19,0,-64       # start address -
                                # sizeof line
 lcache:
        lbzu    r18,64(r19)     # touch
        bdnz    lcache
 fcache:
        dcbf    0,r19           # flush each sector
        dcbf    r18,r19
        addic.  r19,r19, -64
        bge     fcache

        mfspr   r2,HID0         # Read HID0 into r2
        addi    r3,0,0x0008     # Bit for LE
        or      r2,r2,r3        # HID0 image in r2

# Access Ref Implementation Endian Port
# (different for other bridges)
        addi    r6,0,0x92       # Load r6 endian
        oris    r6,r6,0x8000    # port 92 address
        lbz     r5,0(r6)        # Read port 92
        ori     r5,r5,0x02      # Set r5 with LE
        addi    r6,0,0x95       # Load r6 with munged
        oris    r6,r6,0x8000    # port 92 address

# Do endian switching
        sync
        sync
        sync
        mtspr   HID0,r2         # Enable LE on 601
```

Figure 3

```
        sync
        sync
        sync

# Initiate system endian switching
         stb     r5,0(r6)

# Following palindromic instructions serve two purposes:
# Instructions in I cache will be executed correctly
# Wait for system endian changing to become effective
# 24 is enough for 66 MHz.But more are required for higher
# speed processor
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op

# Little-endian instructions follow ..
```

Figure 3 - Continued

```
# 601 endian changing - second sequence
# Entry:cache enabled,interrupt disabled,sr8 is mapped
# into memory force segment for I/O, translation is on,
# Supervisor mode,begin with 16B aligned,no page fault.
#

# duplicate code of sequence 1 won't show here

# Access Ref Implementation Endian Port
# (different for other bridges)
        addi    r6,0,0x92       # Load r6 endian
        oris    r6,r6,0x8000    # port 92 address
        lbz     r5,0(r6)        # Read port 92
        ori     r5,r5,0x02      # Set r5 with LE
# Initiate system endian switching
         isync
         stb     r5,0(r6)

# Do endian switching
        sync
        sync
        sync
        mtspr   HID0,r2         # Enable LE on 601
        sync
        sync
        sync

# Following palindromic instructions serve two purposes:
# . . . duplicate code won't show here
```

Figure 3a

7. Other PowerPC Endian Switch Code

Figure 4 shows the first sequence of endian switching code of PowerPC processors other than 601. Figure 4a shows the code of second sequence. These codes were tested on a 603 and a 604 implementations and are a simplified version of [5]. With the exception of the D-cache flush algorithm, these codes should work for a 620 also. Note that the address of endian port is munged for first sequence. Note also how the branch address of "rfi" instruction is generated.

```
# 603/604 endian changing - first sequence
# Entry:cache enabled,interrupt disabled,translation on,
# address mapped, supervisor mode,begin with 16B aligned,
# no page fault.
#

# flush d cache --from PPC NT code. good on LRU
        addi    r20,0,0x400    # number cache sectors
        mtspr   CTR,r20        # move to count reg.
        addi    r19,0,-64      # start address -
                               # sizeof line
 lcache:
        lbzu    r18,64(r19)    # touch
        bdnz    lcache
 fcache:
        dcbf    0,r19          # flush each sector
        dcbf    r18,r19
        addic.  r19,r19, -64
        bge     fcache

# Invalidate I-cache
         mfspr   r2,HID0        # read HID0 register
        ori     r9,r2,0x0800   # set bit 20 -ICFI
        rlwinm  r9,r9,0,17,15  # reset bit 16 -ICE
        rlwinm  r2,r2,0,21,19  # reset bit 20 -ICFI
        ori     r2,r2,0x8000   # set bit 16 -ICE
        isync
        mtspr   HID0,r9        # invalidate I-cache
        mtspr   HID0,r2        # enable I-cache

# set MSR ILE (bit15). Since this bit is not copied
# from SRR1. Here we set it manually
         mfmsr   r9             # get current msr register
        oris    r9,r9,0x0001   # set msr(15)
        mtmsr   r9
        rlwinm  r9,r9,0,16,14  #reset bit 15, we shouldn't
                               #load SRR1 with this bit set
        ori     r9,r9,0x0001   # set SRR1(31)-LE bit
        mtspr   SRR1,r9        # move r9 to SRR1 register
```

Figure 4

```
# Access Ref Implementation Endian Port
# (different for other bridges)
        addi    r6,0,0x92       # Load r6 endian
        oris    r6,r6,0x8000    # port 92 address
        lbz     r5,0(r6)        # Read port 92
        ori     r5,r5,0x02      # Set r5 with LE
        addi    r6,0,0x95       # Load r6 with munged
        oris    r6,r6,0x8000    # port 92 address

# Load SRR0 with branch address

        bl      x               # Link reg =addr of x
   x:
        mfspr   r2,LR           #move from LR register to r2
        addi    r2,r2,y-x       # point to inst after rfi
        mtspr   SRR0,r2         # store address into SRR0
        rfi                     # cpu endian changing

# Initiate system endian switching
   y:
        stb     r5,0(r6)

# Following palindromic instructions serve two purposes:
# Instructions in I cache will be executed correctly
# Wait for system endian changing to become effective
# 24 is enough for 66 MHz.But more is required for higher
# speed processor
        addi    r0,r1,0x138     # no-op
        addi    r0,r1,0x138     # no-op
        addi    r0,r1,0x138     # no-op
        addi    r0,r1,0x138     # no-op
        addi    r0,r1,0x138     # no-op
```

Figure 4 -Continued

```
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op
        addi    r0,r1,0x138    # no-op

# Littel-endian instructions follow ..
```

Figure 4 - Continued

```
# 603/604 endian changing - second sequence
# Entry:cache enabled,interrupt disabled,translation on,
# address mapped, supervisor mode,begin with 16B aligned,
# no page fault.
#
# Same code of first sequence won't duplicate here


# Access Ref Implementation Endian Port
# (different for other bridges)
        addi    r6,0,0x92       # Load r6 endian
        oris    r6,r6,0x8000    # port 92 address
        lbz     r5,0(r6)        # Read port 92
        ori     r5,r5,0x02      # Set r5 with LE
# Initiate system endian switching

        isync
         stb     r5,0(r6)       # Note 6-cycle latency!

# Load SRR0 with branch address

        bl      x               # Link reg =addr of x
   x:
        mfspr   r2,LR           #move from LR register to r2
        addi    r2,r2,y-x       # point to inst after rfi
        mtspr   SRR0,r2         # store address into SRR0
        rfi                     # cpu endian changing
   y:

# Following palindromic instructions serve two purposes:
# Instructions in I cache will be executed correctly
# Wait for system endian changing effective
# 24 is enough for 66 MHz.But more is required for higher
# speed processor
        addi    r0,r1,0x138     # no-op
# Duplicate code won't show here . . .
```

Figure 4a

8. Conclusion

The PowerPC endian switching can be done with caches enabled. Certain rules should be observed. The ordering of endian changing between PowerPC and system can be done in either way. The system port address must be munged when initiating the system endian changing in the first sequence. Using the palindromic instructions is a powerful concept. It serves as a delay timer, buffering uncertainty and for 604 and 620 to solve the "aggressive prefetch" problem. The aggressive prefetch occurs even when the "rfi" instruction is executed. Instead of discarding previous fetched instructions, 604 and 620 continue to use some pre-fetched instructions after rfi. If not for the palindromic instructions, the only solution for aggressive prefetch is to force the first little-endian instruction to a different cache line!

Different BBMC design deserves special attention. In this report, we propose several ways to allow the operating system to have a single binary image on the medium to cover different BBMCs.

The endian switching for current PowerPC is not straightforward, but for "static" endian switching, it works. The complete solution will be the one, specified in Appendix B.5 of PowerPC Reference Platform Specification,which enables the truly dynamic cross-endian applications.

9. Trademarks

PowerPC, PowerPC 601, PowerPC 603, PowerPC 604 and PowerPC 620 are trademarks of International Business Machines Corporation.

10. Reference

[1] M. Stafford, Private communicationsReferenceReferenceReferenceReference

[2] PowerPC 601 RISC Microprocessor User’s Manual

[3] PowerPC Architecture

[4] Shin-Tai Pan, "Bi-Endian Designs in PowerPC Reference Platform" IBM PPS white paper, March 1994

[5] Early PowerPC endian switch code for NT