SECTION 8
DEVELOPMENT SUPPORT

Development tools are used by a microcomputer system developer to debug the hardware and software of a target system. These tools are used to give the developer some control over the execution of the target program. In-circuit emulators and bus state analyzers are the most frequently used debugging tools. In order for these tools to function properly, they must have full visibility of the microprocessor's buses.

Visibility extends beyond the address and data portions of the buses and includes attribute and handshake signals. In some cases it may also include bus arbitration signals and signals which cause processor exceptions such as interrupts and resets. The visibility requirements of emulators and bus analyzers are in opposition to the trend of modern microcomputers and microprocessors where the CPU bus may be hidden behind a memory management unit or cache or where bus cycles to internal resources are not visible externally.

The development tool visibility requirements may be reduced if some of the development support functions are included in the silicon. For example, if the bus comparator part of a bus analyzer or breakpoint generator is included on the chip, it is not necessary for the entire bus to be visible at all times. In many cases the visibility requirements may be reduced to instruction fetch cycles for tracking program execution. If some additional status information is also available to assist in execution tracking and the development tool has access to the source code, then the only need for bus visibility is often the destination address of indirect change-of-flow instructions (return from subroutine, return from interrupt, and indexed branches and jumps).

Since full bus visibility reduces available bus bandwidth and processor performance, certain development support functions have been included in the MCU. These functions include the following:

- Controls to limit which internal bus cycles are reflected on the external bus (show cycles)
- CPU status signals to allow instruction execution tracking with minimal visibility of the instructions being fetched
- Watchpoint comparators that can generate breakpoints or signal an external bus analyzer
- A serial development port for general emulation control

8.1 Program Flow Tracking

The exact program flow is visible on the external bus only when the processor is programmed to show all fetch cycles on the external bus. This mode is selected by programming the ISCTL (instruction fetch show cycle control) field in the I-bus support control register (ICTRL), as shown in **Table 8-2**. 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.

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

The RCPU implements a prefetch queue combined with parallel, out of order, pipelined execution. Instructions progress inside the processor from fetch to retire. An instruction retires from the machine only after it, and all preceding instructions, finish execution with no exception. Therefore only retired instructions can be considered architecturally executed.

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** and **mtmsr** because they may cause a context switch.
- The number of instructions canceled each clock.

Reporting on program trace during retirement would significantly complicate the visibility support and increase the die size. (Complications arise because more than one instruction can retire in a clock cycle, and because it is harder to report on indirect branches during retirement.) Therefore, program trace is reported during fetch. Since not all fetched instructions eventually retire, an indication on canceled instructions is reported.

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.

8.1.1 Indirect Change-of-Flow Cycles

An *indirect change-of-flow* attribute is attached to all fetch cycles that result from indirect flow changes. Indirect flow changes include the following types of instructions or events:

- Assertion or negation of VSYNC
- Exception taken
- Indirect branch taken
- Execution of the following sequential instructions: **rfi**, **isync**, **mtmsr**, and **mtspr** to CMPA–CMPF, ICTRL, ECR, and DER

When a program trace recording is needed, the user can ensure that cycles which result from an indirect change-of-flow are visible on the external bus. The user can do this in one of two ways: by setting the VSYNC bit, or by programming the ISCTL bits in the I-bus support control register. Refer to **8.1.2 Instruction Fetch Show Cycle Control** for more information.

When the processor is programmed to generate show cycles on the external bus resulting from indirect change-of-flow, these cycles can generate regular bus cycles (address phase and data phase) when the instructions reside in one of the external devices, or they can generate address-only show cycles for instructions that reside in an internal device such as I-cache or internal ROM.

8.1.1.1 Marking the Indirect Change-of-Flow Attribute

When an instruction fetch cycle that results from an indirect change-of-flow is an internal access (e.g., access to an internal memory location, or a cache hit during an access to an external memory address), the indirect change-of-flow attribute is indicated by the assertion (low) of the $\overline{WR}$ pin during the external bus show cycle.

When an instruction fetch cycle that results from an indirect change-of-flow is an access to external memory not resulting in a cache hit, the indirect change-of-flow attribute is indicated by the value 0001 on the CT[0:3] pins.

Table 8-1 summarizes the encodings that represent the indirect change-of-flow attribute. In all cases the AT1 pin is asserted (high), indicating the cycle is an instruction fetch cycle.

Table 8-1 Program Trace Cycle Attribute Encodings

CT[0:3]	AT1	$\overline{WR}$	Type of Bus Cycle
0001	1	1	External bus cycle
01xx, 10xx, 110x	1	0	Show cycle on the external bus reflecting an access to internal register or memory or a cache hit

Refer to **8.1.3 Program Flow-Tracking Pins** for more information on the use of these pins for program flow tracking.

8.1.1.2 Sequential Instructions with the Indirect Change-of-Flow Attribute

Because certain sequential instructions (**rfi**, **isync**, **mtmsr**, and **mtspr** to CMPA–CMPF, ICTRL, ECR, and DER) affect the machine in a manner similar to indirect branch instructions, the processor marks these instructions as indirect branch instruc-

tions (VF = 101, see **Table 8-3**) and marks the subsequent instruction address with the indirect change-of-flow attribute, as if it were an indirect branch target. Therefore, when the processor detects one of these instructions, the address of the following instruction is visible externally. This enables the reconstructing software to correctly evaluate the effect of these instructions.

8.1.2 Instruction Fetch Show Cycle Control

Instruction fetch show cycles are controlled by the bits in the ICTRL and the state of VSYNC, as illustrated in **Table 8-2**.

Table 8-2 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

When the value of the ISCTL field is changed (with the **mtspr** instruction), the new value does not take effect until two instructions after the **mtspr** instruction. The instruction immediately following **mtspr** is under control of the old ISCTL value.

In order to keep the pin count of the chip as low as possible, VSYNC is not implemented as an external pin; rather, it is asserted and negated using the development port serial interface. For more information on this interface refer to **8.3.5 Trap-Enable Input Transmissions**.

The assertion and negation of VSYNC forces the machine to synchronize and the first fetch after this synchronization to be marked as an indirect change-of-flow cycle and to be visible on the external bus. This enables the external hardware to synchronize with the internal activity of the processor.

When either VSYNC is asserted or the ISCTL bits in the I-bus control register are programmed to a value of 0b10, cycles resulting from an indirect change-of-flow are shown on the external bus. By programming the ISCTL bits to show all indirect flow changes, the user can thus ensure that the processor maintains exactly the same behavior when VSYNC is asserted as when it is negated. The loss of performance the user can expect from the additional external bus cycles is minimal.

For additional information on the ISCTL bits and the ICTRL register, refer to **8.8 Development Support Registers**. For more information on the use of VSYNC during program trace, refer to **8.1.4 External Hardware During Program Trace**.

8.1.3 Program Flow-Tracking Pins

The following sets of pins are used in program flow tracking:

- Instruction queue status pins (VF[0:2]) denote the type of the last fetched instruction or how many instructions were flushed from the instruction queue.
- History buffer flushes status pins (VFLS [0:1]) denote how many instructions were flushed from the history buffer during the current clock cycle.
- Address type pin 1 (AT1) indicates whether the cycle is transferring an instruction or data.
- The write/read pin ($\overline{WR}$), when asserted during an instruction fetch show cycle, indicates the current cycle results from an indirect change-of-flow.
- Cycle type pins (CT[0:3]) indicate the type of bus cycle and are used to determine the address of an internal memory or register that is being accessed.

8.1.3.1 Instruction Queue Status Pins

Instruction queue status pins VF[0:2] indicate 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 occur only during clock cycles in which there is no fetch type information to be reported.

Table 8-3 shows the possible instruction types.

Table 8-3 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 program trace cycle 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 **8.1.1.2 Sequential Instructions with the Indirect Change-of-Flow Attribute**.

Table 8-4 shows VF[0:2] encodings for instruction queue flush information.

Table 8-4 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 8-3**.

There is one special case in which although queue flush information is expected on the VF[0:2] pins (according to the immediately preceding value on these pins), regular instruction type information is reported. The only instruction type information that can appear in this case is VF[0:2] = 111, indicating branch (direct or indirect) not taken. Since the maximum queue flushes possible is five, identifying this special case is not a problem.

8.1.3.2 History Buffer Flush Status Pins

History buffer flush status pins VFLS[0:1] indicate how many instructions are flushed from the history buffer this clock. **Table 8-4** shows VFLS encodings.

Table 8-5 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.

8.1.3.3 Flow-Tracking Status Pins in Debug Mode

When the processor is in debug mode, the VF[0:2] signals are low (000) and the VFLS[0:1] signals are high (11).

If VSYNC is asserted or negated while the processor is in debug mode, this information is reported as the first VF pins report when the processor returns to regular mode. If VSYNC is not changed while the processor is in debug mode, the first VF pins report

is of an indirect branch taken (VF[0:2] = 101), appropriate 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.

8.1.3.4 Cycle Type, Write/Read, and Address Type Pins

Cycle type pins (CT[0:3]) indicate the type of bus cycle being performed. During show cycles, these pins are used to determine the internal address being accessed. **Table 8-6** summarizes cycle type encodings.

Table 8-6 Cycle Type Encodings

CT[0:3]	Description
0000	Normal external bus cycle
0001	If address type is data (AT1 = 0), this is a data access to the external bus and the start of a reservation. If address type is instruction (AT1 = 1), this cycle type indicates that an external address is the destination of an indirect change-of-flow.
0010	External bus cycle to emulation memory replacing internal I-bus or L-bus memory. An instruction access (AT1 = 1) with an address that is the target of an indirect change-of-flow is indicated as a logic level zero on the $\overline{WR}$ output.
0011	Normal external bus cycle access to a port replacement chip used for emulation support.
0100	Access to internal I-bus memory. An instruction access (AT1 = 1) with an address that is the target of an indirect change-of-flow is indicated as a logic level zero on the $\overline{WR}$ output.
0101	Access to internal L-bus memory. An instruction access (AT1 = 1) with an address that is the target of an indirect change-of-flow is indicated as a logic level zero on the $\overline{WR}$ output.
0110	Cache hit on external memory address not controlled by chip selects. An instruction access (AT1 = 1) with an address that is the target of an indirect change-of-flow is indicated as a logic level zero on the $\overline{WR}$ output.
0111	Access to an internal register.
1000 1001 1010 1011 1100 1101	Cache hit on external memory address controlled by $\overline{CSBOOT}$. Cache hit on external memory address controlled by $\overline{CS1}$. Cache hit on external memory address controlled by $\overline{CS2}$. Cache hit on external memory address controlled by $\overline{CS3}$. Cache hit on external memory address controlled by $\overline{CS4}$. Cache hit on external memory address controlled by $\overline{CS5}$. An instruction access (AT1 = 1) with an address that is the target of an indirect change-of-flow is indicated as a logic level zero on the $\overline{WR}$ output.
1110	Reserved
1111	

Notice in **Table 8-6** that during an instruction fetch (AT1 = 1) to internal memory or to external memory resulting in a cache hit, a logic level of zero on the $\overline{WR}$ pin indicates that the cycle is the result of an indirect change-of-flow. The indirect change-of-flow attribute is also indicated by a cycle type encoding of 0001 when AT1 = 1. Refer to **8.1.1.1 Marking the Indirect Change-of-Flow Attribute** for additional information.

8.1.4 External Hardware During Program Trace

When program trace is needed, external hardware needs to record the status pins (VF[0:2] and VFLS[0:1]) of each clock and record the address of all cycles marked with the indirect change-of-flow attribute.

Program trace can be used in various ways. Two types of traces that can be implemented are the back trace and the window trace.

8.1.4.1 Back Trace

A back trace provides a record of the program trace *before* some event occurred. An example of such an event is some system failure.

When a back trace is needed, the external hardware should start sampling the status pins and the address of all cycles marked with the indirect change-of-flow attribute immediately after reset is negated. Since the ISCTL field in the ICTRL has a value of is 0b00 (show all cycles) out of reset, all cycles marked with the indirect change-of-flow attribute are visible on the external bus. VSYNC should be asserted sometime after reset and negated when the programmed event occurs. VSYNC must be asserted before the ISCTL encoding is changed to 0b11 (no show cycles), if such an encoding is selected.

NOTE

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.

8.1.4.2 Window Trace

Window trace provides a record of the program trace *between* two events. VSYNC should be asserted between these two events.

After VSYNC is negated, the trace buffer will contain information describing the program trace of the program executed between the two events.

8.1.4.3 Synchronizing the Trace Window to Internal CPU Events

In order to synchronize the assertion or negation of VSYNC to an event internal to the processor, internal breakpoints can be used together with debug mode. This method is available only when debug mode is enabled. (Refer to **8.4 Debug Mode Functions**.)

The following steps enable the user to synchronize the trace window to events internal to the processor:

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 **8.8 Development Support Registers**.
3. Enable debug mode entry for the programmed breakpoint in the debug enable register (DER).
4. Return to the regular code run.
5. The hardware generates a breakpoint when the programmed event is detected, and the machine enters 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[0:2] = 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. The first report on the VF pins is a VSYNC (VF[0:2] = 011).
13. The external hardware stops sampling the program trace information upon the report on the VF pins of VSYNC.

A second method allows the trace window to be synchronized to internal processor events without stopping execution and entering debug mode at the two events.

1. Enter debug mode, either immediately out of reset or using the debug mode request.
2. Program a watchpoint for the event that marks the start of the trace window using the control registers defined in **8.8 Development Support Registers**.
3. Program a second watchpoint for the event that marks the end of the trace window.
4. Return to regular code execution by exiting debug mode.
5. The watchpoint logic signals the starting event by asserting the appropriate watchpoint pin.
6. Upon detecting the first watchpoint, assert VSYNC using the development port serial interface.
7. The external program trace hardware starts sampling the program trace infor-

mation upon the report on the VF pins of VSYNC.

8. The watchpoint logic signals the ending event by asserting the appropriate watchpoint pin.
9. Upon detecting the second watchpoint, negate VSYNC using the development port serial interface.
10. The external program trace hardware stops sampling the program trace information upon the report on VF[0:1] of VSYNC.

The second method is not as precise as the first method because of the delay between the assertion of the watchpoint pins and the assertion or negation of VSYNC using the development port serial interface. It has the advantage, however, of allowing the program to run in quasi-real time (slowed only by show cycles on the external bus), instead of stopping execution at the starting and ending events.

8.1.4.4 Detecting the Trace Window Starting Address

For a back trace, the value of the status pins (VF[0:2] and VFLS[0:1]) and the address of the cycles marked with the indirect change-of-flow attribute should be latched starting immediately after the negation of reset. The starting address is the first address in the program trace cycle buffer.

For a window trace, the value of the status pins and the address of the cycles marked with the indirect change-of-flow attribute should be latched beginning immediately after the first VSYNC is reported on the VF pins. The starting address of the trace window should be calculated according to the first two VF pin reports.

Assume VF1 and VF2 are the two first VF pin reports and T1 and T2 are the addresses of the first two cycles marked with the indirect change-of-flow attribute that were latched in the trace buffer. Use **Table 8-7** to calculate the trace window starting address.

Table 8-7 Detecting the Trace Buffer Starting Point

VF1	VF2	Starting Point	Description
011 VSYNC	001 Sequential	T1	VSYNC asserted followed by a sequential instruction. The starting address is T1.
011 VSYNC	110 Branch direct taken	T1 – 4 + offset (T1 – 4)	VSYNC asserted followed by a taken direct branch. The starting address is the target of the direct branch.
011 VSYNC	101 Branch indirect taken	T2	VSYNC asserted followed by a taken indirect branch. The starting address is the target of the indirect branch.

8.1.4.5 Detecting the Assertion or Negation of VSYNC

Since the VF pins are used for reporting both instruction type information and queue flush information, special care must be taken when trying to detect the assertion or

negation of VSYNC. A VF[0:2] encoding of 011 indicates the assertion or negation of VSYNC only if the previous VF[0:2] pin values were 000, 001, or 010.

8.1.4.6 Detecting the Trace Window Ending Address

The information on the VF and VFLS status pins changes every clock. Cycles marked with the indirect change-of-flow 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 time it is 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 the user negates VSYNC, the processor delays the report of the assertion or negation of VSYNC on the VF pins until all addresses marked with the indirect change-of-flow attribute have been made 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 with the program trace cycle attribute immediately after the VSYNC report on the VF pins.

CAUTION

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.

8.1.5 Compress

In order to store all the information generated on the pins during program trace (5 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.

8.2 Watchpoint and Breakpoint Support

The RCPU provides the ability to detect specific bus cycles, as defined by a user (watchpoints). It also provides the ability to conditionally respond to these watchpoints by taking an exception (internal breakpoints). Breakpoints can also be caused by an event or state in a peripheral or through the development port (external breakpoints, i.e., breakpoints external to the processor).

When a watchpoint is detected, it is reported to external hardware on dedicated pins. Watchpoints do not change the timing or flow of the processor. Because bus cycles on the internal MCU buses are not necessarily visible on the external bus, the watchpoints are a convenient way to signal an external instrument (such as a bus state analyzer or oscilloscope) that the internal bus cycle occurred.

An internal breakpoint occurs when a particular watchpoint is enabled to generate a breakpoint. A watchpoint may be enabled to generate a breakpoint from a software

monitor or by using the development port serial interface. A watchpoint output may also be counted. When the counter reaches zero, an internal breakpoint is generated.

An external breakpoint occurs when a development system or external peripheral requests a breakpoint through the development port serial interface. In addition, if an on-chip peripheral requests a breakpoint, an external breakpoint is generated.

All internal breakpoints are masked by the MSR[RI] bit unless the non-masked control bit (BRKNOMSK) in LCTRL2 is set. The development port maskable breakpoint and breakpoints from internal peripherals are masked by the MSR[RI] bit. The development port non-maskable breakpoint is *not* masked by this bit.

Figure 8-1 is a diagram of watchpoint and breakpoint support in the RCPU.

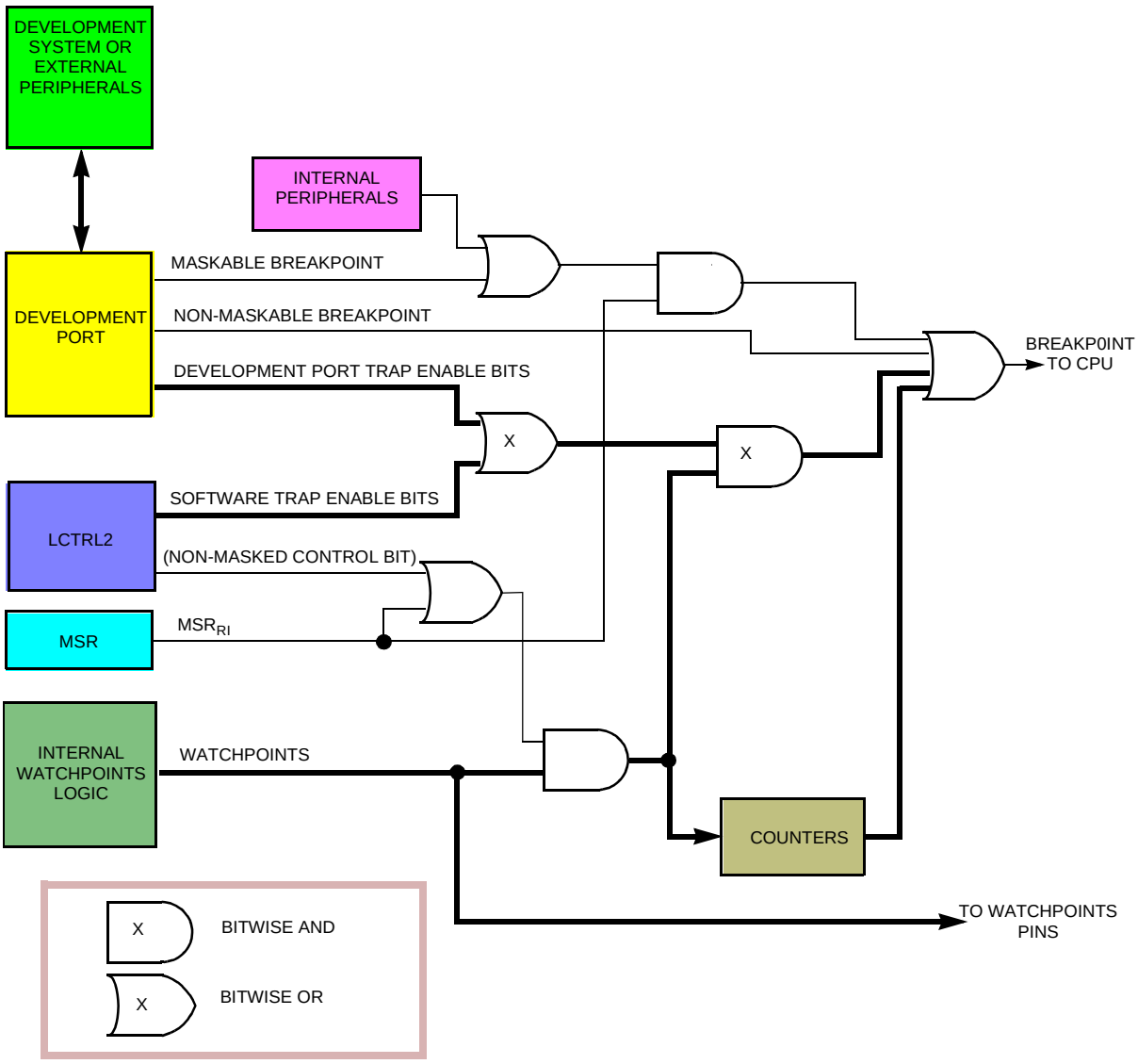


Figure 8-1 Watchpoint and Breakpoint Support in the RCPU

8.2.1 Watchpoints

Watchpoints are based on eight comparators on the I-bus and L-bus, two counters, and two AND-OR logic structures. There are four comparators on the instruction address bus (I-address), two comparators on the load/store address bus (L-address), and two comparators on the load/store data bus (L-data).

The comparators are able to detect the following conditions: equal, not equal, greater than, and less than. Greater than or equal and less than or equal are easily obtained from these four conditions. (For more information refer to **8.2.1.3 Generating Six Compare Types**.) Using the AND-OR logic structures, in range and out of range detection (on address and on data) are supported. Using the counters, it is possible to program a breakpoint to be generated after an event is detected a predefined number of times.

The L-data comparators can operate on integer data, floating-point single-precision data, and the integer value stored using the **stfiwx** instruction. Integer comparisons can be performed on bytes, half words, and words. The operands can be treated as signed or unsigned values.

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

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

L-bus watchpoints can be qualified by I-bus watchpoints. If qualified, the L-bus watchpoint occurs only if the L-bus cycle was the result of executing an instruction that caused the qualifying I-bus watchpoint.

A watchpoint progresses in the machine along with the instruction that caused it (fetch or load/store cycle). Watchpoints are reported on the external pins when the associated instruction is retired.

8.2.1.1 Restrictions on Watchpoint Detection

There are cases when the same watchpoint can be detected more than once during the execution of a single instruction. For example, the processor may detect an L-bus watchpoint on more than one transfer when executing a load/store multiple or string instruction or may detect an L-bus watchpoint on more than one byte when working in byte mode. In 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, separate watchpoint 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 is reported only once. However, the internal counters still count correctly.

8.2.1.2 Byte and Half-Word Working Modes

Watchpoint and breakpoint support enables the user to detect matches on bytes and half words even when accessed using a load/store instruction of larger data widths, e.g. when loading a table of bytes using a series of load word instructions.

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, the user cannot predict the exact value of the L-address lines when the requested byte or half word is accessed. For example, 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 processor 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.

The following examples illustrate how to detect matches on bytes and half words.

1. A fully supported scenario:
   - Looking for:
     - — Data size: byte
     - — Address: 0x0000 0003
     - — Data value: greater than 0x07 and less than 0x0C
   - Programming option:
     - — One L-address comparator = 0x0000 0003 and program for equal
     - — One L-data comparator = 0xXXXX XXX7 and program for greater than
     - — One L-data comparator = 0xXXXX XXXC and program for less than
     - — Both byte masks = 0b0001
     - — Both L-data comparators program to byte mode
   - Result: The event will be detected regardless of the instruction the compiler chooses for this access.
2. A fully supported scenario:
   - Looking for:
     - — Data size: half word
     - — Address: greater than 0x0000 0000 and less than 0x0000 000C
     - — Data value: greater than 0x4E20 and less than 0x9C40
   - Programming option:
     - — One L-address comparator = 0x0000 0000 and program for greater than

— One L-address comparator = 0x0000 000C and program for less than
— One L-data comparator = 0x4E20 4E20 and program for greater than
— One L-data comparator = 0x9C40 9C40 and program for less than
— Both byte masks = 0b1111
— Both L-data comparators program to half word mode

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

3. A partially supported scenario:

• Looking for:
— Data size: half word
— Address: greater than 0x0000 0002 and less than 0x0000 000E
— Data value: greater than 0x4E20 and less than 0x9C40

• Programming option:
— One L-address comparator = 0x0000 0002 and program for greater than
— One L-address comparator = 0x0000 000E and program for less than
— One L-data comparator = 0x4E20 4E20 and program for greater than
— One L-data comparator = 0x9C40 9C40 and program for less than
— Both byte masks = 0b1111
— Both L-data comparators program to half-word mode or to word mode

4. Result: The event will be detected correctly 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. **Figure 8-2** illustrates this partially supported scenario.

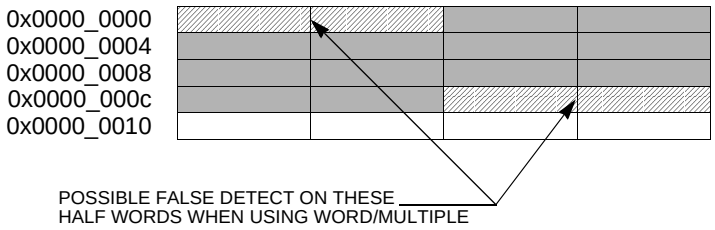


Figure 8-2 Partially Supported Watchpoint/Breakpoint Example

8.2.1.3 Generating Six Compare Types

Using the four basic compare types (equal, not equal, greater than, less than), it is possible to generate two additional compare types: “greater than or equal” and “less than or equal”.

The “greater than or equal” compare type can be generated using the greater than compare type and programming the comparator to the needed value minus one.

The “less than or equal” compare type can be generated using the less than compare type and programming the comparator to the needed value plus one.

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 L-bus watchpoint programming (refer to **8.8 Development Support Registers**).

8.2.1.4 I-Bus Support Detailed Description

There are four I-bus address comparators (comparators A, B, C, D). Each is 30 bits long and generates 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. **Figure 8-3** shows the general structure of I-bus support.

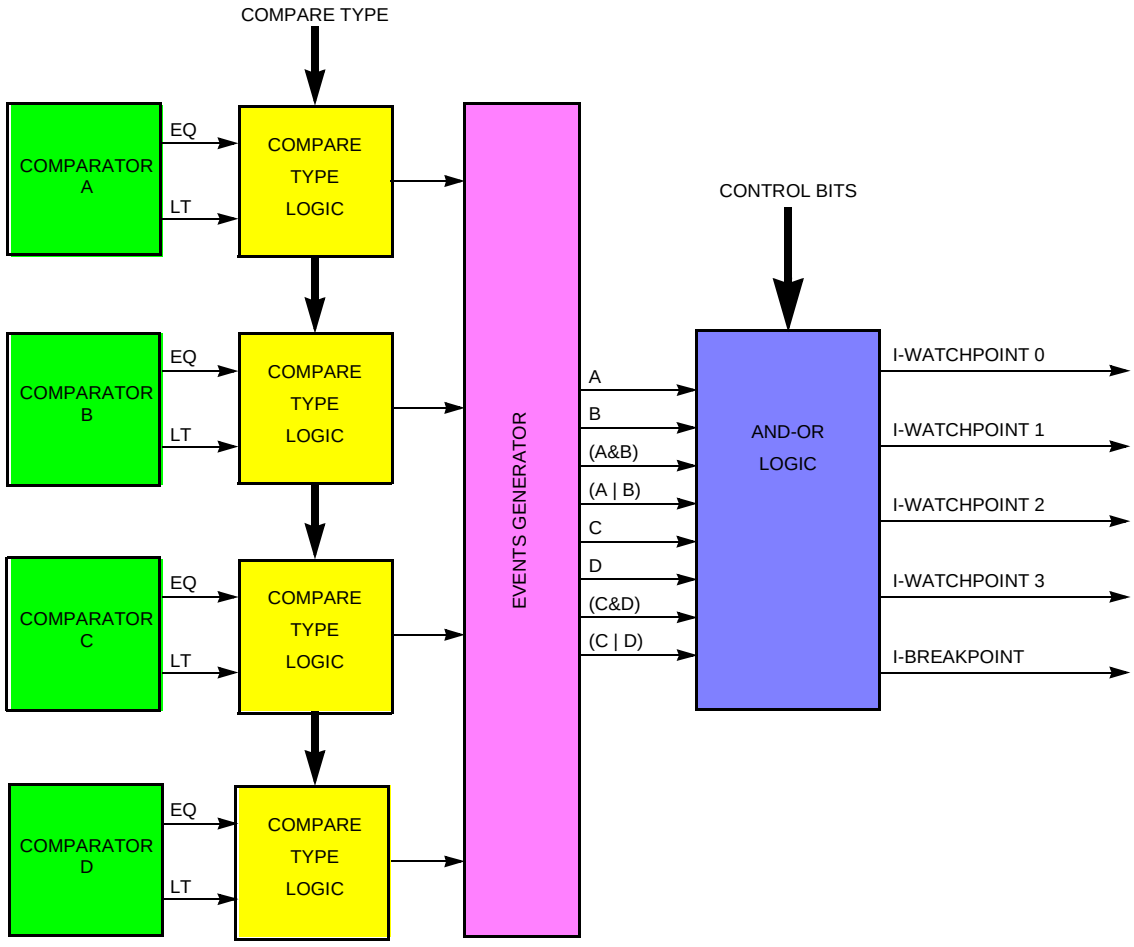


Figure 8-3 I-Bus Support General Structure

The I-bus watchpoints and breakpoint are generated using these events and according to the user's programming of the CMPA, CMPB, CMPC, CMPD, and ICTRL registers. **Table 8-8** shows how watchpoints are determined from the programming options. Note that using the OR option enables "out of range" detection.

Table 8-8 I-bus Watchpoint Programming Options

Name	Description	Programming Options
IW0	First I-bus watchpoint	Comparator A Comparators (A&B)
IW1	Second I-bus watchpoint	Comparator B Comparator (A | B)
IW2	Third I-bus watchpoint	Comparator C Comparators (C&D)
IW3	Fourth I-bus watchpoint	Comparator D Comparator (C | D)

8.2.1.5 L-Bus Support Detailed Description

There are two L-bus address comparators (comparators E and F). Each compares the 32 address bits and the cycle's read/write attribute. 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 **8.2.1.2 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 L-bus data comparators (comparators G and 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). In byte mode all signals are significant. In half-word mode only four signals from each 32-bit comparator are significant. In word mode only two signals from each 32-bit comparator are significant.

From the new "equal and less than" signals, depending on the compare type programmed by the user, one of the following four match events is 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 is asserted.

Figure 8-4 shows the general structure of L-bus support.

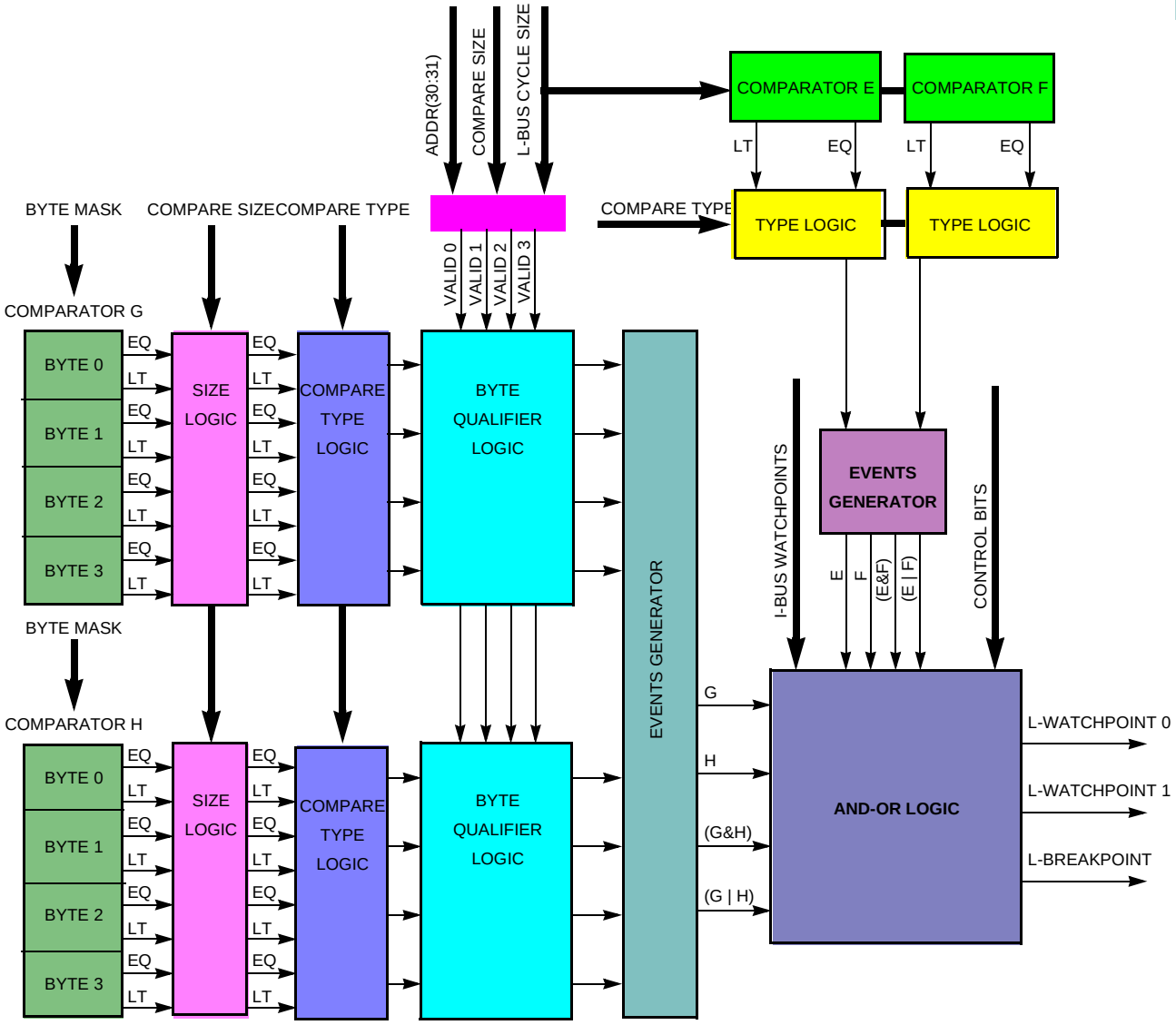


Figure 8-4 L-Bus Support General Structure

Using the match indication signals, four L-bus data events are generated as shown in **Table 8-9**.

Table 8-9 L-Bus 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 L-bus data events together with the match events of the L-bus address comparators and the I-bus watchpoints are used to generate the L-bus watchpoints and breakpoint according to the user's programming of the CMPE, CMPF, CMPG, CMPH, LCTRL1, and LCTRL2 registers. **Table 8-10** shows how the watchpoints are determined from the programming options.

Table 8-10 L-Bus Watchpoints Programming Options

Name	Description	I-Bus Events Programming Options	L-Address Events Programming Options	L-Data Events Programming Options
LW0	First L-bus watchpoint	IW0, IW1, IW2, IW3 or don't care	Comparator E Comparator F Comparators (E&F) Comparators (E | F) or don't care	Comparator G Comparator H Comparators (G&H) Comparators (G | H) or don't care
LW1	Second L-bus watchpoint	IW0, IW1, IW2, IW3 or don't care	Comparator E Comparator F Comparators (E&F) Comparators (E | F) or don't care	Comparator G Comparator H Comparators (G&H) Comparators (G | H) or don't care

8.2.1.6 Treating Floating-Point Numbers

The data comparators can detect match events on floating-point single precision values in floating point load/store instructions. When floating point values are compared, the comparators must be programmed to operate in signed word mode.

During the execution of a load/store instruction of a floating-point double operand, the L-data comparators never generate a match. If L-data events are programmed for don't care (i.e., LCTRL2[LWOLADC] = 0), L-bus watchpoint and breakpoint events can be generated from the L-address events, even if the instruction is a load/store double instruction.

8.2.2 Internal Breakpoints

Internal breakpoints are generated from the watchpoints. The user may enable a watchpoint to create a breakpoint by setting the associated software trap enable bit in the ICTRL or LCTRL2 register. This can be done by a software monitor program executed by the MCU. An external development tool can also enable internal breakpoints from watchpoints by setting the associated development port trap enable bit using the development port serial interface.

Internal breakpoints can also be generated by assigning a breakpoint counter to a particular watchpoint. The counter counts down for each watchpoint, and a breakpoint is generated when the counter reaches zero.

An internal breakpoint progresses in the machine along with the instruction that caused it (fetch or load/store cycle). When a breakpoint reaches the top of the history buffer, the machine processes the breakpoint exception.

An instruction that causes an I-bus breakpoint is not retired. The processor branches to the breakpoint exception routine *before* it executes the instruction. An instruction that causes an L-bus breakpoint is executed. The processor branches to the breakpoint exception routine *after* it executes the instruction. The address of the load/store cycle that generated the L-bus breakpoint is stored in the breakpoint address register (BAR).

8.2.2.1 Breakpoint Counters

There are two 16-bit down counters. Each counter is able to count one of the I-bus watchpoints or one of the L-bus watchpoints. Both generate the corresponding breakpoint when they reach zero. If the instruction associated with the watchpoint is not retired, the counter is adjusted back so that it reflects actual execution.

In the masked mode, the counters do not count watchpoints detected when MSR[RI] = 0. See **8.2.4 Breakpoint Masking**.

When counting watchpoints programmed on the actual instructions that alter the counters, the counters will have unpredictable values. A **sync** instruction should be inserted before a read of an active counter.

8.2.2.2 Trap-Enable Programming

The trap enable bits can be programmed by regular, supervisor-level software (by writing to the ICTRL or LCTRL2 with the **mtspr** instruction) or “on the fly” using the development port interface. For more information on the latter method, refer to **8.3.5 Trap-Enable Input Transmissions**.

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 **Table 8-30** and **Table 8-32**.

8.2.2.3 Ignore First Match

In order to facilitate the debugger utilities of “continue” and “go from x”, the option to ignore the first match is supported for the I-bus breakpoints. When an I-bus breakpoint is first enabled (as a result of the first write to the I-bus support control register or as a result of the assertion of the MSR[RI] bit in masked mode), the first instruction will not cause an I-bus breakpoint if the IFM (ignore first match) bit in the I-bus support control register (ICTRL) is set (used for “continue”). This allows the processor to be stopped at a breakpoint and then later to “continue” from that point without the breakpoint immediately stopping the processor again before executing the first instruction.

When the IFM bit is cleared, every matched instruction can cause an I-bus breakpoint (used for “go from x,” where x is an address that would not cause a breakpoint).

The IFM bit is set by the software and cleared by the hardware after the first I-bus breakpoint match is ignored.

Since L-bus breakpoints are treated after the instruction is executed, L-bus breakpoints and counter-generated I-bus breakpoints are not affected by this mode.

8.2.3 External Breakpoints

Breakpoints external to the processor can come from either an on-chip peripheral or from the development port. For additional information on breakpoints from on-chip peripherals, consult the user’s manual for the microcontroller of interest or the reference manual for the peripheral of interest.

The development port serial interface can be used to assert either a maskable or non-maskable breakpoint. Refer to **8.3.5 Trap-Enable Input Transmissions** for more information about generating breakpoints from the development port. The development port breakpoint bits remain asserted until they are cleared; however, they cause a breakpoint only when they change from cleared to set. If they remain set, they do not cause an additional breakpoint until they are cleared and set again.

External breakpoints are not referenced to any particular instruction; they are referenced to the current or following L-bus cycle. The breakpoint is taken as soon as the processor completes an instruction that uses the L-bus.

8.2.4 Breakpoint Masking

The processor responds to two different types of breakpoints. The maskable breakpoint is taken only if the processor is in a recoverable state. This means that taking the breakpoint will not destroy any of the internal machine context. The processor is defined to be in a recoverable state when the MSR[RI] (recoverable exception) bit is set. Maskable breakpoints are generated by the internal breakpoint logic, modules on the IMB2, and the development port.

Non-maskable breakpoints cause the processor to stop without regard to the state of the MSR[RI] bit. If the processor is in a non-recoverable state when the breakpoint occurs, the state of the SRR0, SRR1, and the DAR may have been overwritten by the breakpoint. It will not be possible to restart the processor, since the restart address and MSR context may not be available in SRR0 and SRR1.

Only the development port and the internal breakpoint logic are capable of generating a non-maskable breakpoint. This allows the user to stop the processor in cases where it would otherwise not stop, but with the penalty that it may not be restartable. The value of the MSR[RI] bit as saved in the SRR1 register indicates whether the processor stopped in a recoverable state or not.

Internal breakpoints are made maskable or non-maskable by clearing or setting the BRKNOMSK bit of the LCTRL2 register. Refer to **8.8.7 L-Bus Support Control Register 2**.

8.3 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 MCU is shown in **Figure 8-5**. Although the development port is implemented as part of the system interface unit (SIU), it is used in conjunction with RCPU development support features and is therefore described in this section.

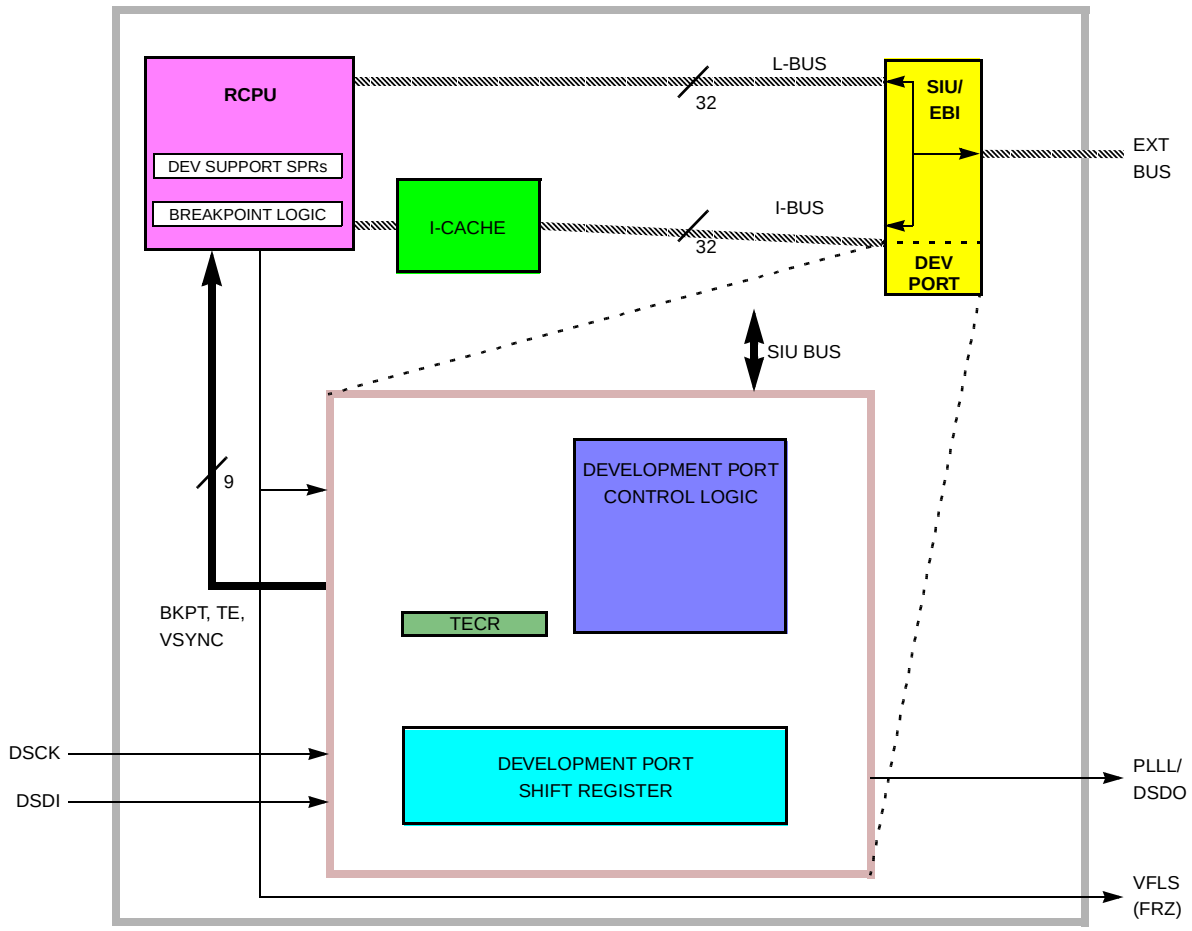


Figure 8-5 Development Port Support Logic

8.3.1 Development Port Signals

The following development port signals are provided:

- Development serial clock (DSCK)
- Development serial data in (DSDI)
- Development serial data out (DSDO)

The development port signal DSDO shares a pin with the PLLL signal.

8.3.1.1 Development Serial Clock

In clocked mode (see **8.3.3 Development Port Clock Mode Selection**), the development serial clock (DSCK) is used to shift data into and out of the development port shift register. The DSCK and DSDI inputs are synchronized to the on-chip system clock, thereby minimizing the chance of propagating metastable states into the serial state machine. The values of the pins are sampled during the low phase of the system clock. At the rising edge of the system clock, the sampled values are latched internally. One quarter clock later, the latched values are made available to the development support logic.

In clocked mode, detection of the rising edge of the synchronized clock causes the synchronized data from the DSDI pin to be loaded into the least significant bit of the shift register. This transfer occurs one quarter clock after the next rising edge of the system clock. At the same time, the new most significant bit of the shift register is presented at the PLLL/DSDO pin. Future 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.

To allow the synchronizers to operate correctly, the development serial clock frequency must not exceed one half of the system clock frequency. The clock may be implemented as a free-running clock. The shifting of data is controlled by ready and start signals so the clock does not need to be gated with the serial transmissions. Refer to **8.3.5 Trap-Enable Input Transmissions** and **8.3.6 CPU Input Transmissions**.

The DSCK pin is also used during reset to enable debug mode and immediately following reset to optionally cause immediate entry into debug mode following reset. This is described in section **8.4.1 Enabling Debug Mode** and **8.4.2 Entering Debug Mode**.

8.3.1.2 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, transitions on the DSDI pin should occur at least a setup time ahead of the rising edge of the DSCK signal (if in clocked mode) or a setup time ahead of the rising edge of the system clock, whichever is greater. This will allow operation of the development port either asynchronously or synchronously with the system clock. The DSDI 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.

When the processor is not in debug mode (freeze not indicated on VFLS[0:1] pins) the data received on the DSDI pin is transferred to the trap enable control register. When the processor is in debug mode, the data received on the DSDI pin is provided to the debug mode interface. Refer to **8.3.5 Trap-Enable Input Transmissions** and **8.3.6 CPU Input Transmissions** for additional information.

The DSDI pin is also used at reset to control overall chip reset configuration and immediately following reset to determine the development port clock mode. See **8.3.3 Development Port Clock Mode Selection** for more information.

8.3.1.3 Development Serial Data Out

When the processor is not in reset, the development port shifts data out of the development port shift register using the development serial data out (PLLL/DSDO) pin. When the processor is in reset, the PLLL/DSDO pin indicates the state of lock of the system clock phase-locked loop. This can be used to determine when a reset is caused by a loss of lock on the system clock PLL.

8.3.2 Development Port Registers

The development port consists of two registers: the development port shift register and the trap enable control register. These registers are described in the following paragraphs. **Figure 8-6** illustrates the development port registers and data paths.

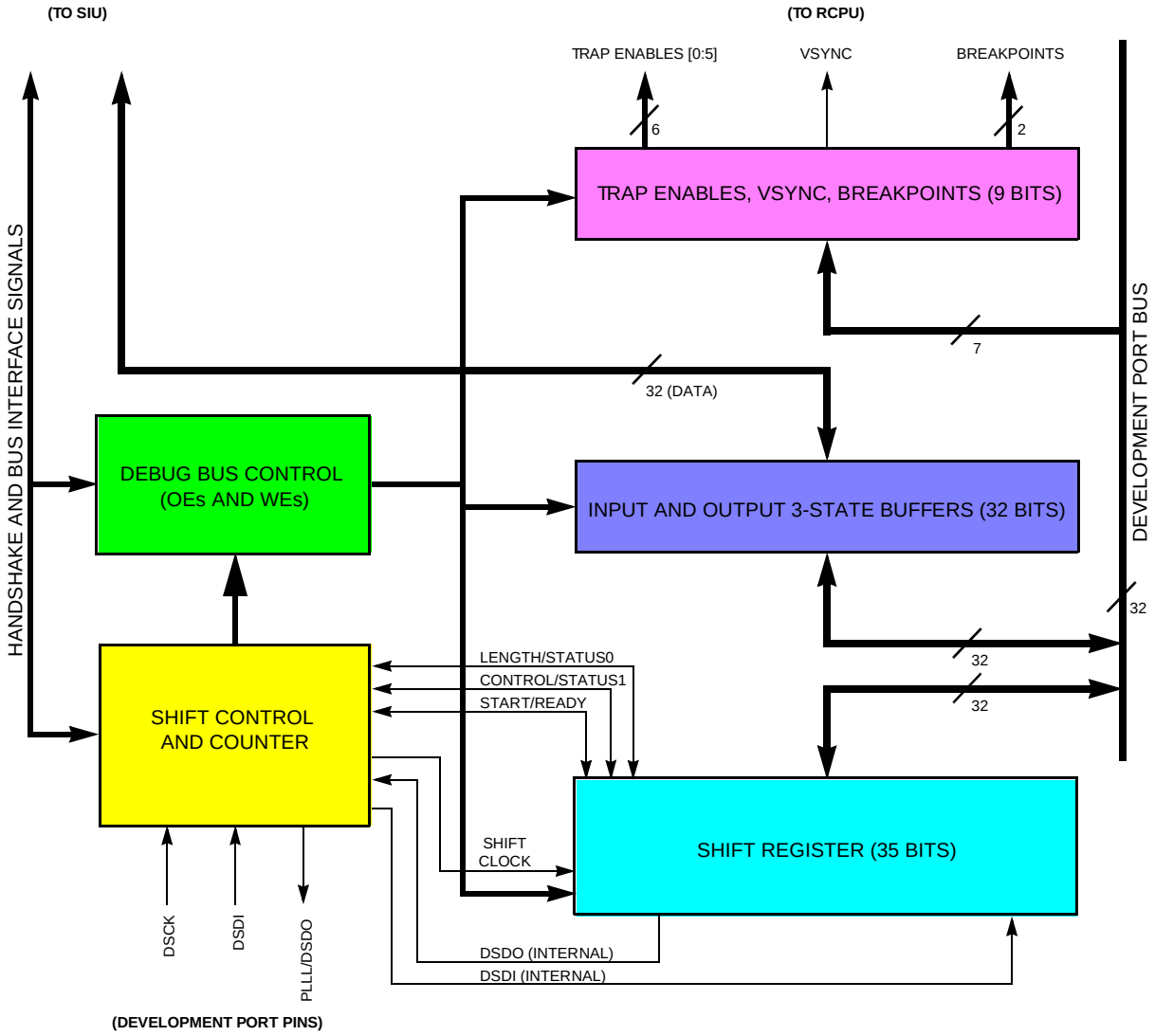


Figure 8-6 Development Port Registers and Data Paths

8.3.2.1 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. These instructions or data are then transferred in parallel to the processor or the trap enable control register (TECR).

When the processor enters debug mode, it fetches instructions from the development port shift register. These instructions are serially loaded into the shift register from DSDI.

When the processor is in debug mode, data is transferred to the CPU by shifting it into the shift register. The processor then reads the data as the result of executing a “move from special-purpose register DPDR” (development port data register) instruction.

In debug mode, 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 PLLL/DSDO.

8.3.2.2 Trap Enable Control Register

The trap enable control register (TECR) is a nine-bit register that is loaded from the development port shift register. The contents of the TECR are used to drive the six trap enable signals, the two breakpoint signals, and the VSYNC signal to the processor. Trap-enable transmissions to the development port cause the appropriate bits of the development port shift register to be transferred to the control register.

8.3.3 Development Port Clock Mode Selection

All of the development port serial transmissions are clocked transmissions. The transmission clock can be either synchronous or asynchronous with the system clock (CLKOUT). The development port supports three methods for clocking the serial transmissions. The first method allows the transmission to occur without being externally synchronized with CLKOUT but at more restricted data rates. The two faster communication methods require the clock and data to be externally synchronized with CLKOUT.

The first clock mode is called *asynchronous clocked* since the input clock (DSCK) is asynchronous with CLKOUT. The input synchronizers on the DSCK and DSDI pins sample the inputs to ensure that the signals used internally have no metastable oscillations. To be sure that data on DSDI is sampled correctly, transitions on DSDI must occur a setup time ahead of the rising edge of DSCK. Data on DSDI must also be held for one CLKOUT cycle plus one hold time after the rising edge of DSCK. This ensures that after the signals have passed through the input synchronizers, the data will be valid at the rising edge of the serial clock even if DSCK and DSDI do not meet the setup and hold time requirements of the pins.

Asynchronous clocked mode allows communications with the port from a development tool that does not have access to the CLKOUT signal or where the CLKOUT signal has been delayed or skewed. Because of the asynchronous nature of the inputs and the setup and hold time requirements on DSDI, this clock mode must be clocked at a frequency less than or equal to one third of CLKOUT.

The second clock mode is called *synchronous clocked* because the input clock and input data meet all setup and hold time requirements with respect to CLKOUT. Since the input synchronizers must sample the input clock in both the high and low state, DSCK cannot be faster than one half of CLKOUT.

The third clock mode is called *synchronous self-clocked* because it does not require an input clock. Instead, the port is clocked by the system clock. The DSDI input is required to meet all setup and hold time requirements with respect to CLKOUT. The

data rate for this mode is always the same as the system clock rate, which is at least twice as fast as in synchronous clocked mode. In this mode, an undelayed CLKOUT signal must be available to the development tool, and extra care must be taken to avoid noise and crosstalk on the serial lines.

The selection of clocked or self-clocked mode is made immediately following reset. The state of the DSDI input is latched eight clocks after $\overline{\text{RESETOUT}}$ is negated. If it is latched low, external clocked mode is enabled. If it is latched high, then self-clocked mode is enabled. When external clocked mode is enabled, the use of asynchronous or synchronous mode is determined by the design of the external development tool.

Since DSDI is used during reset to configure the MCU and 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 does not begin scanning for the start bit of a serial transmission until 16 clocks after the negation of $\overline{\text{RESETOUT}}$. If DSDI is asserted 16 clocks after $\overline{\text{RESETOUT}}$ negation, the port will wait until DSDI is negated to begin scanning for the start bit.

The selection of clocked/self-clocked mode is shown in **Figure 8-7**. The timing diagrams in **Figure 8-8**, **Figure 8-9**, and **Figure 8-10** show the serial communications for both trap enable mode and debug mode for all clocking schemes.

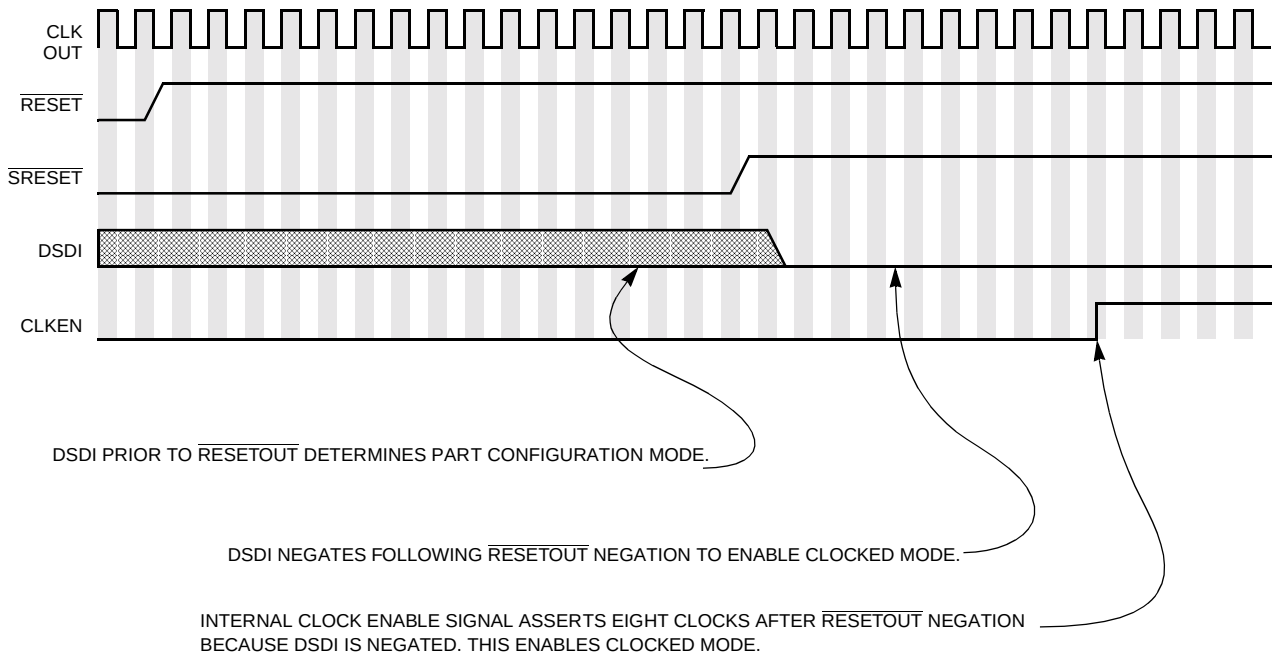


Figure 8-7 Enabling Clock Mode Following Reset

Examples of serial communications using the three clock modes are shown in **Figure 8-8**, **Figure 8-9**, and **Figure 8-10**.

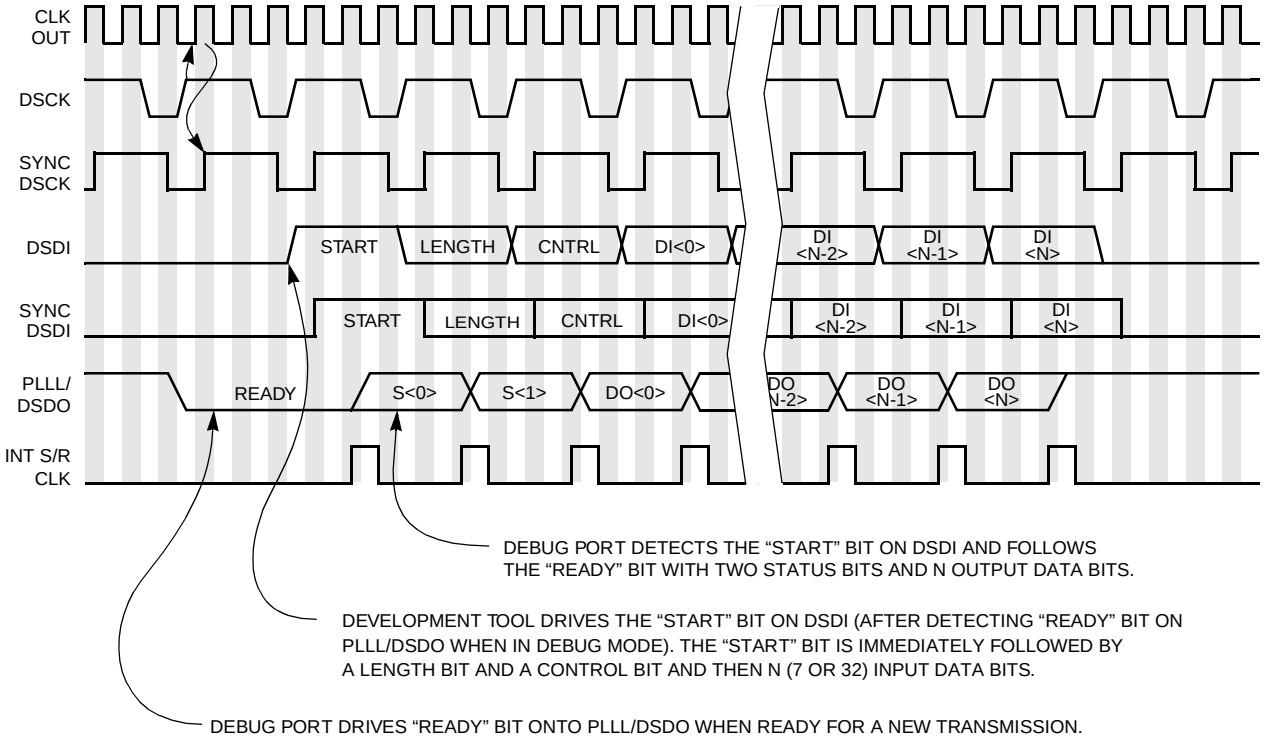


Figure 8-8 Asynchronous Clocked Serial Communications

In **Figure 8-8**, the frequency on the DSCK pin is equal to CLKOUT frequency divided by three. This is the maximum frequency allowed for the asynchronous clocked mode. DSCK and DSDI transitions are not required to be synchronous with CLKOUT.

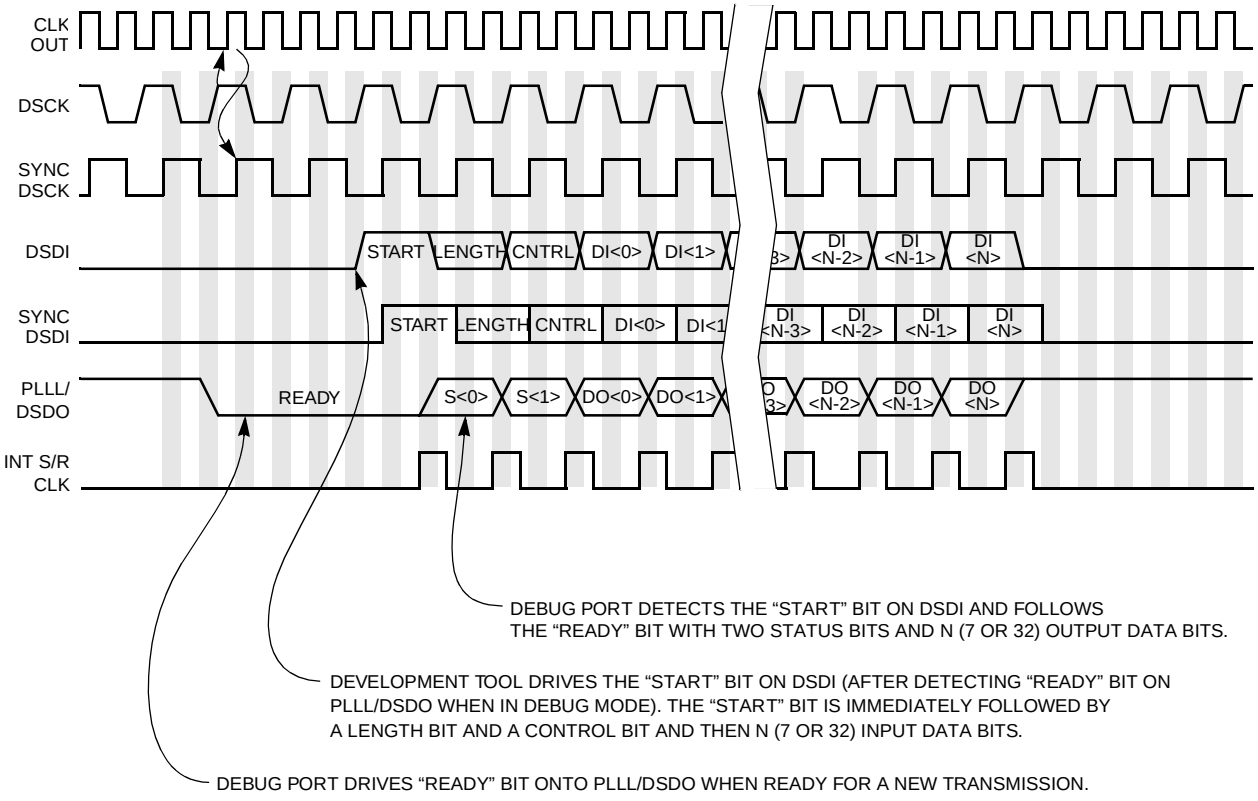


Figure 8-9 Synchronous Clocked Serial Communications

In **Figure 8-9**, the frequency on the DSCK pin is equal to CLKOUT frequency divided by two. DSDI and DSCK transitions must meet setup and hold timing requirements with respect to CLKOUT.

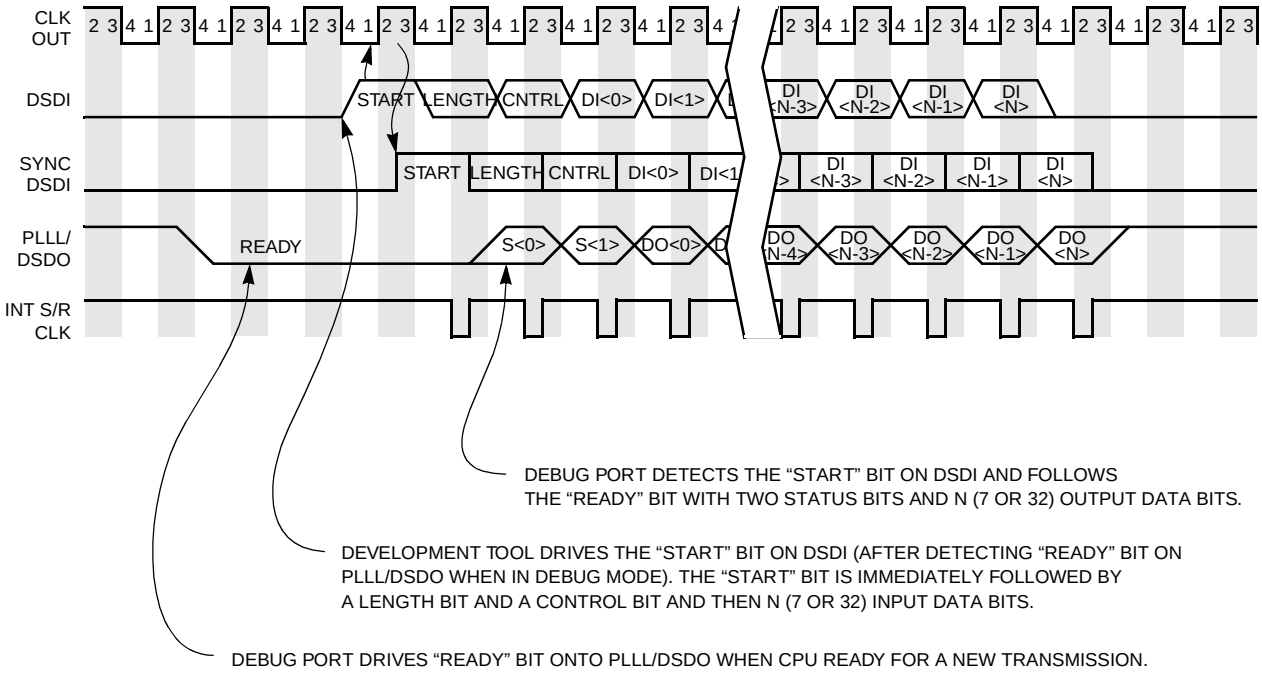


Figure 8-10 Synchronous Self-Clocked Serial Communications

In **Figure 8-10**, the DSCK pin is not used, and the transmission is clocked by CLKOUT. DSDI transitions must meet setup and hold timing requirements with respect to CLKOUT.

8.3.4 Development Port Transmissions

The development port starts communications by setting PLLL/DSDO (the *ready* bit, or MSB of the 35-bit development port shift register) low to indicate that all activity related to the previous transmission is 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 on the DSDI pin.

The start bit also signals the development port that it can begin driving data on the DSDO pin. While data is shifting into the LSB of the shift register from the DSDI pin, it is simultaneously shifting out of the MSB of the shift register onto the DSDO pin.

A *length* bit defines the transmission as being to either the trap-enable register (length bit = 1, indicating seven data bits) or the CPU (length bit = 0, indicating 32 data bits). Transmissions of data and instructions to the CPU are allowed only when the processor is in debug mode. The two types of transmissions are discussed in **8.3.5 Trap-Enable Input Transmissions** and **8.3.6 CPU Input Transmissions**.

8.3.5 Trap-Enable Input Transmissions

If the length bit is set, the input transmission will only be ten bits long. These trap-enable transmissions into the development port include a start bit, a length bit, a control bit, and seven data bits. Only the seven data bits are shifted into the 35-bit shift

register. These seven bits are then latched into the TECR. The 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, as shown in **Table 8-11** and **Table 8-12**.

Table 8-11 Trap Enable Data Shifted Into Development Port Shift Register

Start	Length	Con-trol	1st	2nd	3rd	4th	1st	2nd	VSYNC	Usage
			I-bus				L-bus			
			Watchpoint Trap Enables							
1	1	0	0 = disabled; 1 = enabled							Input data for trap enable control register

Table 8-12 Breakpoint Data Shifted Into Development Port Shift Register

Start	Length	Con-trol	Non-Maskable	Maskable	Reserved bits					Usage
			Breakpoints							
1	1	1	0 = negate; 1 = assert		1	1	1	1	1	Input data for trap enable control register

8.3.6 CPU Input Transmissions

If the length bit in the serial input sequence is cleared, the transmission is an input to the CPU. This transmission type is legal only when the processor is in debug mode.

For transmissions to the CPU, the 35 bits of the development port shift register are interpreted as a start bit, a length bit, a control bit, and 32 bits of instructions or data. The encoding of data shifted into the development port shift register (through the DSDI pin) is shown in **Table 8-13**.

Table 8-13 CPU Instructions/Data Shifted into Shift Register

Start	Length	Control	Instruction/Data (32 Bits)	Usage
1	0	0	CPU Instruction	Input instruction for the CPU
1	0	1	CPU Data	Input data for the CPU

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.

8.3.7 Serial Data Out of Development Port — Non-Debug Mode

The encoding of data shifted out of the development port shift register when the processor is not in debug mode is shown in **Table 8-14**.

Table 8-14 Status Shifted Out of Shift Register — Non-Debug Mode

Ready	Status [0:1]		Data (7 or 32 Bits[1])	Indication
(0)	0	1	Ones	Sequencing Error
(0)	1	1	Ones	Null

NOTES:
1. Depending on input mode.

When the processor is not in debug mode, the sequencing error encoding indicates that the transmission from the external development tool was a transmission to the CPU (length = 0). When a sequencing error occurs, the development port ignores the data being shifted in while the sequencing error is shifting out.

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

When the processor is not in debug mode, the ready bit is 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, and the DSDO pin can be used for untimed I/O.

8.3.8 Serial Data Out of Development Port — Debug Mode

The encoding of data shifted out of the development port shift register when the processor is in debug mode is shown in **Table 8-14**.

Table 8-15 Status/Data Shifted Out of Shift Register

Ready	Status [0:1]		Data (7 or 32 Bits[1])	Indication
(0)	0	0	Data	Valid Data from CPU
(0)	0	1	Ones	Sequencing Error
(0)	1	0	Ones	CPU Exception
(0)	1	1	Ones	Null

NOTES:
1. Depending on input mode.

8.3.8.1 Valid Data Output

The *valid data* encoding is used when data has been transferred from the CPU to the development port shift register. This is the result of executing an instruction in debug mode to move the contents of a general-purpose register to the development port data register (DPDR).

The valid data encoding has the highest priority of all status outputs and is reported even if an exception occurs at the same time. Any exception that is recognized during the transmission of valid data is not related to the execution of an instruction. There-

fore, a status of valid data is output and the CPU exception status is saved for the next transmission. Since it is not possible for a sequencing error to occur and for valid data to be received on the same transmission, there is no conflict between a valid data status and the sequencing error status.

8.3.8.2 Sequencing Error Output

The *sequencing error* encoding indicates that the inputs from the external development tool are not what the development port or the CPU was expecting. Two cases could cause this error: 1) the processor was trying to read instructions and data was shifted into the development port, or 2) the processor was trying to read data and an instruction was shifted into the development port.

When a sequencing error occurs, the port terminates the CPU read or fetch cycle with a bus error. This bus error causes the CPU to signal the development port that an exception occurred. Since a status of sequencing error has a higher priority than a status of exception, the port reports the sequencing error. The development port ignores the data being shifted in while the sequencing error is shifting out. The next transmission to the port should be a new instruction or trap enable data.

Table 8-16 illustrates a typical sequence of events when a sequencing error occurs. This example begins with CPU data being shifted into the shift register (control bit = 1) when the processor is expecting an instruction. During the next transmission, a sequencing error is shifted out of the development port, and the data shifted into the shift register is thrown away. During the third transmission, the “CPU exception” status is output, and again the data shifted into the shift register is thrown away. During the fourth transmission, an instruction is again shifted into the development port and fetched by the CPU for execution. Notice in this example that the development port throws away the first two input transmissions following the one causing the sequencing error.

Table 8-16 Sequencing Error Activity

Trans #	Input to Development Port	Output from Development Port	Port Action	CPU Action
1	CPU data (Control bit = 1)	Depends on previous transmissions	Cause bus error, set sequence error latch	Fetch instruction, take exception because of bus error
2	X (Thrown away)	Sequencing error	Set exception latch, clear sequencing error latch	Signal exception to port, begin new fetch from port
3	X (Thrown away)	CPU exception	Clear exception latch	Continue to wait for instruction from port
4	CPU instruction	Null	Send instruction to CPU at end of transmission	Fetch instruction from port

8.3.8.3 CPU Exception Output

The *CPU exception* encoding is used to indicate that the CPU encountered an exception during the execution of the previous instruction in debug mode. Exceptions may occur as the result of instruction execution (such as unimplemented opcode or arithmetic error), because of a memory access fault, or from an external interrupt. The exception is recognized only if the associated bit in the DER is set. When an exception occurs, the development port ignores the data being shifted in while the CPU exception status is shifting out. The port terminates the current CPU access with a bus error. The next transmission to the port should be a new instruction or trap enable data.

8.3.8.4 Null Output

Finally, the *null* encoding is used to indicate that no data has been transferred from the CPU to the development port shift register. It also indicates that the previous transmission did not have any associated errors.

8.3.9 Use of the Ready Bit

To minimize the overhead required to detect and correct errors, the external development system should wait for the ready bit on DSDO before beginning each input transmission. This ensures that all CPU activity (if any) relative to the previous transmission has been completed and that any errors have been reported.

When the ready bit is used to pace the transmissions, the error status is reported during the transmission following the error. Since any transmission into the port which occurs while shifting out an error status is ignored by the port, the error handler in the external development tool does not need to undo the effects of an intervening instruction.

To improve system performance, however, an external development system may begin transmissions before the ready bit is asserted. If the next transmission does not wait until the port indicates ready, the port will not assert ready again until this next transmission completes and all activity associated with it has finished. Transmissions that begin before ready is asserted on DSDI are subject to the following limitations and problems.

First, if the previous transmission results in a sequence error, or the CPU reports an exception, that status may not be reported until two transmissions after the transmission that caused the error. (When the ready bit is used, the status is reported in the following transmission.) This is because an error condition which occurs after the start of a transmission cannot be reported until the next transmission.

Second, if a transmitted instruction causes the CPU to write to the DPDR and the transmission that follows does not wait for the assertion of ready, the CPU data may not be latched into the development port shift register, and the valid data status is not output. Despite this, no error is indicated in the status outputs. To ensure that the CPU has had enough time to write to the DPDR, there must be at least four CLKOUT cycles between the time the last bit of the instruction (move to SPR) is clocked into the port and the time the start bit for the next transmission is clocked into the port.

8.4 Debug Mode Functions

In debug mode, the CPU fetches all instructions from the development port. In addition, data can be read from and written to the development port. This allows memory and registers to be read and modified by an external development tool (emulator) connected to the development port.

8.4.1 Enabling Debug Mode

Debug mode is enabled by asserting the DSCK pin during reset. The state of this pin is sampled immediately before the negation of RESETOUT. If the DSCK pin is sampled low, debug mode is disabled until a subsequent reset when the DSCK pin is sampled high. When debug mode is disabled, the internal watchpoint/breakpoint hardware is still operational and can be used by a software monitor program for debugging purposes.

The DSCK pin is sampled again eight clock cycles following the negation of RESETOUT. If DSCK is negated following reset, the processor jumps to the reset vector and begins normal execution. If DSCK is asserted following reset and debug mode is enabled, the processor enters debug mode before executing any instructions.

A timing diagram for enabling debug mode is shown in **Figure 8-11**.

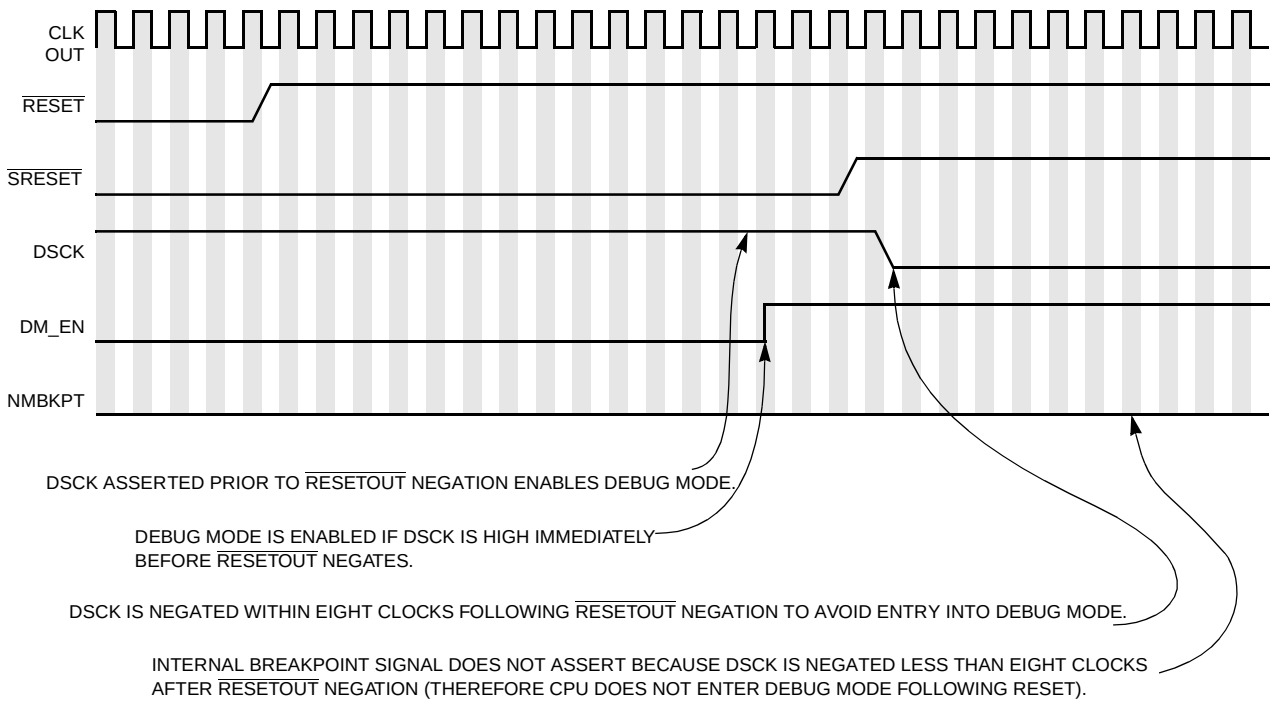


Figure 8-11 Enabling Debug Mode at Reset

8.4.2 Entering Debug Mode

Debug mode is entered whenever debug mode is enabled, an exception occurs, and the corresponding bit is set in the debug enable register (DER). The processor performs normal exception processing, (i.e., saving the next instruction address and the current state of MSR in SRR0 and SRR1 and modifying the contents of the MSR). The processor then enters debug mode and fetches the next instruction from the development port instead of from the vector address. The exception cause register (ECR) shows which event caused entry into debug mode. The freeze indication is encoded on the VFLS pins to show that the CPU is in debug mode.

Debug mode may also be entered immediately following reset. If the DSCK pin continues to be asserted following reset (after debug mode is enabled), the processor takes a breakpoint exception and enters debug mode directly after fetching (but not executing) the reset vector. To avoid entering debug mode following reset, the DSCK pin must be negated no later than seven clock cycles after $\overline{\text{RESETOUT}}$ is negated.

A timing diagram for entering debug mode following reset is shown in **Figure 8-12**.

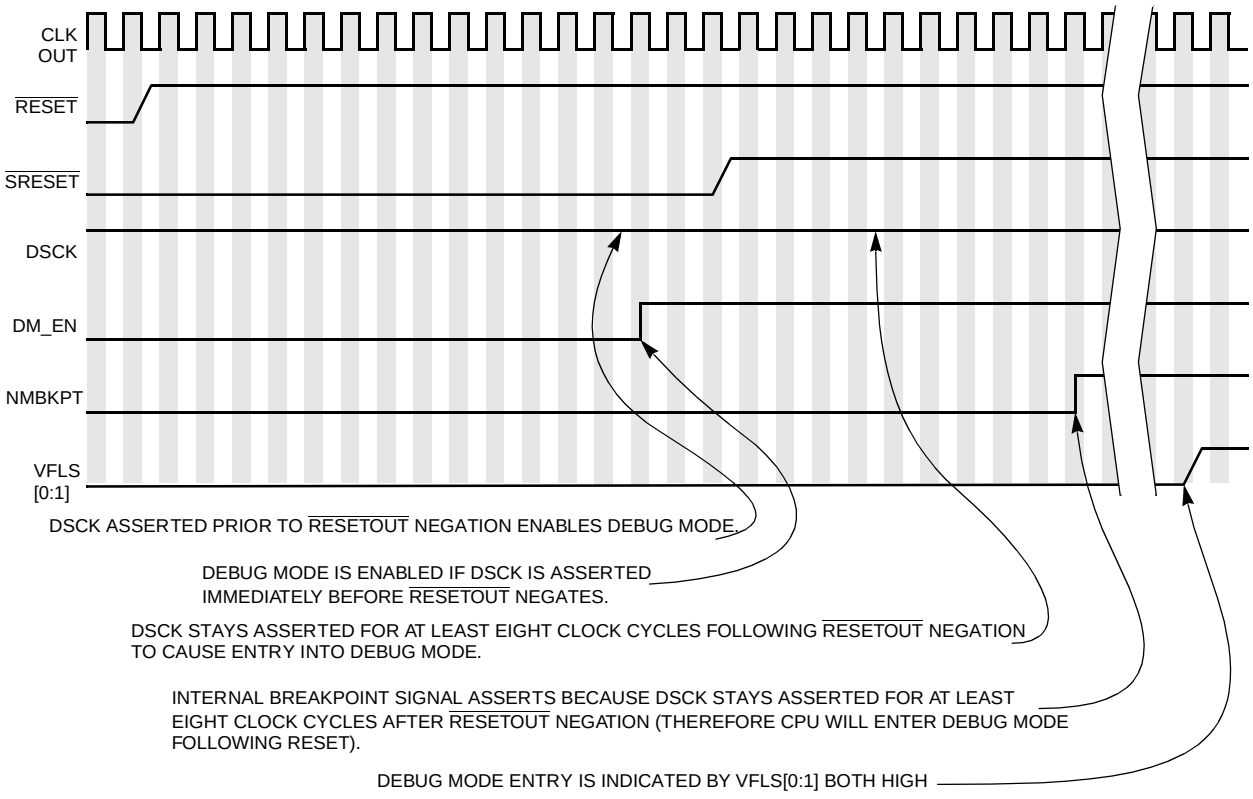


Figure 8-12 Entering Debug Mode Following Reset

8.4.3 Debug Mode Operation

In debug mode, the CPU fetches instructions from the development port. It can also read and write data at the development port. In debug mode the prefetch mechanism

in the CPU is disabled. This forces all data accesses to the development port to occur immediately following the fetch of the associated instruction.

In debug mode, if an exception occurs during the execution of an instruction, normal exception processing does not result. (That is, the processor does not save the MSR and instruction address and does not branch to the exception handler.) Instead, a flag is set that results in a CPU exception status indication in the data shifted out of the development port shift register. The same thing happens if the processor detects an external interrupt. (This can occur only when the associated DER bit is clear and MSR[EE] is set.) When the data in the development port shift register is shifted out, the exception status is detected by the external development tool. The cause of the exception can be determined by reading the ECR.

8.4.4 Freeze Function

While the processor is in debug mode, the freeze indication is broadcast throughout the MCU. This signal is generated by the CPU when debug mode is entered, or when a software debug monitor program is entered as the result of an exception and the associated bit in the DER is set. The software monitor can only assert freeze when debug mode is not enabled. Refer to **8.7 Software Monitor Support** for more information.

Freeze is indicated by the value 11 on the VFLS[0:1] pins. This encoding is not used for pipeline tracking and is left on the VFLS[0:1] pins when the processor is in debug mode. **Figure 8-14** shows how the internal freeze signal is generated.

8.4.5 Exiting Debug Mode

Executing the **rfi** instruction in debug mode causes the processor to leave debug mode and return to normal execution. The freeze indication on the VFLS pins is negated to indicate that the CPU has exited debug mode.

Software must read the ECR (to clear it) before executing the **rfi** instruction. Otherwise, if a bit in the ECR is asserted and its corresponding enable bit in the DER is also asserted, the processor re-enters debug mode and re-asserts the freeze signal immediately after executing the **rfi** instruction.

8.4.6 Checkstop State and Debug Mode

When debug mode is disabled, the processor enters the checkstop state if, when a machine check exception is detected, the machine check exception is disabled (MSR[ME] = 0). However, when debug mode is enabled, if a machine check exception is detected when MSR[ME] = 0 and the checkstop enable bit in the DER is set, the processor enters debug mode rather than the checkstop state. This allows the user to determine why the checkstop state was entered. **Table 8-17** shows what happens when a machine check exception occurs under various conditions.

Table 8-17 Checkstop State and Debug Mode

MSR[ME]	Debug Mode Enable	CHSTPE[1]	MCIE[2]	Action Performed when CPU Detects a Machine Check Interrupt	ECR Value
0	0	X	X	Enter the checkstop state	0x2000 0000
0	1	0	X	Enter the checkstop state	0x2000 0000
0	1	1	X	Enter debug mode	0x2000 0000
1	0	X	X	Take machine check exception	0x1000 0000
1	1	X	0	Take machine check exception	0x1000 0000
1	1	X	1	Enter debug mode	0x1000 0000

NOTES:
1. Checkstop enable bit in the DER
2. Machine check interrupt enable bit in the DER

8.5 Development Port Transmission Sequence

The following sections describe the sequence of events for communication with the development port in both debug and normal mode and provide specific sequences for prologues, epilogues, and poking and peeking operations.

8.5.1 Port Usage in Debug Mode

The sequence of events for communication with the development port in debug mode (freeze is indicated on the VFLS pins) is shown in **Table 8-18**. The sequence starts with the processor trying to read an instruction in step 1. The sequence ends when the processor is ready to read the next instruction. Reading an instruction is the first action the processor takes after entering debug mode. The processor and development port activity is determined by the instruction or data shifted into the shift register. The instruction or data shifted into the shift register also determines the status shifted out during the next transmission. The next step column indicates which step has the appropriate status response.

Table 8-18 Debug Mode Development Port Usage

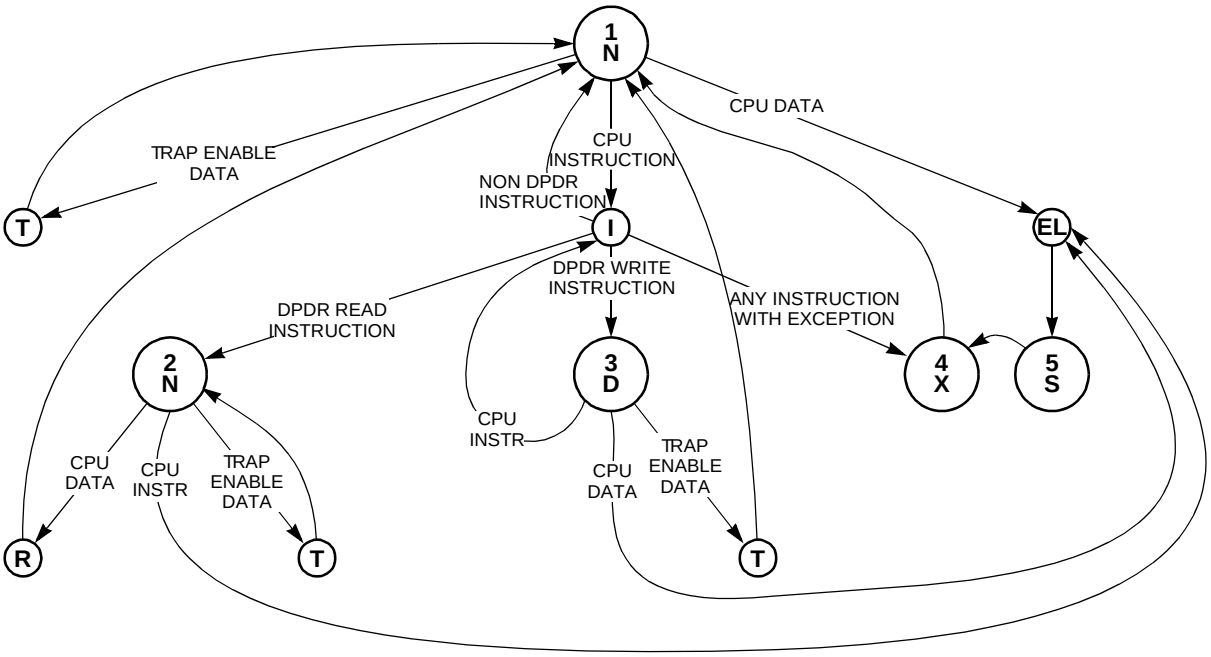
This Step	Serial Data Shifted In (DSDO Indicates "READY")	Shifted Out This Transmission	Development Port Activity; Processor Activity	Next Step
1	CPU instruction (non-DPDR)	Null	Port transfers instruction to CPU; CPU executes instruction, fetches next instruction	1
	CPU instruction (DPDR read)		Port transfers instruction to CPU; CPU executes instruction, reads DPDR	2
	CPU instruction (DPDR write)		Port transfers instruction to CPU; CPU writes DPDR, fetches next instruction	3
	CPU instruction (instruction execution causes exception)		Port transfers instruction to CPU; CPU signals exception to port, fetches next instruction	4
	Data for CPU		Port ignores data, terminates fetch with error, latches sequence error; CPU signals exception to port, fetches next instruction	5
	Data for Trap Enable Control Register		Port updates trap enable control register; CPU waits (continues fetch)	1
2	Any CPU instruction	Null	Port ignores data, terminates DPDR read with error; latches sequence error; CPU signals exception to port, fetches next instruction	5
	Data for CPU		Port transfers data to CPU; CPU reads data from DPDR, fetches next instruction	1
	Data for trap enable control register		Port updates TECR CPU waits (continue data read)	2

Table 8-18 Debug Mode Development Port Usage (Continued)

This Step	Serial Data Shifted In (DSDO Indicates "READY")	Shifted Out This Transmission	Development Port Activity; Processor Activity	Next Step
3	CPU instruction (non-DPDR)	CPU data	Port transfers instruction to CPU; CPU executes instruction, fetches next instruction	1
	CPU instruction (DPDR read)		Port transfers instruction to CPU; CPU executes instruction, reads DPDR	2
	CPU instruction (DPDR write)		Port transfers instruction to CPU; CPU writes DPDR, fetches next instruction	3
	CPU instruction (with exception)		Port transfers instruction to CPU; CPU signals exception to port, fetches next instruction	4
	Data for CPU		Port ignores data, terminates fetch with error, latches sequence error; CPU signals exception to port, fetches next instruction	5
	Data for trap enable control register	7 MSB of CPU data	Port updates TECR; CPU waits (continues fetch)	1
4	Any (ignored by port)	Exception	Port ignores data; CPU waits (continues fetch)	1
5	Any (ignored by port)	Sequence Error	Port ignores data; CPU waits (continues fetch)	4

8.5.2 Debug Mode Sequence Diagram

The sequence of activity shown in **Table 8-18** is summarized below in **Figure 8-13**. The numbers in the large circles correspond to the steps in **Table 8-18**. The letters in the large circles indicate the status that will be shifted out during the transmission. The letters in the small circles show the activity of the development port and the CPU as a result of the transmission.



N - SHIFT OUT NULL STATUS
X - SHIFT OUT EXCEPTION STATUS
S - SHIFT OUT SEQUENCE ERROR STATUS
D - SHIFT OUT DATA FROM CPU
E - TERMINATE CPU READ WITH ERROR
L - LATCH SEQUENCE ERROR
I - TRANSFER INSTR TO CPU
R - TRANSFER DATA TO CPU (READ)
T - TRANSFER DATA TO TECR

Figure 8-13 General Port Usage Sequence Diagram

8.5.3 Port Usage in Normal (Non-Debug) Mode

The sequence of events for communication with the development port when the CPU is not in debug mode (freeze is not indicated on the VFLS pins) is shown below in **Table 8-18**. Note that any instructions or data for the CPU result in a sequence error status response when the processor is not in debug mode. Only data for the trap enable control register is allowed.

Table 8-19 Non-Debug Mode Development Port Usage

This Step	Serial Data Shifted into DPDI (Not in Debug Mode)	Shifted out of DPDO this Transmission	Development Port Activity	Next Step
6	Any CPU instruction or data	Null	Port ignores data and latches sequence error	7
	Data for trap enable control register		Port updates trap enable control register	6
7	Any (ignored by port)	Sequence Error	Port ignores data	6

8.6 Examples of Debug Mode Sequences

The tables that follow show typical sequences of instructions that are used in a development activity. They assume that no bus errors or sequence errors occur and that no writes occur to the trap enable control register.

8.6.1 Prologue Instruction Sequence

The prologue sequence of instructions is used to unload the machine context when entering debug mode. The sequence starts by unloading two general-purpose registers (R0 and R1) to be used as a data transfer register and an address pointer. Since SRR0 and SRR1 are not changed while in debug mode except by explicitly writing to them, there is no need to save and restore these registers. Finally, the ECR is unloaded to determine the cause of entry into debug mode. Any registers that will be used while in debug mode in addition to R0 and R1 will also need to be saved.

Table 8-20 Prologue Events

Development Port Activity	Instruction	Processor Activity	Purpose
Shift in instruction	**mtspr** DPDR, R0	Transfer R0 to DPDR	Save R0 so the register can be used
Shift out R0 data, shift in instruction	**mfspr** R0, ECR	Transfer ECR to R0	Read the debug mode cause register
Shift in instruction	**mtspr** DPDR, R0	Transfer from R0 to DPDR	Output reason for debug mode entry
Shift out stop cause data, shift in instruction	**mtspr** DPDR, R1	Transfer R1 to DPDR	Save R1 so the register can be used
Shift out R1 data, shift in instruction	First instruction of next sequence	Execute next instruction	Continue instruction processing

8.6.2 Epilogue Instruction Sequence

The epilogue sequence of instructions is used to restore the machine context when leaving debug mode. It restores the two general-purpose registers and then issues the **rfi** instruction. If additional registers were used while in debug mode, they also need to be restored before the **rfi** instruction is executed.

Table 8-21 Epilogue Events

Development Port Activity	Instruction	Processor Activity	Purpose
Shift in instruction, shift in saved R0	**mfspr** R0, DPDR	Transfer from DPDR to R0	Restores value of R0 when stopped
Shift in instruction, shift in saved R1	**mfspr** R1, DPDR	Transfer from DPDR to R1	Restores value of R1 when stopped
Shift in instruction	**rfi**	Return from exception	Restart execution

8.6.3 Peek Instruction Sequence

The peek sequence of instructions is used to read a memory location and transfer the data to the development port. It starts by moving the memory address into R1 from the development port. Next the location is read and the data loaded into R0. Finally, R0 is transferred to the development port.

Table 8-22 Peek Instruction Sequence

Development Port Activity	Instruction	Processor Activity	Purpose
Shift in instruction	**mfspr** R1, DPDR	Transfer address from DPDR to R1	Point to memory address
Shift in instruction	**lwzu** R0, D(R1)	Load data from memory address (R1) into R0	Read data from memory
Shift in instruction	**mtspr** DPDR, R0	Transfer data from R0 to DPDR	Write memory data to the port
Shift in instruction, shift out memory data	First instruction of next sequence	Execute next instruction	Output memory data

8.6.4 Poke Instruction Sequence

The poke sequence of instructions is used to write data entered at the development serial port to a memory location. It starts by moving the memory address into R1 from the development port. Next the data is moved into R0 from the development port. Finally, R0 is written to the address in R1.

Table 8-23 Poke Instruction Sequence

Development Port Activity	Instruction	Processor Activity	Purpose
Shift in instruction	**mfspr** R1, DPDR	Transfer address from DPDR to R1	Point to memory address
Shift in instruction, shift in memory data	**mfspr** R0, DPDR	Transfer data from DPDR to R0	Read memory data from the port
Shift in instruction	**stwu** R0, D(R1)	Store data from R0 to memory address (R1)	Write data to memory

8.7 Software Monitor Support

When debug mode is disabled, a software monitor debugger can make use of all of the processor's development support features. With debug mode disabled, all events result in regular exception handling, (i.e., the processor resumes execution in the appropriate exception handler). The ECR and the DER only influence the assertion and negation of the 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 executing, it is possible to assert and negate the internal freeze signal when debug mode is disabled. (The freeze signal can be asserted externally only when the processor enters debug mode.)

The internal freeze signal is asserted whenever an enabled event occurs, regardless of whether debug mode is enabled or disabled. To enable an event to cause freeze assertion, software needs to set the relevant bit in the DER. To clear the freeze signal, software needs to read the ECR to clear the register and then perform an **rfi** instruction.

If the ECR is not cleared before the **rfi** instruction is executed, freeze is not negated. It is therefore possible to nest inside a software monitor debugger without affecting the value of the freeze signal, even though **rfi** is performed. Only before the last **rfi** does the software need to clear the ECR.

Figure 8-14 shows how the ECR and DER control the assertion and negation of the freeze signal and the internal debug mode signal.

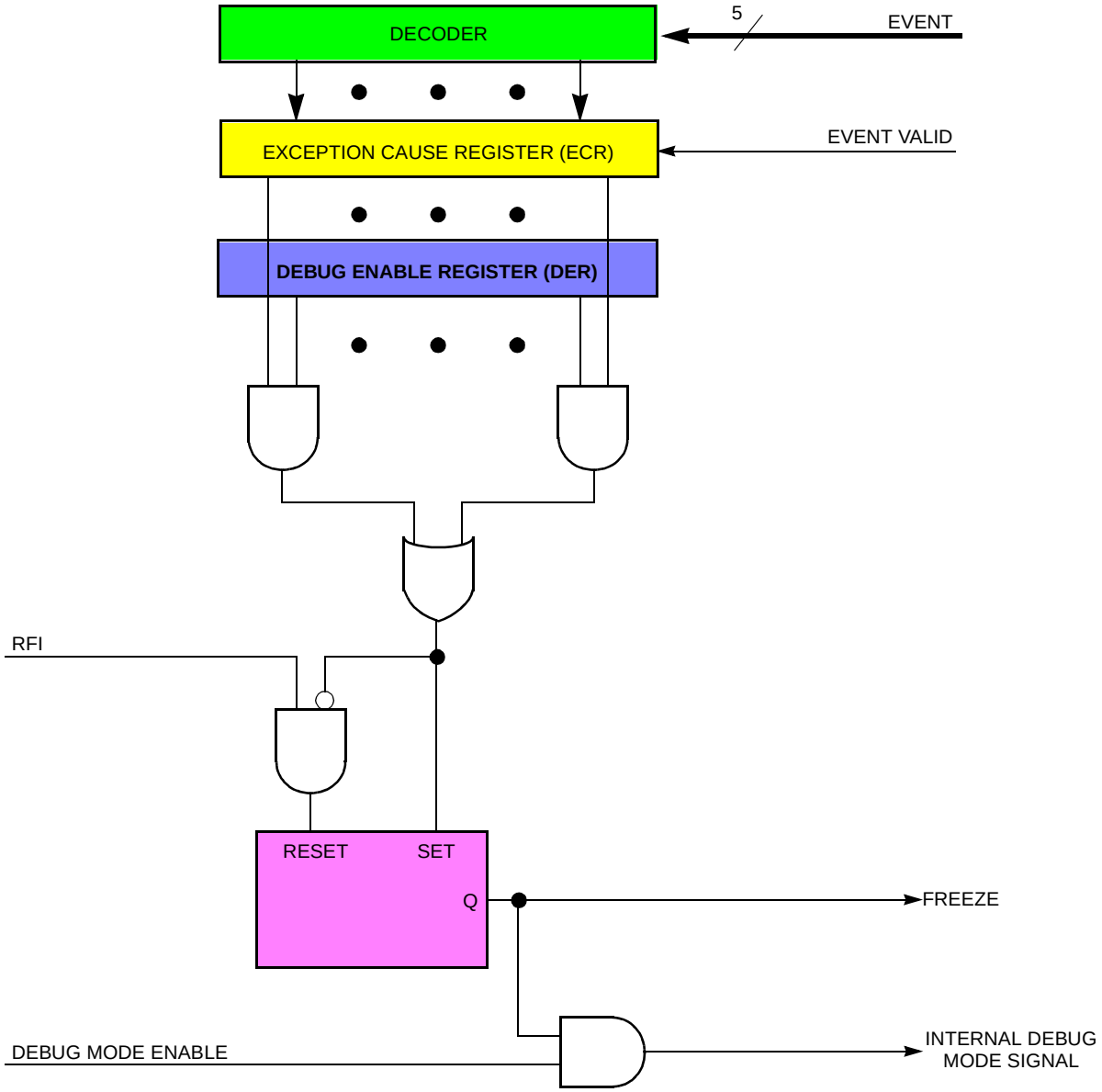


Figure 8-14 Debug Mode Logic

8.8 Development Support Registers

Table 8-24 lists the registers used for development support. The registers are accessed with the **mtspr** and **mfspr** instructions.

Table 8-24 Development Support Programming Model

SPR Number (Decimal)	Mnemonic	Name
144	CMPA	Comparator A value register
145	CMPB	Comparator B value register
146	CMPC	Comparator C value register
147	CMPD	Comparator D value register
148	ECR	Exception cause register
149	DER	Debug enable register
150	COUNTA	Breakpoint counter A value and control register
151	COUNTB	Breakpoint counter B value and control register
152	CMPE	Comparator E value register
153	CMPF	Comparator F value register
154	CMPG	Comparator G value register
155	CMPH	Comparator H value register
156	LCTRL1	L-bus support control register 1
157	LCTRL2	L-bus support control register 2
158	ICTRL	I-bus support control register
159	BAR	Breakpoint address register
630	DPDR	Development port data register

8.8.1 Register Protection

Table 8-25 and **Table 8-26** summarize protection features of development support registers during read and write accesses, respectively.

Table 8-25 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 8-26 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.

8.8.2 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: UNDEFINED

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

RESET: UNDEFINED

Table 8-27 CMPA–CMPD Bit Settings

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

The reset state of these registers is undefined.

8.8.3 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: UNDEFINED

Table 8-28 CMPE–CMPF Bit Settings

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

The reset state of these registers is undefined.

8.8.4 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: UNDEFINED																															

Table 8-29 CMPG–CMPH Bit Settings

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

The reset state of these registers is undefined.

8.8.5 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			IW0		IW1	
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
IW2		IW3		SIW0 EN	SIW1 EN	SIW2 EN	SIW3 EN	DIW0 EN	DIW1 EN	DIW2 EN	DIW3 EN	IIFM	ISCT_SER		
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-30 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	IW0	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	IW2	I-bus 3rd watchpoint programming	0x - not active (reset value) 10 - match from comparator C 11 - match from comparators (C&D)
18:19	IW3	I-bus 4th watchpoint programming	0x - not active (reset value) 10 - match from comparator D 11 - match from comparators (C | D)
20	SIW0EN	Software trap enable selection of the 1st I-bus watchpoint	0 - trap disabled (reset value) 1 - trap enabled
21	SIW1EN	Software trap enable selection of the 2nd I-bus watchpoint	
22	SIW2EN	Software trap enable selection of the 3rd I-bus watchpoint	
23	SIW3EN	Software trap enable selection of the 4th I-bus watchpoint	
24	DIW0EN	Development port trap enable selection of the 1st I-bus watchpoint (read only bit)	0 - trap disabled (reset value) 1 - trap enabled
25	DIW1EN	Development port trap enable selection of the 2nd I-bus watchpoint (read only bit)	
26	DIW2EN	Development port trap enable selection of the 3rd I-bus watchpoint (read only bit)	
27	DIW3EN	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 8-30 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

The ICTRL is cleared following reset.

NOTE

The machine is fetch serialized whenever SER = 0b0 or ISCTL = 0b00.

8.8.6 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 8-31 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.

8.8.7 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 NOM-SK	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 8-32 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

Table 8-32 LCTRL2 Bit Settings (Continued)

Bits	Mnemonic	Description	Function
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	—
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.

8.8.8 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: UNDEFINED															

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 8-33 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 undefined.

8.8.9 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: UNDEFINED															

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 8-34 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 undefined.

8.8.10 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
RESERVED		CHSTP	MCE	DSE	ISE	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										LBRK	IBRK	EBRKD	DPI
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-35 ECR Bit Settings

Bit(s)	Name	Description
0:1	—	Reserved
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	DSE	Data storage exception bit. Set when a machine check exception caused by a data storage error is asserted.
5	ISE	Instruction storage exception bit. Set when a machine check exception caused by an instruction storage error is asserted.
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 is asserted.
16	—	Reserved
17	SEE	Software emulation exception. Set when the software emulation exception is asserted.
18:27		Reserved
28	LBRK	L-bus breakpoint exception bit. Set when an L-bus breakpoint is asserted.
29	IBRK	I-bus breakpoint exception bit. Set when an I-bus breakpoint is asserted.
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).
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.

8.8.11 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
RESERVED		CHSTPE	MCEE	DSEE	ISEE	EXTIE	ALEE	PREE	FPUVEE	DECEE	RESERVED		SYSEE	TRE	FPASEE
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
0	SEEE	RESERVED										LBRKE	IBRKE	EBRKE	DPIE
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Table 8-36 DER Bit Settings

Bit(s)	Name	Description
0:1	—	Reserved
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	DSEE	Data storage exception (type of machine check exception) enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
5	ISEE	Instruction storage exception (type of machine check exception) enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
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

Table 8-36 DER Bit Settings (Continued)

Bit(s)	Name	Description
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
18:27	—	Reserved
28	LBRKE	L-bus breakpoint exception enable bit 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
29	IBRKE	I-bus breakpoint exception enable bit 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
30	EBRKE	External breakpoint exception enable bit 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)