

19 BLACKFIN PROCESSOR DEBUG AND EMULATION

The Blackfin processor's debug functionality is used for software debugging. It also complements some services often found in an operating system (OS) kernel. The functionality is implemented in the processor hardware and is grouped into multiple levels.

A summary of available debug features is shown in [Table 19-1](#).

Table 19-1. Blackfin Debug Features

Debug Feature	Description
Watchpoints	Specify address ranges and conditions that halt the processor when satisfied.
Trace History	Stores the last 16 discontinuous values of the Program Counter in an on-chip trace buffer.
Cycle Count	Provides functionality for all code profiling functions.
Performance Monitoring	Allows internal resources to be monitored and measured non-intrusively.

Watchpoint Unit

By monitoring the addresses on both the instruction bus and the data bus, the Watchpoint Unit provides several mechanisms for examining program behavior. After counting the number of times a particular address is matched, the unit schedules an event based on this count.

Watchpoint Unit

In addition, information that the Watchpoint Unit provides helps in the optimization of code. The unit also makes it easier to maintain executables through code patching.

The Watchpoint Unit contains these memory-mapped registers (MMRs), which are accessible in Supervisor and Emulator modes:

- The Watchpoint Status register (WPSTAT)
- Six Instruction Watchpoint Address registers (WPIA[5:0])
- Six Instruction Watchpoint Address Count registers (WPIACNT[5:0])
- The Instruction Watchpoint Address Control register (WPIACTL)
- Two Data Watchpoint Address registers (WPDA[1:0])
- Two Data Watchpoint Address Count registers (WPDACNT[1:0])
- The Data Watchpoint Address Control register (WPDACTL)

Two operations implement instruction watchpoints:

- The values in the six Instruction Watchpoint Address registers, WPIA[5:0], are compared to the address on the instruction bus.
- Corresponding count values in the Instruction Watchpoint Address Count registers, WPIACNT[5:0], are decremented on each match.

The six Instruction Watchpoint Address registers may be further grouped into three ranges of instruction-address-range watchpoints. The ranges are identified by the addresses in WPIA0 to WPIA1, WPIA2 to WPIA3, and WPIA4 to WPIA5.



The address ranges stored in `WPIA0`, `WPIA1`, `WPIA2`, `WPIA3`, `WPIA4`, and `WPIA5` must satisfy these conditions:

`WPIA0 <= WPIA1`

`WPIA2 <= WPIA3`

`WPIA4 <= WPIA5`

Two operations implement data watchpoints:

- The values in the two Data Watchpoint Address registers, `WPDA[1:0]`, are compared to the address on the data buses.
- Corresponding count values in the Data Watchpoint Address Count registers, `WPDACNT[1:0]`, are decremented on each match.

The two Data Watchpoint Address registers may be further grouped together into one data-address-range watchpoint, `WPDA[1:0]`.

The instruction and data count value registers must be loaded with the number of times the watchpoint must match minus one. After the count value reaches zero, the subsequent watchpoint match results in an exception or emulation event.



Note count values must be reinitialized after the event has occurred.

An event can also be triggered on a combination of the instruction and data watchpoints. If the `WPAND` bit in the `WPIACTL` register is set, then an event is triggered only when both an instruction address watchpoint matches *and* a data address watchpoint matches. If the `WPAND` bit is 0, then an event is triggered when any of the enabled watchpoints or watchpoint ranges match.

Watchpoint Unit

To enable the Watchpoint Unit, the `WPPWR` bit in the `WPIACTL` register must be set. If `WPPWR = 1`, then the individual watchpoints and watchpoint ranges may be enabled using the specific enable bits in the `WPIACTL` and `WPDACTL` MMRs. If `WPPWR = 0`, then all watchpoint activity is disabled.

Instruction Watchpoints

Each instruction watchpoint is controlled by three bits in the `WPIACTL` register, as shown in [Table 19-2](#).

Table 19-2. WPIACTL Control Bits

Bit Name	Description
EMUSW _x	Determines whether an instruction-address match causes either an emulation event or an exception event.
WPICNTEN _x	Enables the 16-bit counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPIAEN _x	Enables the address watchpoint activity.

When two watchpoints are associated to form a range, two additional bits are used, as shown in [Table 19-3](#).

Table 19-3. WPIACTL Watchpoint Range Control Bits

Bit Name	Description
WPIREN _{xy}	Indicates the two watchpoints that are to be associated to form a range.
WPIRINV _{xy}	Determines whether an event is caused by an address within the range identified or outside of the range identified.

Code patching allows software to replace sections of existing code with new code. The watchpoint registers are used to trigger an exception at the start addresses of the earlier code. The exception routine then vectors to the location in memory that contains the new code.

On the processor, code patching can be achieved by writing the start address of the earlier code to one of the `WPIAn` registers and setting the corresponding `EMUSWx` bit to trigger an exception. In the exception service routine, the `WPSTAT` register is read to determine which watchpoint triggered the exception. Next, the code writes the start address of the new code in the `RETX` register, and then returns from the exception to the new code. Because the exception mechanism is used for code patching, event service routines of the same or higher priority (exception, NMI, and reset routines) cannot be patched.

A write to the `WPSTAT` MMR clears all the sticky status bits. The data value written is ignored.

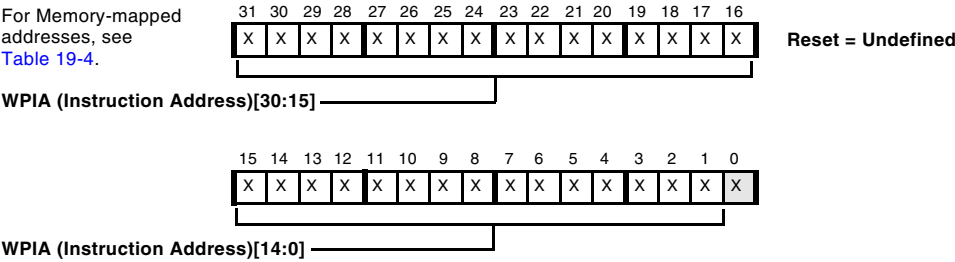
Instruction Watchpoint Address Registers (WPIAn)

When the Watchpoint Unit is enabled, the values in the `WPIAn` registers are compared to the address on the instruction bus. Corresponding count values in the Instruction Watchpoint Address Count registers (`WPIACNTn`) are decremented on each match.

Figure 19-1 shows the Instruction Watchpoint Address registers, `WPIA[5:0]`.

Watchpoint Unit

Instruction Watchpoint Address Registers (WPIAn)



Instruction Watchpoint Address Count Registers (WPIACNTn)

For Memory-mapped addresses, see [Table 19-5](#).

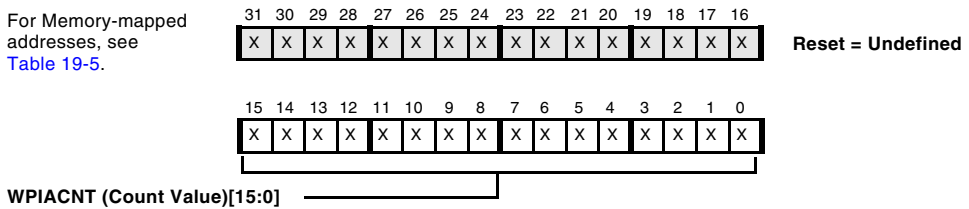


Figure 19-2. Instruction Watchpoint Address Count Registers

Table 19-5. Instruction Watchpoint Address Count Register Memory-mapped Addresses

Register Name	Memory-mapped Address
WPIACNT0	0xFFE0 7080
WPIACNT1	0xFFE0 7084
WPIACNT2	0xFFE0 7088
WPIACNT3	0xFFE0 708C
WPIACNT4	0xFFE0 7090
WPIACNT5	0xFFE0 7094

Instruction Watchpoint Address Control Register (WPIACTL)

Three bits in the Instruction Watchpoint Address Control register (WPIACTL) control each instruction watchpoint. [Figure 19-3](#) describes the upper half of the register. [Figure 19-4 on page 19-9](#) describes the lower half of the register. For more information about the bits in this register, see [“Instruction Watchpoints” on page 19-4](#).



The bits in the WPIACTL register have no effect unless the WPPWR bit is set.

Watchpoint Unit

Instruction Watchpoint Address Control Register (WPIACTL)

In range comparisons, IA = instruction address

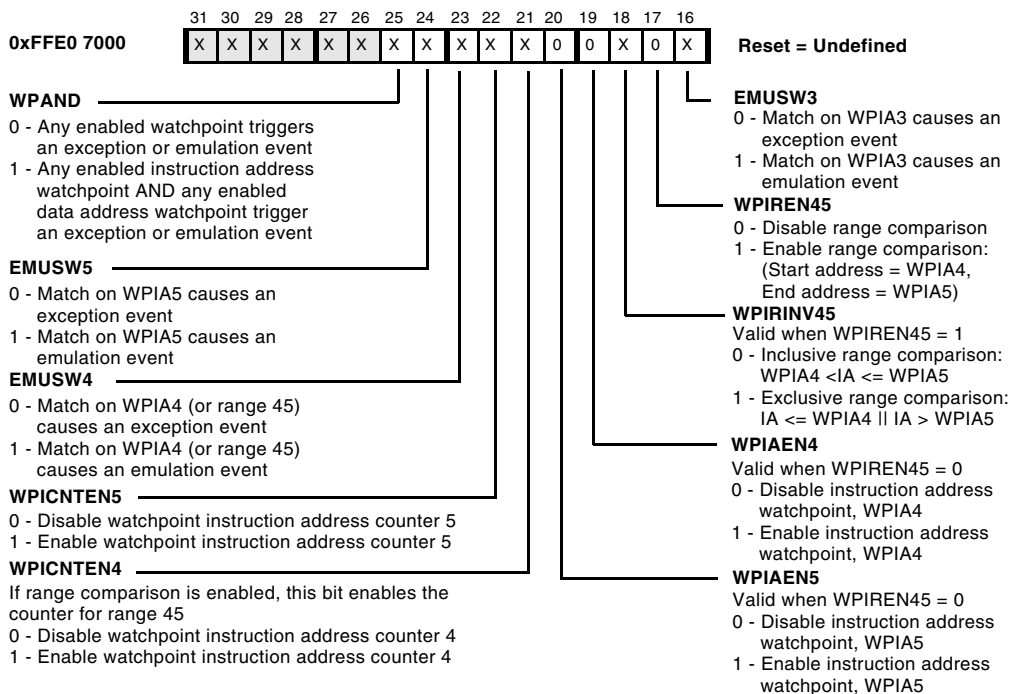


Figure 19-3. Instruction Watchpoint Address Control Register (WPIACTL)[31:16]

Instruction Watchpoint Address Control Register (WPIACTL)

In range comparisons, IA = instruction address

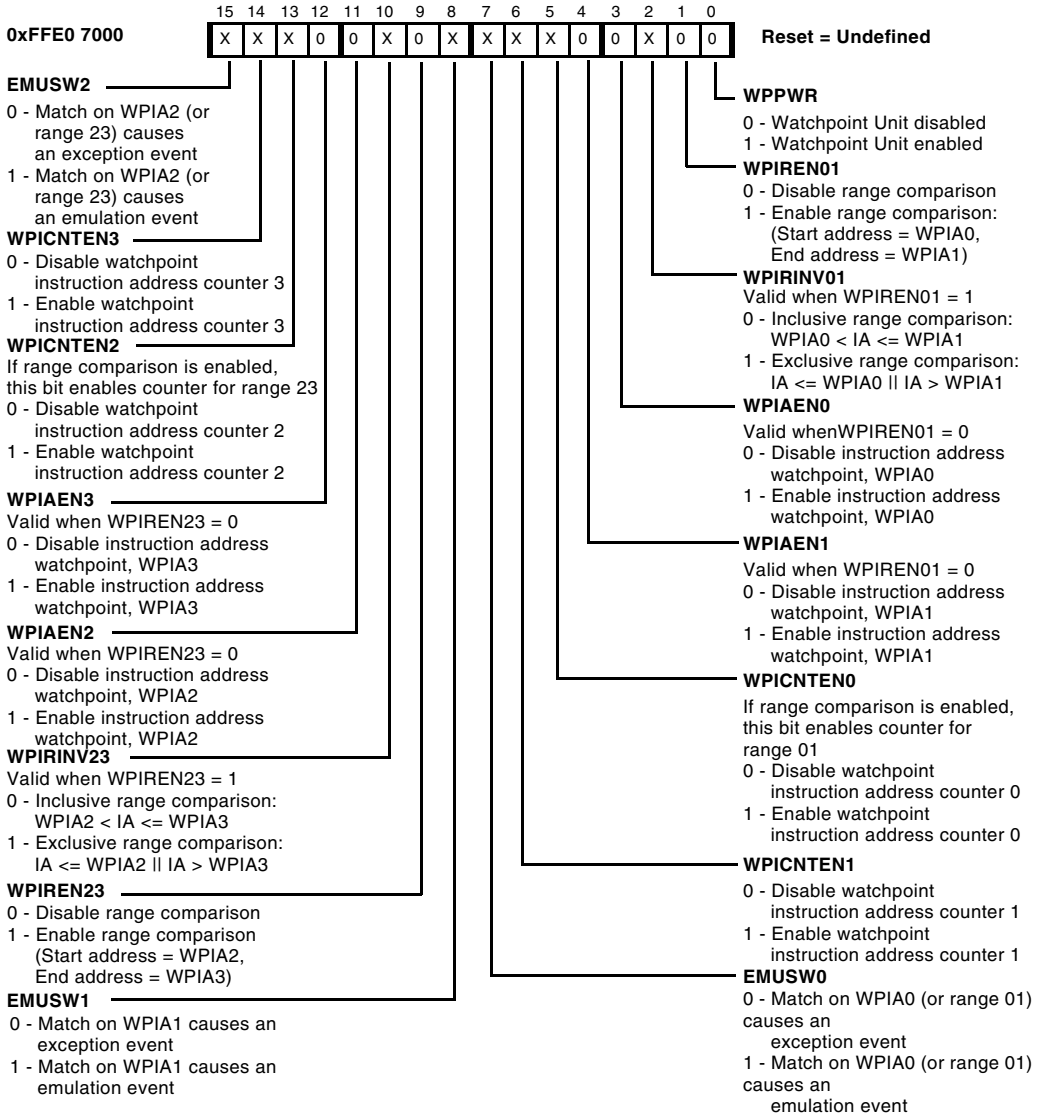


Figure 19-4. Instruction Watchpoint Address Control Register (WPIACTL)[15:0]

Data Address Watchpoints

Each data watchpoint is controlled by four bits in the WPDACTL register, as shown in [Table 19-6](#).


Table 19-6. Data Address Watchpoints

Bit Name	Description
WPDACC _n	Determines whether the match should be on a read or write access.
WPDSRC _n	Determines which DAG the unit should monitor.
WPDCTEN _n	Enables the counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPDAEN _n	Enables the data watchpoint activity.

When the two watchpoints are associated to form a range, two additional bits are used. See [Table 19-7](#).

Table 19-7. WPDACTL Watchpoint Control Bits

Bit Name	Description
WPDREN01	Indicates the two watchpoints associated to form a range.
WPDRIINV01	Determines whether an event is caused by an address within the range identified or outside the range.

 Note data address watchpoints always trigger emulation events.

Data Watchpoint Address Registers (WPDAn)

When the Watchpoint Unit is enabled, the values in the WPDAn registers are compared to the address on the data buses. Corresponding count values in the Data Watchpoint Address Count registers (WPDACNTn) are decremented on each match.

Figure 19-5 shows the Data Watchpoint Address registers, WPDA[1:0].

Data Watchpoint Address Registers (WPDAn)

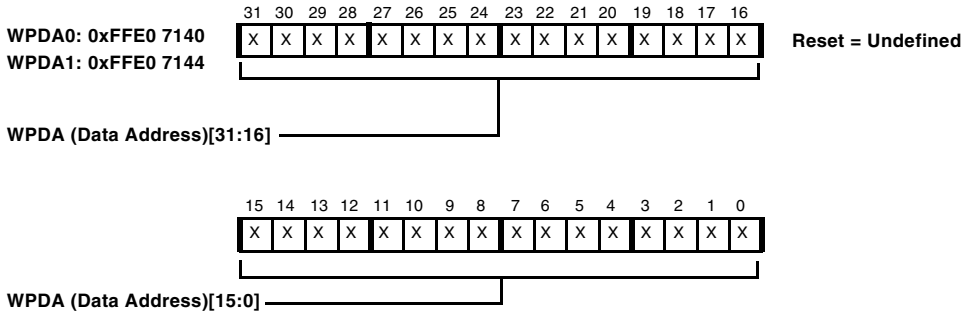


Figure 19-5. Data Watchpoint Address Registers

Data Watchpoint Address Count Value Registers (WPDACNTn)

When the Watchpoint Unit is enabled, the count values in these registers are decremented each time the address or the address bus matches a value in the WPDAn registers. Load this WPDACNTn register with a value that is one less than the number of times the watchpoint must match before triggering an event. The WPDACNTn register will decrement to 0x0000 when the programmed count expires. [Figure 19-6](#) shows the Data Watchpoint Address Count Value registers, WPDACNT[1:0].

Data Watchpoint Address Count Value Registers (WPDACNTn)

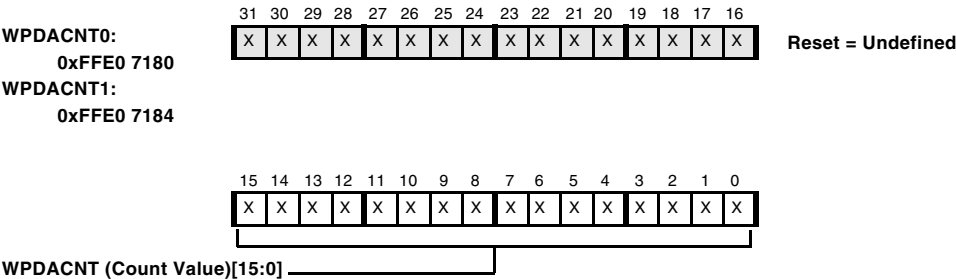


Figure 19-6. Data Watchpoint Address Count Value Registers

Data Watchpoint Address Control Register (WPDACTL)

For more information about the bits in the WPDACTL register, see [“Data Address Watchpoints” on page 19-10](#).

Data Watchpoint Address Control Register (WPDACCTL)

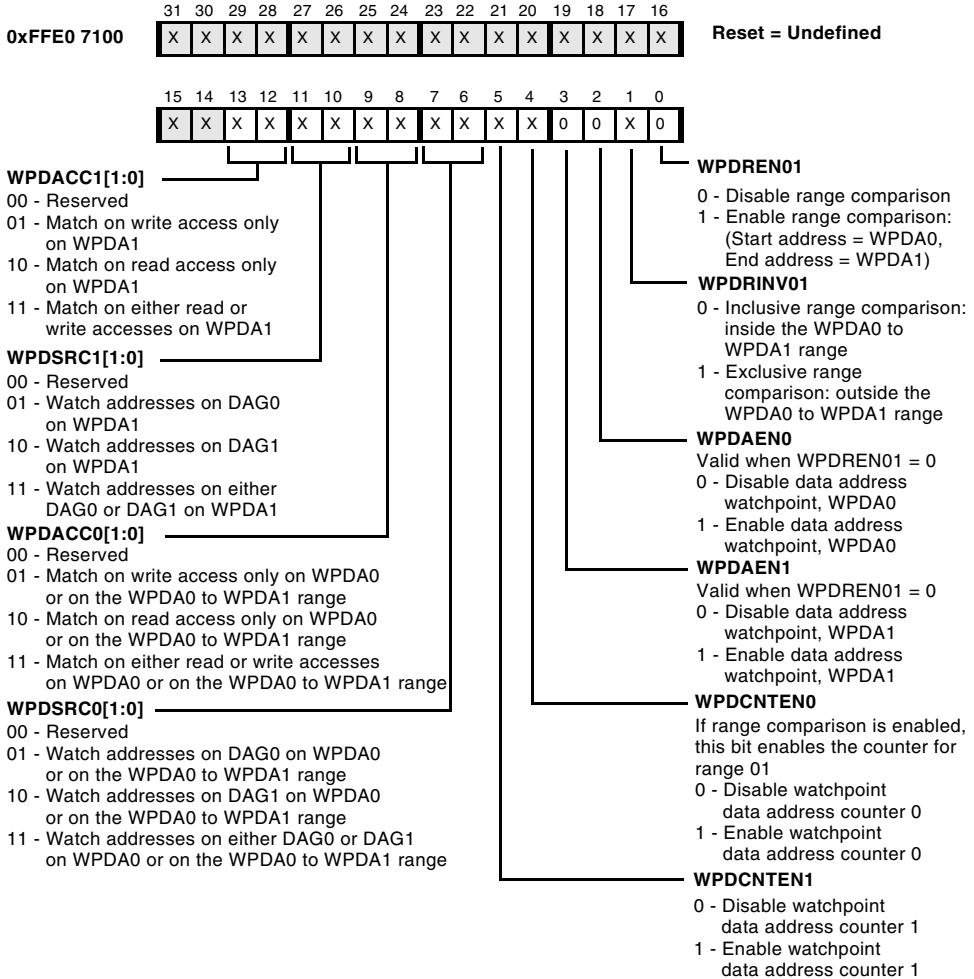


Figure 19-7. Data Watchpoint Address Control Register

Watchpoint Status Register (WPSTAT)

The WPSTAT register monitors the status of the watchpoints. It may be read and written in Supervisor or Emulator modes only. When a watchpoint or watchpoint range matches, this register reflects the source of the watchpoint. The status bits in the WPSTAT register are sticky, and all of them are cleared when any write, regardless of the value, is performed to the register.

Figure 19-8 shows the Watchpoint Status register.

Watchpoint Status Register (WPSTAT)

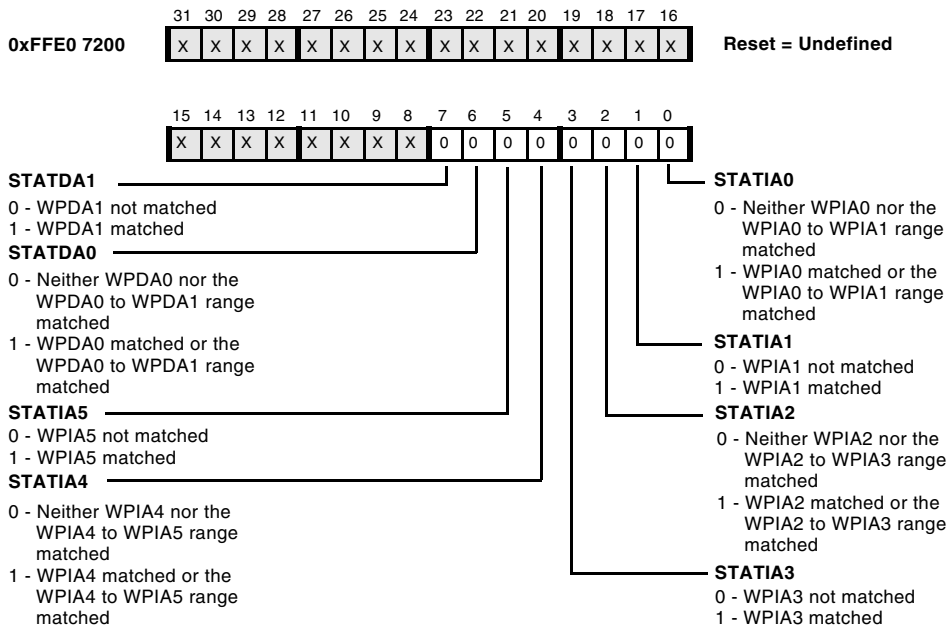


Figure 19-8. Watchpoint Status Register

Trace Unit

The Trace Unit stores a history of the last 16 changes in program flow taken by the program sequencer. The history allows the user to recreate the program sequencer's recent path.

The trace buffer can be enabled to cause an exception when full. The exception service routine associated with the exception saves trace buffer entries to memory. Thus, the complete path of the program sequencer since the trace buffer was enabled can be recreated.


Changes in program flow because of zero-overhead loops are not stored in the trace buffer. For debugging code that is halted within a zero-overhead loop, the iteration count is available in the Loop Count registers, `LC0` and `LC1`.


The trace buffer can be configured to omit the recording of changes in program flow that match either the last entry or one of the last two entries. Omitting one of these entries from the record prevents the trace buffer from overflowing because of loops in the program. Because zero-overhead loops are not recorded in the trace buffer, this feature can be used to prevent trace overflow from loops that are nested four deep.

When read, the Trace Buffer register (`TBUF`) returns the top value from the Trace Unit stack, which contains as many as 16 entries. Each entry contains a pair of branch source and branch target addresses. A read of `TBUF` returns the newest entry first, starting with the branch destination. The next read provides the branch source address.

Trace Unit

The number of valid entries in `TBUF` is held in the `TBUFCNT` field of the `TBUFSTAT` register. On every second read, `TBUFCNT` is decremented. Because each entry corresponds to two pieces of data, a total of $2 \times \text{TBUFCNT}$ reads empties the `TBUF` register.

 Discontinuities that are the same as either of the last two entries in the trace buffer are not recorded.

 Because reading the trace buffer is a destructive operation, it is recommended that `TBUF` be read in a non-interruptible section of code.

Note, if single-level compression has occurred, the least significant bit (LSB) of the branch target address is set. If two-level compression has occurred, the LSB of the branch source address is set.

Trace Buffer Control Register (TBUFCTL)

The Trace Unit is enabled by two control bits in the Trace Buffer Control register (`TBUFCTL`) register. First, the Trace Unit must be activated by setting the `TBUFPWR` bit. If `TBUFPWR` = 1, then setting `TBUFEN` to 1 enables the Trace Unit.

[Figure 19-9](#) describes the Trace Buffer Control register (`TBUFCTL`). If `TBUFOVF` = 1, then the Trace Unit does not record discontinuities in the exception, NMI, and reset routines.

Trace Buffer Control Register (TBUFCTL)

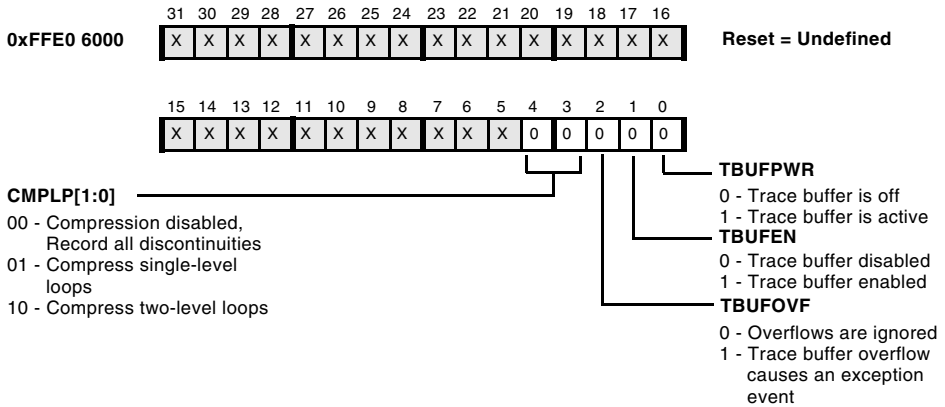


Figure 19-9. Trace Buffer Control Register

Trace Buffer Status Register (TBUFSTAT)

Figure 19-10 shows the Trace Buffer Status register (TBUFSTAT). Two reads from TBUF decrements TBUFCNT by one.

Trace Buffer Status Register (TBUFSTAT)

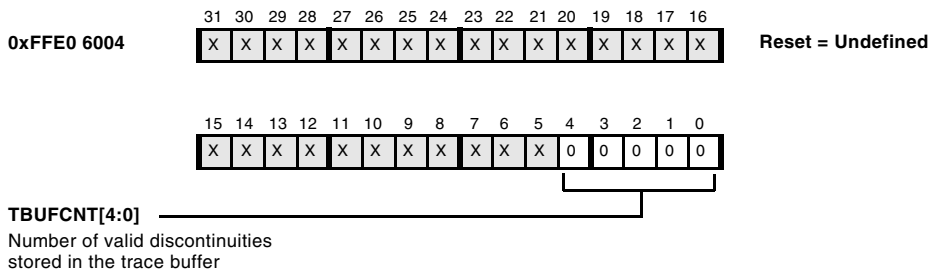


Figure 19-10. Trace Buffer Status Register

Trace Buffer Register (TBUF)

Figure 19-11 shows the Trace Buffer register (TBUF). The first read returns the latest branch target address. The second read returns the latest branch source address.

Trace Buffer Register (TBUF)

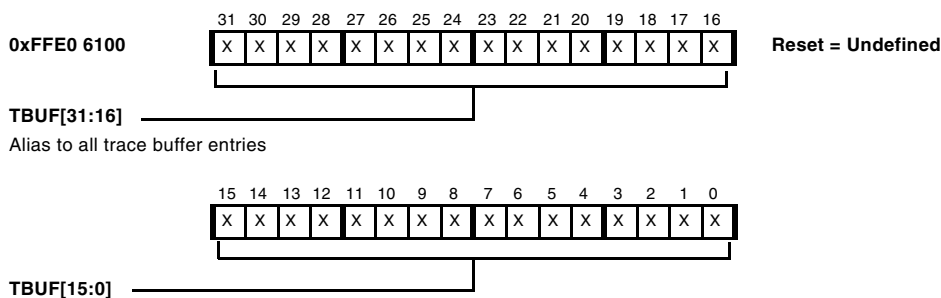


Figure 19-11. Trace Buffer Register

The Trace Unit does not record changes in program flow in:

- Emulator mode
- The exception or higher priority service routines (if $TBUF_{OVF} = 1$)

In the exception service routine, the program flow discontinuities may be read from TBUF and stored in memory by the code shown in Listing 19-1.



While TBUF is being read, be sure to disable the trace buffer from recording new discontinuities.

Code to Recreate the Execution Trace in Memory

Listing 19-1 provides code that recreates the entire execution trace in memory.

Listing 19-1. Recreating the Execution Trace in Memory

```
[--sp] = (r7:7, p5:2); /* save registers used in this routine */
p5 = 32; /* 32 reads are needed to empty TBUF */
    p2.l = buf; /* pointer to the header (first location) of the
        software trace buffer */
    p2.h = buf; /* the header stores the first available empty buf
        location for subsequent trace dumps */

p4 = [p2++]; /* get the first available empty buf location from
the buf header */
p3.l = TBUF & 0xffff; /* low 16 bits of TBUF */
p3.h = TBUF >> 16; /* high 16 bits of TBUF */

lsetup(loop1_start, loop1_end) lc0 = p5;
loop1_start: r7 = [p3]; /* read from TBUF */
loop1_end: [p4++] = r7; /* write to memory and increment */
[p2] = p4; /* pointer to the next available buf location is
saved in the header of buf */
(r7:7, p5:3) = [sp++]; /* restore saved registers */
```

Performance Monitoring Unit

Two 32-bit counters, the Performance Monitor Counter registers (PFCNTR[1:0]) and the Performance Control register (PFCTL), count the number of occurrences of an event from within a processor core unit during a performance monitoring period. These registers provide feedback indicating the measure of load balancing between the various resources on the chip so that expected and actual usage can be compared and analyzed. In addition, events such as mispredictions and hold cycles can also be monitored.

Performance Monitor Counter Registers (PFCNTRn)

Figure 19-12 shows the Performance Monitor Counter registers, PFCNTR[1:0]. The PFCNTR0 register contains the count value of performance counter 0. The PFCNTR1 register contains the count value of performance counter 1.

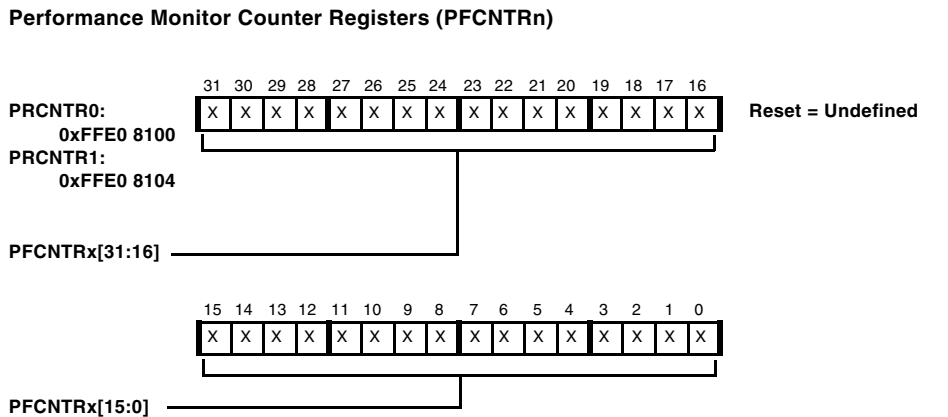


Figure 19-12. Performance Monitor Counter Registers

Performance Monitor Control Register (PFCTL)

To enable the Performance Monitoring Unit, set the PFPWR bit in the PFCTL register (see [Figure 19-13](#)). Once the unit is enabled, individual count-enable bits (PFCENn) take effect. Use the PFCENx bits to enable or disable the performance monitors in User mode, Supervisor mode, or both. Use the PEMUSWx bits to select the type of event triggered.

Performance Monitor Control Register (PFCTL)

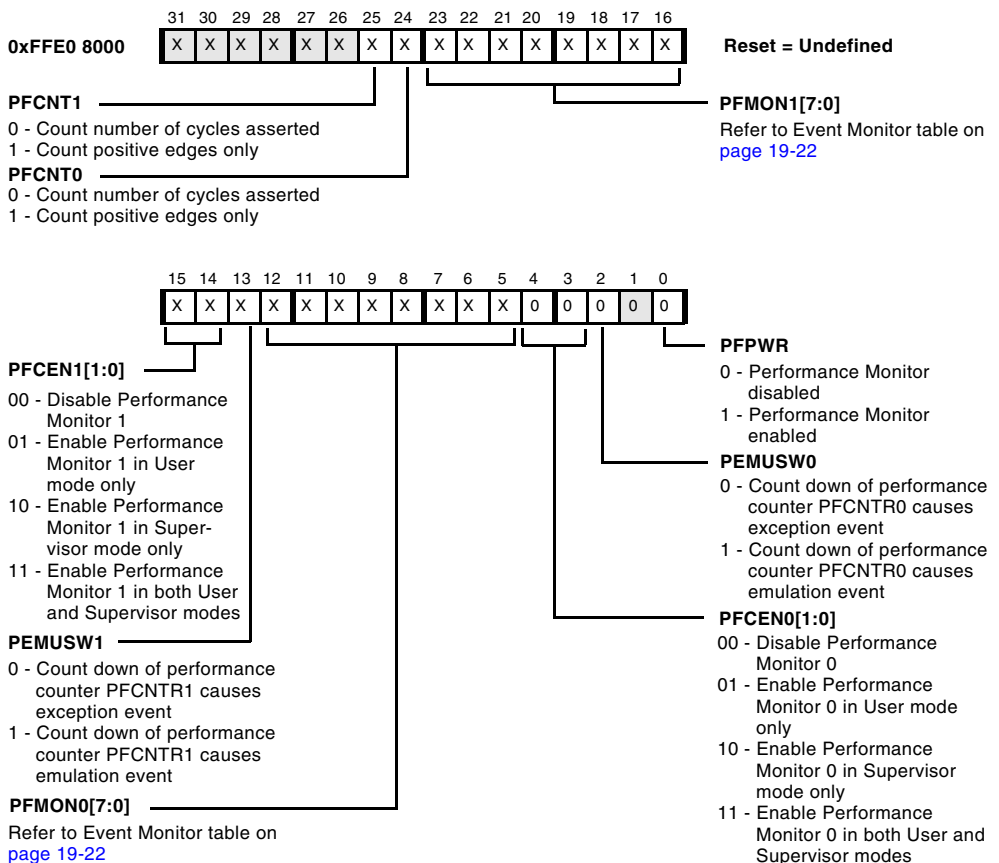


Figure 19-13. Performance Monitor Control Register

Event Monitor Table

Table 19-8 identifies events that cause the Performance Monitor Counter registers (PFMON0 or PFMON1) to increment.

Table 19-8. Event Monitor Table

PFMONx Fields	Events That Cause the Count Value to Increment
0x00	Loop 0 iterations
0x01	Loop 1 iterations
0x02	Loop buffer 0 not optimized
0x03	Loop buffer 1 not optimized
0x04	PC invariant branches (requires trace buffer to be enabled, see “Trace Buffer Control Register (TBUFCTL)” on page 19-16)
0x06	Conditional branches
0x09	Total branches including calls, returns, branches, but not interrupts (requires trace buffer to be enabled, see “Trace Buffer Control Register (TBUFCTL)” on page 19-16)
0x0A	Stalls due to CSYNC, SSYNC
0x0B	EXCPT instructions
0x0C	CSYNC, SSYNC instructions
0x0D	Committed instructions
0x0E	Interrupts taken
0x0F	Misaligned address violation exceptions
0x10	Stall cycles due to read after write hazards on DAG registers
0x13	Stall cycles due to RAW data hazards in computes
0x80	Code memory fetches postponed due to DMA collisions (minimum count of two per event)
0x81	Code memory TAG stalls (cache misses, or FlushI operations, count of 3 per FlushI). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.


Table 19-8. Event Monitor Table (Cont'd)

PFMONx Fields	Events That Cause the Count Value to Increment
0x82	Code memory fill stalls (cacheable or non-cacheable). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.
0x83	Code memory 64-bit words delivered to processor instruction assembly unit
0x90	Processor stalls to memory
0x91	Data memory stalls to processor not hidden by processor stall
0x92	Data memory store buffer full stalls
0x93	Data memory write buffer full stalls due to high-to-low priority code transition
0x94	Data memory store buffer forward stalls due to lack of committed data from processor
0x95	Data memory fill buffer stalls
0x96	Data memory array or TAG collision stalls (DAG to DAG, or DMA to DAG)
0x97	Data memory array collision stalls (DAG to DAG or DMA to DAG)
0x98	Data memory stalls
0x99	Data memory stalls sent to processor
0x9A	Data memory cache fills completed to Bank A
0x9B	Data memory cache fills completed to Bank B
0x9C	Data memory cache victims delivered from Bank A
0x9D	Data memory cache victims delivered from Bank B
0x9E	Data memory cache high priority fills requested
0x9F	Data memory cache low priority fills requested

Cycle Counter

The cycle counter counts `CCLK` cycles while the program is executing. All cycles, including execution, wait state, interrupts, and events, are counted while the processor is in User or Supervisor mode, but the cycle counter stops counting in Emulator mode.

The cycle counter is 64 bits and increments every cycle. The count value is stored in two 32-bit registers, `CYCLES` and `CYCLES2`. The least significant 32 bits (LSBs) are stored in `CYCLES`. The most significant 32 bits (MSBs) are stored in `CYCLES2`.

 To ensure the correct cycle count, a read from `CYCLES` must occur before reading from `CYCLES2`.

In User mode, these two registers may be read, but not written. In Supervisor and Emulator modes, they are read/write registers.

To enable the cycle counters, set the `CCEN` bit in the `SYSCFG` register. The following example shows how to use the cycle counter:

```
R2 = 0;
CYCLES = R2;
CYCLES2 = R2;
R2 = SYSCFG;
BITSET(R2,1);
SYSCFG = R2;
/* Insert code to be benchmarked here. */
R2 = SYSCFG;
BITCLR(R2,1);
SYSCFG = R2;
```


Execution Cycle Count Registers (CYCLES and CYCLES2)

The Execution Cycle Count registers, shown in [Figure 19-14](#), consist of `CYCLES` and `CYCLES2`. This 64-bit counter increments every `CCLK` cycle. The `CYCLES` register contains the least significant 32 bits of the cycle counter's 64-bit count value. The most significant 32 bits are contained by `CYCLES2`.



The `CYCLES` and `CYCLES2` registers are not system MMRs, but are instead system registers. Please see [page 2-9](#) for a full listing of system registers.

Execution Cycle Count Registers (CYCLES and CYCLES2)

RO in User mode, RW in Supervisor and Emulator modes

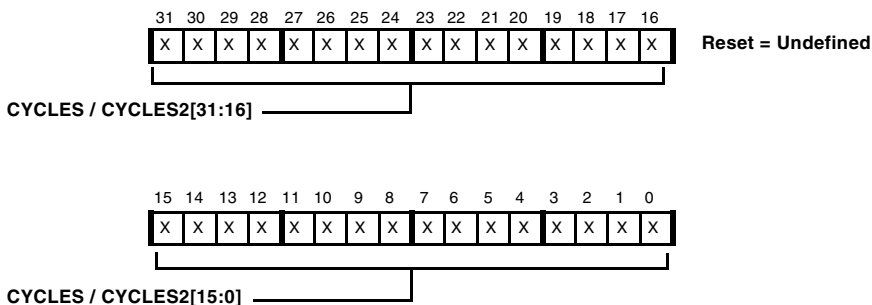


Figure 19-14. Execution Cycle Count Registers

Product Identification Registers

The 32-bit Chip ID register (`CHIPID`) is a system MMR that contains the product identification and revision fields for the ADSP-BF532. The 32-bit DSP Device ID register (`DSPID`) is a core MMR that contains core identification and revision fields for the core.

Chip ID Register (CHIPID)

The Chip ID register (CHIPID) is a read-only register. The definition of this register complies with Joint Electronic Device Engineering Council (JEDEC) standard JEP106, Standard Manufacturer’s Identification Code, as follows:

- CHIPID[31:28]: silicon revision number
- CHIPID[27:12]: part number 0x27A5
- CHIPID[11:1]: Analog Devices, Inc., ID number 0xE5 – drop the MSB (parity) and set MSBs to 0 (no ID extension) in accordance with the JTAG standard: 0x065.
- CHIPID[0]: set to 1

Chip ID Register (CHIPID)
RO

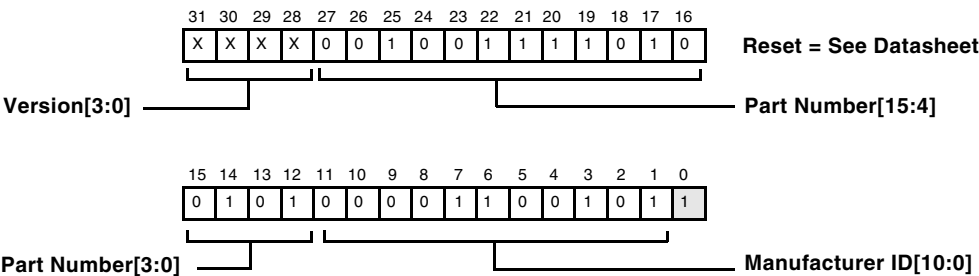


Figure 19-15. Chip ID Register

DSP Device ID Register (DSPID)

The DSP Device ID register (DSPID), shown in [Figure 19-16](#), is a read-only register and is part of the core.

DSP Device ID Register (DSPID)

RO

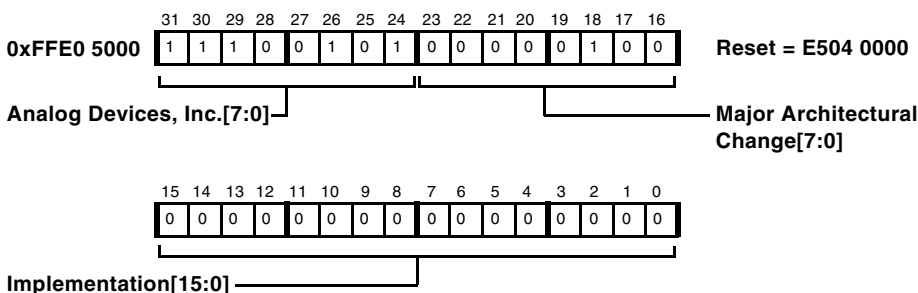


Figure 19-16. DSP Device ID Register

Debug Control Register (DBGCTL)

The Debug Control register (DBGCTL) contains control bits that enable the debug and emulation features; it is accessible only through the JTAG DBGCTL chain. Although Watchpoint, Trace, and Performance Monitor Units have separate control registers, events such as watchpoint match or external emulation exceptions are ignored unless the DBGCTL EMFEN and EMPWR bits are set (see [Figure 19-19](#), “Debug Control Register,” on page 19-34).

When the Test Access Port (TAP) controller is reset, the EMFEN and EMPWR bits are cleared.

Product Identification Registers

Debug Control Register (DBGCTL)

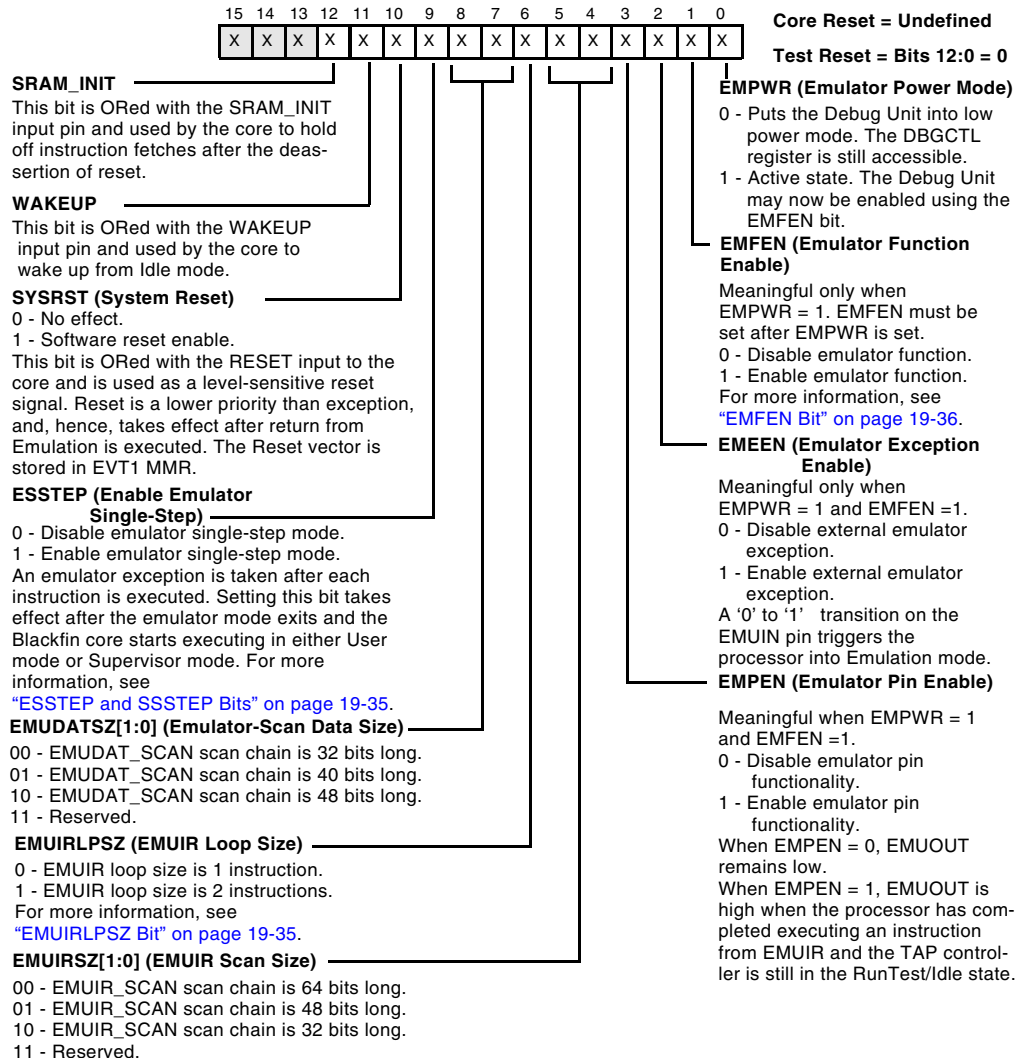


Figure 19-17. Debug Control Register

ESSTEP and SSSTEP Bits

The Emulator Single Step (ESSTEP) and the Supervisor Single Step (SSSTEP) bits determine whether either an Emulation exception or a Supervisor exception is taken. If the SSSTEP bit is set in the SYSCFG register, a Supervisor exception is taken after executing each instruction in the User and Interrupt Service Routine (ISR) code. In the Exception and higher-priority event service routines, the SSSTEP bit is ignored.

If both ESSTEP and SSSTEP are active, a Supervisor exception is taken after executing each instruction of the User and ISR coded, then an Emulation exception is taken after executing each instruction in the exception, NMI, or reset service routines. This feature allows the emulator to single step through the Supervisor Single Step code.

EMUIRLPSZ Bit

The EMUIR Loop Size (EMUIRLPSZ) bit in the DBGCTL register determines the size of the two-entry stack formed by registers EMUIR_A and EMUIR_B. The two-entry stack can be configured as a single instruction or as a two-instruction loop. Each time the TAP controller enters the Run-Test/Idle state, EMUIRLPSZ also determines whether EMUIR_A or EMUIR_B supplies the instruction to be executed, as follows:

- If EMUIRLPSZ = 0, the instruction in EMUIR_A is executed if valid. If the instruction is invalid, then a NOP is issued.
- If EMUIRLPSZ = 1, the instruction in EMUIR_A is executed if valid; then the instruction in EMUIR_B is executed if it is valid. If either instruction is invalid, a NOP is issued in its place

While the processor executes instructions from EMUIR, the EMUREADY bit in the DBGCTL register is cleared. After the instructions execute, the EMUREADY bit is set again.

EMFEN Bit

When the Emulator Function Enable (EMFEN) bit is set to 0, the EMUEXCPT instruction triggers an illegal instruction exception; Watchpoint matches or Performance Monitor events that are configured to invoke Emulation mode are ignored. When the EMFEN and EMPWR bits are both set to 1, the EMUEXCPT instruction invokes Emulation mode.

Debug Status Register (DBGSTAT)

The Debug Status register (DBGSTAT) is a pseudo read-only status register that is accessible through both the JTAG and the MMR interfaces. The two sticky bits of the DBGSTAT register (EMUDIOVF and EMUDOOVF) can be cleared by writing 1s to the corresponding bits.

Debug Status Register (DBGSTAT)

Core Reset = b#0000 0000 0000 0000 00001 0XXX XXX0 XXXX

Test Reset = b#0000 0000 0000 0000 0XXX 0XXX XXXX 0000

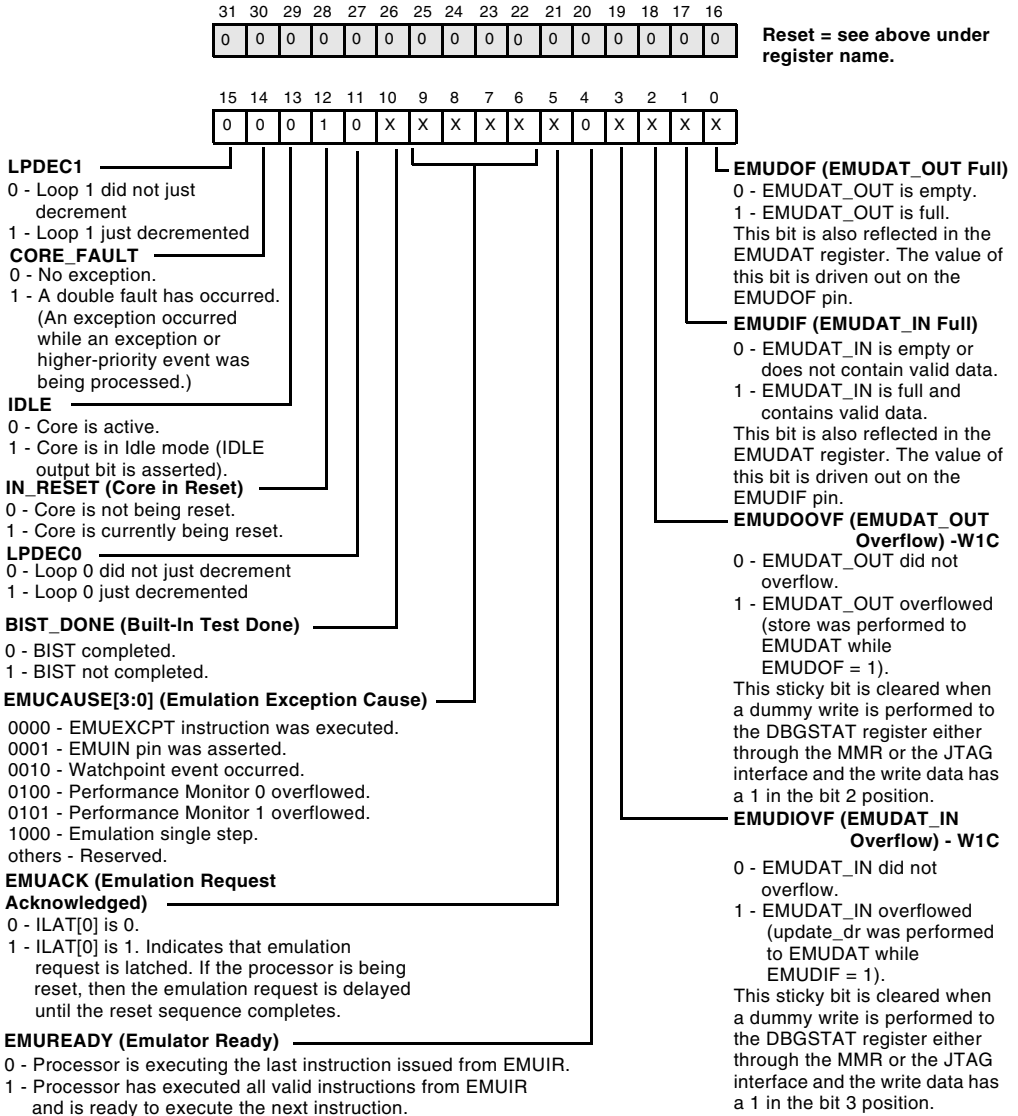


Figure 19-18. Debug Status Register

Emulation Data Register (EMUDAT)

The Emulation Data register (EMUDAT) is a complex register that is used for data exchange through JTAG. EMUDAT consists of two 32-bit registers, EMUDAT_IN and EMUDAT_OUT, that map to the same memory address and share the same JTAG scan chain, EMUDAT_SCAN, to allow full-duplex data transfers. EMUDAT_SCAN may be configured as a 32-bit scan chain, a 40-bit scan chain, or as a 48-bit scan chain, using the EMUDATSZ bit in the DBGCTL register.

Stores from the Blackfin core to the EMUDAT MMR are written to EMUDAT_OUT. Loads to the Blackfin core from the EMUDAT MMR are read from EMUDAT_IN. EMUDAT is a special register, because loads from it do not return the value that the previous store wrote in EMUDAT unless the JTAG TAP intervenes with a capture_dr followed immediately with an update_dr. Full duplex data transfers are accomplished by cycling through capture_dr, shift_dr, and then update_dr.

Status bits EMUDOF and EMUDIF in the DBGSTAT register indicate whether EMUDAT_OUT and EMUDAT_IN are full, respectively:

- EMUDOF is set by a store to the EMUDAT register; it is cleared by a capture_dr on the EMUDAT_SCAN scan chain.
- EMUDIF is set by an update_dr operation if the corresponding bit is set in the EMUDAT_SCAN scan chain; it is cleared by a load from the EMUDAT register.

The EMUDAT_SCAN scan chain includes the EMUDOF and EMUDIF flag bits. These bits are part of the DBGSTAT register and are duplicated in the EMUDAT_SCAN scan chain to speed up the data upload/download process.

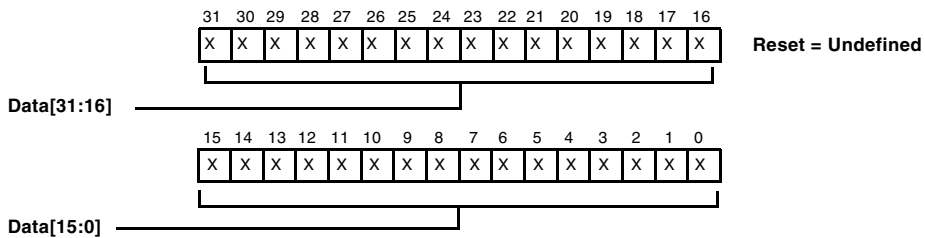
When configured as a 40- or 48-bit register, the EMUDIF bit serves as a write enable for the JTAG to write the EMUDAT_IN register on an update_dr. 1 enables the write, 0 disables the write.

When configured as a 32-bit register, the JTAG interface will always write the EMUDAT_IN register on an update_dr.

Emulation Data Register (EMUDAT)

Consists of two 32-bit registers, EMUDAT_IN and EMUDAT_OUT, that map to the same memory address and share the same JTAG scan chain. The scan chain may be configured as a 32-bit scan chain, a 40-bit scan chain, or a 48-bit scan chain.

When configured as a 32-bit register.



When configured as a 40-bit register

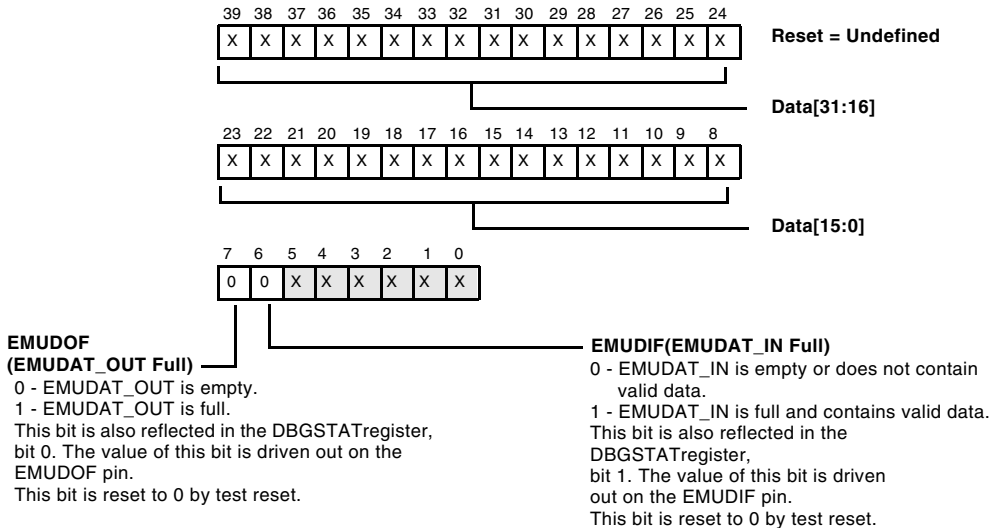


Figure 19-19. Emulation Data Register, 32- and 40-bit Configurations

Product Identification Registers

Emulation Data Register (EMUDAT)

When configured as a 48-bit register

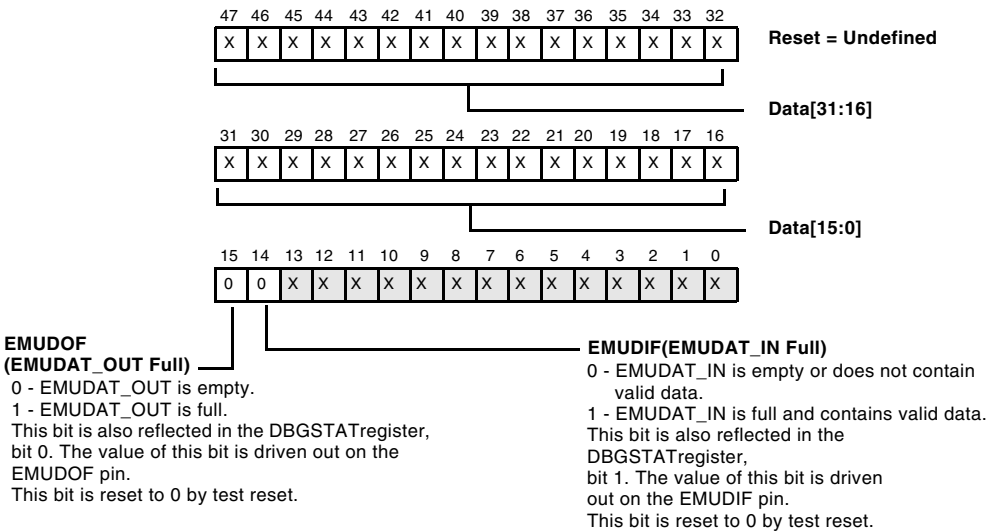


Figure 19-20. Emulation Data Register, 48-bit Configuration

Emulation Instruction Register (EMUIR)

The Emulation Instruction register (EMUIR) is accessible only through the JTAG EMUIR_SCAN scan chain. The MMR interface does not provide access to this register, which consists of a 64-bit scan that can optionally be configured as a 32- or 48-bit scan chain. Bits in the DBGCTL register define the EMUIR loop size and scan size, as follows:

- EMUIRLPSZ, Bit 6, defines the EMUIR loop size.
- EMUIRSZ, Bits [5:4], define the size of the EMUIR scan chain.

The two-entry stack may be configured as a single instruction or as a two-instruction loop, using the `EMUIRLPSZ` bit in the `DBGCTL` register. For information about the two-entry stack, see “`EMUIRLPSZ` Bit” on page 19-35.

When configured as a two instruction loop, each time the JTAG TAP controller enters the `update_dr` state, the contents of the `EMUIR_SCAN` scan chain are pushed into the two-entry stack formed by registers `EMUIR_A` and `EMUIR_B`. The first instruction scanned in is stored in `EMUIR_B` and the second instruction scanned in is stored in `EMUIR_A`.



If more than two instructions are scanned, the processor behavior is unpredictable.

When configured as a single instruction, instructions are always scanned into `EMUIR_A`.

The instructions issued from the `EMUIR` register bypass the `Icache` and `Align` units and are sent directly to the decode unit. In Emulator mode, pipelining is disabled. Each instruction completes execution before the next instruction begins. Each time the TAP controller enters the `update-dr` state while the `EMUIR_SCAN` is selected as the JTAG instruction, the instruction Pointer resets to the first instruction (stored in `EMUIR_A`).

Hardware loops and branches are not supported in the Emulation mode. If these instructions are issued from the `EMUIR` register, the processor behavior is unpredictable.

Figure 19-23 describes the `EMUIR` register.

Product Identification Registers

Emulation Instruction Register (EMUIR)

Consists of a 64-bit scan chain that can optionally be configured as a 32-bit or a 48-bit scan chain. The size of the EMUIR scan chain is controlled by the EMUIRSZ bits in the DBGCTL register. When configured as a 32-bit register.

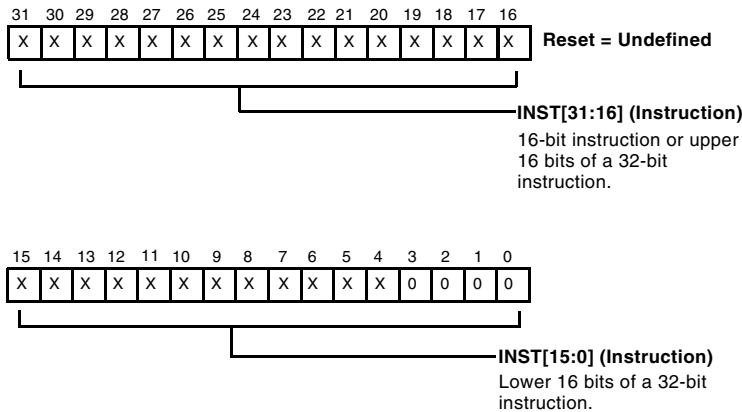


Figure 19-21. Emulation Instruction Register Configured as a 32-bit Register

Examples of EMUIR and EMUDAT Use

The following sequence of operations illustrates a possible use of the EMUIR and EMUDAT registers.

1. Activate the Emulator Unit by setting the EMPWR bit in the DBGCTL register.
2. Enable the emulator functionality (set the EMFEN bit) by scanning the DBGCTL register. Set the EMUIRSZ bits to select the 32-bit scan chain and set the EMUIRLPSZ bits to configure the EMUIR stack as a two-instruction loop.

Emulation Instruction Register (EMUIR)

Consists of a 64-bit scan chain that can optionally be configured as a 32-bit or a 48-bit scan chain. The size of the EMUIR scan chain is controlled by the EMUIRSZ bits in the DBGCTL register.
When configured as a 48-bit register.

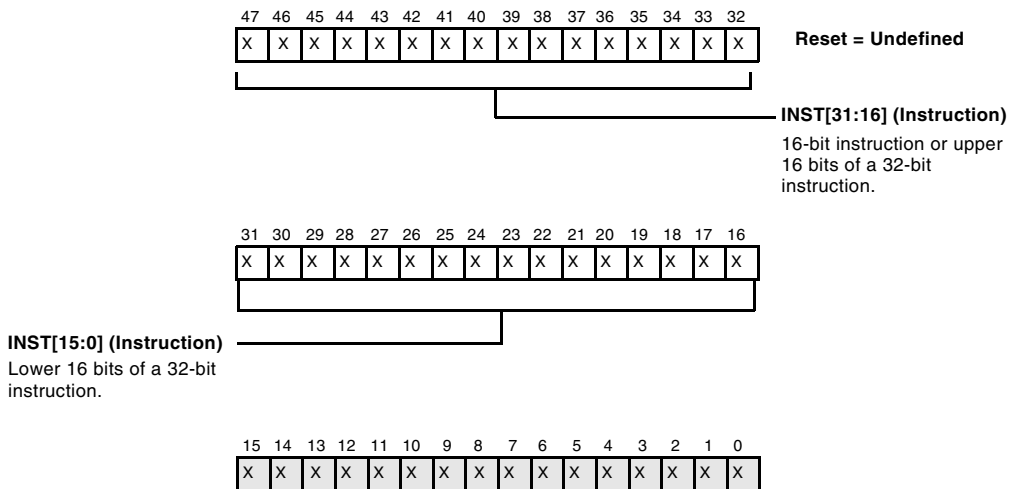


Figure 19-22. Emulation Instruction Register Configured as a 48-bit Register

3. Select the EMUIR scan chain and scan in the two instructions in the loop. The second instruction in the loop is scanned in first. The EMUIR registers now contain:
- ```
R0 = EMUDAT ; // in EMUIR_A (INST_SIZE=01),
[P0++] = R0 ; // in EMUIR_B (INST_SIZE=01)
```
4. Trigger an external emulator exception by asserting the EMUIN pin.
  5. Select the EMUDAT scan chain and download data into the EMUDAT register.

# Product Identification Registers

## Emulation Instruction Register (EMUIR)

Consists of a 64-bit scan chain that can optionally be configured as a 32-bit or a 48-bit scan chain. The size of the EMUIR scan chain is controlled by the EMUIRSZ bits in the DBGCTL register. When configured as a 64-bit register.

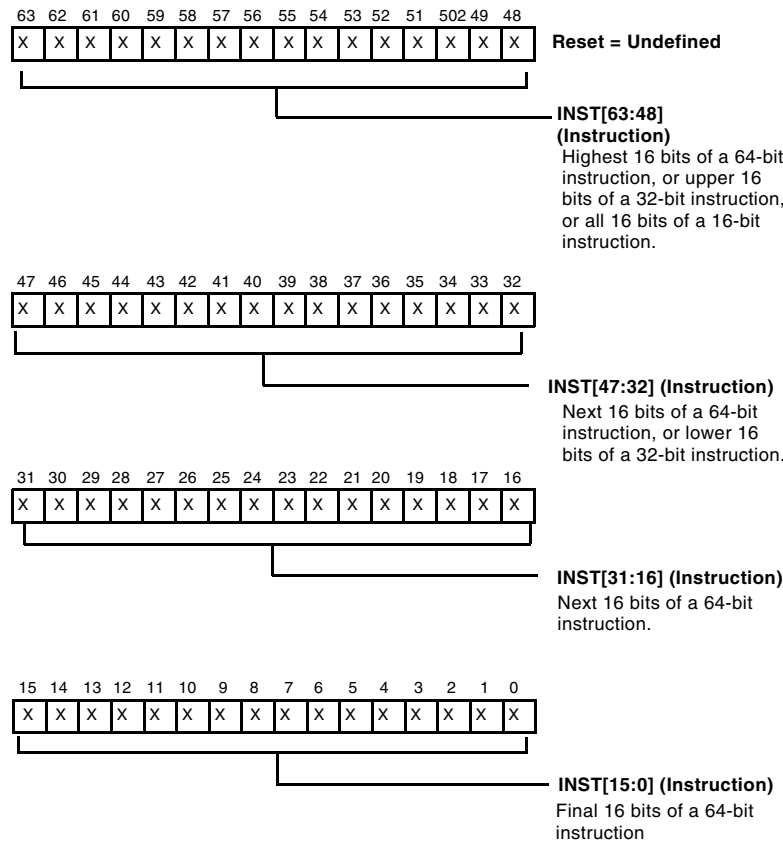


Figure 19-23. Emulation Instruction Register Configured as a 64-bit Register

6. Enter the Run-Test/Idle state to pulse the RT/I input to the Blackfin core. This causes the processor to execute the instruction stored in EMUIR\_A, then the instruction stored in EMUIR\_B. The instruction pointer now returns to EMUIR\_A. The EMUREADY bit in the DBGCTL register is cleared while the processor is executing both instructions. After both instructions finish executing, the EMUREADY bit is set again.
7. The external controller can use the EMUREADY bit as a signal to repeat steps 5 and 6 to download multiple data elements to memory.

If loop behavior is not needed, the emulator may send instructions to EMUIR and execute them using the following sequence of operations.

1. Activate the Emulator Unit by setting the EMPWR bit in the DBGCTL register.
2. Enable the emulator functionality (EMFEN) and external emulator exception (EMEEN) by scanning the DBGCTL register. Set the EMUIRSZ bits to select the 34-bit scan chain and set the EMUIRLPSZ bit to configure the EMUIR stack as a single instruction loop.
3. Select the EMUIR scan chain and scan in the first instruction to be executed. The EMUIR registers now contain:

```
first_instruction; // in EMUIR_A
-- don't care -- // in EMUIR_B
```

4. Trigger an external emulator exception by asserting the EMUIR pin.
5. Enter the Run-Test/Idle state to pulse the RT/I input to the core. This causes the core to execute the instruction stored in EMUIR\_A. Because EMUIRLPSZ=0, the instruction pointer continues to point to EMUIR\_A. The EMUREADY bit in the DBGSTAT register is cleared while the processor is executing this instruction. After the instruction finishes executing, the EMUREADY bit is set again.

6. The external controller can use the EMUREADY bit as a signal to select the EMUIR scan chain and scan in the second instruction to be executed. The update\_dr state resets the instruction pointer to EMUIR\_A. The EMUIR registers now contain:

```
second_instruction; // in EMUIR_A
-- don't care -- // in EMUIR_B
```

7. Enter the Run-Test/Idle state to pulse the RT/I input to the core. This causes the processor to execute the instruction stored in EMUIR\_A. Because EMUIRLPSZ=0, the instruction pointer continues to point to EMUIR\_A. The EMUREADY bit in the DBGCTL register is cleared while the processor is executing this instruction. After the instruction finishes executing, the EMUREADY bit is set again.
8. Repeat steps 6 and 7 to execute as many instructions as needed.

## Emulation Program Counter Register (EMUPC)

Figure 19-26 describes the Emulation Program Counter register (EMUPC), which is accessible only through the JTAG EMUPC\_SCAN scan chain. The EMUPC\_SCAN scan chain is available through JTAG even when the ADSP-BF53x core is not in Emulation mode.

The address of the last instruction that completed execution in User or Supervisor mode is saved in the EMUPC register. This allows the emulator to debug a hung system. This feature is active only when EMFEN = 1 and EMPWR = 1 in the DBGCTL register. Note that, when the processor is in Emulation mode, the RETE register contains the address of the next instruction to be executed on return from Emulation.



## Emulation Counter Register (EMUPC)

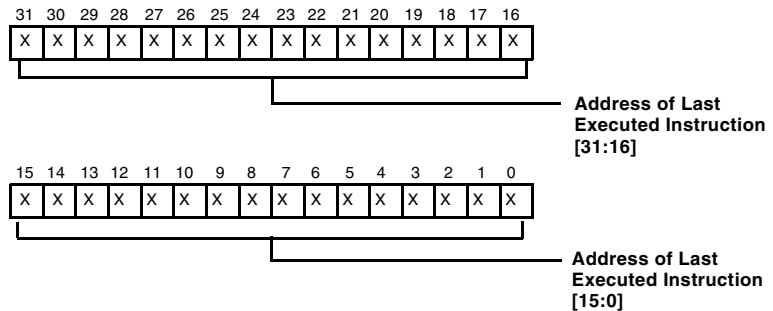


Figure 19-24. Emulation Counter Register

## JTAG Functionality

The Blackfin core does not include a JTAG Test Access Port (TAP), but it does include the control and data signals that interface to a TAP controller. The chip designer is responsible for implementing a TAP controller, decoding the JTAG instruction register, and driving these input signals. This feature allows a multi-core, single-chip processor to use a common JTAG TAP controller for all the cores and peripherals.

The JTAG TCK clock frequency can be as high as one-half the core clock CLK frequency.

The debug registers are accessible through JTAG.

### Scan Chains

Table 19-9 lists the Blackfin scan chains accessible through JTAG. According to the IEEE 1149.1 JTAG specification, when registers are serially scanned out of the core, the MSB is the first bit to clock out of the core.

Table 19-9. Blackfin Scan Chains

| Scan Chain Name | Size (in Bits) | Contents           |
|-----------------|----------------|--------------------|
| DBGSTAT_SCAN    | 16             | DBGSTAT bits 15:0  |
| DBGCTL_SCAN     | 16             | DBGCTL             |
| EMUIR_SCAN      | 32/48/64       | EMUIR              |
| EMUDAT_SCAN     | 32/40/48       | EMUDAT             |
| EMUPC_SCAN      | 32             | EMUPC              |
| BOUNDARY_SCAN   | 197            | Chip Boundary scan |
| MEMTEST_SCAN    | 64             | MEMTEST            |

IDCODE is the JTAG Chip ID register prescribed by the IEEE 1149.1 specifications. The Blackfin core does not implement IDCODE; the silicon processor designer must implement it. IDCODE allows an external emulator to read the type and version of the processor through the JTAG port without processor intervention. IDCODE identifies the silicon processor and not merely the Blackfin core. The Device ID register is at least one of the fields of the IDCODE register.

### JTAG Interface Signals

The JTAG TAP controller is not part of the Blackfin core. Thus, the processor provides the following interface signals for communication between the Blackfin core and the JTAG TAP controller.

# Blackfin Processor Debug and Emulation

Table 19-10. JTAG TAP Controller Interface Signals

| Signal Name          | Direction from the Blackfin Core | Description                                                                                                                                    |
|----------------------|----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| ci_tck               | Input                            | TAP test clock.                                                                                                                                |
| ci_trst              | Input                            | TAP test-logic-reset state.                                                                                                                    |
| ci_te_shift_t2f      | Input                            | TAP shift-dr state.                                                                                                                            |
| ci_te_capture_t2f    | Input                            | TAP capture-dr state.                                                                                                                          |
| ci_te_update_t2f     | Input                            | TAP update-dr state.                                                                                                                           |
| ci_te_runtstidle_t2f | Input                            | TAP run-test-idle state.                                                                                                                       |
| ci_te_si_t2f         | Input                            | Scan data in.                                                                                                                                  |
| ci_te_securityen     | Input                            | Allows emulator functionality (gates EMFEN and EMPWR).<br>Note: Assumed to change only when DSP is not running or when EMFEN and/or EMPWR = 0. |
| ci_te_emuin_t2f      | Input                            | External emulator interrupt (conditioned in core with EMEEN).                                                                                  |
| ci_te_selemupc_t2f   | Input                            | Select EMUPC scan chain.                                                                                                                       |
| ci_te_selemudat_t2f  | Input                            | Select EMUDAT scan chain.                                                                                                                      |
| ci_te_selemuir_t2f   | Input                            | Select EMUIR scan chain.                                                                                                                       |
| ci_te_selemuctl_t2f  | Input                            | Select EMUCTL scan chain.                                                                                                                      |
| ci_te_selemustat_t2f | Input                            | Select EMUSTAT scan chain.                                                                                                                     |
| mem_bist_done_2f     | Input                            | Memory is not actively running bist.                                                                                                           |
| ci_dspid<7:0>        | Input                            | Core ID.                                                                                                                                       |
| co_te_so_t1f         | Output                           | Scan data out.                                                                                                                                 |
| co_emuout_?dsp       | Output                           | For emupin.                                                                                                                                    |
| co_emudif_?dsp       | Output                           | EMUDAT IN is full.                                                                                                                             |

Table 19-10. JTAG TAP Controller Interface Signals (Cont'd)

| Signal Name    | Direction from the Blackfin Core | Description                        |
|----------------|----------------------------------|------------------------------------|
| co_emudof_?dsp | Output                           | EMUDAT OUT is full.                |
| co_emuspace_2f | Output                           | Core is in emulation space (int0). |

## Compatibility Notes

For compatibility with the ADI SHARC debug boards, the JTAG instruction register is 5-bits long and follows the encoding shown below for the EMUIR and EMUDAT registers.

Table 19-11. JTAG Instruction Codes for Compatible Registers

| JTAG Instruction | Instruction Code | Blackfin Register Being Accessed |
|------------------|------------------|----------------------------------|
| Bypass           | 11111            | BYPASS                           |
| Extest           | 00000            | BOUNDARY                         |
| Sample/Preload   | 00001            | BOUNDARY                         |
| EmuIr            | 00010            | EMUIR                            |
| Emctl            | 00100            | EMUCTL                           |
| Emudat           | 00101            | EMUDAT                           |
| Emustat          | 00110            | EMUSTAT                          |
| Idcode           | 01000            | IDCODE                           |
| membist          | 01010            | MEMBIST                          |
| tmkey            | 01100            | TMKEY                            |
| cskey            | 01101            | CSKEY                            |
| samplepc         | 01111            | EMUPC                            |

For single-core processors, `Implementation[15:0]` covers the lower 16 bits of this MMR. Otherwise, `Implementation[7:0]` covers bits 15:8 and `DSPID[7:0]` covers the lower 8 bits of the MMR.

## DMA Bus Debug Registers

During normal operation, the core has no direct visibility into the DMA bus. However, the SBIU includes two registers that provide visibility into the bus and control of it. Additional information about the SBIU is available in Chapter 7, “Chip Bus Hierarchy.”

If a DMA channel does not behave as expected, these two registers provide a debug capability:

- DMA Bus Control Comparator register (`DB_CCOMP`)
- DMA Bus Address Comparator register (`DB_ACOMP`)

## DMA Bus Control Comparator Register (`DB_CCOMP`)

The DMA Bus Control Comparator register (`DB_CCOMP`), shown in Figure 19-25, provides control for the debug register pair. Program this register with the type of bus operation that you wish to detect. Program the address associated with this control field into the DMA Bus Address Comparator register (`DB_ACOMP`). See “DMA Bus Address Comparator Register (`DB_ACOMP`)” on page 19-47.

When enabled, this function compares the address and control fields of each bus operation on the DMA bus to those programmed into the address and control comparator fields. If a match occurs, the function sets the Compare Hit (`CH`) status bit in `DB_CCOMP`. If the interrupt is unmasked, the hit also generates a Hardware Error interrupt to the core. The source of the interrupt can be determined by reading the core MMR `HWE Cause` field in `SEQSTAT`, and/or by reading `DB_CCOMP` status.

# JTAG Functionality

## DMA Bus Control Comparator Register (DB\_CCOMP)

Read/write

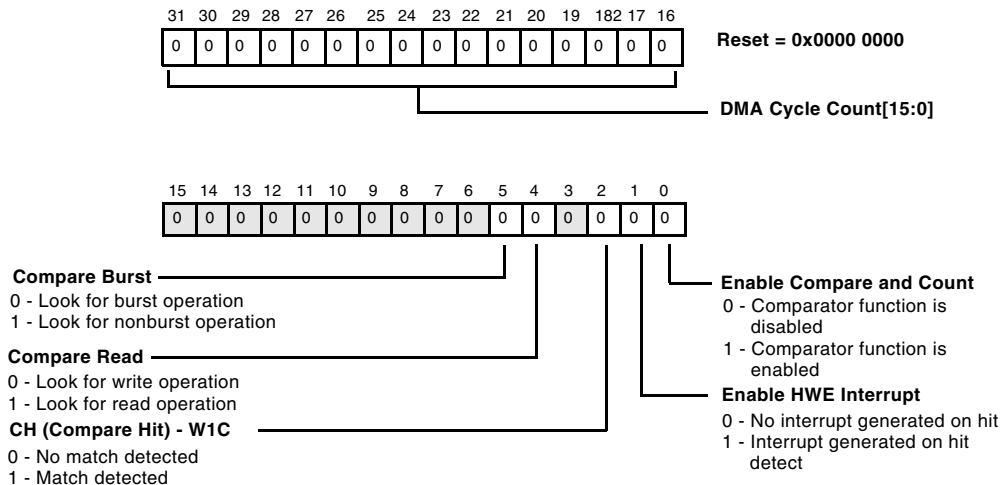


Figure 19-25. DMA Bus Control Comparator Register

The upper 16 bits of the DMA Bus Control Comparator register provide a cycle count function that helps determine when a given DMA access has occurred. The counter is clocked by the DMA bus clock.

The counter can be cleared in either of two ways:

- By a hardware reset
- By clearing the Enable Compare and Count bit (DB\_CCOMP[0])

When enabled, the counter increments from 0 to a maximum value of 65,535 until a match is detected or the compare-and-count function is disabled. A comparison match freezes the counter within two cycles of the assertion of CH.

## DMA Bus Address Comparator Register (DB\_ACOMP)

Figure 19-26 shows the DMA Bus Address Comparator register (DB\_ACOMP). Program into this register the DMA address that is associated with the type of bus information that you wish to detect.

For information about detecting bus information, see “DMA Bus Control Comparator Register (DB\_CCOMP)” on page 19-45.

### DMA Bus Address Comparator Register (DB\_ACOMP)

Read/write

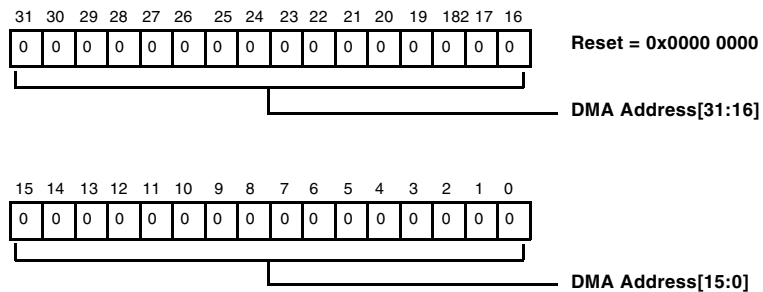


Figure 19-26. DMA Bus Address Comparator Register

## More Debug Registers and Emulation Registers

The ADSP-BF532 provides five debug registers that are accessible through JTAG, as follows:

- DBGCTL
- DBGSTAT
- EMUDAT

## JTAG Functionality

- EMUIR
- EMUPC



Note that the `CCLK` must be enabled before debug or emulation is attempted. The MMRs in the system-level clock controller must be on a separate JTAG scan chain to allow the host system to turn on the `CCLK` before accessing the debug or emulation features.