SECTION 21
DEVELOPMENT SUPPORT

21.1 Overview

The visibility and controllability requirements of emulators and bus analyzers are in opposition to the trend of modern microcomputers and microprocessors where many bus cycles are directed to internal resources and are not visible externally.

In order to enhance the development tool visibility and controllability, some of the development support functions are implemented in silicon. These functions include program flow tracking, internal watchpoint, breakpoint generation, and emulation while in debug mode.

This section covers program flow tracking support, breakpoint/watchpoint support, development system interface support (debug mode) and software monitor debugger support. These features allow the user to efficiently debug systems based on the MPC555.

21.2 Program Flow Tracking

The mechanism described below allows tracking of program instruction flow with almost no performance degradation. The information provided may be compressed and captured externally and then parsed by a post-processing program using the microarchitecture defined below.

The program instructions flow is visible on the external bus when the MPC555 is programmed to operate in serial mode and show all fetch cycles on the external bus. This mode is selected by programming the ISCT_SER (instruction fetch show cycle control) field in the I-bus support control register (ICTRL), as shown in **Table 21-21**. In this mode, the processor is fetch serialized, and all internal fetch cycles appear on the external bus. Processor performance is, therefore, much lower than when working in regular mode.

These features, together with the fact that most fetch cycles are performed internally (e.g., from the I-cache), increase performance but make it very difficult to provide the user with the real program trace.

In order to reconstruct a program trace, the program code and the following additional information from the MCU are needed:

- A description of the last fetched instruction (stall, sequential, branch not taken, branch direct taken, branch indirect taken, exception taken)
- The addresses of the targets of all indirect flow change. Indirect flow changes include all branches using the link and count registers as the target address, all exceptions, and **rfi**, **mtmsr** and **mtspr** (to some registers) because they may cause a context switch.
- The number of instructions canceled each clock

Instructions are fetched sequentially until branches (direct or indirect) or exceptions appear in the program flow or some stall in execution causes the machine not to fetch the next address. Instructions may be architecturally executed, or they may be canceled in some stage of the machine pipeline.

The following sections define how this information is generated and how it should be used to reconstruct the program trace. The issue of data compression that could reduce the amount of memory needed by the debug system is also mentioned.

21.2.1 Program Trace Cycle

To allow visibility of the events happening in the machine a few dedicated pins are used and a special bus cycle attribute, program trace cycle, is defined.

The program trace cycle attribute is attached to all fetch cycles resulting from indirect flow changes. When program trace recording is needed, the user can make sure these cycles are visible on the external bus.

The VSYNC indication, when asserted, forces all fetch cycles marked with the program trace cycle attribute to be visible on the external bus even if their data is found in one of the internal devices. To enable the external hardware to properly synchronize with the internal activity of the CPU, the assertion and negation of VSYNC forces the machine to synchronize. The first fetch after this synchronization is marked as a program trace cycle and is visible on the external bus. For more information on the activity of the external hardware during program trace refer to **21.2.4 The External Hardware**.

In order to keep the pin count of the chip as low as possible, VSYNC is not implemented as one of the chip's external pins. It is asserted and negated using the serial interface implemented in the development port. For more information on this interface refer to **21.5 Development Port**

Forcing the CPU to show all fetch cycles marked with the program trace cycle attribute can be done either by asserting the VSYNC pin (as mentioned above) or by programming the fetch show cycle bits in the instruction support control register, ICTRL. For more information refer to **21.2.5 Instruction Fetch Show Cycle Control**

When the VSYNC indication is asserted, all fetch cycles marked with the program trace cycle attribute are made visible on the external bus. These cycles can generate regular bus cycles (address phase and data phase) when the instructions reside only in one of the external devices. Or, they can generate address-only cycles when the instructions reside in one of the internal devices (internal memory, etc.).

When VSYNC is asserted, some performance degradation is expected due to the additional external bus cycles. However, since this performance degradation is expected to be very small, it is possible to program the machine to *show all indirect flow changes*. In this way, the machine will always perform the additional external bus cycles and maintain exactly the same behavior both when VSYNC is asserted and when it is negated. For more information refer to **21.7.6 I-Bus Support Control Register**.

The status pins are divided into two groups and one special case listed below:

21.2.1.1 Instruction Queue Status Pins — VF [0:2]

Instruction queue status pins denote the type of the last fetched instruction or how many instructions were flushed from the instruction queue. These status pins are used for both functions because queue flushes only happen in clocks that there is no fetch type information to be reported.

Possible instruction types are defined in **Table 21-1**.

Table 21-1 VF Pins Instruction Encodings

VF[0:2]	Instruction Type	VF Next Clock Will Hold
000	None	More instruction type information
001	Sequential	More instruction type information
010	Branch (direct or indirect) **not** taken	More instruction type information
011	VSYNC was asserted/negated and therefore the next instruction will be marked with the indirect change-of-flow attribute	More instruction type information
100	Exception taken — the target will be marked with the indirect change-of-flow attribute	Queue flush information[1]
101	Branch indirect taken, **rfi**, **mtmsr**, **isync** and in some cases **mtspr** to CMPA-F, ICTRL, ECR, or DER — the target will be marked with the indirect change-of-flow attribute[2]	Queue flush information[1]
110	Branch direct taken	Queue flush information[1]
111	Branch (direct or indirect) **not** taken	Queue flush information[1]

NOTES:

1. Unless next clock VF=111. See below.
2. The sequential instructions listed here affect the machine in a manner similar to indirect branch instructions. Refer to **21.2.3 Sequential Instructions Marked as Indirect Branch**.

Table 21-2 shows VF[0:2] encodings for instruction queue flush information.

Table 21-2 VF Pins Queue Flush Encodings

VF[0:2]	Queue Flush Information
000	0 instructions flushed from instruction queue
001	1 instruction flushed from instruction queue
010	2 instructions flushed from instruction queue
011	3 instructions flushed from instruction queue
100	4 instructions flushed from instruction queue
101	5 instructions flushed from instruction queue
110	Reserved
111	Instruction type information[1]

NOTES:
1. Refer to **Table 21-1**.

21.2.1.2 History Buffer Flushes Status Pins— VFLS [0..1]

The history buffer flushes status pins denote how many instructions are flushed from the history buffer this clock due to an exception.**Table 21-3** shows VFLS encodings.

Table 21-3 VFLS Pin Encodings

VFLS[0:1]	History Buffer Flush Information
00	0 instructions flushed from history queue
01	1 instruction flushed from history queue
10	2 instructions flushed from history queue
11	Used for debug mode indication (FREEZE). Program trace external hardware should ignore this setting.

21.2.1.3 Queue Flush Information Special Case

There is one special case when although queue flush information is expected on the VF pins, (according to the last value on the VF pins), regular instruction type information is reported. The only instruction type information that can appear in this case is VF = 111, branch (direct or indirect) NOT taken. Since the maximum queue flushes possible is five, it is easy to identify this special case.

21.2.2 Program Trace when in Debug Mode

When entering debug mode an *interrupt/exception taken* is reported on the VF pins, (VF = 100) and a cycle marked with the program trace cycle is made visible externally.

When the CPU is in debug mode, the VF pins equal ‘000’ and the VFLS pins equal ‘11’. For more information on debug mode refer to **21.4 Development System Interface**

If VSYNC is asserted/negated while the CPU is in debug mode, this information is reported as the first VF pins report when the CPU returns to regular mode. If VSYNC was not changed while in debug mode. the first VF pins report will be of an indirect branch taken (VF = 101), suitable for the **rfi** instruction that is being issued. In both cases the first instruction fetch after debug mode is marked with the program trace cycle attribute and therefore is visible externally.

21.2.3 Sequential Instructions Marked as Indirect Branch

There are cases when non-branch (sequential) instructions may effect the machine in a manner similar to indirect branch instructions. These instructions include **rfi**, **mtmsr**, **isync** and **mtspr** to CMPA-F, ICTRL, ECR and DER.

These instructions are marked by the CPU as indirect branch instructions (VF = 101) and the following instruction address is marked with the same program trace cycle attribute as if it were an indirect branch target. Therefore, when one of these special instructions is detected in the CPU, the address of the following instruction is visible externally. In this way the reconstructing software is able to evaluate correctly the effect of these instructions.

21.2.4 The External Hardware

When program trace is needed, the external hardware needs to sample the status pins (VF and VFLS) each clock cycle and the address of all cycles marked with the program trace cycle attribute.

Program trace can be used in various ways. Below are two examples of how program trace can be used:

- **Back trace —** Back trace is useful when a record of the program trace *before* some event occurred is needed. An example of such an event is some system failure.
  In case back trace is needed the external hardware should start sampling the status pins (VF and VFLS) and the address of all cycles marked with the program trace cycle attribute immediately when reset is negated. If *show cycles* is programmed out of reset to *show all,* all cycles marked with program trace cycle attribute are visible on the external bus. VSYNC should be asserted sometime after reset and negated when the programmed event occurs. If *no show* is programmed for *show cycles*, make sure VSYNC is asserted before the Instruction show cycles programming is changed from *show all*.
  Note that in case the timing of the programmed event is unknown it is possible to use cyclic buffers.
  After VSYNC is negated the trace buffer will contain the program flow trace of the program executed before the programmed event occurred.
- **Window trace —** Window trace is useful when a record of the program trace between two events is needed. In case window trace is needed the VSYNC pin should be asserted between these two events.
  After the VSYNC pin is negated the trace buffer will contain information describing the program trace of the program executed *between* the two events.

21.2.4.1 Synchronizing the Trace Window to the CPU Internal Events

The assertion/negation of VSYNC is done using the serial interface implemented in the development port. In order to synchronize the assertion/negation of VSYNC to an internal event of the CPU, it is possible to use the internal breakpoints together with debug mode. This method is available only when debug mode is enabled. For more information on debug mode refer to **21.4 Development System Interface**

The following is an example of steps that enable the user to synchronize the trace window to the CPU internal events:

1. Enter debug mode, either immediately out of reset or using the debug mode request
2. Program the hardware to break on the event that marks the start of the trace window using the control registers defined in **21.3 Watchpoints and Breakpoints Support**
3. Enable debug mode entry for the programmed breakpoint in the debug enable register (DER). See **21.7.12 Debug Enable Register (DER)**)
4. Return to the regular code run (see **21.4.1.6 Exiting Debug Mode**)
5. The hardware generates a breakpoint when the programmed event is detected and the machine enters debug mode (see **21.4.1.2 Entering Debug Mode**)
6. Program the hardware to break on the event that marks the end of the trace window
7. Assert VSYNC
8. Return to the regular code run. The first report on the VF pins is a VSYNC (VF = 011).
9. The external hardware starts sampling the program trace information upon the report on the VF pins of VSYNC
10. The hardware generates a breakpoint when the programmed event is detected and the machine enters debug mode
11. Negate VSYNC
12. Return to the regular code run (issue an **rfi**). The first report on the VF pins is a VSYNC (VF = 011)
13. The external hardware stops sampling the program trace information upon the report on the VF pins of VSYNC

21.2.4.2 Detecting the Trace Window Start Address

When using *back trace*, latching the value of the status pins (VF and VFLS), and the address of the cycles marked as program trace cycle, should start immediately after the negation of reset. The start address is the first address in the program trace cycle buffer.

When using *window trace*, latching the value of the status pins (VF and VFLS), and the address of the cycles marked as program trace cycle, should start immediately after the first VSYNC is reported on the VF pins. The start address of the trace window should be calculated according to first two VF pins reports.

Assuming that VF1 and VF2 are the two first VF pins reports and T1 and T2 are the two addresses of the first two cycles marked with the program trace cycle attribute that were latched in the trace buffer, use the following table to calculate the trace window start address.

Table 21-4 Detecting the Trace Buffer Start Point

VF1	VF2	Starting point	Description
011 **VSYNC**	001 sequential	T1	**VSYNC** asserted followed by a sequential instruction. The start address is T1
011 **VSYNC**	110 branch direct taken	T1 - 4 + offset (T1 - 4)	**VSYNC** asserted followed by a taken direct branch. The start address is the target of the direct branch
011 **VSYNC**	101 branch indirect tak- en	T2	**VSYNC** asserted followed by a taken indirect branch. The start address is the target of the indirect branch

21.2.4.3 Detecting the Assertion/Negation of VSYNC

Since the VF pins are used for reporting both instruction type information and queue flush information, the external hardware must take special care when trying to detect the assertion/negation of VSYNC. When VF = 011 it is a VSYNC assertion/negation report only if the previous VF pins value was one of the following values: 000, 001, or 010.

21.2.4.4 Detecting the Trace Window End Address

The information on the status pins that describes the last fetched instruction and the last queue/history buffer flushes, changes every clock. Cycles marked as program trace cycle are generated on the external bus only when possible (when the SIU wins the arbitration over the external bus). Therefore, there is some delay between the information reported on the status pins that a cycle marked as program trace cycle will be performed on the external bus and the actual time that this cycle can be detected on the external bus.

When VSYNC is negated by the user (through the serial interface of the development port), the CPU delays the report of the of the assertion/negation of VSYNC on the VF pins (VF = 011) until all addresses marked with the program trace cycle attribute were visible externally. Therefore, the external hardware should stop sampling the value of the status pins (VF and VFLS), and the address of the cycles marked as program trace cycle immediately after the VSYNC report on the VF pins.

The last two instructions reported on the VF pins are not always valid. Therefore at the last stage of the reconstruction software, the last two instructions should be ignored.

21.2.4.5 Compress

In order to store all the information generated on the pins during program trace (five bits per clock + 30 bits per show cycle) a large memory buffer may be needed. However, since this information includes events that were canceled, compression can be very effective. External hardware can be added to eliminate all canceled instructions and report only on branches (taken and not taken), indirect flow change, and the number of sequential instructions after the last flow change.

21.2.5 Instruction Fetch Show Cycle Control

Instruction fetch show cycles are controlled by the bits in the ICTRL and the state of VSYNC. The following table defines the level of fetch show cycles generated by the CPU. For information on the fetch show cycles control bits refer to **Table 21-5**

Table 21-5 Fetch Show Cycles Control

VSYNC	ISCTL Instruction Fetch Show Cycle Control Bits	Show cycles generated
X	00	All fetch cycles
X	01	All change of flow (direct & indirect)
X	10	All indirect change of flow
0	11	No show cycles are performed
1	11	All indirect change of flow

NOTE

A cycle marked with the program trace cycle attribute is generated for any change in the VSYNC state (assertion or negation).

21.3 Watchpoints and Breakpoints Support

Watchpoints, when detected, are reported to the external world on dedicated pins but do not change the timing and the flow of the machine. Breakpoints, when detected, force the machine to branch to the appropriate exception handler. The CPU supports internal watchpoints, internal breakpoints, and external breakpoints.

Internal watchpoints are generated when a user programmable set of conditions are met. Internal breakpoints can be programmed to be generated either as an immediate result of the assertion of one of the internal watchpoints, or after an internal watchpoint is asserted for a user programmable times. Programming a certain internal watchpoint to generate an internal breakpoint can be done either in software, by setting the corresponding software trap enable bit, or on the fly using the serial interface implemented in the development port to set the corresponding development port trap enable bit.

External breakpoints can be generated by any of the peripherals of the system, including those found on the MPC555 or externally, and also by an external development system. Peripherals found on the external bus use the serial interface of the development port to assert the external breakpoint.

In the CPU, as in other RISC processors, saving/restoring machine state on the stack during exception handling, is done mostly in software. When the software is in the middle of saving/restoring machine state, the MSRRI bit is cleared. Exceptions that occur and that are handled by the CPU when the MSRRI bit is clear result in a non-restartable machine state. For more information refer to **3.15.4 Interrupts**

In general, breakpoints are recognized in the CPU is only when the MSRRI bit is set, which guarantees machine restartability after a breakpoint. In this working mode breakpoints are said to be *masked*. There are cases when it is desired to enable breakpoints even when the MSRRI bit is clear, with the possible risk of causing a non-restartable machine state. Therefore internal breakpoints have also a programmable *non-masked* mode, and an external development system can also choose to assert a *non-maskable* external breakpoint.

Watchpoints are not *masked* and therefore always reported on the external pins, regardless of the value of the MSRRI bit. The counters, although counting watchpoints, are part of the internal breakpoints logic and therefore are not decremented when the CPU is operating in the masked mode and the MSRRI bit is clear.

The following figure illustrates the watchpoints and breakpoints support of the CPU.

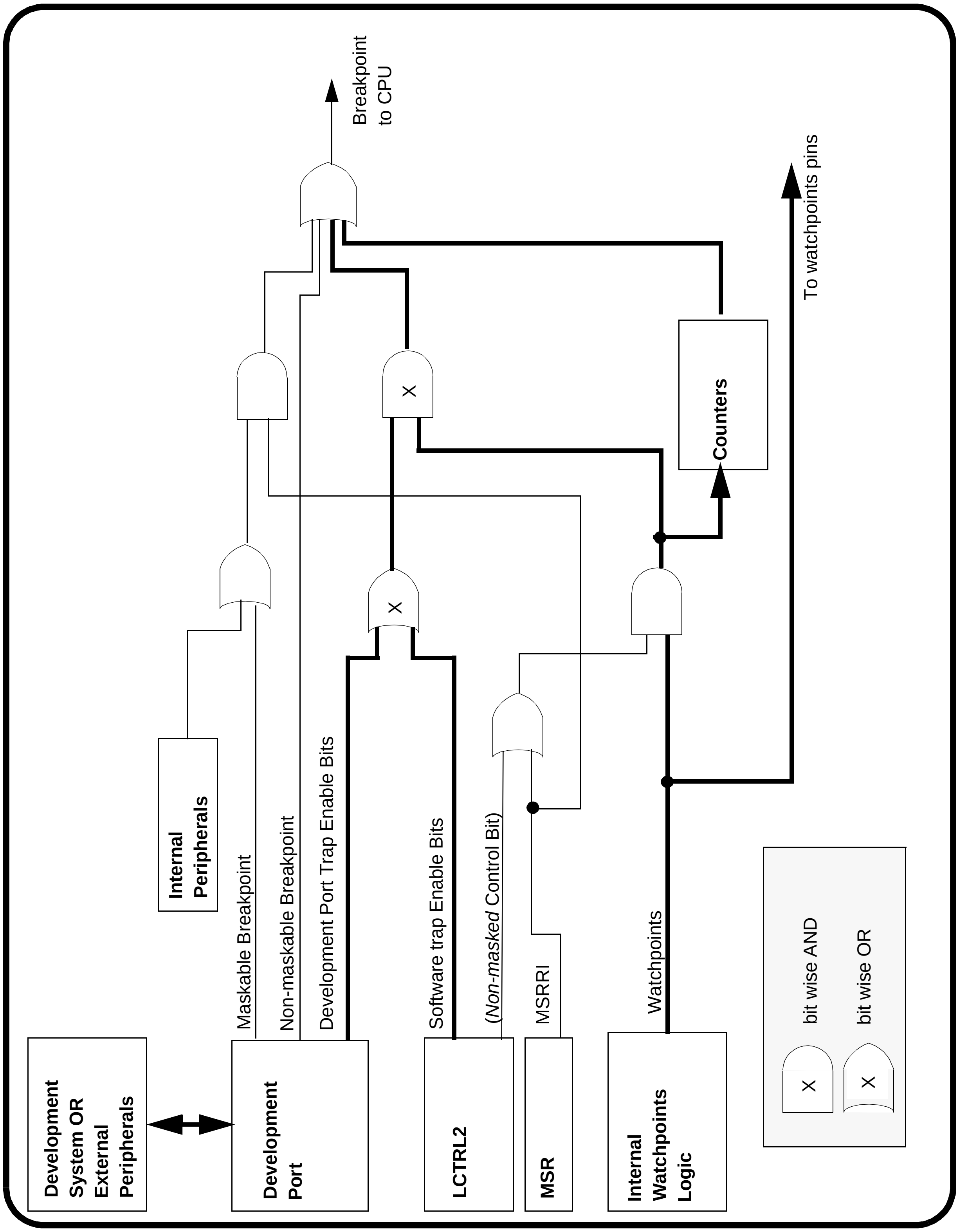

Figure 21-1 Watchpoints and Breakpoint Support in the CPU

21.3.1 Internal Watchpoints and Breakpoints

This section describes the internal breakpoints and watchpoints support of the CPU. For information on external breakpoints support refer to **21.4 Development System Interface**

Internal breakpoint and watchpoint support is based on eight comparators comparing information on instruction and load/store cycles, two counters, and two AND-OR logic structures. The comparators perform compare on the Instruction address (I-address), on the load/store address (L-address) and on the load/store data (L-data).

The comparators are able to detect the following conditions: equal, not equal, greater than, less than (greater than or equal and less than or equal are easily obtained from these four conditions, for more information refer to **21.3.1.6 Generating Six Compare Types**). Using the AND-OR logic structures "in range" and "out of range" detections (on address and on data) are supported. Using the counters, it is possible to program a breakpoint to be recognized after an event was detected a predefined number of times.

The L-data comparators can operate on fix point data of load or store. When operating on fix point data the L-data comparators are able to perform compare on bytes, half-words and words and can treat numbers either as signed or as unsigned values.

The comparators generate match events. The match events enter the instruction AND-OR logic where the instruction watchpoints and breakpoint are generated. The instruction watchpoints, when asserted, may generate the instruction breakpoint. Two of them may decrement one of the counters. If one of the instruction watchpoints expires in a counter that is counting, the instruction breakpoint is asserted.

The instruction watchpoints and the load/store match events (address and data) enter the load/store AND-OR logic where the load/store watchpoints and breakpoint are generated. The load/store watchpoints, when asserted, may generate the load/store breakpoint or they may decrement one of the counters. When a counter that is counting one of the load/store watchpoints expires, the load/store breakpoint is asserted.

Watchpoints progress in the machine and are reported on retirement. Internal breakpoints progress in the machine until they reach the top of the history buffer when the machine branches to the breakpoint exception routine.

In order to enable the user to use the breakpoint features without adding restrictions on the software, the address of the load/store cycle that generated the load/store breakpoint is not stored in the DAR (data address register), like other load/store type exceptions. In case of a load/store breakpoint, the address of the load/store cycle that generated the breakpoint is stored in an implementation-dependent register called the BAR (breakpoint address register).

Key features of internal watchpoint and breakpoint support are:

- Four I-address comparators (each supports equal, not equal, greater than, less than)
- Two L-address comparators (each supports equal, not equal, greater than, less than) including least significant bits masking according to the size of the bus cycle for the byte and half-word working modes. Refer to **21.3.1.2 Byte and Half-Word Working Modes**

- Two L-data comparators (each supports equal, not equal, greater than, less than) including byte, half-word and word operating modes and four byte mask bits for each comparator. Can be used for fix point data. Match is detected only on the valid part of the data bus (according to the cycle's size and the two address least significant bits).
- *No internal breakpoint/watchpoint matching support for unaligned words and half-words*
- The L-data comparators can be programmed to treat fix point numbers as signed values or as unsigned values
- Combine comparator pairs to detect in and out of range conditions (including either signed or unsigned values on the L-data)
- A programmable AND-OR logic structure between the four instruction comparators results with five outputs, four instruction watchpoints and one instruction breakpoint
- A programmable AND-OR logic structure between the four instruction watchpoints and the four load/store comparators results with three outputs, two load/store watchpoints and one load/store breakpoint
- Five watchpoint pins, three for the instruction and two for the load/store
- Two dedicated 16-bit down counters. Each can be programmed to count either an instruction watchpoint or an load/store watchpoint. Only *architecturally executed* events are counted, (count up is performed in case of recovery).
- On the fly trap enable programming of the different internal breakpoints using the serial interface of the development port (refer to **21.5 Development Port**). Software control is also available.
- Watchpoints do not change the timing of the machine
- Internal breakpoints and watchpoints are detected on the instruction during instruction fetch
- Internal breakpoints and watchpoints are detected on the load/store during load/store bus cycles
- Both instruction and load/store breakpoints and watchpoints are handled and reported on retirement. Breakpoints and watchpoints on recovered instructions (as a result of exceptions, interrupts or miss prediction) are not reported and do not change the timing of the machine.
- Instructions with instruction breakpoints are not executed. The machine branches to the breakpoint exception routine BEFORE it executes the instruction.
- Instructions with load/store breakpoints are executed. The machine branches to the breakpoint exception routine AFTER it executes the instruction. The address of the access is placed in the BAR (breakpoint address register).
- Load/store multiple and string instructions with load/store breakpoints first finish execution (all of it) and then the machine branches to the breakpoint exception routine.
- Load/store data compare is done on the load/store, AFTER swap in store accesses and BEFORE swap in load accesses (as the data appears on the bus).
- Internal breakpoints may operate either in *masked* mode or in *non-masked* mode.
- Both "go to x" and "continue" working modes are supported for the instruction breakpoints.

21.3.1.1 Restrictions

There are cases when the same watchpoint can be detected more than once during the execution of a single instruction, e.g. a load/store watchpoint is detected on more than one transfer when executing a load/store multiple/string or a load/store watchpoint is detected on more than one byte when working in byte mode. In all these cases only one watchpoint of the same type is reported for a single instruction. Similarly, only one watchpoint of the same type can be counted in the counters for a single instruction.

Since watchpoint events are reported upon the retirement of the instruction that caused the event, and more than one instruction can retire from the machine in one clock, consequent events may be reported in the same clock. Moreover the same event, if detected on more than one instruction (e.g., tight loops, range detection), in some cases will be reported only once. Note that the internal counters count correctly in these cases.

Do not put a breakpoint on an **mtspr ICTRL** instruction. When a breakpoint is set on an **mtspr ICTRL** Rx instruction and the value of bit 28 (IFM) is one, the result will be unpredictable. A breakpoint can be taken or not on the instruction and the value of the IFM bit can be either zero or one. Also, do not put a breakpoint on an mtspr ICTRL Rx instruction when Rx contains one in bit 28.

21.3.1.2 Byte and Half-Word Working Modes

The CPU watchpoints and breakpoints support enables the user to detect matches on bytes and half-words even when accessed using a load/store instruction of larger data widths, for example when loading a table of bytes using a series of load word instructions. In order to use this feature, the user needs to program the byte mask for each of the L-data comparators and to write the needed match value to the correct half-word of the data comparator when working in half-word mode and to the correct bytes of the data comparator when working in byte mode.

Since bytes and half-words can be accessed using a larger data width instruction, it is impossible for the user to predict the exact value of the L-address lines when the requested byte/half-word is accessed, (e.g., if the matched byte is byte two of the word and it is accessed using a load word instruction), the L-address value will be of the word (byte zero). Therefore, the CPU masks the two least-significant bits of the L-address comparators whenever a word access is performed and the least-significant bit whenever a half-word access is performed.

Address range is supported only when aligned according to the access size. (See examples)

21.3.1.3 Examples

- A fully supported scenario:

  Looking for:

  Data size: Byte
  Address: 0x00000003
  Data value: greater than 0x07 and less than 0x0c

  Programming options:

  One L-address comparator = 0x00000003 and program for equal
  One L-data comparator = 0x00000007 and program for greater than
  One L-data comparator = 0x0000000c and program for less than
  Both byte masks = 0xe
  Both L-data comparators program to byte mode

  Result: The event will be correctly detected regardless of the load/store instruction the compiler chooses for this access

- A fully supported scenario:

  Looking for:

  Data size: half-word
  Address: greater than 0x00000000 and less than 0x0000000c
  Data value: greater than 0x4e204e20 and less than 0x9c409c40

  Programming option:

  One L-address comparator = 0x00000000 and program for greater than
  One L-address comparator = 0x0000000c and program for less than
  One L-data comparator = 0x4e204e20 and program for greater than
  One L-data comparator = 0x9c409c40 and program for less than
  Both byte masks = 0x0
  Both L-data comparators program to half-word mode

  Result: The event will be correctly detected as long as the compiler does not use a load/store instruction with data size of byte.

- A **partially** supported scenario:

  Looking for:

  Data size: half-word
  Address: greater than or equal 0x00000002 and less than 0x0000000e
  Data value: greater than 0x4e204e20 and less than 0x9c409c40

  Programming option:

  One L-address comparator = 0x00000001 and program for greater than
  One L-address comparator = 0x0000000e and program for less than
  One L-data comparator = 0x4e204e20 and program for greater than
  One L-data comparator = 0x9c409c40 and program for less than
  Both byte masks = 0x0
  Both L-data comparators program to half-word mode or to word mode

  Result: The event will be correctly detected if the compiler chooses a load/store instruction with data size of half-word. If the compiler chooses load/store instructions with data size greater than half-word (word, multiple), there might be some false detections.

These can be ignored only by the software that handles the breakpoints. The following figure illustrates this **partially** supported scenario.

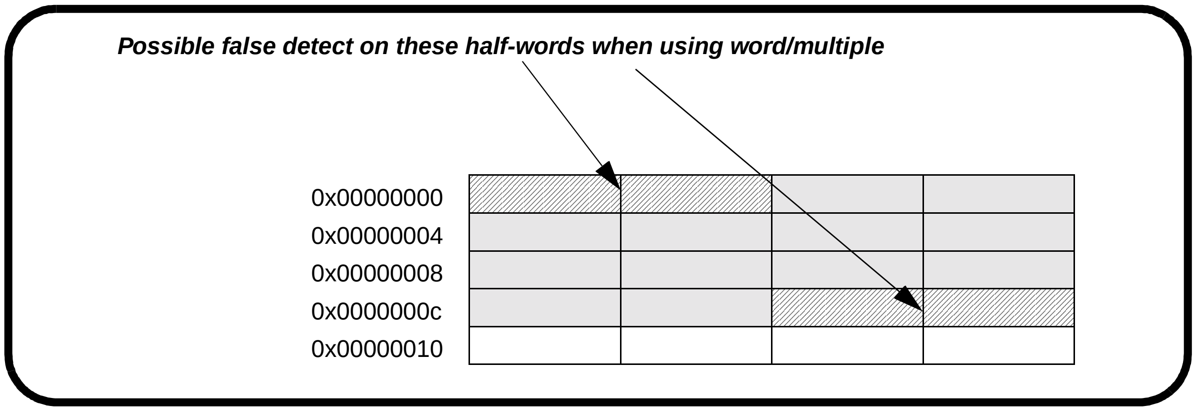
Possible false detect on these half-words when using word/multiple
0x00000000
0x00000004
0x00000008
0x0000000c
0x00000010

Figure 21-2 Partially Supported Watchpoint/Breakpoint Example

21.3.1.4 Context Dependent Filter

The CPU can be programmed to either recognize internal breakpoints only when the recoverable interrupt bit in the MSR is set (*masked* mode) or it can be programmed to always recognize internal breakpoints (*non-masked* mode).

When the CPU is programmed to recognize internal breakpoints only when MSRRI = 1, it is possible to debug all parts of the code except when the machine status save/restore registers (SRR0 and SRR1), DAR (data address register) and DSISR (data storage interrupt status register) are busy and, therefore, MSRRI = 0, (in the prologues and epilogues of interrupt/exception handlers).

When the CPU is programmed always to recognize internal breakpoints, it is possible to debug all parts of the code. However, if an internal breakpoint is recognized when MSRRI = 0 (SRR0 and SRR1 are busy), the machine enters into a non-restartable state. For more information refer to **3.15.4 Interrupts**

When working in the *masked* mode, all internal breakpoints detected when MSRRI = 0 are lost. Watchpoints detected in this case are not counted by the debug counters. Watchpoints detected are always reported on the external pins, regardless of the value of the MSRRI bit.

Out of reset, the CPU is in *masked* mode. Programming the CPU to be in *non-masked* mode is done by setting the BRKNOMSK bit in the LCTRL2 register. Refer to **21.7.8 L-Bus Support Control Register 2** The BRKNOMSK bit controls all internal breakpoints (I-breakpoints and L-breakpoints).

21.3.1.5 Ignore First Match

In order to facilitate the debugger utilities “continue” and “go from x”, the ignore first match option is supported for instruction breakpoints. When an instruction breakpoint is first enabled (as a result of the first write to the instruction support control register or as a result of the assertion of the MSRRI bit when operating in the *masked* mode), the first instruction will not cause an instruction breakpoint if the ignore first match (IFM) bit in the instruction support control register (ICTRL) is set (used for “continue”).

When the IFM bit is clear, every matched instruction can cause an instruction breakpoint (used for “go from x”). This bit is set by the software and cleared by the hardware after the first instruction breakpoint match is ignored. Load/store breakpoints and all counter generated breakpoints (instruction and load/store) are not affected by this mode.

21.3.1.6 Generating Six Compare Types

Using the four compare types mentioned above (equal, not equal, greater than, less than) it is possible to generate also two more compare types: greater than or equal and less than or equal.

- Generating the greater than or equal compare type can be done by using the greater than compare type and programming the comparator to the needed value minus 1.
- Generating the less than or equal compare type can be done by using the less than compare type and programming the comparator to the needed value plus 1.

This method does not work for the following boundary cases:

- Less than or equal of the largest unsigned number (1111...1)
- Greater than or equal of the smallest unsigned number (0000...0)
- Less than or equal of the maximum positive number when in signed mode (0111...1)
- Greater than or equal of the maximum negative number when in signed mode (1000...)

These boundary cases need no special support because they all mean ‘always true’ and can be programmed using the ignore option of the load/store watchpoint programming (refer to **21.3 Watchpoints and Breakpoints Support**).

21.3.2 Instruction Support

There are four instruction address comparators A,B,C, and D. Each is 30 bits long, generating two output signals: equal and less than. These signals are used to generate one of the following four events: equal, not equal, greater than, less than.

The instruction watchpoints and breakpoint are generated using these events and according to user programming. Note that using the OR option enables “out of range” detect.

Table 21-6 Instruction Watchpoints Programming Options

Name	Description	Programming options
IWP0	First instruction watchpoint	Comparator A Comparators (A&B)
IWP1	Second instruction watchpoint	Comparator B Comparator (A | B)
IWP2	Third instruction watchpoint	Comparator C Comparators (C&D)
IWP3	Fourth instruction watchpoint	Comparator D Comparator (C | D)

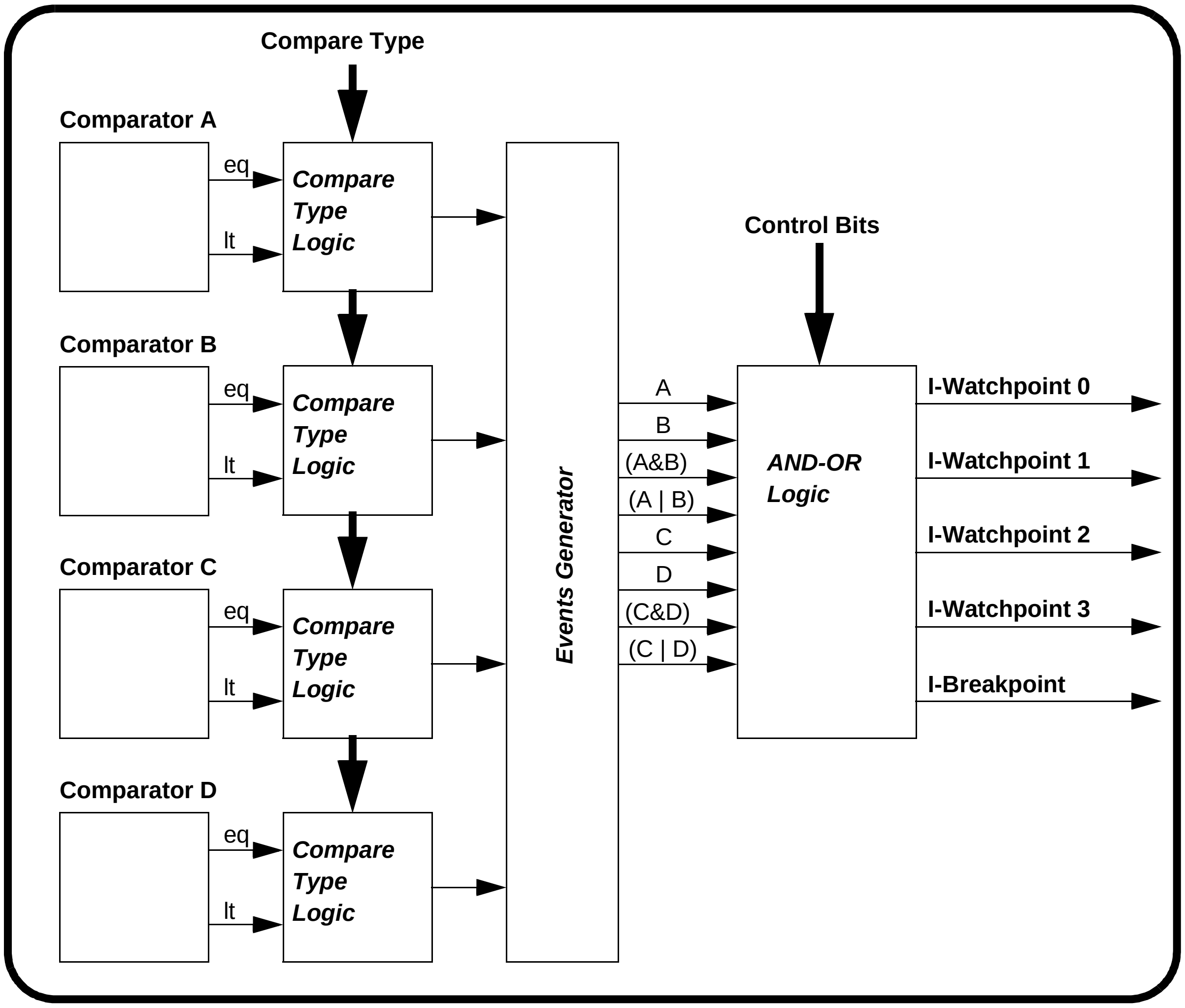

Figure 21-3 Instruction Support General Structure

21.3.2.1 Load/Store Support

There are two load/store address comparators E, and F. Each compares the 32 address bits and the cycle's attributes (read/write). The two least-significant bits are masked (ignored) whenever a word is accessed and the least-significant bit is masked whenever a half-word is accessed. (For more information refer to **21.3.1.2 Byte and Half-Word Working Modes**). Each comparator generates two output signals: equal and less than. These signals are used to generate one of the following four events (one from each comparator): equal, not equal, greater than, less than.

There are two load/store data comparators (comparators G,H) each is 32 bits wide and can be programmed to treat numbers either as signed values or as unsigned values. Each data comparator operates as four independent byte comparators. Each byte comparator has a mask bit and generates two output signals: equal and less than, if the mask bit is not set. Therefore, each 32 bit comparator has eight output signals.

These signals are used to generate the “equal and less than” signals according to the compare size programmed by the user (byte, half-word, word). When operating in byte mode all signals are significant, when operating in half-word mode only four signals from each 32 bit comparator are significant. When operating in word mode only two signals from each 32 bit comparator are significant.

From the *new* “equal and less than” signals and according to the compare type programmed by the user one of the following four match events are generated: equal, not equal, greater than, less than. Therefore, from the two 32-bit comparators eight match indications are generated: Gmatch[0:3], Hmatch[0:3].

According to the lower bits of the address and the size of the cycle, only match indications that were detected on bytes that have valid information are validated, the rest are negated. Note that if the cycle executed has a smaller size than the compare size (e.g., a byte access when the compare size is word or half-word) no match indication will be asserted.

Using the match indication signals four load/store data events are generated in the following way.

Table 21-7 Load/Store Data Events

Event Name	Event Function[1]
G	(Gmatch0 | Gmatch1 | Gmatch2 | Gmatch3)
H	(Hmatch0 | Hmatch1 | Hmatch2 | Hmatch3)
(G&H)	((Gmatch0 & Hmatch0) | (Gmatch1 & Hmatch1) | (Gmatch2 & Hmatch2) | (Gmatch3 & Hmatch3))
(G | H)	((Gmatch0 | Hmatch0) | (Gmatch1 | Hmatch1) | (Gmatch2 | Hmatch2) | (Gmatch3 | Hmatch3))

NOTES:
1. ‘&’ denotes a logical AND, ‘|’ denotes a logical OR

The four load/store data events together with the match events of the load/store address comparators and the instruction watchpoints are used to generate the load/store watchpoints and breakpoint according to the users programming.

Table 21-8 Load/Store Watchpoints Programming Options

Name	Description	Instruction Events Programming Options	L-address Events Programming Options	L-data Events Programming Options
LWP0	First Load/store watch-point	IWP0, IWP1, IWP2, IWP3, ignore instruction events	Comparator E Comparator F Comparators (E&F) Comparators (E | F) ignore L-addr events	Comparator G Comparator H Comparators (G&H) Comparators (G | H) ignore L-data events
LWP1	Second Load/store watch-point	IWP0, IWP1, IWP2, IWP3, ignore instruction events	Comparator E Comparator F Comparators (E&F) Comparators (E | F) ignore I-addr events	Comparator G Comparator H Comparators (G&H) Comparators (G | H) ignore L-data events

Note that when programming the load/store watchpoints to ignore L-addr events and L-data events, it does not reduce the load/store watchpoints detection logic to be instruction watchpoint detection logic since the instruction must be a load/store instruction for the load/store watchpoint event to trigger.

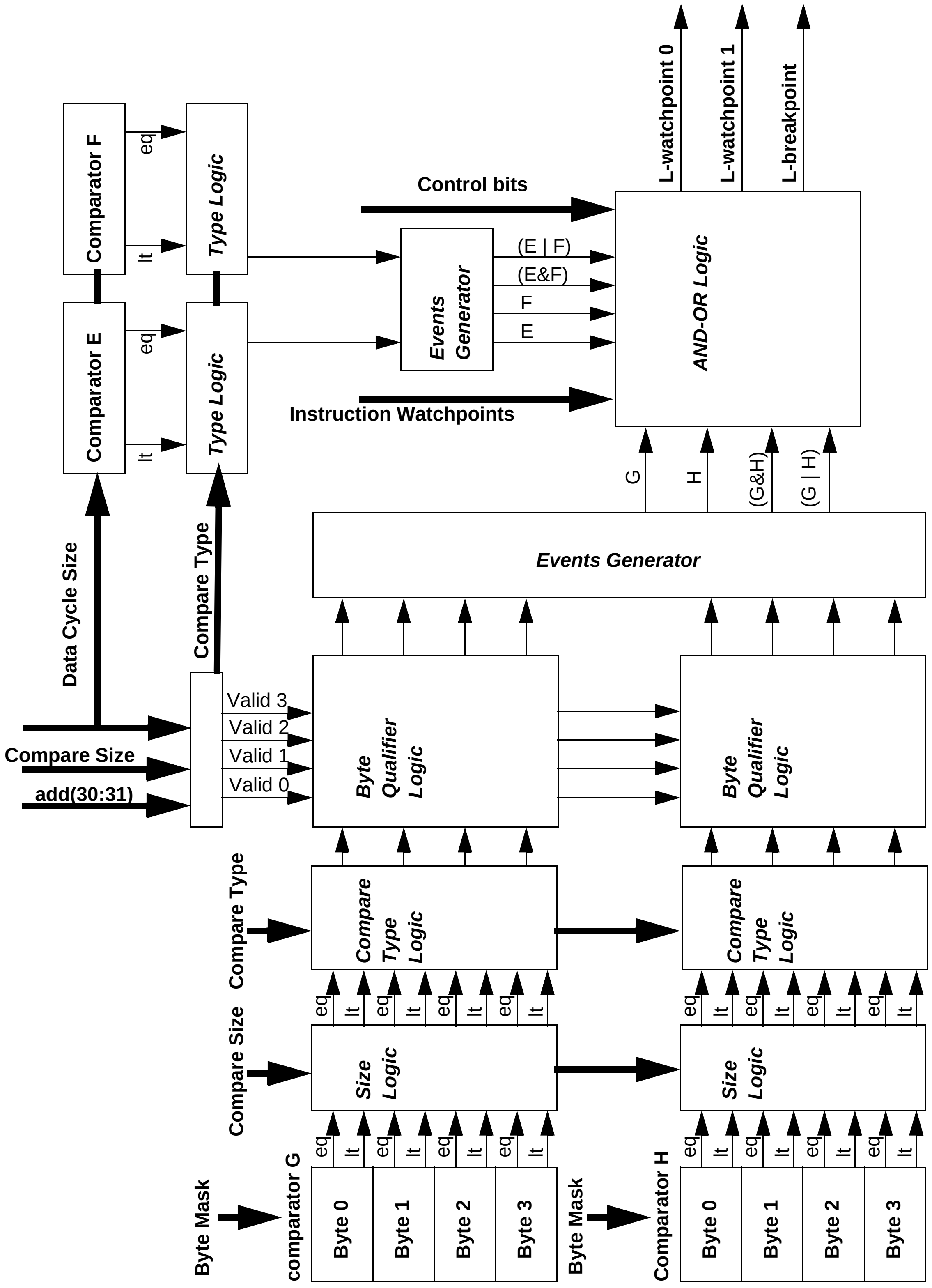

Figure 21-4 Load/Store Support General Structure

21.3.3 Watchpoint Counters

There are two 16-bit watchpoint counters. Each counter is able to count one of the instruction watchpoints or one of the load/store watchpoints. Both generate the corresponding breakpoint when they reach ZERO.

When working in the *masked* mode, the counters do **not** count watchpoints detected when MSRRI = 0. See **21.3.1.4 Context Dependent Filter**

The counters value when counting watchpoints programmed on the actual instructions that alter the counters, are not predictable. Reading values from the counters when they are active, must be synchronized by inserting a sync instruction before the actual read is performed.

NOTE

When programmed to count instruction watchpoints, the last instruction which decrements the counter to ZERO is treated like any other instruction breakpoint in the sense that it is not executed and the machine branches to the breakpoint exception routine BEFORE it executes this instruction. As a side effect of this behavior, the value of the counter inside the breakpoint exception routine equals ONE and not ZERO as might be expected.

When programmed to count load/store watchpoints, the last instruction which decrements the counter to ZERO is treated like any other load/store breakpoint in the sense that it is executed and the machine branches to the breakpoint exception routine AFTER it executes this instruction. Therefore, the value of the counter inside the breakpoint exception routine equals ZERO.

21.3.3.1 Trap Enable Programming

The trap enable bits can be programmed by regular software (only if MSRPR = 0) using THE **mtspr** instruction or “on the fly” using the special development port interface. For more information refer to section **21.5.6.5 Development Port Serial Communications — Trap Enable Mode**.

The value used by the breakpoints generation logic is the bit wise **OR** of the software trap enable bits, (the bits written using the **mtspr**) and the development port trap enable bits (the bits serially shifted using the development port).

All bits, the software trap enable bits and the development port trap enable bits, can be read from ICTRL and the LCTRL2 using **mfspr**. For the exact bits placement refer to **21.7.6 I-Bus Support Control Register** and to **21.7.8 L-Bus Support Control Register 2**

21.4 Development System Interface

When debugging an existing system, it is sometimes desirable to be able to do so without the need to insert any changes in the existing system. In some cases it is not desired, or even impossible, to add load to the lines connected to the existing system. The development system interface of the CPU supports such a configuration.

The development system interface of the CPU uses a dedicated serial port (the development port) and, therefore, does not need any of the regular system interfaces. Controlling the activity of the system from the development port is done when the CPU is in the debug mode. The development port is a relatively economical interface (three pins) that allows the development system to operate in a lower frequency than the frequency of the CPU. Note that it is also possible to debug the CPU using monitor debugger software, for more information refer to **21.6 Software Monitor Debugger Support**.

Debug mode is a state where the CPU fetches all instructions from the development port. In addition, when in debug mode, data can be read from the development port and written to the development port. This allows memory and registers to be read and modified by a development tool (emulator) connected to the development port.

For protection purposes, two possible working modes are defined: debug mode enable and debug mode disable. These working modes are selected only during reset. For more information refer to **21.4.1.1 Debug Mode Enable vs. Debug Mode Disable**

The user can work in debug mode starting from reset or the CPU can be programmed to enter debug mode as a result of a predefined list of events. These events include all possible interrupts and exceptions in the CPU system, including the internal breakpoints, together with two levels of development port requests (*masked* and *non-masked*) and one peripheral breakpoint request that can be generated by any one of the peripherals of the system (including internal and external modules). Each event can be programmed either to be treated as a regular interrupt that causes the machine to branch to its interrupt vector, or to be treated as a special interrupt that causes debug mode entry.

When in debug mode an **rfi** instruction will return the machine to its regular work mode.

The relationship between the debug mode logic to the rest of the CPU chip is shown in the following figure.

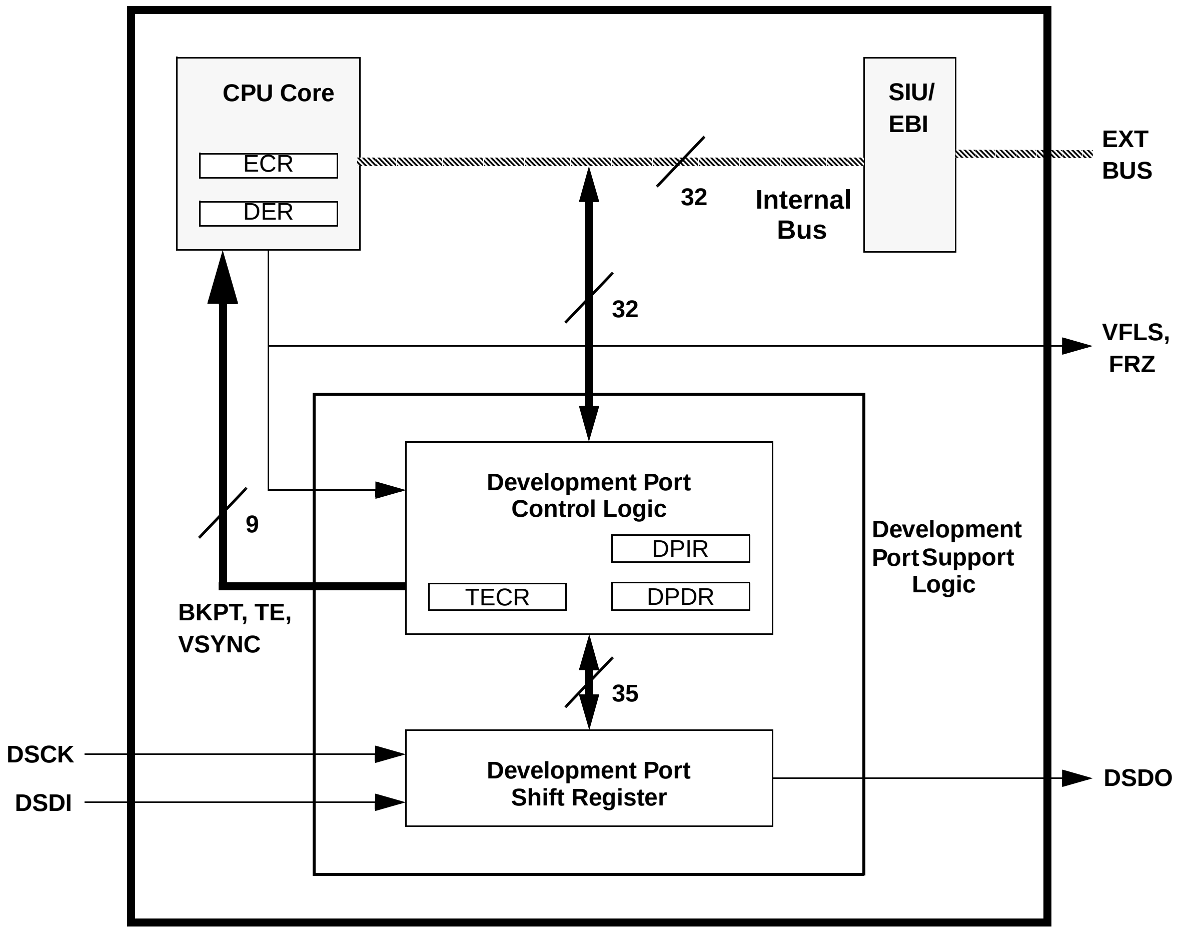

Figure 21-5 Functional Diagram of MPC555 Debug Mode Support

The development port provides a full duplex serial interface for communications between the internal development support logic of the CPU and an external development tool. The development port can operate in two working modes: the trap enable mode and the debug mode.

The trap enable mode is used in order to shift into the CPU internal development support logic the following control signals:

1. Instruction trap enable bits, used for on the fly programming of the instruction breakpoint
2. Load/store trap enable bits, used for on the fly programming of the load/store breakpoint
3. Non-maskable breakpoint, used to assert the non-maskable external breakpoint
4. Maskable breakpoint, used to assert the maskable external breakpoint
5. VSYNC, used to assert and negate VSYNC

In debug mode the development port controls also the debug mode features of the CPU. For more information **21.5 Development Port**

21.4.1 Debug Mode Support

The debug mode of the CPU provides the development system with the following basic functions:

- Gives an ability to control the execution of the processor and maintain control on it under all circumstances. The development port is able to force the CPU to enter to the debug mode even when external interrupts are disabled.
- It is possible to enter debug mode immediately out of reset thus allowing the user even to debug a ROM-less system.
- The user can selectively define, using an enable register, the events that will cause the machine to enter into the debug mode.
- When in debug mode the user can detect the reason upon which the machine entered debug mode by reading a cause register.
- Entering into the debug mode in all regular cases is restartable in the sense that the user is able to continue to run his regular program from the location where it entered the debug mode.
- When in debug mode all instructions are fetched from the development port but load/store accesses are performed on the real system memory.
- Data Register of the development port is accessed using **mtspr** and **mfspr** instructions via special load/store cycles. (This feature together with the last one enables easy memory dump & load).
- Upon entering debug mode, the processor gets into the privileged state (MSRPR = 0). This allows execution of any instruction, and access to any storage location.
- An OR signal of all exception cause register (ECR) bits (ECR_OR) enables the development port to detect pending events while already in debug mode. An example is the ability of the development port to detect a debug mode access to a non existing memory space.

The following figure illustrates the debug mode logic implemented in the CPU.

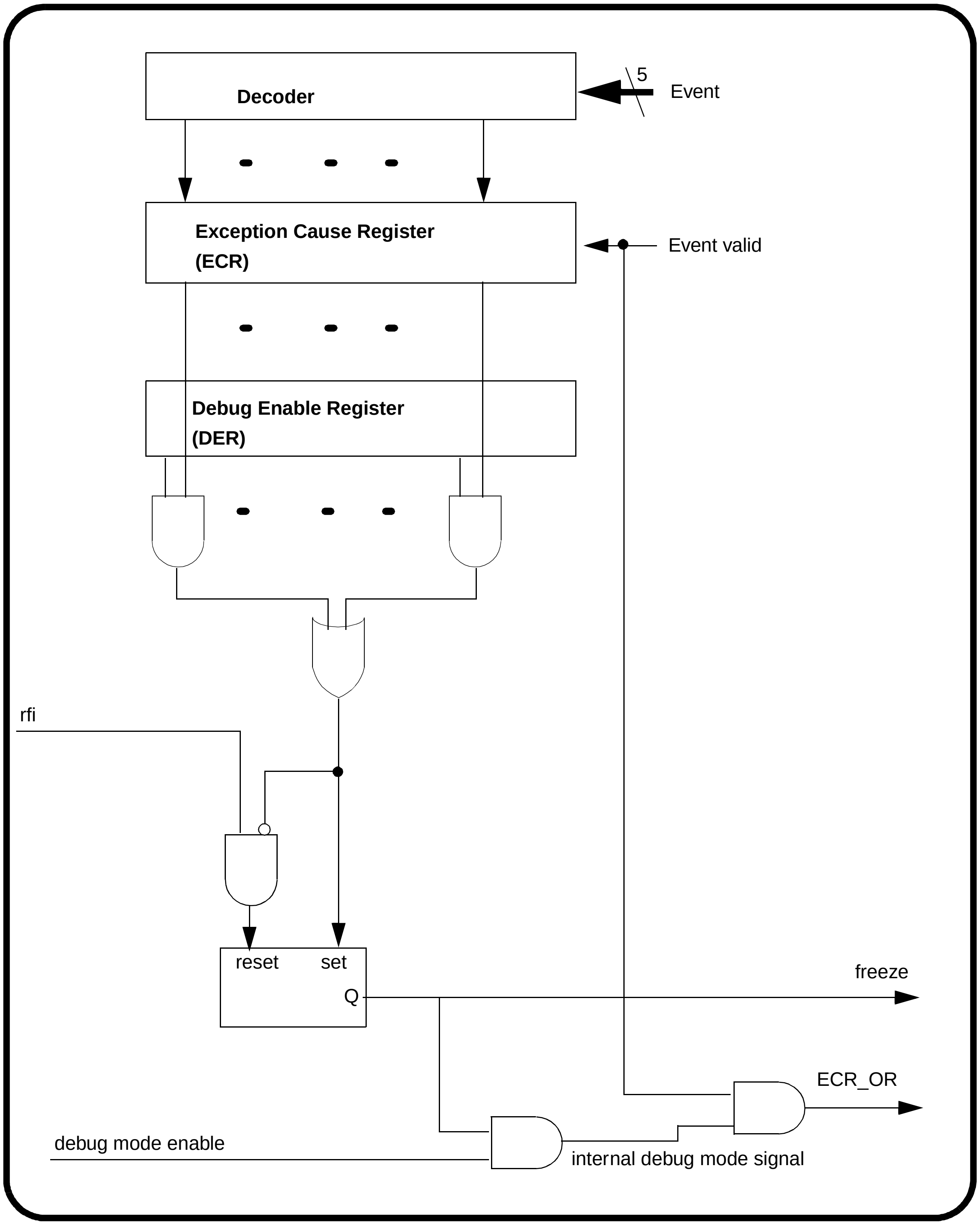

Figure 21-6 Debug Mode Logic

21.4.1.1 Debug Mode Enable vs. Debug Mode Disable

For protection purposes two possible working modes are defined: debug mode enable and debug mode disable. These working modes are selected only during reset.

Debug mode is enabled by asserting the DSCK pin during reset. The state of this pin is sampled three clocks before the negation of $\overline{\text{SRESET}}$.

NOTE

Since $\overline{\text{SRESET}}$ negation is done by an external pull up resistor any reference here to $\overline{\text{SRESET}}$ negation time refers to the time the MPC555 releases $\overline{\text{SRESET}}$. If the actual negation is slow due to large resistor, set up time for the debug port signals should be set accordingly.

If the DSCK pin is sampled negated, debug mode is disabled until a subsequent reset when the DSCK pin is sampled in the asserted state. When debug mode is disabled the internal watchpoint/breakpoint hardware will still be operational and may be used by a software monitor program for debugging purposes.

When working in debug mode disable, all development support registers (see list in **Table 21-14**) are accessible to the supervisor code (MSRPR = 0) and can be used by a monitor debugger software. However, the processor *never* enters debug mode and, therefore, the exception cause register (ECR) and the debug enable register (DER) are used only for asserting and negating the freeze signal. For more information on the software monitor debugger support refer to **21.6 Software Monitor Debugger Support**.

When working in debug mode enable, all development support registers are accessible *only* when the CPU is in debug mode. Therefore, even supervisor code that may be still under debug cannot prevent the CPU from entering debug mode. The development system has full control of all development support features of the CPU through the development port. Refer to **Table 21-16**

21.4.1.2 Entering Debug Mode

Entering debug mode can be a result of a number of events. All events have a programmable enable bit so the user can selectively decide which events result in debug mode entry and which in regular interrupt handling.

Entering debug mode is also possible immediately out of reset, thus allowing the user to debug even a ROM-less system. Using this feature is possible by special programming of the development port during reset. If the DSCK pin continues to be asserted following $\overline{\text{SRESET}}$ negation (after enabling debug mode) the processor will take a breakpoint exception and go directly to debug mode instead of fetching the reset vector. To avoid entering debug mode following reset, the DSCK pin must be negated no later than seven clock cycles after $\overline{\text{SRESET}}$ negates. In this case, the processor will jump to the reset vector and begin normal execution. When entering debug mode immediately after reset, bit 31 (development port interrupt) of the exception cause register (ECR) is set.

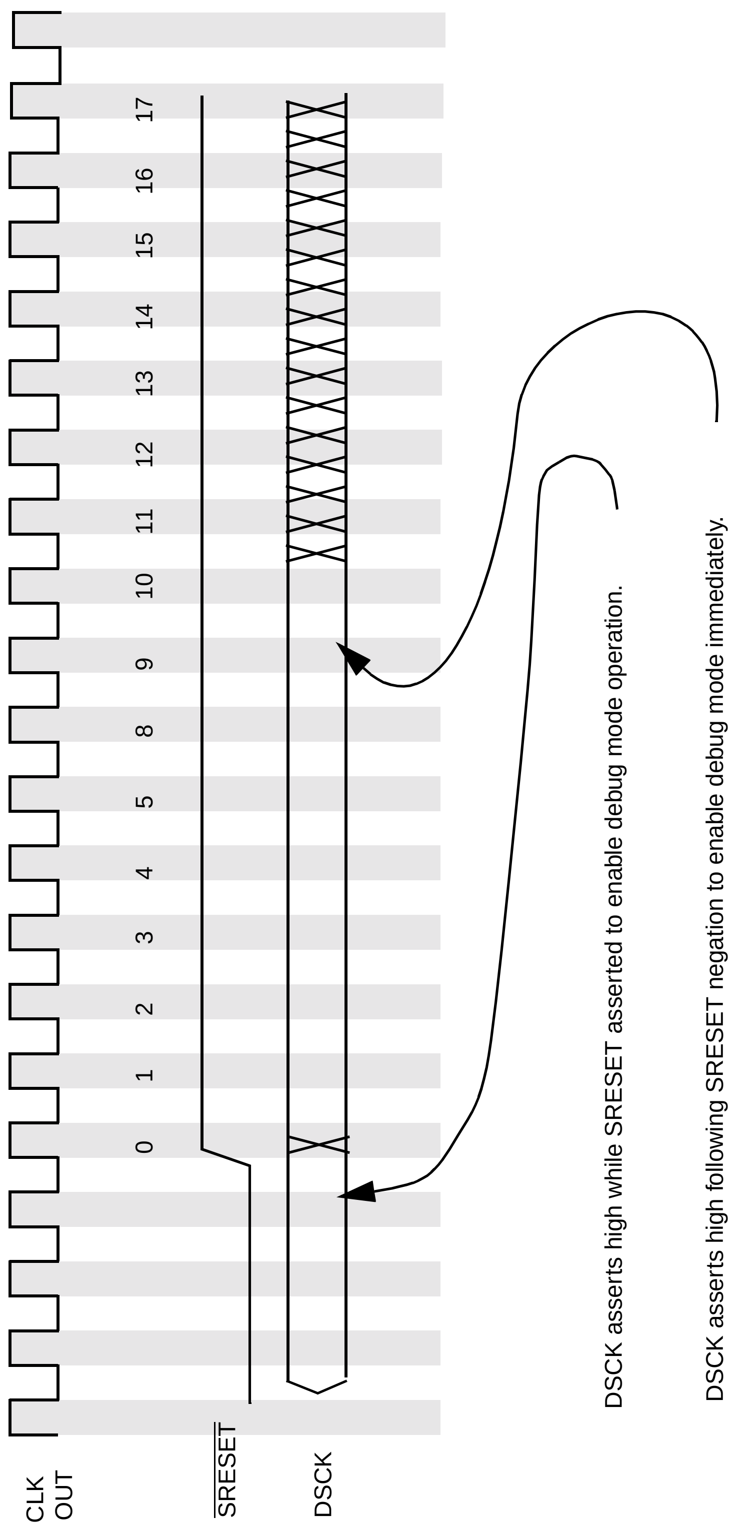
CLK OUT
SRESET
DSCK
0
1
2
3
4
5
8
9
10
11
12
13
14
15
16
17
DSCK asserts high while SRESET asserted to enable debug mode operation.
DSCK asserts high following SRESET negation to enable debug mode immediately.

Figure 21-7 Debug Mode Reset Configuration

When debug mode is disabled all events result in regular interrupt handling.

The internal freeze signal is asserted whenever an enabled event occurs, regardless if debug mode is enabled or disabled. The internal freeze signal is connected to all relevant internal modules. These modules can be programmed to stop all operations in response to the assertion of the freeze signal. Refer to **21.6.1 Freeze Indication**.

The freeze indication is negated when exiting debug mode. Refer to **21.4.1.6 Exiting Debug Mode**

The following list contains the events that can cause the CPU to enter debug mode. Each event results in debug mode entry if debug mode is enabled and the corresponding enable bit is set. The reset values of the enable bits let the user, in most cases, to use of the debug mode features without the need to program the debug enable register (DER). For more information refer to **21.7.12 Debug Enable Register (DER)**.

- NMI exception as a result of the assertion of the IRQ0_B pin. For more information refer to **3.15.4.1 System Reset Interrupt**
- Check stop. Refer to **21.4.1.3 The Check Stop State and Debug Mode**
- Machine check exception
- Implementation specific instruction protection error
- Implementation specific data protection error
- External interrupt, recognized when MSREE = 1
- Alignment interrupt
- Program interrupt
- Floating point unavailable exception
- Floating point assist exception
- Decrementer exception, recognized when MSREE = 1
- System call exception
- Trace, asserted when in single trace mode or when in branch trace mode (refer to **3.15.4.10 Trace Interrupt**)
- Implementation dependent software emulation exception
- Instruction breakpoint, when breakpoints are *masked* (BRKNOMSK bit in the LCTRL2 is clear) recognized only when MSRRI = 1, when breakpoints are not *mask*ed (BRKNOMSK bit in the LCTRL2 is set) always recognized
- Load/store breakpoint, when breakpoints are *masked* (BRKNOMSK bit in the LCTRL2 is cleared) recognized only when MSRRI = 1, when breakpoints are not *masked* (BRKNOMSK bit in the LCTRL2 is set) always recognized
- Peripherals breakpoint, from the development port, internal and external modules. are recognized only when MSRRI = 1.
- Development port non-maskable interrupt, as a result of a debug station request. Useful in some catastrophic events like an endless loop when MSRRI = 0. As a result of this event the machine may enter a non-restartable state, for more information refer to **3.15.4 Interrupts**.

The processor enters into the debug mode state when at least one of the bits in the exception cause register (ECR) is set, the corresponding bit in the debug enable register (DER) is enabled and debug mode is enabled. When debug mode is enabled and an enabled event occurs, the processor waits until its pipeline is empty and then starts fetching the next instructions from the development port. For information on the exact value of machine status save/restore registers (SRR0 and SRR1) refer to **3.15.4 Interrupts**

When the processor is in debug mode the freeze indication is asserted thus allowing any peripheral that is programmed to do so to stop. The fact that the CPU is in debug mode is also broadcast to the external world using the value b11 on the VFLS pins.

NOTE

The freeze signal can be asserted by software when debug mode is disabled.

The development port should read the value of the exception cause register (ECR) in order to get the cause of the debug mode entry. *Reading the exception cause register (ECR) clears all its bits*.

21.4.1.3 The Check Stop State and Debug Mode

The CPU enters the check stop state if the machine check interrupt is disabled (MSRME = 0) and a machine check interrupt is detected. However, if a machine check interrupt is detected when MSRME = 0, debug mode is enabled and the check stop enable bit in the debug enable register (DER) is set, the CPU enters debug mode rather then the check stop state.

The different actions taken by the CPU when a machine check interrupt is detected are shown in the following table.

Table 21-9 The Check Stop State and Debug Mode

MSR_{ME}	Debug Mode Enable	CHSTPE[1]	MCIE[2]	Action Performed by the CPU when Detecting a Machine Check Interrupt	Exception Cause Register (ECR) Value
0	0	X	X	Enter the check stop state	0x20000000
1	0	X	X	Branch to the machine check interrupt	0x10000000
0	1	0	X	Enter the check stop state	0x20000000
0	1	1	X	Enter Debug Mode	0x20000000
1	1	X	0	Branch to the machine check interrupt	0x10000000
1	1	X	1	Enter Debug Mode	0x10000000

NOTES:

1. Check stop enable bit in the debug enable register (DER)
2. Machine check interrupt enable bit in the debug enable register (DER)

21.4.1.4 Saving Machine State upon Entering Debug Mode

If entering debug mode was as a result of any load/store type exception, and therefore the DAR (data address register) and DSISR (data storage interrupt status register) have some significant value, these two registers must be saved before any other operation is performed. Failing to save these registers may result in loss of their value in case of another load/store type exception inside the development software.

Since exceptions are treated differently when in debug mode (refer to **21.4.1.5 Running in Debug Mode**), there is no need to save machine status save/restore zero register (SRR0) and machine status save/restore one register (SRR1).

21.4.1.5 Running in Debug Mode

When running in debug mode all fetch cycles access the development port regardless of the actual address of the cycle. All load/store cycles access the real memory system according to the cycle's address. The data register of the development port is mapped as a special control register therefore it is accessed using **mtspr** and **mfspr** instructions via special load/store cycles (refer to **21.7.13 Development Port Data Register (DPDR)**).

Exceptions are treated differently when running in debug mode. When already in debug mode, upon recognition of an exception, the exception cause register (ECR) is updated according to the event that caused the exception, a special error indication (ECR_OR) is asserted for one clock cycle to report to the development port that an exception occurred and execution continues in debug mode without any change in SRR0 and SRR1. ECR_OR is asserted before the next fetch occurs to allow the development system to detect the excepting instruction.

Not all exceptions are recognized when in debug mode. Breakpoints and watchpoints are not generated by the hardware when in debug mode (regardless of the value of MSRRI). Upon entering debug mode MSREE is cleared by the hardware thus forcing the hardware to ignore external and decrementer interrupts.

Setting the MSREE bit while in debug mode, (by the debug software), is strictly forbidden. The reason for this restriction is that the external interrupt event is a level signal, and since the CPU only reports exceptions while in debug mode but do not treat them, the CPU does not clear the MSREE bit and, therefore, this event, if enabled, is recognized again every clock cycle.

When the ECR_OR signal is asserted the development station should investigate the exception cause register (ECR) in order to find out the event that caused the exception.

Since the values in SRR0 and SRR1 do not change if an exception is recognized while already in debug mode, they only change once when entering debug mode, saving them when entering debug mode is not necessary.

21.4.1.6 Exiting Debug Mode

The **rfi** instruction is used to exit from debug mode in order to return to the normal processor operation and to negate the freeze indication. The development system may monitor the freeze status to make sure the MPC555 is out of debug mode. It is the responsibility of the software to read the exception cause register (ECR) before performing the **rfi**. Failing to do so will force the CPU to immediately re-enter to debug mode and to re-assert the freeze indication in case an asserted bit in the interrupt cause register (ECR) has a corresponding enable bit set in the debug enable register (DER).

21.5 Development Port

The development port provides a full duplex serial interface for communications between the internal development support logic including debug mode and an external development tool.

The relationship of the development support logic to the rest of the CPU chip is shown in **Figure 21-5**. The development port support logic is shown as a separate block for clarity. It is implemented as part of the SIU module.

21.5.1 Development Port Pins

The following development port pin functions are provided:

1. Development serial clock (DSCK)
2. Development serial data in (DSDI)
3. Development serial data out (DSDO)

21.5.2 Development Serial Clock

The development serial clock (DSCK) is used to shift data into and out of the development port shift register. At the same time, the new most significant bit of the shift register is presented at the DSDO pin. In all further discussions references to the DSCK signal imply the internal synchronized value of the clock. The DSCK input must be driven either high or low at all times and not allowed to float. A typical target environment would pull this input low with a resistor.

The clock may be implemented as a free running clock or as gated clock. As discussed in section **21.5.6.5 Development Port Serial Communications — Trap Enable Mode** and section **21.5.6.8 Development Port Serial Communications — Debug Mode**, the shifting of data is controlled by ready and start signals so the clock does not need to be gated with the serial transmissions.

The DSCK pin is also used at reset to enable debug mode and immediately following reset to optionally cause immediate entry into debug mode following reset.

21.5.3 Development Serial Data In

Data to be transferred into the development port shift register is presented at the development serial data in (DSDI) pin by external logic. To be sure that the correct value is used internally. When driven asynchronous (synchronous) with the system clock, the data presented to DSDI must be stable a setup time before the rising edge of DSCK (CLKOUT) and a hold time after the rising edge of DSCK (CLKOUT).

The DSDI pin is also used at reset to control the overall chip configuration mode and to determine the development port clock mode. See section **21.5.6.4 Development Port Serial Communications — Clock Mode Selection** for more information.

21.5.4 Development Serial Data Out

The debug mode logic shifts data out of the development port shift register using the development serial data out (DSDO) pin. All transitions on DSDO are synchronous with DSCK or CLKOUT depending on the clock mode. Data will be valid a setup time before the rising edge of the clock and will remain valid a hold time after the rising edge of the clock.

Refer to **Table 21-12** for DSDO data meaning.

21.5.5 Freeze Signal

The freeze indication means that the processor is in debug mode (i.e., normal processor execution of user code is frozen). On the MPC555, the freeze state can be indicated by three different pins. The FRZ signal is generated synchronously with the system clock. This indication may be used to halt any off-chip device while in debug mode as well as a handshake means between the debug tool and the debug port. The internal freeze status can also be monitored through status in the data shifted out of the debug port.

21.5.5.1 SGPIO6/FRZ/$\overline{\text{PTR}}$ Pin

The SGPIO6/FRZ/$\overline{\text{PTR}}$ pin powers up as the $\overline{\text{PTR}}$ function and its function is controlled by the GPC bits in the SIUMCR.

21.5.5.2 IWP[0:1]/VFLS[0:1] Pins

The IWP[0:1]/VFLS[0:1] pins power up as the VFLS[0:1] function and their function can be changed via the DBGC bits in the SIUMCR (see **6.13.1.1 SIU Module Configuration Register**). They can also be set via the reset configuration word (See **7.5.2 Hard Reset Configuration Word)**. The FRZ state is indicated by the value b11 on the VFLS[0:1] pins.

21.5.5.3 VFLS[0:1]_MPIO32B[3:4] Pins

The VFLS[0:1]_MPIO32B[3:4] Pins power up as the MPIO32B[3:4] function and their function can be changed via the VFLS bit in the MIOS1TPCR register (see section 15.15.1.1). The FRZ state is indicated by the value b11 on the VFLS[0:1] pins.

21.5.6 Development Port Registers

The development port consists logically of the three registers: development port instruction register (DPIR), development port data register (DPDR), and trap enable control register (TECR). These registers are physically implemented as two registers, development port shift register and trap enable control register. The development port shift register acts as both the DPIR and DPDR depending on the operation being performed. It is also used as a temporary holding register for data to be stored into the TECR. These registers are discussed below in more detail.

21.5.6.1 Development Port Shift Register

The development port shift register is a 35-bit shift register. Instructions and data are shifted into it serially from DSDI using DSCK (or CLKOUT depending on the debug port clock mode, refer to **21.5.6.4 Development Port Serial Communications — Clock Mode Selection**) as the shift clock. These instructions or data are then transferred in parallel to the CPU, the trap enable control register (TECR). When the processor enters debug mode it fetches instructions from the DPIR which causes an access to the development port shift register. These instructions are serially loaded into the shift register from DSDI using DSCK (or CLKOUT) as the shift clock. In a similar way, data is transferred to the CPU by moving it into the shift register which the processor reads as the result of executing a “move from special purpose register DPDR” instruction. Data is also parallel-loaded into the development port shift register from the CPU by executing a “move to special purpose register DPDR” instruction. It is then shifted out serially to DSDO using DSCK (or CLKOUT) as the shift clock.

21.5.6.2 Trap Enable Control Register

The trap enable control register is a 9-bit register that is loaded from the development port shift register. The contents of the control register are used to drive the six trap enable signals, the two breakpoint signals, and the VSYNC signal to the CPU. The “transfer data to trap enable control register” commands will cause the appropriate bits to be transferred to the control register.

The trap enable control register is not accessed by the CPU, but instead supplies signals to the CPU. The trap enable bits, VSYNC bit, and the breakpoint bits of this register are loaded from the development port shift register as the result of trap enable mode transmissions. The trap enable bits are reflected in ICTRL and LCTRL2 special registers. See **21.7.6 I-Bus Support Control Register** and **21.7.8 L-Bus Support Control Register 2**.

21.5.6.3 Development Port Registers Decode

The development port shift register is selected when the CPU accesses DPIR or DPDR. Accesses to these two special purpose registers occur in debug mode and appear on the internal bus as an address and the assertion of an address attribute signal indicating that a special purpose register is being accessed. The DPIR register is read by the CPU to fetch all instructions when in debug mode and the DPDR register is read and written to transfer data between the CPU and external development tools. The DPIR and DPDR are pseudo registers. Decoding either of these registers will cause the development port shift register to be accessed. The debug mode logic knows whether the CPU is fetching instructions or reading or writing data. If what the CPU is expecting and what the register receives from the serial port do not match (instruction instead of data) the mismatch is used to signal a sequence error to the external development tool.

21.5.6.4 Development Port Serial Communications — Clock Mode Selection

All of the serial transmissions are clock transmissions and are therefore synchronous communications. However, the transmission clock may be either synchronous or asynchronous with the system clock (CLKOUT). The development port allows the user to select two methods for clocking the serial transmissions. The first method allows the transmission to occur without being externally synchronized with CLKOUT, in this mode a serial clock DSCK must be supplied to the MPC555. The other communication method requires a data to be externally synchronized with CLKOUT.

The first clock mode is called “asynchronous clock” since the input clock (DSCK) is asynchronous with CLKOUT. To be sure that data on DSDI is sampled correctly, transitions on DSDI must occur a setup time ahead and a hold time after the rising edge of DSCK. This clock mode allows communications with the port from a development tool which does not have access to the CLKOUT signal or where the CLKOUT signal has been delayed or skewed. Refer to the timing diagram in **Figure 21-8**

The second clock mode is called “synchronous self clock”. It does not require an input clock. Instead the port is timed by the system clock. The DSDI input is required to meet setup and hold time requirements with respect to CLKOUT rising edge. The data rate for this mode is always the same as the system clock. Refer to the timing diagram in **Figure 21-9**.

The selection of clock or self clock mode is made at reset. The state of the DSDI input is latched eight clocks after $\overline{\text{SRESET}}$ negates. If it is latched low, asynchronous clock mode is enabled. If it is latched high then synchronous self clock mode is enabled.

Since DSDI is used to select the development port clocking scheme, it is necessary to prevent any transitions on DSDI during this time from being recognized as the start of a serial transmission. The port will not begin scanning for the start bit of a serial transmission until 16 clocks after the negation of $\overline{\text{SRESET}}$. If DSDI is asserted 16 clocks after $\overline{\text{SRESET}}$ negation, the port will wait until DSDI is negated to begin scanning for the start bit.

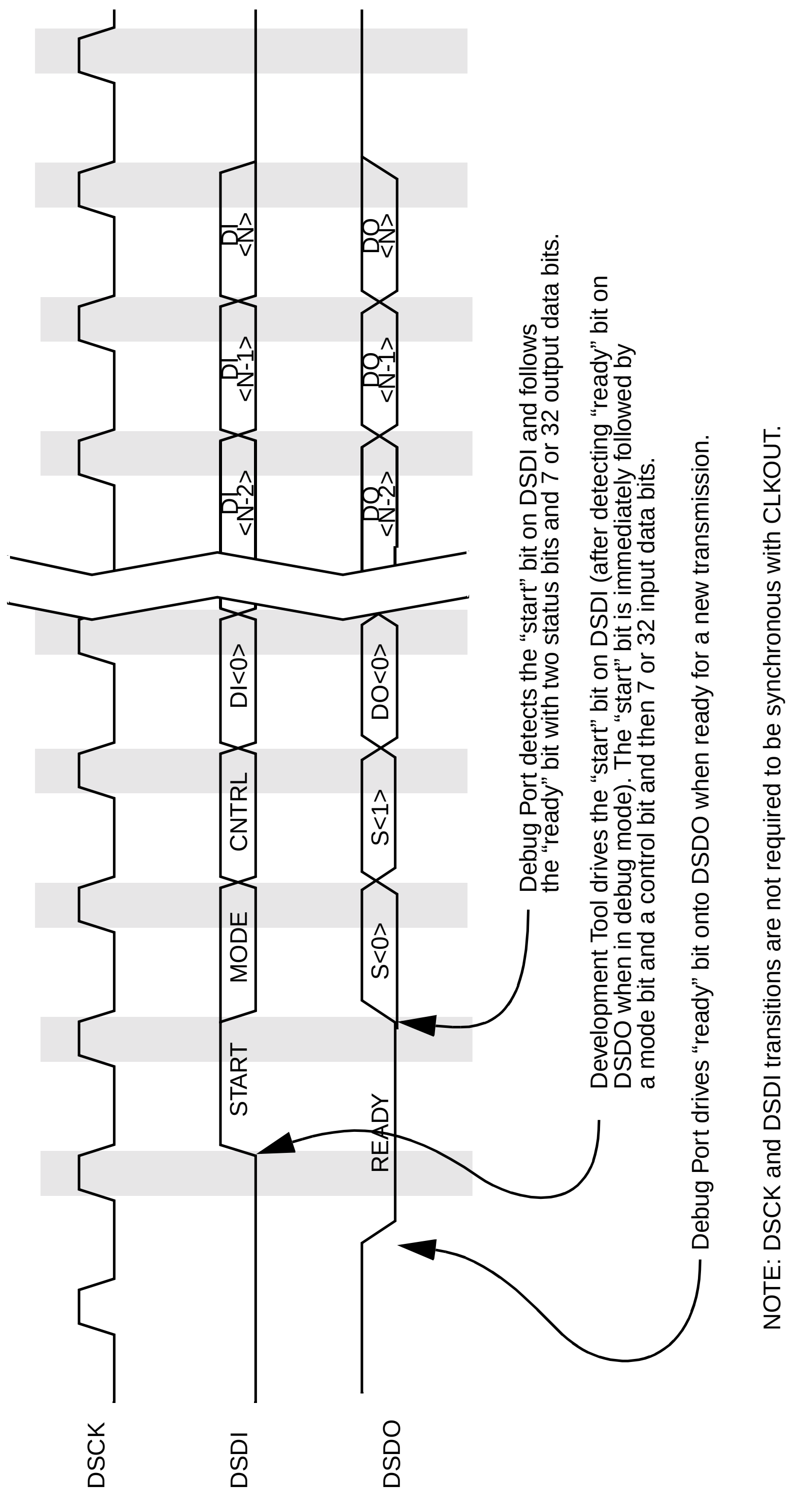
DSCK
DSDI
DSDO
START
MODE
CNTRL
DI<0>
DI <N-2>
DI <N-1>
DI <N>
READY
S<0>
S<1>
DO<0>
DO <N-2>
DO <N-1>
DO <N>
Debug Port detects the “start” bit on DSDI and follows the “ready” bit with two status bits and 7 or 32 output data bits.
Development Tool drives the “start” bit on DSDI (after detecting “ready” bit on DSDO when in debug mode). The “start” bit is immediately followed by a mode bit and a control bit and then 7 or 32 input data bits.
Debug Port drives “ready” bit onto DSDO when ready for a new transmission.
NOTE: DSCK and DSDI transitions are not required to be synchronous with CLKOUT.

Figure 21-8 Asynchronous Clock Serial Communications

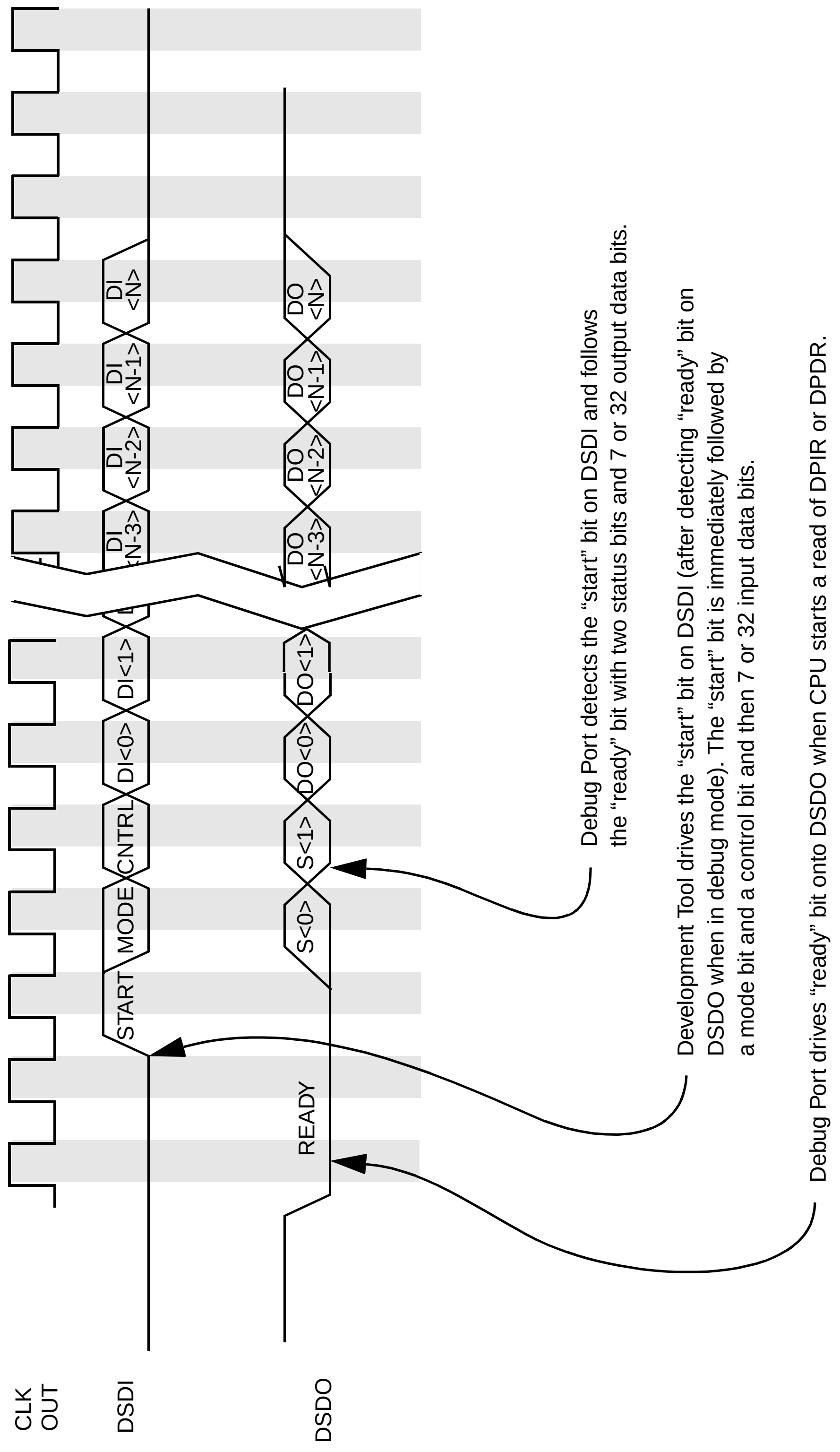
CLK OUT
DSDI
DSDO
START
MODE
CNTRL
DI<0>
DI<1>
DI <N-3>
DI <N-2>
DI <N-1>
DI <N>
READY
S<0>
S<1>
DO<0>
DO<1>
DO <N-3>
DO <N-2>
DO <N-1>
DO <N>
Debug Port detects the “start” bit on DSDI and follows the “ready” bit with two status bits and 7 or 32 output data bits.
Development Tool drives the “start” bit on DSDI (after detecting “ready” bit on DSDO when in debug mode). The “start” bit is immediately followed by a mode bit and a control bit and then 7 or 32 input data bits.
Debug Port drives “ready” bit onto DSDO when CPU starts a read of DPIR or DPDR.

Figure 21-9 Synchronous Self Clock Serial Communication

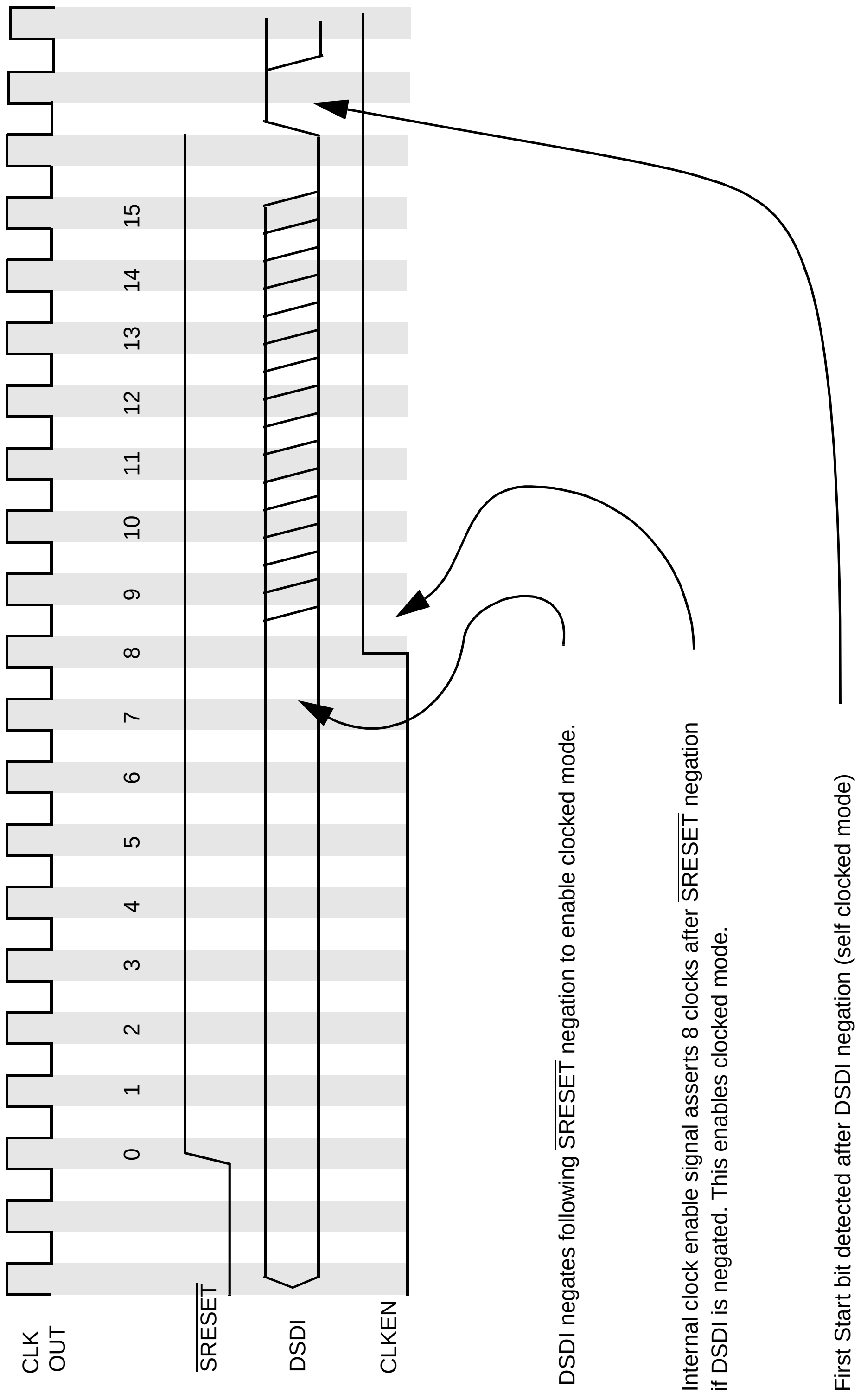
CLK OUT
SRESET
DSDI
CLKEN
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
DSDI negates following SRESET negation to enable clocked mode.
Internal clock enable signal asserts 8 clocks after SRESET negation if DSDI is negated. This enables clocked mode.
First Start bit detected after DSDI negation (self clocked mode)

Figure 21-10 Enabling Clock Mode Following Reset

21.5.6.5 Development Port Serial Communications — Trap Enable Mode

When in not in debug mode the development port starts communications by setting DSDO (the MSB of the 35-bit development port shift register) low to indicate that all activity related to the previous transmission are complete and that a new transmission may begin. The start of a serial transmission from an external development tool to the development port is signaled by a start bit. A mode bit in the transmission defines the transmission as either a trap enable mode transmission or a debug mode transmission. If the mode bit is set the transmission will only be 10 bits long and only seven data bits will be shifted into the shift register. These seven bits will be latched into the TECR. A control bit determines whether the data is latched into the trap enable and VSYNC bits of the TECR or into the breakpoints bits of the TECR.

21.5.6.6 Serial Data into Development Port — Trap Enable Mode

The development port shift register is 35 bits wide but trap enable mode transmissions only use the start/ready bit, a mode/status bit, a control/status bit, and the seven least significant data bits. The encoding of data shifted into the development port shift register (through the DSDI pin) is shown in **Table 21-10** and **Table 21-11** below:

Table 21-10 Trap Enable Data Shifted into Development Port Shift Register

Start	Mode	Control	1st	2nd	3rd	4th	1st	2nd	VSYNC	Function
			- - - - - - Instruction- - - - - -				- - Data- -			
			Watchpoint Trap Enables							
1	1	0	0 = disabled; 1 = enabled							Transfer Data to Trap Enable Control Register

Table 21-11 Debug Port Command Shifted Into Development Port Shift Register

Start	Mode	Control	Extended Opcode		Major Opcode	Function
1	1	1	x	x	00000	NOP
					00001	Hard Reset request
					00010	Soft Reset request
			0	x	00011	Reserved
			1	0	00011	End Download procedure
			1	1	00011	Start Download procedure
			x	x	00100... 11110	Reserved
			x	0	11111	Negate Maskable breakpoint.
			x	1	11111	Assert Maskable breakpoint.
			0	x	11111	Negate Non Maskable breakpoint.
			1	x	11111	Assert Non Maskable breakpoint.

The watchpoint trap enables and VSYNC functions are described in section **21.3 Watchpoints and Breakpoints Support** and section **21.2 Program Flow Tracking**.

The debug port command function allows the development tool to either assert or negate breakpoint requests, reset the processor, activate or deactivate the fast download procedure.

21.5.6.7 Serial Data Out of Development Port — Trap Enable Mode

In trap enable mode the only response out of the development port is “sequencing error.”

Data that can come out of the development port is shown in **Table 21-12**. “Valid data from CPU” and “CPU interrupt” status cannot occur in trap enable mode.

Table 21-12 Status / Data Shifted Out of Development Port Shift Register

Ready	Status [0:1]		Data			Function
			Bit 0	Bit 1	Bits 2:31 or 2:6 — (Depending on Input Mode)	
(0)	0	0	Data			Valid Data from CPU
(0)	0	1	Freeze status[1]	Download Procedure in progress[2]	1’s	Sequencing Error
(0)	1	0			1’s	CPU Interrupt
(0)	1	1			1’s	Null

NOTES:

1. The “Freeze” status is set to (1) when the CPU is in debug mode and to (0) otherwise.
2. The “Download Procedure in progress” status is asserted (0) when Debug port in the Download procedure and is negated (1) otherwise.

When not in debug mode the sequencing error encoding indicates that the transmission from the external development tool was a debug mode transmission. When a sequencing error occurs the development port will ignore the data shifted in while the sequencing error was shifting out. It will be treated as a NOP function.

Finally, the null output encoding is used to indicate that the previous transmission did not have any associated errors.

When not in debug mode, ready will be asserted at the end of each transmission. If debug mode is not enabled and transmission errors can be guaranteed not to occur, the status output is not needed.

21.5.6.8 Development Port Serial Communications — Debug Mode

When in debug mode the development port starts communications by setting DSDO low to indicate that the CPU is trying to read an instruction from DPIR or data from DPDR. When the CPU writes data to the port to be shifted out the ready bit is not set. The port waits for the CPU to read the next instruction before asserting ready. This allows duplex operation of the serial port while allowing the port to control all transmissions from the external development tool. After detecting this ready status the external development tool begins the transmission to the development port with a start bit (logic high) on the DSDI pin.

21.5.6.9 Serial Data Into Development Port

In debug mode the 35 bits of the development port shift register are interpreted as a start/ready bit, a mode/status bit, a control/status bit, and 32 bits of data. All instructions and data for the CPU are transmitted with the mode bit cleared indicating a 32-bit data field. The encoding of data shifted into the development port shift register (through the DSDI pin) is shown below in **Table 21-13**

Table 21-13 Debug Instructions / Data Shifted Into Development Port Shift Register

Start	**Mode**	**Control**	**Instruction / Data (32 Bits)**		**Function**
			Bits 0:6	**Bits 7:31**	
1	0	0	CPU Instruction		Transfer Instruction to CPU
1	0	1	CPU Data		Transfer Data to CPU
1	1	0	Trap enable[1]	Not exist	Transfer data to Trap Enable Control Register
1	1	1	0011111	Not exist	Negate breakpoint requests to the CPU.
1	1	1	0	Not exist	nop

NOTES:
1. Refer to **Table 21-10**

Data values in the last two functions other than those specified are reserved.

All transmissions from the debug port on DSDO begin with a “0” or “ready” bit. This indicates that the CPU is trying to read an instruction or data from the port. The external development tool must wait until it sees DSDO go low to begin sending the next transmission.

The control bit differentiates between instructions and data and allows the development port to detect that an instruction was entered when the CPU was expecting data and vice versa. If this occurs a sequence error indication is shifted out in the next serial transmission.

The trap enable function allows the development tool to transfer data to the trap enable control register.

The debug port command function allows the development tool to either negate breakpoint requests, reset the processor, activate or deactivate the fast down load procedure.

The NOP function provides a null operation for use when there is data or a response to be shifted out of the data register and the appropriate next instruction or command will be determined by the value of the response or data shifted out.

21.5.6.10 Serial Data Out of Development Port

The encoding of data shifted out of the development port shift register in debug mode (through the DSDO pin) is the same as for trap enable mode and is shown in **Table 21-12**.

Valid data encoding is used when data has been transferred from the CPU to the development port shift register. This is the result of an instruction to move the contents of a general purpose register to the debug port data register (DPDR). The valid data encoding has the highest priority of all status outputs and will be reported even if an interrupt occurs at the same time. Since it is not possible for a sequencing error to occur and also have valid data there is no priority conflict with the sequencing error status. Also, any interrupt that is recognized at the same time that there is valid data is not related to the execution of an instruction. Therefore, a valid data status will be output and the interrupt status will be saved for the next transmission.

The sequencing error encoding indicates that the inputs from the external development tool are not what the development port and/or the CPU was expecting. Two cases could cause this error:

1. The processor was trying to read instructions and there was data shifted into the development port, or
2. The processor was trying to read data and there was instruction shifted into the development port. The port will terminate the read cycle with a bus error.

This bus error will cause the CPU to signal that an interrupt (exception) occurred. Since a status of sequencing error has a higher priority than exception, the port will report the sequencing error first, and the CPU interrupt on the next transmission. The development port will ignore the command, instruction, or data shifted in while the sequencing error or CPU interrupt is shifted out. The next transmission after all error status is reported to the port should be a new instruction, trap enable or command (possibly the one that was in progress when the sequencing error occurred).

The interrupt-occurred encoding is used to indicate that the CPU encountered an interrupt during the execution of the previous instruction in debug mode. Interrupts may occur as the result of instruction execution (such as unimplemented opcode or arithmetic error), because of a memory access fault, or from an unmasked external interrupt. When an interrupt occurs the development port will ignore the command, instruction, or data shifted in while the interrupt encoding was shifting out. The next transmission to the port should be a new instruction, trap enable or debug port command.

Finally, the null encoding is used to indicate that no data has been transferred from the CPU to the development port shift register.

21.5.6.11 Fast Download Procedure

The download procedure is used to download a block of data from the debug tool into system memory. This procedure can be accomplished by repeating the following sequence of transactions from the development tool to the debug port for the number of data words to be down loaded:

```
INIT:    Save RX, RY
         RY <- Memory Block address- 4

         •••

repeat:  mfspr       RX, DPDR
         DATA word to be moved to memory
         stwu        RX, 0x4(RY)
until here


         •••

         Restore RX,RY
```

Figure 21-11 Download Procedure Code Example

For large blocks of data this sequence may take significant time to complete. The "fast download procedure" of the debug port may be used to reduce this time. This time reduction is achieved by eliminating the need to transfer the instructions in the loop to the debug port. The only transactions needed are those required to transfer the data to be placed in system memory. **Figure 21-12** and **Figure 21-13** illustrate the time benefit of the "fast download procedure".

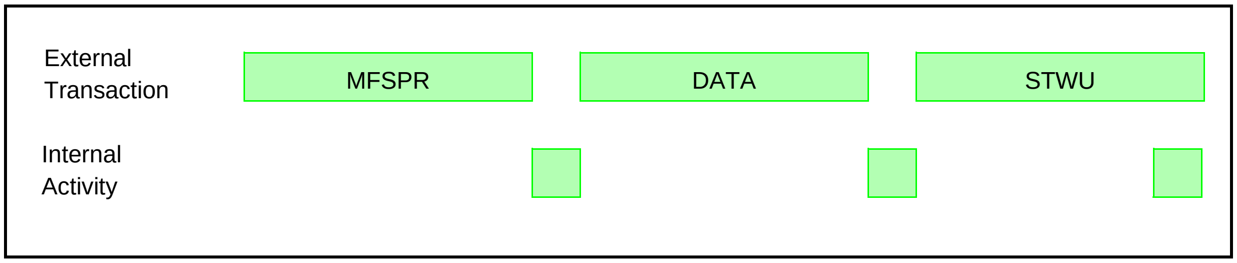

Figure 21-12 Slow Download Procedure Loop

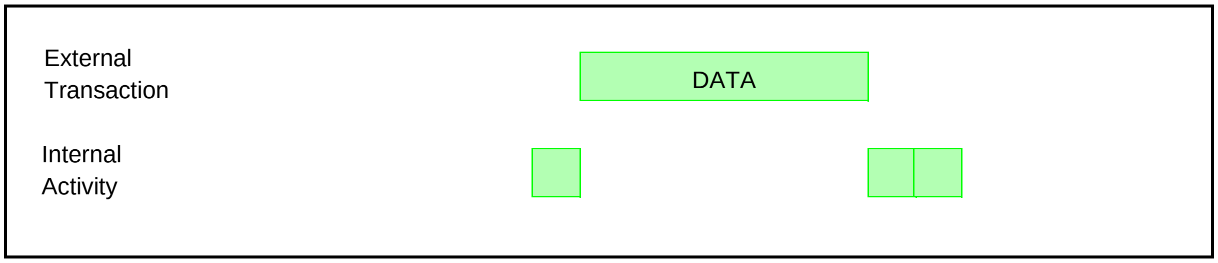

Figure 21-13 Fast Download Procedure Loop

The sequence of the instructions used in the “fast download procedure” is the one illustrated in **Figure 21-11** with RX = r31 and RY = r30. This sequence is repeated infinitely until the “end download procedure” command is issued to the debug port.

Note that, the internal general purpose register 31 is used for temporary storage data value. Before beginning the “fast download procedure” by the “start download procedure command”, The value of the first memory block address, – 4, must be written to the general purpose register 30.

To end a download procedure, an “end download procedure” command should be issued to the debug port, and then, additional DATA transaction should be sent by the development tool. This data word will NOT be placed into the system memory, but it is needed to stop the procedure gracefully.

21.6 Software Monitor Debugger Support

When in debug mode disable, a software monitor debugger can make use of all of the development support features defined in the CPU. When debug mode is disabled all events result in regular interrupt handling, i.e. the processor resumes execution in the corresponding interrupt handler. The exception cause register (ECR) and the debug enable register (DER) only influence the assertion and negation of the freeze signal.

21.6.1 Freeze Indication

The internal freeze signal is connected to all relevant internal modules. These modules can be programmed to stop all operations in response to the assertion of the freeze signal. In order to enable a software monitor debugger to broadcast the fact that the debug software is now executed, it is possible to assert and negate the internal freeze signal also when debug mode is disabled.

The assertion and negation of the freeze signal when in debug mode disable is controlled by the exception cause register (ECR) and the debug enable register (DER) as described in **Figure 21-6**. In order to assert the freeze signal the software needs to program the relevant bits in the debug enable register (DER). In order to negate the freeze line the software needs to read the exception cause register (ECR) in order to clear it and perform an **rfi** instruction.

If the exception cause register (ECR) is not cleared before the **rfi** is performed the freeze signal is not negated. Therefore it is possible to nest inside a software monitor debugger without affecting the value of the freeze line although **rfi** may be performed a few times. Only before the last **rfi** the software needs to clear the exception cause register (ECR).

The above mechanism enables the software to accurately control the assertion and the negation of the freeze signal.

21.7 Development Support Registers

Table 21-14 lists the registers used for development support. The registers are accessed with the **mtspr** and **mfspr** instructions.

Table 21-14 Development Support Programming Model

SPR Number (Decimal)	Name
144	Comparator A Value Register (CMPA) See **Table 21-17** for bit descriptions.
145	Comparator B Value Register (CMPB) See **Table 21-17** for bit descriptions.
146	Comparator C Value Register (CMPC) See **Table 21-17** for bit descriptions.
147	Comparator D Value Register (CMPD) See **Table 21-17** for bit descriptions.
148	Exception Cause Register (ECR) See **Table 21-26** for bit descriptions.
149	Debug Enable Register (DER) See **Table 21-27** for bit descriptions.
150	Breakpoint Counter A Value and Control Register (COUNTA) See **Table 21-24** for bit descriptions.
151	Breakpoint Counter B Value and Control Register (COUNTB) See **Table 21-25** for bit descriptions.
152	Comparator E Value Register (CMPE) See **Table 21-18** for bit descriptions.
153	Comparator F Value Register (CMPF) See **Table 21-18** for bit descriptions.
154	Comparator G Value Register (CMPG) See **Table 21-20** for bit descriptions.
155	Comparator H Value Register (CMPH) See **Table 21-20** for bit descriptions.
156	L-Bus Support Control Register 1 (LCTRL1) See **Table 21-22** for bit descriptions.
157	L-Bus Support Control Register 2 (LCTRL2) See **Table 21-23** for bit descriptions.
158	I-Bus Support Control Register (ICTRL) See **Table 21-21** for bit descriptions.
159	Breakpoint Address Register (BAR) See **Table 21-19** for bit descriptions.
630	Development Port Data Register (DPDR) See **21.7.13 Development Port Data Register (DPDR)** for bit descriptions.

21.7.1 Register Protection

Table 21-15 and **Table 21-16** summarize protection features of development support registers during read and write accesses, respectively.

Table 21-15 Development Support Registers Read Access Protection

MSR[PR]	Debug Mode Enable	In Debug Mode	Result
0	0	X	Read is performed. ECR is cleared when read. Reading DPDR yields indeterminate data.
0	1	0	Read is performed. ECR is *not* cleared when read. Reading DPDR yields indeterminate data.
0	1	1	Read is performed. ECR is cleared when read.
1	X	X	Program exception is generated. Read is not performed. ECR is *not* cleared when read.

Table 21-16 Development Support Registers Write Access Protection

MSR[PR]	Debug Mode Enable	In Debug Mode	Result
0	0	X	Write is performed. Write to ECR is ignored. Writing to DPDR is ignored.
0	1	0	Write is *not* performed. Writing to DPDR is ignored.
0	1	1	Write is performed. Write to ECR is ignored.
1	X	X	Write is *not* performed. Program exception is generated.

21.7.2 Comparator A–D Value Registers (CMPA–CMPD)

CMPA–CMPD — Comparator A–D Value Register **SPR 144 – SPR 147**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CMPAD															

RESET: UNAFFECTED

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
CMPAD														RESERVED	

RESET: UNAFFECTED

Table 21-17 CMPA-CMPD Bit Settings

Bits	Mnemonic	Description
0:29	CMPAD	Address bits to be compared
30:31	—	Reserved

These registers are unaffected by reset.

21.7.3 Comparator E–F Value Registers

CMPE–CMPF — Comparator E–F Value Registers **SPR 152, 153**

```
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31
                                                        CMPEF
RESET: UNAFFECTED
```

Table 21-18 CMPE-CMPF Bit Settings

Bits	Mnemonic	Description
0:31	CMPV	Address bits to be compared

These registers are unaffected by reset.

21.7.4 Breakpoint Address Register (BAR)

BAR — Breakpoint Address Register **SPR 159**

```
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31
                                                        CMPEF
RESET: UNAFFECTED
```

Table 21-19 BAR Bit Settings

Bits	Mnemonic	Description
0:31	BARV[0:31]	The address of the load/store cycle that generated the breakpoint

21.7.5 Comparator G–H Value Registers (CMPG–CMPH)

CMPG–CMPH — Comparator G–H Value Registers **SPR 154, 155**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
CMPGH																															

RESET: UNAFFECTED

Table 21-20 CMPG-CMPH Bit Settings

Bits	Mnemonic	Description
0:31	CMPGH	Data bits to be compared

These registers are unaffected by reset.

21.7.6 I-Bus Support Control Register

ICTRL — I-Bus Support Control Register **SPR 158**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CTA			CTB			CTC			CTD			IWP0		IWP1	
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
IWP2		IWP3		SIWP0 EN	SIWP1 EN	SIWP2 EN	SIWP3 EN	DIWP0 EN	DIWP 1 EN	DIWP 2 EN	DIWP 3 EN	IIFM	ISCT_SER*		
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

*Changing the instruction show cycle programming starts to take effect only from the second instruction after the actual **mtspr** to ICTRL.

If the processor aborts a fetch of the target of a direct branch (due to an exception), the target is not always visible on the external pins. Program trace is not affected by this phenomenon.

Table 21-21 ICTRL Bit Settings

Bits	Mnemonic	Description	Function
0:2	CTA	Compare type of comparator A	0xx = not active (reset value) 100 = equal 101 = less than 110 = greater than 111 = not equal
3:5	CTB	Compare type of comparator B	
6:8	CTC	Compare type of comparator C	
9:11	CTD	Compare type of comparator D	
12:13	IWP0	I-bus 1st watchpoint programming	0x = not active (reset value) 10 = match from comparator A 11 = match from comparators (A&B)
14:15	W1	I-bus 2nd watchpoint programming	0x = not active (reset value) 10 = match from comparator B 11 = match from comparators (A | B)
16:17	IWP2	I-bus 3rd watchpoint programming	0x = not active (reset value) 10 = match from comparator C 11 = match from comparators (C&D)
18:19	IWP3	I-bus 4th watchpoint programming	0x = not active (reset value) 10 = match from comparator D 11 = match from comparators (C | D)
20	SIWP0EN	Software trap enable selection of the 1st I-bus watchpoint	0 = trap disabled (reset value) 1 = trap enabled
21	SIWP1EN	Software trap enable selection of the 2nd I-bus watchpoint	
22	SIWP2EN	Software trap enable selection of the 3rd I-bus watchpoint	
23	SIWP3EN	Software trap enable selection of the 4th I-bus watchpoint	
24	DIWP0EN	Development port trap enable selection of the 1st I-bus watchpoint (read only bit)	0 = trap disabled (reset value) 1 = trap enabled
25	DIWP1EN	Development port trap enable selection of the 2nd I-bus watchpoint (read only bit)	
26	DIWP2EN	Development port trap enable selection of the 3rd I-bus watchpoint (read only bit)	
27	DIWP3EN	Development port trap enable selection of the 4th I-bus watchpoint (read only bit)	
28	IIFM	Ignore first match, only for I-bus breakpoints	0 = Do not ignore first match, used for “go to x” (reset value) 1 = Ignore first match (used for “continue”)

Table 21-21 ICTRL Bit Settings (Continued)

Bits	Mnemonic	Description	Function
29:31	ISCT_SER	Instruction fetch show cycle and RCPU serialize control	000 = RCPU is fully serialized and show cycle will be performed for all fetched instructions (reset value) 001 = RCPU is fully serialized and show cycle will be performed for all changes in the program flow 010 = RCPU is fully serialized and show cycle will be performed for all indirect changes in the program flow 011 = RCPU is fully serialized and no show cycles will be performed for fetched instructions 100 = Illegal 101 = RCPU is not serialized (normal mode) and show cycle will be performed for all changes in the program flow 110 = RCPU is not serialized (normal mode) and show cycle will be performed for all indirect changes in the program flow 111 = RCPU is not serialized (normal mode) and no show cycles will be performed for fetched instructions **NOTE**: Changing the instruction show cycle programming starts to take effect only from the second instruction after the actual **mtspr** to ICTRL.

21.7.7 L-Bus Support Control Register 1

LCTRL1 — L-Bus Support Control Register 1 **SPR 156**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CTE			CTF			CTG			CTH			CRWE		CRWF	
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
CSG		CSH		SUSG	SUSH	CGBMSK				CHBMSK				UNUSED	
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 21-22 LCTRL1 Bit Settings

Bits	Mnemonic	Description	Function
0:2	CTE	Compare type, comparator E	0xx = not active (reset value) 100 = equal 101 = less than 110 = greater than 111 = not equal
3:5	CTF	Compare type, comparator F	
6:8	CTG	Compare type, comparator G	
9:11	CTH	Compare type, comparator H	
12:13	CRWE	Select match on read/write of comparator E	0X = don't care (reset value) 10 = match on read 11 = match on write
14:15	CRWF	Select match on read/write of comparator F	
16:17	CSG	Compare size, comparator G	00 = reserved 01 = word 10 = half word 11 = byte (Must be programmed to word for floating point compares)
18:19	CSH	Compare size, comparator H	
20	SUSG	Signed/unsigned operating mode for comparator G	0 = unsigned 1 = signed (Must be programmed to signed for floating point compares)
21	SUSH	Signed/unsigned operating mode for comparator H	
22:25	CGBMSK	Byte mask for 1st L-data comparator	0000 = all bytes are **not** masked 0001 = the last byte of the word is masked . . . 1111 = all bytes are masked
26:29	CHBMSK	Byte mask for 2nd L-data comparator	
30:31	—	Reserved	—

LCTRL1 is cleared following reset.

21.7.8 L-Bus Support Control Register 2

LCTRL2 — L-Bus Support Control Register 2 **SPR 157**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LW0EN	LW0IA		LW0 IADC	LW0LA		LW0 LADC	LW0LD		LW0 LDDC	LW1EN	LW1IA		LW1 IADC	LW1LA	
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
LW1 LADC	LW1LD		LW1 LDDC	BRK NOMSK	RESERVED							DLW0 EN	DLW1 EN	SLW0 EN	SLW1 EN
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 21-23 LCTRL2 Bit Settings

Bits	Mnemonic	Description	Function
0	LW0EN	1st L-bus watchpoint enable bit	0 = watchpoint not enabled (reset value) 1 = watchpoint enabled
1:2	LW0IA	1st L-bus watchpoint I-addr watchpoint selection	00 = first I-bus watchpoint 01 = second I-bus watchpoint 10 = third I-bus watchpoint 11 = fourth I-bus watchpoint
3	LW0IADC	1st L-bus watchpoint care/don't care I-addr events	0 = Don't care 1 = Care
4:5	LW0LA	1st L-bus watchpoint L-addr events selection	00 = match from comparator E 01 = match from comparator F 10 = match from comparators (E&F) 11 = match from comparators (E | F)
6	LW0LADC	1st L-bus watchpoint care/don't care L-addr events	0 = Don't care 1 = Care
7:8	LW0LD	1st L-bus watchpoint L-data events selection	00 = match from comparator G 01 = match from comparator H 10 = match from comparators (G&H) 11 = match from comparators (G | H)
9	LW0LDDC	1st L-bus watchpoint care/don't care L-data events	0 = Don't care 1 = Care
10	LW1EN	2nd L-bus watchpoint enable bit	0 = watchpoint not enabled (reset value) 1 = watchpoint enabled
11:12	LW1IA	2nd L-bus watchpoint I-addr watchpoint selection	00 = first I-bus watchpoint 01 = second I-bus watchpoint 10 = third I-bus watchpoint 11 = fourth I-bus watchpoint
13	LW1IADC	2nd L-bus watchpoint care/don't care I-addr events	0 = Don't care 1 = Care
14:15	LW1LA	2nd L-bus watchpoint L-addr events selection	00 = match from comparator E 01 = match from comparator F 10 = match from comparators (E&F) 11 = match from comparators (E | F)
16	LW1LADC	2nd L-bus watchpoint care/don't care L-addr events	0 = Don't care 1 = Care
17:18	LW1LD	2nd L-bus watchpoint L-data events selection	00 = match from comparator G 01 = match from comparator H 10 = match from comparators (G&H) 11 = match from comparator (G | H)
19	LW1LDDC	2nd L-bus watchpoint care/don't care L-data events	0 = Don't care 1 = Care
20	BRKNOMSK	Internal breakpoints non-mask bit	0 = masked mode; breakpoints are recognized only when MSR[RI]=1 (reset value) 1 = non-masked mode; breakpoints are always recognized
21:27	—	Reserved	—

Table 21-23 LCTRL2 Bit Settings (Continued)

Bits	Mnemonic	Description	Function
28	DLW0EN	Development port trap enable selection of the 1st L-bus watchpoint (read only bit)	0 = trap disabled (reset value) 1 = trap enabled
29	DLW1EN	Development port trap enable selection of the 2nd L-bus watchpoint (read only bit)	
30	SLW0EN	Software trap enable selection of the 1st L-bus watchpoint	
31	SLW1EN	Software trap enable selection of the 2nd L-bus watchpoint	

LCTRL2 is cleared following reset.

For each watchpoint, three control register fields (LWxIA, LWxLA, LWxLD) must be programmed. For a watchpoint to be asserted, all three conditions must be detected.

21.7.9 Breakpoint Counter A Value and Control Register

COUNTA — Breakpoint Counter A Value and Control Register **SPR 150**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CNTV															

RESET: UNAFFECTED

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESERVED														CNTC	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

RESET:

Table 21-24 Breakpoint Counter A Value and Control Register (COUNTA)

Bit(s)	Name	Description
0:15	CNTV	Counter preset value
16:29	—	Reserved
30:31	CNTC	Counter source select 00 = not active (reset value) 01 = I-bus first watchpoint 10 =L-bus first watchpoint 11 = Reserved

COUNTA[16:31] are cleared following reset; COUNTA[0:15] are unaffected by reset.

21.7.10 Breakpoint Counter B Value and Control Register

COUNTB — Breakpoint Counter B Value and Control Register **SPR 151**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CNTV															

RESET: UNAFFECTED

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESERVED														CNTC	

RESET:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 21-25 Breakpoint Counter B Value and Control Register (COUNTB)

Bit(s)	Name	Description
0:15	CNTV	Counter preset value
16:29	—	Reserved
30:31	CNTC	Counter source select 00 = not active (reset value) 01 = I-bus second watchpoint 10 = L-bus second watchpoint 11 = Reserved

COUNTB[16:31] are cleared following reset; COUNTB[0:15] are unaffected by reset.

21.7.11 Exception Cause Register (ECR)

The ECR indicates the cause of entry into debug mode. All bits are set by the hardware and cleared when the register is read when debug mode is disabled, or if the processor is in debug mode. Attempts to write to this register are ignored. When the hardware sets a bit in this register, debug mode is entered only if debug mode is enabled and the corresponding mask bit in the DER is set.

All bits are cleared to zero following reset.

ECR — Exception Cause Register **SPR 148**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	RST	CHSTP	MCE	RESERVED		EXTI	ALE	PRE	FPUVE	DECE	RESERVED		SYSE	TR	FPASE
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	SEE	RESERVED	ITLBER	RESERVED	DTLBER	RESERVED						LBRK	IBRK	EBRKD	DPI
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 21-26 ECR Bit Settings

Bit(s)	Name	Description
0	—	Reserved
1	RST	Reset interrupt bit. This bit is set when the system reset pin is asserted.
2	CHSTP	Checkstop bit. Set when the processor enters checkstop state.
3	MCE	Machine check interrupt bit. Set when a machine check exception (other than one caused by a data storage or instruction storage error) is asserted.
4:5	—	Reserved
6	EXTI	External interrupt bit. Set when the external interrupt is asserted.
7	ALE	Alignment exception bit. Set when the alignment exception is asserted.
8	PRE	Program exception bit. Set when the program exception is asserted.
9	FPUVE	Floating point unavailable exception bit. Set when the program exception is asserted.
10	DECE	Decrementer exception bit. Set when the decrementer exception is asserted.
11:12	—	Reserved
13	SYSE	System call exception bit. Set when the system call exception is asserted.
14	TR	Trace exception bit. Set when in single-step mode or when in branch trace mode.
15	FPASE	Floating point assist exception bit. Set when the floating point assist exception occurs.
16	—	Reserved
17	SEE	Software emulation exception. Set when the software emulation exception is asserted.
18	—	Reserved
19	ITLBER	Implementation specific instruction protection error This bit is set as a result of an instruction protection error. Results in debug mode entry if debug mode is enabled and the corresponding enable bit is set.
20	—	Reserved
21	DTLBER	Implementation specific data protection error This bit is set as a result of an data protection error. Results in debug mode entry if debug mode is enabled and the corresponding enable bit is set.

Table 21-26 ECR Bit Settings (Continued)

Bit(s)	Name	Description
22:27	—	Reserved
28	LBRK	L-bus breakpoint exception bit. This bit is set as a result of the assertion of a load/store breakpoint. Results in debug mode entry if debug mode is enabled and the corresponding enable bit is set.
29	IBRK	I-bus breakpoint exception bit. This bit is set as a result of the assertion of an Instruction breakpoint. Results in debug mode entry if debug mode is enabled and the corresponding enable bit is set.
30	EBRK	External breakpoint exception bit. Set when an external breakpoint is asserted (by an on-chip IMB or L-bus module, or by an external device or development system through the development port). This bit is set as a result of the assertion of an external breakpoint. Results in debug mode entry if debug mode is enabled and the corresponding enable bit is set.
31	DPI	Development port interrupt bit. Set by the development port as a result of a debug station non-maskable request or when debug mode is entered immediately out of reset.

21.7.12 Debug Enable Register (DER)

This register enables the user to selectively mask the events that may cause the processor to enter into debug mode.

DER — Debug Enable Register **SPR 149**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	RSTE	CHSTPE	MCEE	RESERVED		EXTIE	ALEE	PREE	FPUVEE	DECEE	RESERVED		SYSEE	TRE	FPASE
RESET:															
0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESERVED	SEEE	RESERVED	ITLBERE	RESERVED	DTLBERE	RESERVED						LBRKE	IBRKE	EBRKE	DPIE
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Table 21-27 DER Bit Settings

Bit(s)	Name	Description
0:1	—	Reserved
1	RSTE	Reset enable 0 = Debug entry is disabled (reset value) 1 = Debug entry is enabled
2	CHSTPE	Checkstop enable bit 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
3	MCEE	Machine check exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
4:5	—	Reserved

Table 21-27 DER Bit Settings (Continued)

Bit(s)	Name	Description
6	EXTIE	External interrupt enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
7	ALEE	Alignment exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
8	PREE	Program exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
9	FPUVEE	Floating point unavailable exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
10	DECEE	Decrementer exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
11:12	—	Reserved
13	SYSEE	System call exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
14	TRE	Trace exception enable bit 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
15	FPASEE	Floating point assist exception enable bit. 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
16	—	Reserved
17	SEEE	Software emulation exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
18	—	Reserved
19	ITLBERE	Implementation specific instruction protection error enable bit. 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
20	—	Reserved
21	DTLBERE	Implementation specific data protection error enable bit. 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
22:27	—	Reserved
28	LBRKE	Load/store breakpoint enable bit. 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
29	IBRKE	Instruction breakpoint interrupt enable bit. 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
30	EBRKE	External breakpoint interrupt enable bit (development port, internal or external modules). 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
31	DPIE	Development port interrupt enable bit 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)

21.7.13 Development Port Data Register (DPDR)

This 32-bit special purpose register physically resides in the development port logic. It is used for data interchange between the core and the development system. An access to this register is initiated using **mtspr** and **mfspr** (SPR 630) and implemented using a special bus cycle on the internal bus.