Errata: EP9302 - Silicon revision: E0

Reference EP9302 Data Sheet revision DS653PP3 dated March 2005.

Determining the Silicon Revision of the Integrated Circuit

On the front of the integrated circuit, directly under the part number, is an alpha-numeric line. Characters 5 and 6 in this line represent the silicon revision of the chip. For example, this line indicates that the chip is a "E0" revision chip:

EFWA**E0**AM0340

This Errata is applicable only to the E0 revision of the chip.

Please refer to AN273, *"EP93xx Rev E0 Design Guidelines"* for additional information.

AC'97

Description

Disabling audio transmit by clearing the TEN bit in one of the AC97TXCRx registers will not clear out any remaining bytes in the TX FIFO. If the number of bytes left in the FIFO is not equal to a whole sample or samples, this will throw off subsequent audio playback causing distortion or channel swapping.

Workaround

To stop audio playback, do the following:

1) Pause DMA
2) Poll the AC97SRx register until either TXUE or TXFE is set.
3) Clear the TEN bit.

This ensures that the TX FIFO is empty before the transmit channel is disabled.

Ethernet

Description 1

The Ethernet controller does not correctly receive frames that have a size of 64 bytes.

Workaround

In order to receive frames of 64 bytes, enable the RCRCA bit in RxCTL. This will prevent the Ethernet controller from discarding the 64-byte-long frames.

Description 2

When there is inadequate AHB bus bandwidth for data to be transferred from the Ethernet controller FIFO to the receive descriptor, the Ethernet FIFO will overflow and cause the Ethernet controller to fail to receive any more packets.

This problem will also occur if the processor is too busy to service incoming packets in a timely manner. By the time that new receive descriptors are available, the data in the FIFO will contain frames that are corrupted.

It is the job of the system designer to ensure that there is adequate bandwidth for the applications being run.

Workaround

This is a rare occurrence, however at a system level it is important to reserve adequate bandwidth for the Ethernet controller. This can be accomplished by some of the following:

- Reducing the bandwidth use of other bus masters in the system.
- Lowering Ethernet rate to half duplex or 10Mbit if higher bandwidth is not required.
- Insuring that the Ethernet controller receive descriptor processing is given a high enough priority to ensure that the controller never runs out of receive descriptors.

HDLC

Description

When the final byte of a received packet is read into the DMA controller's buffer, the software will be notified by an HDLC RFC interrupt. However, the DMA controller may not have written the currently buffered part of the packet to memory, so that the last one to fifteen bytes of a packet may not be accessible.

Workaround

To insure that the DMA channel empties the buffer, do the following (in the HDLC interrupt handler, for example):

1) Note the values in the MAXCNTx and REMAIN registers for the DMA channel. The difference is the number of bytes read from the UART/HDLC, which is the size of the HDLC packet. Call this number N. Note that the BC field of the UART1HDLCRXInfoBuf register should also be N.

2) Temporarily disable the UART DMA RX interface by clearing the RXDMAE bit in the UART1DMACtrl register.

3) Wait until the difference between the CURRENTx and BASEx registers in the DMA channel is equal to N + 1.

At this point, the rest of the packet is guaranteed to have been written to memory. Using this method will cause an extra byte to be read from the UART by the DMA channel and also written to memory. This last byte should be ignored.

SDRAM Controller

Description 1

Using the SDRAM controller in auto-precharge mode will produce system instability at external bus speeds greater than 50MHz.

Workaround

Do not turn on the auto-precharge feature of the SDRAM controller if the external bus speed will be greater than 50 MHz.

Description 2

When the SDRAM controller is configured for PRECHARGE ALL command, the actual sequence is not always issued to the SDRAM device(s).

Workaround

Do a read from each SDRAM bank so that a PRECHARGE command is issued to each bank of the SDRAM device. This will satisfy the required SDRAM initialization sequence.

Due to the effectiveness and simplicity of the software workaround, no silicon fix is planned.

Reset

Description 1

No SDCLK after system reset. This occurs when switching from the crystal clock to the PLL clock upon a system reset. When this condition occurs the processor will not start its boot sequence.

Workaround

Implement the recommended external reset circuit described in AN258, *"EP93xx Power-up and Reset Luckup Workaround",* which can be found at http://www.cirrus.com/en/pubs/appNote/AN258REV2.pdf

A fix for this bug has been implemented for silicon revision E1.

Description 2

SDRAM controller gets stuck in a busy state after resetting from sync mode. When this condition occurs the processor will not complete its boot sequence.

Workaround

Implement the recommended external reset circuit described in AN258, *"EP93xx Power-up and Reset Luckup Workaround"*, which can be found at http://www.cirrus.com/en/pubs/appNote/AN258REV2.pdf

A fix for this bug has been implemented for silicon revision E1.

Description 3

Double reset after a software reset is issued. This condition occurs when the state of $\overline{\text{CS1}}$ is intended to be latched as a logic 0 during reset. However, because of a slow slew rate, they are latched as a logic 1 before they have a chance to reach their final value. When this happens, the Watchdog reset, which is based on a counter, is directly set and generates another reset.

Workaround

There is no work around at this time.

A fix for this bug has been implemented for silicon revision E1.

Description 4

The internal RTC oscillator is susceptible to noise which can lead to extra clocks on the internal 32.768-kHz signal.

Workaround

Please refer to application note AN265, *"EP93xx RTC Oscillator Circuit"*, which can be found at http://www.cirrus.com/en/pubs/appNote/AN265REV1.pdf

No fix of this bug is planned for future silicon revisions.

Software Reset

Description

The EP93xx device may hang during the boot process after a software or watchdog reset has been issued.

Workaround

Implement the recommended external reset circuit described in AN258, which can be found at http://www.cirrus.com/en/pubs/appNote/AN258REV2.pdf

A fix for this bug has been implemented for silicon revision E1.

USB and DMA Arbitration

Description

The EP93xx device can hang during AHB arbitration if the USB controller and the DMA controller are used at the same time. This issue has been observed only when the EP93xx is configured to use 16-bit memory boot mode. It has not been observed when the device is configured for 32-bit memory boot mode.

Workaround

Either do not use the DMA controller or do not use the USB controller. Items potentially affected by not using the DMA controller are IDE, audio, SSP, UART, IrDA, and external DMA. Most of these items can be used in PIO mode with little performance impact. Refer to the User's Guide for details about DMA controller operation.

A fix for this bug has been implemented for silicon revision E1.

I^{2}S Audio

Description

I^{2}S slave mode operation does not function as expected when I2SonAC97 bit (DeviceCfg[6]) is set in the DeviceCfg register causing the ASYNC pin to be driven as LRCLK input.

Workaround

Use I2SonSSP (DeviceCfg[7]) mode instead of I2SonAC97 (DeviceCfg[6]).

A fix for this bug has been implemented for silicon revision E1.

MaverickCrunch^{TM}

Various MaverickCrunch errata share common features. The individual descriptions will refer to these common features.

1) For several errata, an instruction appears in the coprocessor pipeline, but does not execute for one of the following reasons:

- It fails its condition code check.
- A branch is taken and it is one of the two instructions in the branch delay slot.
- An exception occurs.
- An interrupt occurs.

2) For several errata, the coprocessor must be either operating in serialized mode or not be operating in serialized mode. The coprocessor is operating in serialized mode if and only if both:

- At least one exception type is enabled by setting one of the following bits in the DSPSC: IXE, UFE, OFE, or IOE.
- Serialization is not specifically disabled by setting the AEXC bit in the DSPSC.

3) For several errata, an instruction must update an accumulator. These include all of the following:

- Moves to accumulators: cfmva32, cfmva64, cfmval32, cfmvam32, cfmvah32.
- Arithmetic into accumulators: cfmadd32, cfmadda32, cfmsub32, cfmsuba32.

4) For several errata, an instruction must be any two-word coprocessor load or store. These include cfldr64, cfldrd, cfstr64, and cfstrd.

The following table summarizes MaverickCrunch errata.

Erratum	Failing Coprocessor Instructions	Mode	Result	Workaround
1.	two-word load / store		register or memory corruption	change sequence
2.	instruction with source operand		bad calculation or stored value	change sequence
3.	two-word load / store		register or memory corruption	change sequence
4.	two-word store	forwarding, not serialized	memory corruption	change sequence
5.	cfrshl32, cfrshl64	serialized	bad calculation	unserialized mode, substitute ARM code sequence
6.	ldr32, mv64lr		bad sign extension in register	add correcting code sequence
7.	accumulator updates		accumulator corruption	change sequence
8.	accumulator updates		accumulator corruption	change sequence
9.	accumulator updates		accumulator corruption	change sequence
10.	accumulator updates	serialized	accumulator corruption	unserialized mode
11.	two-word load / store		memory or register corruption	change sequence

Several of the errata are sensitive to certain coprocessor instructions appearing early in an interrupt or exception handler. To avoid seeing any errata due to such instructions, insure that no coprocessor instructions appear in the instruction stream within the first seven instructions after an interrupt or exception. Note that, typically, the first three instructions in this stream would be a branch in the jump table followed by the two instructions in the branch delay slot.

Description 1

Under certain circumstances, data in coprocessor registers or in memory may be corrupted. The following sequence of instructions will cause the corruption:

1) Let the first instruction be both:

- any coprocessor instruction that is not executed[1].
- stalled by the coprocessor due to an internal dependency.

2) Let the second instruction be any two-word coprocessor load or store[4].

If the second instruction is a load, the upper word in the target register will generally get an incorrect value. If the second instruction is a store, the word immediately following the second target memory location will be written; that is, instead of just writing two consecutive 32-bit words (a 64-bit value or a double value) to memory, a third 32-bit word immediately following this will be written, leading to memory corruption.

Consider a simple example with a store instruction:

```
  cfaddne     c0, c1, c2          ; assume this does not execute
  cfstr64     c3, [r2, #0x0]
```

Three words will be written to memory. The correct values will appear at the memory location pointed to by r2, and r2 + 0x4. Another value will be written at r2 + 0x8.

Consider now an example with a load instruction:

```
  cfaddne     c0, c1, c2           ; assume this does not execute
  cfldrd      c3, [r2, #0x0]
```

The final value in c3 will be incorrect. The lower 32 bits will be correct, while the upper 32 bits will be incorrect.

Finally, consider a case where a branch occurs:

```
  target
    cfldrd      c3, [r2, #0x0]
    b           target
    nop
    cfadd       c0, c1, c2          ; though in pipeline, this does not execute
```

Note: The above examples assume that the cfaddne or cfadd would busy-wait (for whatever reason) if actually executed. If not, the execution of the following instruction would be correct.

(Continued)

Workaround

The simplest workaround is to insure that no two such instructions ever appear in the instruction stream consecutively. Specifically, a conditional coprocessor instruction should not precede a load/store 64/double. Simply inserting another ARM or coprocessor instruction accomplishes this:

```
  cfaddne       c0, c1, c2          ; assume this does not execute
  nop                               ; inserted extra instruction here
  cfldrd        c3, [r2, #0x0]
```

Cases where branches may be taken also needs to be handled. In this particular case, the first instruction is moved earlier in the instruction stream by exchanging it with the previous one:

```
  target
    cfldrd      c3, [r2, #0x0]
    b           target
    cfadd       c0, c1, c2          ; though in pipeline, this does not execute
    nop
```

To avoid this error when entering exception and interrupt handlers, the first instruction in an interrupt or exception handler should not be a coprocessor instruction. Since the first instruction is normally a branch, this error should not appear.

Description 2

Under certain circumstances, incorrect values may be used for arithmetic calculations or stored in memory. The error appears as follows.

1) Execute a coprocessor instruction whose target is one of the coprocessor general purpose register c0 through c15.

2) Let the second instruction be an instruction with the same target, but not be executed[1].

3) Execute a third instruction at least one of whose operands is the target of the previous two instructions.

For example, assume no pipeline interlocks other than the dependencies involving register c0 in the following instruction sequence:

```
  cfadd32       c0, c1, c2
  cfsub32ne     c0, c3, c4          ; assume this does not execute
  cfstr32       c0, [r2, #0x0]
```

In this particular case, the incorrect value stored at the address in r2 is the previous value in c0, not the expected one resulting from the cfadd32.

(Continued)

Workaround

Insure that this kind of sequence of instructions does not occur. Note that adding a small number of intervening instructions may not be sufficient to avoid this problem. If such a sequence must occur, insure that the first and third instructions are sufficiently far apart in the instruction stream by placing five other instructions between them:

```
cfadd32      c0, c1, c2
nop                             ; inserted extra instruction here
nop                             ; inserted extra instruction here
cfsub32ne    c0, c3, c4         ; assume this does not execute
nop                             ; inserted extra instruction here
nop                             ; inserted extra instruction here
nop                             ; inserted extra instruction here
cfstr32      c0, [r2, #0x0]
```

The five intervening instructions need not be nops and may appear before or after the second instruction.

Note that it is the instruction stream as executed by the processor, not the instructions as they appear in the source code, which is relevant. Hence, cases where the program flow changes between the first and third instruction must be considered.

To avoid this error when entering exception and interrupt handlers, the first five instructions of an exception or interrupt handler should not be coprocessor instructions.

Description 3

Under certain circumstances, data in coprocessor general purpose registers or in memory may be corrupted. The error appears as follows.

1) Let the first instruction be a serialized instruction that does not execute[1]. For an instruction to be serialized, at least one of the following must be true:

   - The processor must be operating in serialized[2] mode.
   - The instruction must move to or from the DSPSC (either cfmv32sc or cfmvsc32).

2) Let the immediately following instruction be a two-word coprocessor load or store[4].

In the case of a load, only the lower 32 bits (the first word) will be loaded into the target register. For example:

```
cfadd32ne     c0, c1, c2        ; assume this does not execute
cfldr64       c3, [r2, #0x0]
```

The lower 32 bits of c3 will correctly become what is at the memory address in r2, but the upper 32 bits of c3 will not become what is at address r2 + 0x4.

In the case of a store, only the lower 32 bits (the first word) will be stored into memory. For example:

```
cfadd32ne    c4, c5, c6         ; assume this does not execute
cfstr64      c3, [r2, #0x0]
```

The lower 32 bits of c3 will be correctly written to the memory address in r2, but the upper 32 bits of c3 will not be written.

(Continued)

Workaround

Separating the first and second instruction by one instruction will avoid this error whether or not the coprocessor is operating in serialized or unserialized mode. For example:

```
                                ; load sequence
cfadd32ne    c0, c1, c2         ; assume this does not execute
nop                             ; inserted extra instruction here
cfldr64      c3, [r2, #0x0]     ; store sequence
cfadd32ne    c4, c5, c6         ; assume this does not execute
nop                             ; inserted extra instruction here
cfstr64      c3, [r2, #0x0]
```

Note that the effect of branches should also be accounted for, as it is the instruction stream as seen by the coprocessor that matters, not the order of instructions in the source code. The two instructions following a taken branch may be seen by the coprocessor and then not executed, and would be treated exactly as the first instruction above.

To avoid this error when entering exception and interrupt handlers, the first instruction in an interrupt or exception handler should not be a coprocessor instruction. Since the first instruction is normally a branch, this error should not appear.

Description 4

When the coprocessor is not in serialized mode[2] and forwarding is enabled, memory can be corrupted when two types of instructions appear in the instruction stream with a particular relative timing.

1) Execute an instruction that is a data operation (not a move between ARM and coprocessor registers) whose destination is one of the general purpose register c0 through c15.

2) Execute an instruction that is a two-word coprocessor store (either cfstr64 or cfstrd), where the destination register of the first instruction is the source of the store instruction, that is, the second instruction stores the result of the first one to memory.

3) Finally, the first and second instruction must appear to the coprocessor with the correct relative timing; this timing is not simply proportional to the number of intervening instructions and is difficult to predict in general.

The result is that the lower 32 bits of the result stored to memory will be correct, but the upper the 32 bits will be wrong. The value appearing in the target register will still be correct.

Workaround

One workaround is to operate the coprocessor without forwarding enabled, with a possible decrease in performance.

Another is to operate in serialized mode by enabling at least one exception, with significantly reduced performance.

Another workaround is to insure that at least seven instructions appear between the first and second instructions that cause the error.

(Continued)

Note: The effect of branches should also be accounted for, as it is the instruction stream as seen by the coprocessor that matters, not the order of instructions in the source code. To avoid this error when entering exception and interrupt handlers, the first seven instruction in an interrupt or exception handler should not be a coprocessor instructions.

Description 5

When operating in serialized mode[2], cfrshl32 and cfrshl64 do not work properly. The instructions shift by an unpredictable amount, but cause no other side effects.

Workaround

One workaround is to avoid these instructions. With this approach, an alternative instruction sequence may accomplish the shift with the following steps:

- Move the data to be shifted to ARM register(s)
- Shift the data using non-coprocessor instructions
- Move the shifted data back to the coprocessor.

Another workaround is to never operate in serialized mode. With this approach, synchronous exceptions are not possible.

Description 6

If an interrupt occurs during the execution of cfldr32 or cfmv64lr, the instruction may not sign extend the result correctly.

Either instruction places a 32 bit value into the lower half of one of the coprocessor general purpose registers c0 through c15 and sign extends the high (32nd) bit through the upper half of the register. If an IRQ or FIQ to the ARM processor interrupts either of these instructions at the right time, the coprocessor will properly load the low 32 bits of the target register, but instead of sign extending it will replicate the low 32 bit into the upper 32 bits. Code that depends on sign extension will fail to operate correctly.

Workaround

Possible workarounds include:

- Disable interrupts when executing cfldr32 or cfmv64lr instructions.
- Avoid executing these two instructions.
- Do not depend on the sign extension to occur; that is, ignore the upper word in any calculations involving data loaded using these instructions.
- Add extra code to sign extend the lower word after it is loaded by explicitly forcing the upper word to be all zeroes or all ones, as appropriate. It is possible to do this selectively in exception or interrupt handler code. If the instruction preceding the interrupted instruction can be determined, and it is a cfldr32 or cfmv64lr, the instruction may be re-executed or explicitly sign extended before returning from interrupt or exception.

Description 7

The coprocessor can incorrectly update one of its destination accumulators even if the coprocessor instruction should not have been executed or is canceled by the ARM processor. This error can occur if the following is true:

1) The first instruction must be a coprocessor compare instruction, one of cfcmp32, cfcmp64, cfcmps, and cfcmpd.

2) The second instruction:

    - has an accumulator as a destination[3].
    - does not execute[1].

Example 1: In this case the second instruction may modify a2 even if the condition is not matched.

```
  cfcmp32       r15, c0, c5
  cfmva64ne     a2, c8
```

Example 2: In this case the second instruction may modify a2 even if an interrupt or exception causes it to be canceled and re-executed after the interrupt/exception handler returns.

```
  cfcmp32       r15, c0, c5
  cfmadda       a2, a2, c0, c1
```

Workaround

The workaround for this issue is to insure that at least one other instruction appears between these instructions. For example, possible fixes for the instructions sequences above are:

```
  cfcmp32       r15, c0, c5
  nop
  cfmva64ne     a2, c8
```

and

```
  cfcmp32       r15, c0, c5
  nop
  cfmadda       a2, a2, c0, c1
```

Description 8

If a data abort occurs on an instruction preceding a coprocessor data path instruction that writes to one of the accumulators[3], the accumulator may be updated even though the instruction was canceled.

For example:

```
  str              r7, [r0, #0x1d]      ; assume this causes a data abort
  cfmadda32        a0, a2, c0, c1
```

The second instruction will update a0 even though it should be canceled due to the data abort on the previous instruction.

(Continued)

Workaround

A complete software workaround requires ensuring that data aborts do not occur due to any instruction immediately preceding a coprocessor instruction that writes to an accumulator. The only way to ensure this is to not allow memory operations immediately preceding these types of instructions. For example, the fixes for the instructions above are:

```
str               r7, [r0, #0x1d]     ; assume this causes a data abort
nop
cfmadda32         a0, a2, c0, c1
```

Description 9

The coprocessor will erroneously update an accumulator if the coprocessor instruction that updates an accumulator is canceled and is followed by a coprocessor instruction that is *not* a data path instruction. This error will occur under the following conditions:

1) The first instruction:

   - must update a coprocessor accumulator[3].
   - does not execute[1].

2) The second instruction is not a coprocessor datapath instruction. Coprocessor data path instructions include any instruction that does not move data to or from memory or to or from the ARM registers.

For example:

```
cfmva64ne         a2, c3
cfmvr64l          r4, c15
```

If the first instruction should not execute or is interrupted, it may incorrectly update a2.

Workaround

Because any instruction may be canceled due to an asynchronous interrupt, the most general software workaround is to insure that no instruction that updates an accumulator is followed immediately by a non-datapath coprocessor instruction. For example, the fix for the instruction sequence above is:

```
cfmva64ne         a2, c3
nop
cfmvr64l          r4, c15
```

Description 10

An instruction that writes a result to an accumulator[3] may cause corruption of any of the four accumulators when the coprocessor is operating in serialized mode[2].

For example, the following sequence of instructions may corrupt a2 if the second instruction is not executed.

```
cfmadda32       a0, a2, c0, c1
cfmadda32ne     a2, c3, c0, c1
```

Workaround

The only workaround for this issue is to operate the coprocessor in unserialized mode.

Description 11

An erroneous memory transfer to or from any of the coprocessor general purpose registers c0 through c15 can occur given the following conditions are satisfied:

1) The first instruction:

- is a two-word load or store[4].
- fails its condition code check.
- does not busy-wait.

2) The second consecutive instruction:

- is a coprocessor load or store.
- is executed.
- does not busy-wait.

When the error occurs, the result is either coprocessor register or memory corruption. Here are several examples:

```
cfstr64ne     c0, [r0, #0x0]      ; assume does not execute
cfldrs        c2, [r2, #0x8]      ; could corrupt c2!

cfldrdge      c0, [r0, #0x0]      ; assume does not execute
cfstrd        c2, [r2, #0x8]      ; could corrupt memory!

cfldr64ne     c0, [r0, #0x0]      ; assume does not execute
cfldrdgt      c2, [r2, #0x8]      ; could corrupt c2!
```

Workaround

The software workaround involves avoiding a pair of consecutive instructions with these properties. For example, if a conditional coprocessor two-word load or store appears, insure that the following instruction is not a coprocessor load or store:

```
cfstr64ne   c0, [r0, #0x0]        ; assume does not execute
nop                               ; separate two instructions
cfldrs      c2, [r2, #0x8]        ; c2 will be ok
```

Another workaround is to insure that the first instruction is not conditional:

```
cfstr64     c0, [r0, #0x0]        ; executes
cfldrs      c2, [r2, #0x8]        ; c2 will be ok
```

Note: If both instructions depend on the same condition code, the error should not occur, as either both or neither will execute.

Contacting Cirrus Logic Support

For all product questions and inquiries contact a Cirrus Logic Sales Representative.
To find one nearest you go to www.cirrus.com