Differences Between the EOnCE and OnCE Ports

By Barbara Johnson

In the DSP56300 core, the on-chip emulation (OnCE™) port enables programmers to examine registers, memory, and on-device peripherals. It is a non-intrusive interface with the DSP56300 core and its peripherals. However, OnCE does not support real-time debugging functions. Freescale Semiconductor's StarCore™-based DSPs built on the SC140 core do not have this limitation. An advantage of the SC140 core Enhanced On-Chip Emulation (EOnCE) module is its capacity for real-time software debugging. This feature is only one of the EOnCE enhancements discussed in this application note, which compares the EOnCE and the OnCE ports and illustrates the differences with examples. This discussion assumes that you are familiar with the DSP56300 core OnCE and Joint Test Action Group (JTAG) ports. The following reading material is suggested for further reference:

- Chapter 7, Debugging Support, *DSP56300 Family Manual* (DSP56300FM)
- Chapter 4, Emulation and Debug (EOnCE), *SC140 DSP Core Reference Manual* (MSC140CORE)
- Chapter 17, JTAG and IEEE 1149.1 Test Access Port, *MSC8101 Reference Manual* (MSC8101RM)
- Chapter 12, EOnce/JTAG, *MSC8101 User's Guide* (MSC8101UG)

CONTENTS

1 External Pins

The only dedicated OnCE pin on the DSP56300 core is the Debug Event $\overline{\text{DE}}$ pin. As an input, this pin provides entry into Debug mode. As an output, this pin provides acknowledgment that the DSP has entered Debug mode. The $\overline{\text{DE}}$ pin is not available in the EOnCE. The SC140 core has seven dedicated EOnCE pins:

- EOnCE Event EE[0:5] pins
- EOnCE Event EED pin

The functions of the bidirectional EOnCE pins are programmable. As inputs, the EE[0–5] and EED pins can be configured to enable the Event Detection Channels (refer to **Section 8**, *Event Detection Unit*, on page 13). As inputs, pins EE[0–4] cause EOnCE events, such as entering Debug mode, issuing a debug exception, enabling trace, and disabling trace. As outputs, these pins can be configured to indicate detection by the event detection channels.

Some of the dedicated EOnCE pins can be programmed to perform specific functions. For example, as an input, EE0 can allow the SC140 core to enter Debug mode after reset when it is asserted. As an output, EE1 can acknowledge entry into Debug mode. As an input, EE2 can enable the event counter. As an output, EE3 can indicate that the EOnCE Receive (ERCV) register was read by the DSP (refer to **Section 5**, *Real-Time Data Transfer*, on page 6). As an output, EE4 can indicate that the EOnCE Transmit (ETRSMT) register was written by the DSP (see **Section 5**, *Real-Time Data Transfer*, on page 6). The SC140 JTAG pins have exactly the same functionality as the DSP56300 JTAG pins. **Table 1** summarizes the OnCE and EOnCE pin functionality.

Table 1. OnCE and EOnCE External Pins

OnCE Pin	DE	Input: Debug request.
		Output: DSP acknowledge.
EOnCE Pins	EE0	Input: Debug request, enable Address Event Detection Channel 0, or generate one of the EOnCE events.
		Output: Detection by Address Event Detection Channel 0.
	EE1	Input: Enable Address Event Detection Channel 1 or generate one of the EOnCE events.
		Output: Debug acknowledge or detection by Address Event Detection Channel 1.
	EE2	Input: Enable Address Event Detection Channel 2 or generate one of the EOnCE events or enable the Event Counter.
		Output: Detection by the Address Event Detection Channel 2.
	EE3	Input: Enable Address Event Detection Channel 3 or generate one of the EOnCE events.
		Output: ERCV register was read by the DSP.
	EE4	Input: Enable Address Event Detection Channel 4 or generate one of the EOnCE events.
		Output: ETRSMT register was written by the DSP.
	EE5	Input: Enable Address Event Detection Channel 5.
		Output: Detection by Address Event Detection Channel 5.
	EED	Input: Enable the Data Event Detection Channel.
		Output: Detection by the Data Event Detection Channel.

1.1 EE Signals Control Register

The EE Control Register (EE_CTRL) controls the operation of the EE pins (see **Table 2**).

Table 2. EE_CTRL Register

Bit Number	Bit Name	Description
15	EEDDEF	0 EED is an output to indicate detection by EDCD. 1 EED is an input to enable EDCD.
14–11	Reserved	
10	EE5DEF	0 EE5 is an output to indicate detection by EDCA5. 1 EE5 is an input to enable EDCA5.
9–8	EE4DEF	00 EE4 is an output to indicate detection by EDCA4. 01 EE4 is an output to indicate ETRSMT is ready. 10 Reserved. 11 EE4 is an input to enable EDCA4 or to generate an EOnCE event.
7–6	EE3DEF	00 EE3 is an output to indicate detection by EDCA3. 01 EE3 is an output to indicate ERCV is full. 10 Reserved. 11 EE3 is an input to enable EDCA3 or to generate an EOnCE event.
5–4	EE2DEF	00 EE2 is an output to indicate detection by EDCA2. 01 Reserved. 10 Reserved. 11 EE2 is an input to enable EDCA2 or ECNT or to generate an EOnCE event.
3–2	EE1DEF	00 EE1 is an output to indicate detection by EDCA1. 01 EE1 is an output to indicate debug acknowledgment. 10 Reserved. 11 EE1 is an input to enable EDCA1 or to generate an EOnCE event.
1–0	EE0DEF	00 EE0 is an output to indicate detection by EDCA0. 01 Reserved. 10 Input. 11 EE0 is an input to enable debug mode, enable EDCA0 or to generate an EOnCE event.

1.2 Example: EE0 As an Output to Indicate Detection by EDCA0

Example 1 shows how the EE0 pin can be configured as an output to indicate detection by an address event detection channel. The EOnCE registers are configured as follows:

- EDCA0_REFA = 0x80 to set the reference value.
- EE_CTRL[EE0DEF] = 00 to use EE0 as an output to indicate detection by EDCA0.
- EDCA0_CTRL[EDCAEN] = 1111 to enable EDCA0.
- EDCA0_CTRL[CS] = 00 to select Comparator A.
- EDCA0_CTRL[CACS] = 00 to compare the address equal to EDCA0_REFA.
- EDCA0_CTRL[ATS] = 01 to detect a write access.
- EDCA0_CTRL[BS] = 00 to compare to XABA.

Another way to configure the EOnCE registers is to use the EOnCE Configurator feature on the Metrowerks® CodeWarrior® for the StarCore debugger, as follows:

1. Select Debug →EOnCE →**EOnCE Configurator** to open the configuration window.
2. Select the **Control** tab.
   — EE Pin 0: Output: detection by EDCA0

3. Select the **EDCA0** tab.
   - Bus Selection: XABA
   - Access Type: Write
   - Comparator A (Hex 32 bits): 0x80
   - Enable After Event On: Enabled
4. Click **OK**.

The code in **Example 1** implements a loop executed `0x100` times that writes data to a memory location to which address register `r0` points and then reads the data back. When the `move.w d0,(r0)` instruction executes and `r0 = 0x80`, EDCA0 detects the write access to location `0x80`. EE0 is asserted to indicate detection by EDCA0.

Example 1. EE0 As an Output to Indicate Detection by EDCA0

```
        org             p:$20000
        dosetup3        START
        doen3           #$100
        move.w          #0,r0
        move.w          #$dcba,d0
        loopstart3
START   move.w          d0,(r0)
        nop
        move.w          (r0)+,d1
        loopend3
        jmp             *
```

1.3 Example: EE2 As an Input to Enable the Event Counter

Example 2 shows how pin EE2 is configured as an input to enable the Event Counter (ECNT). The EOnCE registers are configured as follows:

- EE_CTRL[EE2DEF] = `11` to use EE2 as an input to enable the ECNT.
- ECNT_CTRL[ECNTWHAT] = 1100 to count core clocks.
- ECNT_CTRL[ECNTEN] = 1010 to enable the counter when EE2 is asserted.
- ECNT_VAL = `0x7FFFFFFF` to initialize the counter value.

Using the EOnCE Configurator tool to configure the EOnCE registers as follows:

1. Select Debug→EOnCE →EOnCE Configurator to open the configuration window.
2. Select the **Control** tab:
   - EE Pin 2: Input: Enable EDCA2 event
3. Select the **Counter** tab:
   - What to Count: Core clock
   - Enable After Event On: EE2
   - Event Counter Value: 0x7FFFFFFF
4. Click **OK**.

ECNT_CTRL is programmed to count core clocks when it is enabled by the assertion of EE2. For simplicity, the code from **Example 1** is used. When you assert EE2, the event counter starts counting SC140 core clocks. When the `break` button is pressed to stop the DSP, the number of core SC140 clocks executed in the interval between enabling the event counter and stopping the DSP is the original ECNT_VAL minus the new ECNT_VAL.

Example 2. EE2 As an Input to Enable the Event Counter

```
        org             p:$1000
        dosetup3        START
        doen3           #$100
        move.w          #0,r0
        move.w          #$dcba,d0
        loopstart3
START   move.w          d0,(r0)
        nop
        move.w          (r0)+,d1
        loopend3
        jmp             *
```

2 Entering Debug Mode

The DSP56300 core enters Debug mode when:

- The $\overline{\text{DE}}$ pin is asserted.
- The DEBUG_REQUEST command executes via JTAG.
- The **debug**/**debugcc** instruction executes in software.
- A memory breakpoint is encountered.
- An instruction is encountered when the trace counter is zero.

The SC140 core enters Debug mode when:

- The DEBUG_REQUEST command executes via JTAG.
- The **debug** instruction executes in software.
- The EE0 pin is set to logic 1 at reset.
- The EE0 pin is asserted when configured as debug request.
- The trace buffer is full.
- The event selector is programmed to enter Debug mode and the proper event occurs.

3 Dedicated Instructions

In the DSP56300 core, when the **debug** instruction executes, the DSP enters Debug mode and awaits OnCE commands from the external host. The **debugcc** instruction enters Debug mode conditionally. Similarly, when the SC140 core decodes the **debug** instruction, the DSP enters Debug mode. The **debugev** instruction generates a debug event. The **mark** instruction writes the program counter (PC) value into the trace buffer when the trace buffer is enabled and TB_CTRL[TMARK] is set. **Table 3** lists the dedicated OnCE and EOnCE instructions.

Table 3. OnCE and EOnCE Dedicated Instructions

OnCE Dedicated Instructions	EOnCE Dedicated Instructions
debug Enter Debug mode.	**debug** Enter Debug mode.
debugcc Enter Debug mode conditionally.	**debugev** Generate a debug event.
	mark Write the PC into the trace buffer.

4 Register Access

In the DSP56300 core, the OnCE registers are accessible only via JTAG. For example, to read from the OnCE Trace Counter (OTC), the OnCE Command Register (OCR) is first configured to read the OTC. The contents of the OCR are then shifted in via JTAG using the TDI signal. The contents of the OTC are read via JTAG using the TDO signal. In the SC140 core, not only are all the EOnCE registers accessible via JTAG but also most are accessible from software. For example, to read the EOnCE Status Register (ESR), the EOnCE Command Register (ECR) is first configured to read the ESR. The contents of the ECR are then shifted in via JTAG using the TDI signal. The contents of the ESR are read via JTAG using the TDO signal. Alternatively, the read from the ESR can be performed in software. For example, while the DSP is running, the ESR can be read using a **move** instruction in the software. Only four of the EOnCE registers are not accessible from software:

- PC_NEXT PC of the next execution set
- PC_LAST PC of the last execution set
- CORE_CMD Core command register
- NOREG No register selected

5 Real-Time Data Transfer

In the DSP56300 core, the DSP must be in Debug mode to read or write OnCE registers via JTAG. When the DSP enters Debug mode, normal operation stops. For example, to access a OnCE register, the DSP must be in Debug mode, and the JTAG instruction `ENABLE_ONCE` must execute. However, most of the SC140 EOnCE registers are read or written via JTAG either when the SC140 core is in Debug mode or when it is operating in Normal mode. Real-time data transfer occurs via a receive or transmit mechanism using the EOnCE Receive (ERCV) and the EOnCE Transmit (ETRSMT) registers:

- The ERCV register transfers data to the SC140 core from the host. The host can write this 64-bit shift register via the TDI input signal, and the SC140 core can read it from software. The SC140 core cannot write it from software.
- The ETRSMT register transfers data from the SC140 core to the host. The host can read this 64-bit shift register via the TDO output signal, and the SC140 core can write it from software. The SC140 core cannot read it from software.

Figure 1 shows an example of a write to the ERCV register via JTAG. This example assumes that the JTAG instructions `ENABLE_EONCE` and `CHOOSE_EONCE` have executed. The DSP does not need to be in Debug mode. The host first writes into the EOnCE Control Register (ECR) to indicate a write operation to the ERCV register. Next, the host sends the 64-bit data to be written into the ERCV on the TDI pin. The RCV bit in the EOnCE Status Register (ESR) is set to indicate that the host has finished writing into the ERCV register and the ERCV is available for the core to read. Alternatively, pin EE3 can be programmed as an output to indicate that the host has finished writing into the ERCV when EE3 goes low.

Figure 2 shows an example of a read from the ETRSMT register via JTAG. This example assumes that the JTAG instructions ENABLE_EONCE and CHOOSE_EONCE have executed. The SC140 core does not need to be in Debug mode. The host first writes into the ECR to indicate a read operation from the ETRSMT register. Next, the host reads the 64-bit data from the ETRSMT on the TDO pin. The TRSMT bit in the ESR is set to indicate that the core has finished writing the MSB of the ETRSMT register and the ETRSMT is available for the host to read. Alternatively, pin EE4 can be programmed as an output to indicate that the core has finished writing the ETRSMT when EE4 goes low.

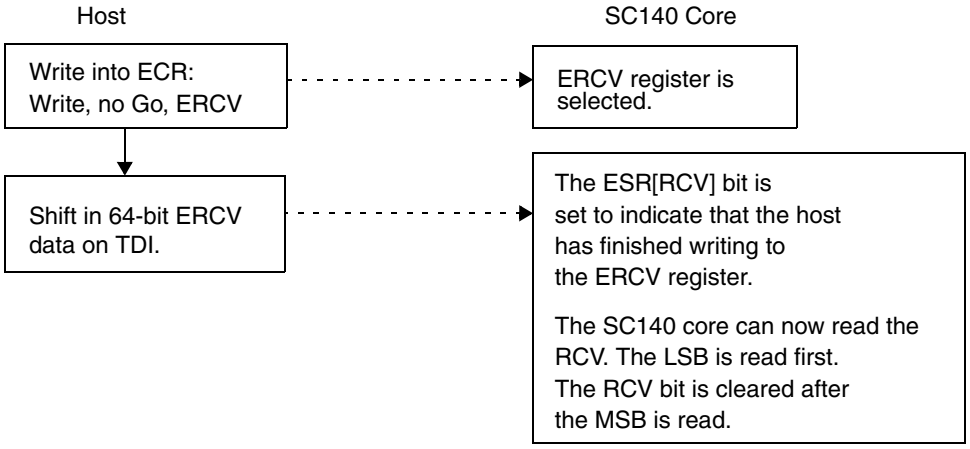

Figure 1. Writing EOnCE Registers via JTAG

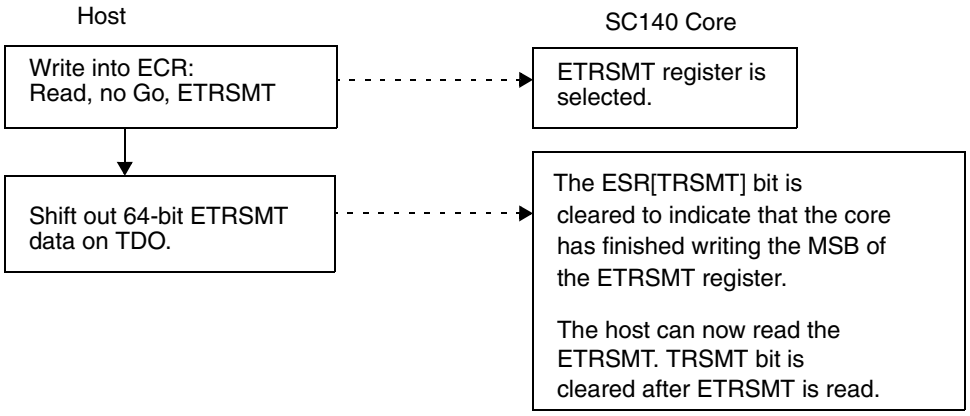

Figure 2. Reading EOnCE Registers via JTAG

6 Executing Instructions in Debug Mode

The EOnCE port can execute instructions while the DSP is in Debug mode. When the Core Command (CORE_CMD) register is written with an instruction and the GO bit in the EOnCE Command Register (ECR) is set, the fetch and dispatch stages are eliminated from the pipeline, and only the decoding and execution stages of the instruction are performed. The CORE_CMD register handles the following types of instructions:

- **move** instructions with all the possible addressing modes
  - ex: `move.w #0x0123,d0`
  - ex: `move.2l d0:d1,(r0)+`

- **jump** and **branch** instructions except delayed jumps and branches
  - ex: `jmp 0x100`
  - ex: `bsr 0x80`
- AGU arithmetic instructions
  - ex: `asla r0`
  - ex: `adda #4,r3`

The 48-bit CORE_CMD register is accessible only via JTAG. Software cannot access it.

6.1 CORE_CMD Instruction Format

The instruction to be executed must be in the CORE_CMD format, as shown in **Figure 3**.

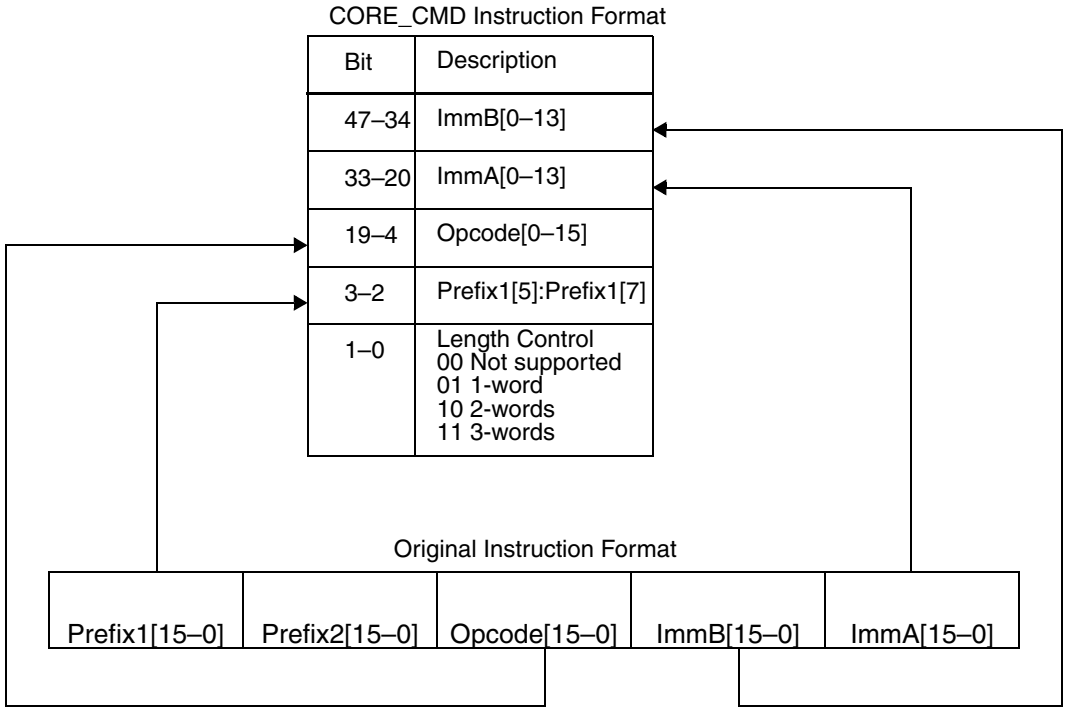

Figure 3. CORE_CMD Instruction Format

6.2 Example: Generating the CORE_CMD Value

If the instruction `move.l #$c0ffee,d8` needs to be executed using the CORE_CMD register, the instruction format must be rearranged to be compatible with the CORE_CMD instruction format. When this instruction is assembled in big-endian mode, the original instruction format is:

Prefix1	Prefix2	Opcode	ImmB	ImmA
0x3820	0xA000	0x30E0	0x3FEE	0x80C0

Not counting the prefix words, this instruction contains three words. Bits 15–0 contain the ImmA value (0x80C0), bits 16–31 contain the ImmB value (0x3FEE), and bits 32–47 contain the Opcode value (0x30E0). The Prefix2 and Prefix1 values are 0xA000 and 0x3820, respectively.

To reformat the instruction for the CORE_CMD register, the bit order is reversed for the ImmA, ImmB, and Opcode values. Only bits 13–0 of the ImmA and ImmB values are used. The Prefix2 value is not used. Only bits 5 and 7 of the Prefix1 value are used. Since the instruction length is three words, the length control bits contain a value of 11. **Table 4** shows how the CORE_CMD value is derived. The 48-bit CORE_CMD register is the concatenation of the bits in boldface. The CORE_CMD instruction format for the instruction `move.l #$c0ffee,d8` is: CORE_CMD Instruction Format:`0x0301 DFF0 70CB`.

Table 4. CORE_CMD Example

	ImmA	ImmB	Opcode	Prefix1	Length
	0x80C0	0x3FEE	0x30E0	0x3820	3 words
Original Format	ImmA[15–0] 1000 0000 1100 0000	ImmB[15–0] 0011 1111 1110 1110	Opcode[15–0] 0011 0000 1110 0000	Prefix1[5] 1 Prefix1[7] 0	
CORE_CMD Format	ImmA[0–13] **0000 0011 0000 00**	ImmB[0–13] **0111 0111 1111 11**	Opcode[0–15] **0000 0111 0000 1100**	Prefix1[5]: Prefix1[7] **10**	**11**

6.3 Software Downloading

The ERCV and CORE_CMD registers can be used for software downloading via JTAG. The ERCV is written with the data to be loaded into the DSP internal memory, and the CORE_CMD is written with the instruction to move the data in the ERCV to the DSP internal memory:

1. Write into the ERCV register the data to be transferred.
2. Write into the CORE_CMD register the instruction to move from the ERCV register to a data register.
3. Write into the CORE_CMD register the instruction to move from the data register to the desired memory location.

Figure 4 shows an example of software downloading via JTAG. The DSP is in Debug mode and the JTAG instructions **ENABLE_EONCE** and **CHOOSE_EONCE** have executed. Address register `r1` points to the address of the ERCV register (0xEFFE08), and address register `r0` points to the start of the memory location where data is to be stored. Since the ERCV register is 64-bits, two **move** instructions execute to move the data into data registers `d0` and `d1`. In this example, the instructions `move.l (r1)+,d1` and `move.l (r1)+,d0` are written into the CORE_CMD register to transfer the contents of the ERCV register into `d0` and `d1`. The first move instruction transfers the lower 32-bit content of the ERCV to `d1` and the second move instruction transfers the upper 32-bit content of the ERCV to `d0`. After the second move instruction, `r1` is reinitialized to point to ERCV. Otherwise, `r1` points to an address different from the ERCV after the second move instruction executes. Finally, the instruction `move.2l d0:d1,(r0)+` is written into the CORE_CMD register to transfer the contents of `d0` and `d1` into the internal memory. This process repeats until all data is downloaded. You must ensure that the previous contents of the registers `d0`, `d1`, `r0`, and `r1` are saved prior to downloading software.

7 Event Counter

Another enhancement of the EOnce is the addition of the Event Counter (ECNT) which provides the capability of counting various events. The ECNT can be programmed to count the following events:

- Core clocks
- Execution of instructions
- Event detection by an event detection channel (Event0–Event5 and EventD)
- Trace events
- Execution of the **debugev** instruction

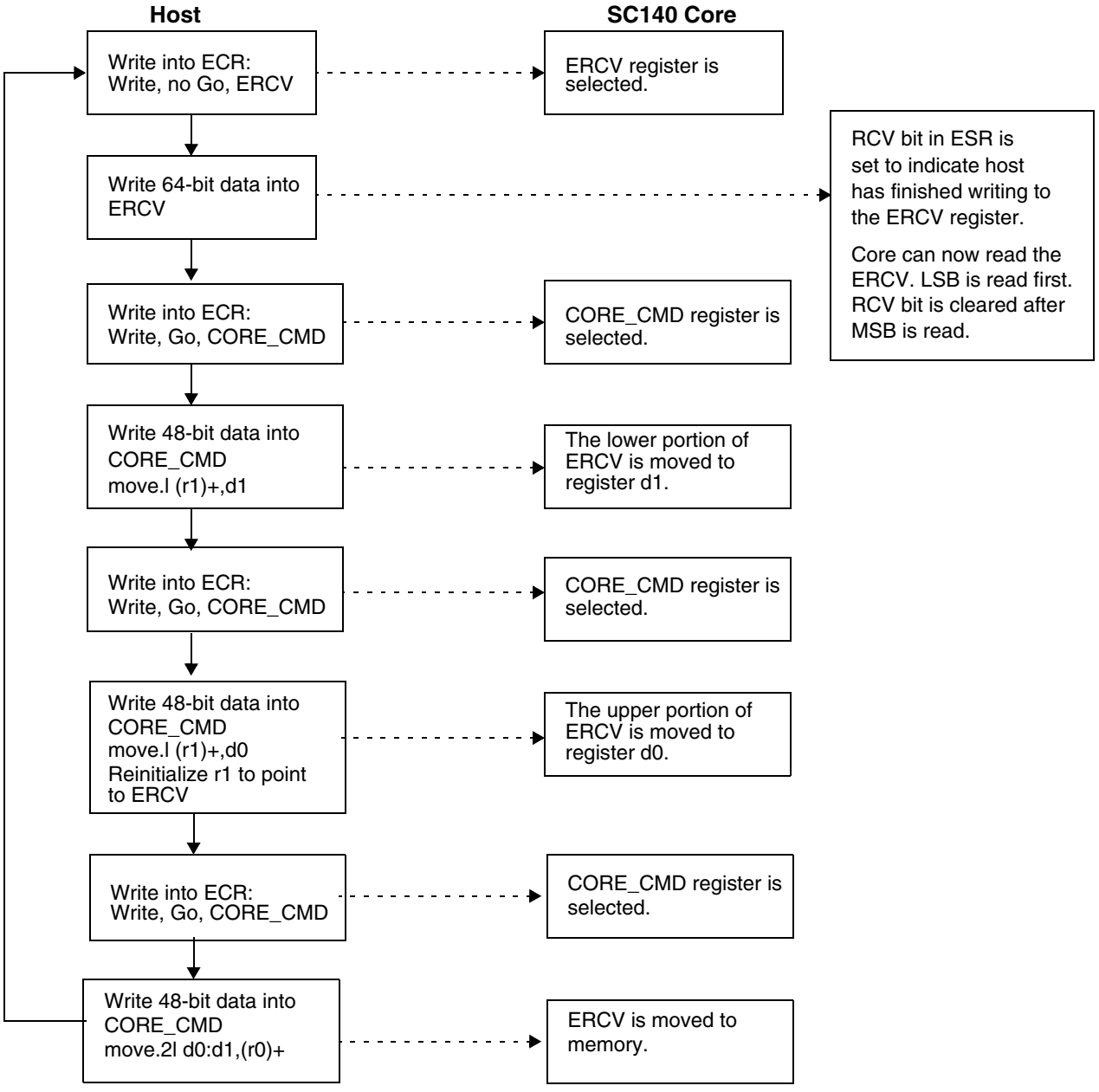

Figure 4. Software Downloading via JTAG

The ECNT can operate in two modes. In the normal mode of operation when the extension counter is disabled, a counter event is generated when the Event Counter Value (ECNT_VAL) reaches zero. In the extended mode of operation when the extension counter is enabled, a counter event is not generated when the ECNT_VAL reaches zero. Instead, the ECNT_VAL wraps around to `0xFFFFFFFF`. The number of wrap-arounds is counted by the Extension Counter (ECNT_EXT). **Figure 5** shows a block diagram of the Event Counter.

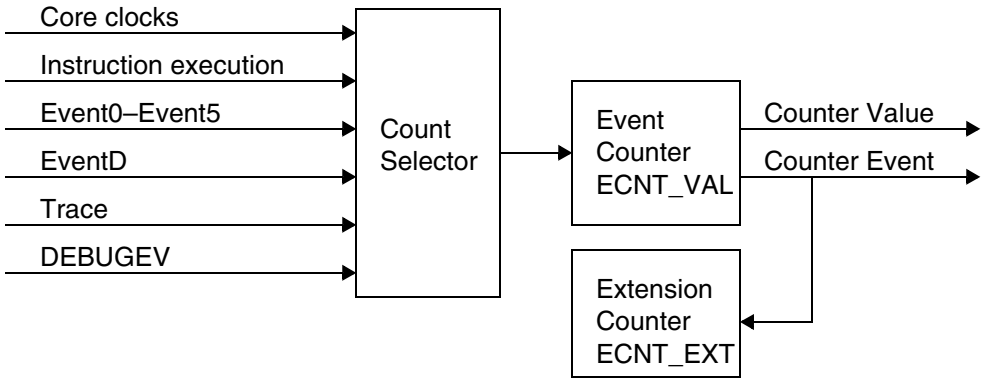

Figure 5. Event Counter Block Diagram

7.1 Event Counter Register Set

Table 5 shows the ECNT register set.

Table 5. ECNT Register Set

Register	Description		
ECNT_CTRL	ECNT Control Register. Controls the operation of the ECNT.		
	Bits 15–9	Reserved	
	Bit 8	EXT	0 ECNT operates in normal mode. 1 ECNT operates in extended mode.
	Bits 7–4	ECNTEN	0000 = ECNT is disabled. 0001 = ECNT is disabled but enabled when EDCA0 detects an event. 0010 = ECNT is disabled but enabled when EDCA1 detects an event. 0011 = ECNT is disabled but enabled when EDCA2 detects an event. 0100 = ECNT is disabled but enabled when EDCA3 detects an event. 0101 = ECNT is disabled but enabled when EDCA4 detects an event. 0110 = ECNT is disabled but enabled when EDCA5 detects an event. 1000 = ECNT is disabled but enabled when EDCD detects an event. 1010 = ECNT is disabled but enabled when an EE2 is asserted and EE2 is an input. 1111 = ECNT is enabled. All other settings are reserved.

Table 5. ECNT Register Set (Continued)

Register	Description		
ECNT_CTRL (Cont.)	Bits 3–0	ECNTWHAT	0000 = Count Event0 occurrence. 0001 = Count Event1 occurrence. 0010 = Count Event2 occurrence. 0011 = Count Event3 occurrence. 0100 = Count Event4 occurrence. 0101 = Count Event5 occurrence. 1000 = Count EventD occurrence. 1001 = Count execution of DEBUGEV instruction. 1010 = Count trace events. 1011 = Count executed execution sets. 1100 = Count core clocks. All other settings are reserved.
ECNT_VAL	Event Counter Value Register. Determines how many events the ECNT should count before it generates a count event signal. It counts down.		
ECNT_EXT	Event Extension Counter Value Register. Counts the number of ECNT_VAL overflows.		

7.2 Example: Counting Core Clocks Using the ECNT

Example 3 shows how the ECNT is configured to count core clock cycles. The EOnCE registers are configured as follows:

- ECNT_CTRL[EXT] = `00` to select normal mode.
- ECNT_CTRL[ECNTEN] = `1111` to enable the ECNT.
- ECNT_CTRL[ECNTWHAT] = `1100` to count core clocks.
- ECNT_VAL = `0x7FFFFFFF` to initialize the counter value.

Using the EOnCE Configurator tool, the EOnCE registers are configured as follows:

1. Select Debug →EOnCE →EOnCE Configurator to open the configuration window.
2. Select the **Counter** tab:
   - What to Count: Core clock
   - Enable After Event On: Enabled
   - Event Counter Value: 0x7FFFFFFF
3. Click **OK**.

The code begins at address p:START and ends at address p:END. When the code executes, the value in ECNT_VAL decrements for each executed cycle. The **debug** instruction is executed at the end of the code, and the SC140 core enters Debug mode. The number of cycles between START and END is the original ECNT_VAL value minus the new ECNT_VAL value.

Example 3. Cycle Counting Using the ECNT

```
org     p:START
;       ...
;       code
;       ...
END     debug
```

8 Event Detection Unit

The EOnCE event detection unit (EDU) performs the following tasks:

- Event detection on program and data memory address bus range or value.
- Event detection on data memory, data bus range or value.
- Detection of data written or read to/from a certain data memory address.
- Upon event detection, cause any of the EOnCE events.

The EDU consists of six address event detection channels (EDCA5–EDCA0), a data event detection channel (EDCD), and an event selector. **Figure 6** shows a block diagram of the EDU.

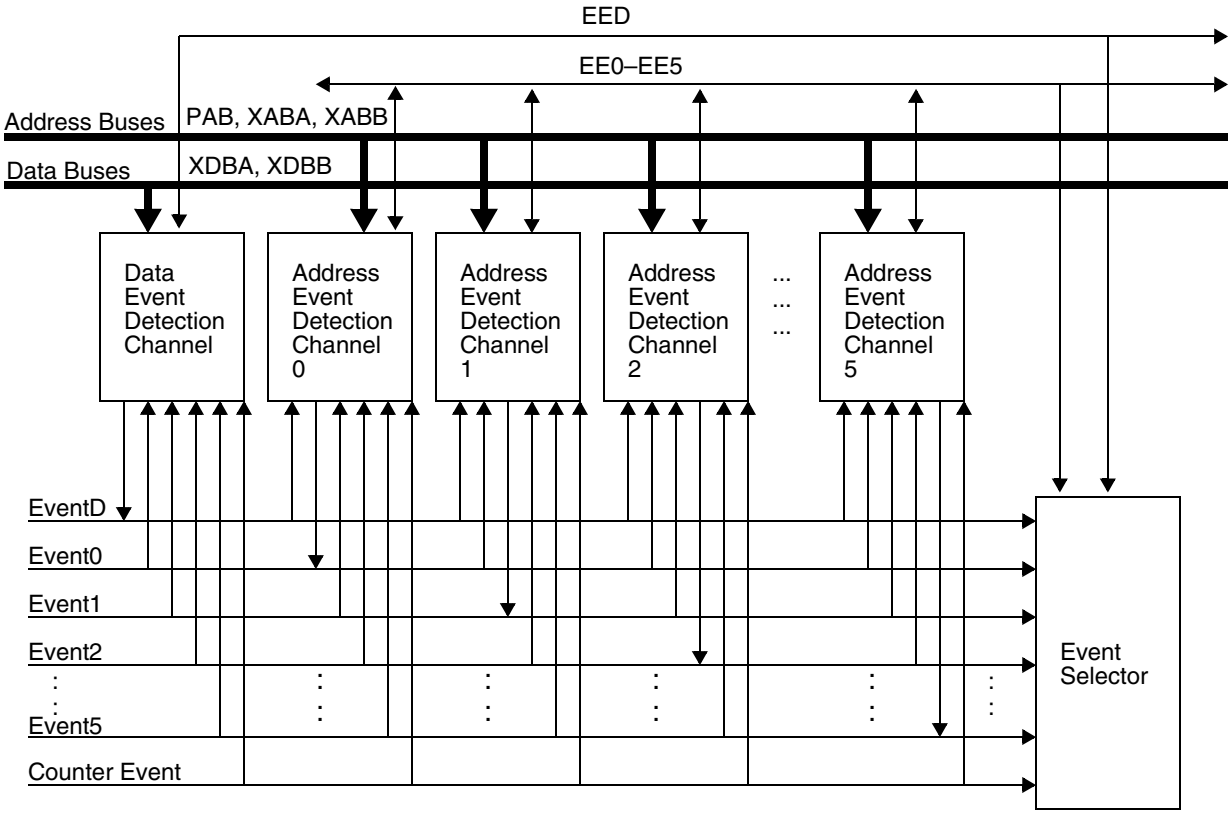

Figure 6. Event Detection Unit

8.1 Address Event Detection Channel

Each EDCAx has two 32-bit comparators that compare the core address buses and the reference values programmed into the 32-bit EDCAx Reference Value Register A (EDCAi_REFA) and EDCAi Reference Value Register B (EDCAi_REFB). The selected address buses that are sampled for comparison are:

- XABA address bus
- XABB address bus
- XABA and XABB address busses
- PAB address bus (program counter)

The EDCA can be programmed to detect read/write accesses to/from the addresses. The selected addresses can be specific addresses or a range of addresses. For example, the EDCA can be programmed to detect the following:

- Read access to x:0x100
- Execution set at p:0x200
- Execution set accesses from p:0x200 to p:0x300
- Write access to memory locations outside the range of x:0x500 to x:0x800

8.1.1 EDCA Register Set

Table 6 shows the EDCA register set.

Table 6. EDCA Register Set

Register	Description		
EDCAi_REFA	EDCAi Reference Value Register A. Contains reference value used by Comparator A to compare to sampled core address.		
EDCAi_REFB	EDCAi Reference Value Register B. Contains reference value used by Comparator B to compare to sampled core address.		
EDCAi_CTRL	EDCAi Control Register. Controls the operation of the EDCA.		
	Bits 15–14	Reserved	
	Bits 13–10	EDCAEN	0000 EDCAi is disabled. 0001 EDCAi is disabled but enabled when EDCA0 detects an event. 0010 EDCAi is disabled but enabled when EDCA1 detects an event. 0011 EDCAi is disabled but enabled when EDCA2 detects an event. 0100 EDCAi is disabled but enabled when EDCA3 detects an event. 0101 EDCAi is disabled but enabled when EDCA4 detects an event. 0110 EDCAi is disabled but enabled when EDCA5 detects an event. 1001 EDCAi is disabled but enabled when EDCD detects an event. 1011 EDCAi is disabled but enabled when an EEi is asserted and EEi is an input. 1111 EDCAi is enabled. All other settings are reserved.
	Bits 9–8	CS	00 Comparator A only. 01 Comparator B only. 10 Comparator A and Comparator B. 11 Comparator A or Comparator B.
	Bits 7–6	CBCS	00 Equal to EDCAi_REFB. 01 Not equal to EDCAi_REFB. 10 Greater than EDCAi_REFB. 11 Less than EDCAi_REFB.
	Bits 5–4	CACS	00 Equal to EDCAi_REFA. 01 Not equal to EDCAi_REFA. 10 Greater than EDCAi_REFA. 11 Less than EDCAi_REFA.
	Bits 3–2	ATS	00 Read access. 01 Write access. 10 Read or write access. 11 Reserved.
	Bits 1–0	BS	00 XABA bus is compared. 01 XABB bus is compared. 10 XABA or XABB is compared. 11 PC is compared.

Table 6. EDCA Register Set (Continued)

Register	Description
EDCAi_MASK	EDCAi Mask Register. Allows masking of bits of the sampled address.

8.1.2 Example: PC Detection Using EDCA

Example 4 shows how the EDCA0 is configured for PC detection. The EOnCE registers are configured as follows:

- EDCA0_REFA = `0x1004` to set the reference value.
- EDCA0_CTRL[EDCAEN] = `1111` to enable EDCA0.
- EDCA0_CTRL[CS] = `00` to select Comparator A.
- EDCA0_CTRL[CACS] = `00` to compare the address equal to EDCA0_REFA.
- EDCA0_CTRL[BS] = `11` to compare to PC.
- ESEL_CTRL[SELDM] = 0 to force core to enter debug mode by any one of the sources selected in the ESEL_DM register.
- ESEL_DM[EDCA0] = 1 to select the EDCA0 event as the cause for entering debug mode

Use the CodeWarrior EOnCE Configurator tool to configure the EOnCE registers as follows:

1. Select Debug→EOnCE →**EOnCE Configurator** to open the configuration window.
2. Select the **EDCA0** tab:
   - Bus Selection: PC
   - Comparator A (Hex 32 bits): 0x1004
   - Comparators Selection: A only
   - Enable After Event On: Enabled
3. Select the **Selector** tab:
   - Event(s) to Enter DEBUG Mode: OR
   - DEBUG Mode Mask: EDCA0
4. Click **OK**.

ESEL_CTRL is programmed to place the SC140 core into Debug mode when the PC matches the reference value. ESEL_DM sets EDCA0 as the source to cause the SC140 core to enter Debug mode.

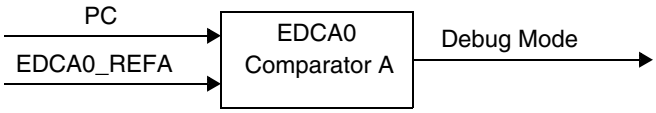

Figure 7. PC Detection Using EDCA0

Example 4 uses the same code as **Example 1** on page 4. The code begins at address 0x1000. A loop executes to write data into a memory location and then read the data back from memory. The DSP stops running after the instruction at address 0x1004 executes. Then the system enters Debug mode since EDCA0 has detected the event.

Example 4. PC Detection Using EDCA0

```
                org             p:$1000
                dosetup3        START   ; p:$1000
                doen3           #$100           ; p:$1004 debug mode is
                                                ; entered after this
                                                ; instruction executes
                move.w          #0,r0           ; p:$1008
                move.w          #$dcba,d0       ; p:$100a
                loopstart3
START           move.w          d0,(r0)         ; p:$100e
                nop                             ; p:$1012
                move.w          (r0)+,d1        ; p:$1014
                loopend3
```

8.1.3 Example: XABA Write Detection Using EDCA

Example 5 shows how the EDCA0 is configured to detect a write access to the XABA address bus. The EOnCE registers are configured as follows:

- EDCA0_REFA = `0x80` to set the reference value.
- EDCA0_CTRL[EDCAEN] = `1111` to enable EDCA0.
- EDCA0_CTRL[CS] = `00` to select Comparator A.
- EDCA0_CTRL[CACS] = `00` to compare the address equal to EDCA0_REFA.
- EDCA0_CTRL[ATS] = `01` to detect a write access.
- EDCA0_CTRL[BS] = `00` to compare to XABA address.
- ESEL_CTRL[SELDM] = 0 to force core to enter debug mode by any one of the sources selected in the ESEL_DM register.
- ESEL_DM[EDCA0] = 1 to select the EDCA0 event as the cause for entering debug mode

Using the CodeWarrior EOnCE Configurator tool, the EOnCE registers are configured as follows:

1. Select Debug →EOnCE →**EOnCE Configurator** to open the configuration window.
2. Select the **EDCA0** tab:
   - Bus Selection: XABA
   - Access Type: Write
   - Comparator A (Hex 32 bits): 0x80
   - Comparators Selection: A only
   - Enable After Event On: Enabled
3. Select the **Selector** tab:
   - Event(s) to Enter DEBUG Mode: OR
   - DEBUG Mode Mask: EDCA0
4. Click **OK**.

ESEL_CTRL is programmed to place the SC140 core in Debug mode when a write access to location `0x80` is detected. ESEL_DM sets EDCA0 as the source to cause the SC140 core to enter Debug mode.

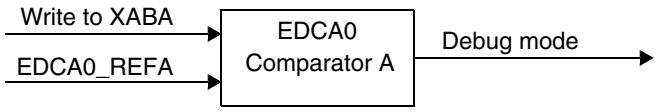

Figure 8. XABA Write Detection Using EDCA0

Example 5 uses the same code as **Example 1** on page 4. A loop executes 0x100 times to write data into a memory location and then read the data back from memory. The DSP stops running after data is written to memory location 0x80. Then the system enters Debug mode since EDCA0 has detected the event.

Example 5. XABA Write Detection Using EDCA0

```
                org     p:$1000
                dosetup3 START ; p:$1000
                doen3   #$100  ; p:$1004
                move.w  #0,r0  ; p:$1008
                move.w  #$dcba,d0; p:$100a
                loopstart3
START           move.w  d0,(r0) ; p:$100e debug mode is
                                ; entered after this
                                ; instruction is executed
                                ; when r0=$80
                move.w  (r0)+,d1; p:$1012
                loopend3
```

8.1.4 Example: XABA Read Detection Using EDCA

Example 6 shows how the EDCA0 is configured to detect a read access from the XABA address bus. The EOnCE registers are configured as follows:

- EDCA0_REFA = `0x10` to set the reference value.
- EDCA0_CTRL[EDCAEN] = `1111` to enable EDCA0.
- EDCA0_CTRL[CS] = `00` to select Comparator A.
- EDCA0_CTRL[CACS] = `00` to compare the address equal to EDCA0_REFA.
- EDCA0_CTRL[ATS] = `00` to detect a read access.
- EDCA0_CTRL[BS] = `00` to compare to XABA address.
- ESEL_CTRL[SELDM] = 0 to force core to enter debug mode by any one of the sources selected in the ESEL_DM register.
- ESEL_DM[EDCA0] = 1 to select the EDCA0 event as the cause for entering Debug mode.

Use the CodeWarrior EOnCE Configurator tool to configure the EOnCE registers as follows:

1. Select Debug →EOnCE →EOnCE Configurator to open the configuration window.
2. Select the **EDCA0** tab:
   - Bus Selection: XABA
   - Access Type: Read
   - Comparator A (Hex 32 bits): 0x10
   - Comparators Selection: A only
   - Enable After Event On: Enabled

3. Select the **Selector** tab:
   — Event(s) to Enter DEBUG Mode: OR
   — DEBUG Mode Mask: EDCA0
4. Click **OK**.

ESEL_CTRL is programmed to place the SC140 core in Debug mode when the a read access from location `0x10` is detected. ESEL_DM sets EDCA0 as the source to cause the core to enter Debug mode.

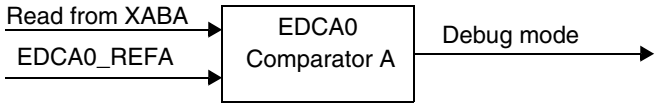

Figure 9. XABA Read Detection Using EDCA0

Example 6 uses the same code as **Example 5**. A loop executes 0x100 times to write data to a memory location and then read the data back from memory. The DSP stops running after data is read from memory location 0x10. The system enters Debug mode since EDCA0 has detected the event.

Example 6. XABA Read Detection Using EDCA0

```
        org             p:$1000
        dosetup3        START                   ; p:$1000
        doen3           #$100                   ; p:$1004
        move.w          #0,r0                   ; p:$1008
        move.w          #$dcba,d0               ; p:$100a
        loopstart3
START   move.w          d0,(r0)                 ; p:$100e
                        move.w          (r0)+,d1                ; p:$1012 debug mode
                                                                ; entered after this
                                                                ; instruction executes
                                                                ; when r0=$10
        loopend3
```

8.2 Data Event Detection Channel

The EDCD has a 32-bit comparator that compares the data values. It can be programmed to detect read or write accesses of data. It supports access widths of byte, word, and long word. For example, the EDCD can be programmed to detect the following:

- Read byte 0x07
- Write word 0x1234
- Write long word 0x12345678

8.2.1 EDCD Register Set

Table 7 shows the EDCD register set.

Table 7. EDCD Register Set

Register	Description
EDCD_REF	EDCD Reference Value Register. Contains reference value used by the comparator to compare to sampled data.

Table 7. EDCD Register Set (Continued)

Register	Description		
EDCD_CTRL	EDCD Control Register. Controls the operation of the EDCD.		
	Bits 15–10	Reserved	
	Bits 9–8	AWS	00 Byte access. 01 Word access. 10 Long word access. 11 Reserved.
	Bit 7	Reserved	
EDCD_CTRL cont.	Bits 6–3	EDCDEN	0000 EDCD is disabled. 0001 EDCD is disabled but enabled when EDCA0 detects an event. 0010 EDCD is disabled but enabled when EDCA1 detects an event. 0011 EDCD is disabled but enabled when EDCA2 detects an event. 0100 EDCD is disabled but enabled when EDCA3 detects an event. 0101 EDCD is disabled but enabled when EDCA4 detects an event. 0110 EDCD is disabled but enabled when EDCA5 detects an event. 1001 EDCD is disabled but enabled when a count event is detected. 1010 EDCD is disabled but enabled when an EEi is asserted and EEi is an input. 1111 EDCD is enabled. All other settings are reserved.
	Bits 2–1	CCS	00 Equal to EDCD_REF. 01 Not equal to EDCD_REF. 10 Greater than EDCD_REF. 11 Less than EDCD_REF.
	Bit 0	ATS	0 Read access. 1 Write access.
EDCD_MASK	EDCD Mask Register. Allows masking of bits of the sampled data.		

8.2.2 Example: Data Write Detection Using EDCD

Example 7 shows how the EDCD is configured for data detection. The EOnCE registers are configured as follows:

- EDCD_REFA = `0x24` to set the reference value.
- EDCD_CTRL[AWS] = `01` to select word-length data access.
- EDCD_CTRL[EDCDEN] = `1111` to enable the EDCD.
- EDCD_CTRL[CCS] = `00` to compare the data equal to EDCD_REF.
- EDCD_CTRL[ATS] = `1` to detect a write access.
- ESEL_CTRL[SELDM] = 0 to force core to enter debug mode by any one of the sources selected in the ESEL_DM register.
- ESEL_DM[EDCD] = 1 to select the EDCD event as the cause for entering debug mode

Use the EOnCE Configurator tool to configure the EOnCE registers as follows:

1. Select Debug →EOnCE →**EOnCE Configurator** to open the configuration window
2. Select the **EDCD** tab:
   — Access Type: Write

— Reference Value (Hex 32 bits): 0x24

— Enable After Event On: Enabled

3. Select the **Selector** tab:

— Event(s) to Enter DEBUG Mode: OR

— DEBUG Mode Mask: EDCD

4. Click **OK**.

ESEL_CTRL is programmed place the SC140 core in Debug mode when the data accessed matches the reference value. ESEL_DM sets EDCD as the source to cause the SC140 core to enter Debug mode.

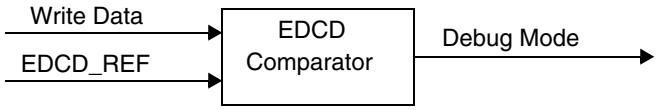

Figure 10. Data Write Detection Using EDCD

The code in **Example 7** implements a loop that writes data to a memory location. The data is incremented by one and written to the next memory location. The DSP stops running after 0x24 is written to memory. Then the system enters Debug mode since EDCD has detected the event.

Example 7. Data Write Detection Using EDCD

```
        org     p:$1000
        move.w  #0,r0
        move.w  #1,d1
        move.w  #0,d0
START   move.w  d0,(r0)+; debug mode is
                                                    ; entered after this
                                                    ; instruction is executed
        add2    d1,d0
        jmp     START
```

9 Event Selector

The event selector (ESEL) selects the source for the generated event. The possible sources are:

- Outputs of the address event detection channels (Event0–Event5)
- Output of the data event detection channel (EventD)
- Output of the event counter
- EE[4–0] pins
- **debugev** instruction

Upon event detection, the ESEL can generate one of the possible events:

- Enter the SC140 core into Debug mode
- Cause a debug exception
- Enable the trace buffer
- Disable the trace buffer

Figure 11 shows a block diagram of the ESEL.

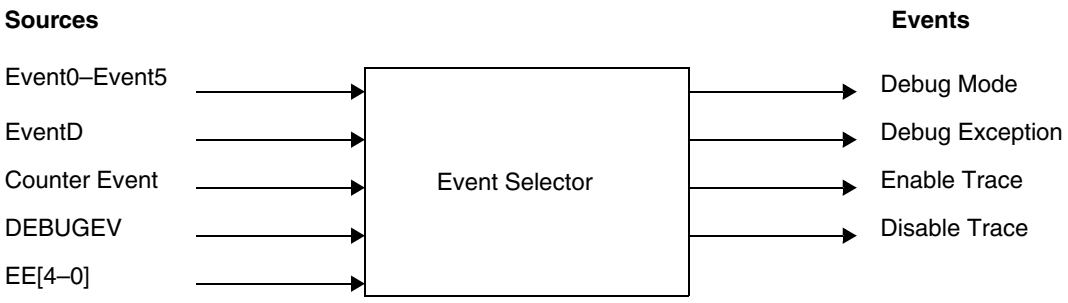

Figure 11. Event Selector Block Diagram

9.1 ESEL Register Set

The ESEL has a Control register (ESEL_CTRL) and four Mask registers (ESEL_DM, ESEL_DI, ESEL_ETB and ESEL_DTB). **Table 8** shows the ESEL register set.

Table 8. ESEL Register Set

Register	Description			
ESEL_CTRL	ESEL Control Register. Controls the operation of the ES.			
	Bits 7–5	Reserved		
	Bit 4	SELDTB	0	Trace is disabled upon detection of the event by any one of the sources selected in ESEL_DTB.
			1	Trace is disabled upon detection of the event by all sources selected in ESEL_DTB.
	Bit 3	SELETB	0	Trace is enabled upon detection of the event by any one of the sources selected in ESEL_ETB.
			1	Trace is enabled upon detection of the event by all sources selected in ESEL_ETB.
	Bit 2	Reserved		
ESEL_CTRL Cont.	Bit 1	SELDI	0	A debug exception is generated upon detection of the event by any one of the sources selected in ESEL_DI.
			1	A debug exception is generated upon detection of the event by all sources selected in ESEL_DI.
	Bit 0	SELDM	0	Debug mode is entered upon detection of the event by any one of the sources selected in ESEL_DM.
			1	Debug mode is entered upon detection of the event by all sources selected in ESEL_DM.
ESEL_DM ESEL_DI ESEL_ETB ESEL_DTB	ESEL Mask Debug Mode Register. Configures the source to cause entry into Debug mode. ESEL Mask Debug Exception Register. Configures the source to cause a debug exception. ESEL Mask Trace Enable Register. Configures the source to enable trace. ESEL Mask Trace Disable Register. Configures the source to disable trace.			
	Bit 15	DEBUGEV	1	DEBUGEV instruction is the source of the event.
	Bits 14–10	EE[4–0]	1	EEi is the source of the event.
	Bit 9	COUNT	1	Count event is the source of the event.
	Bit 8	EDCD	1	EDCD is the source of the event.
	Bits 7–6	Reserved		
	Bits 5–0	EDCA[5–]	1	EDCAi is the source of the event.

9.2 Example: Generating a Debug Exception Using the ESEL

In the EDCA example, the ESEL is programmed to cause the SC140 core to enter Debug mode when an event is detected. This example shows how the ESEL is programmed to generate a debug exception upon EDCA0 detection. The EOnCE registers are configured as shown:

- ESEL_CTRL[SELDI] = `1` to cause a debug exception upon detection of the event by all sources selected in the ESEL_DI register.
- ESEL_DI[EDCA0] = `1` to select EDCA0 as the source to cause the debug exception.
- EDCA0_REFA = `0x14` to set the reference value.
- EDCA0_CTRL[EDCAEN] = `1111` to enable EDCA0.
- EDCA0_CTRL[CS] = `00` to select Comparator A.
- EDCA0_CTRL[CACS] = `00` to compare the address equal to EDCA0_REFA.
- EDCA0_CTRL[ATS] = `00` to detect a read access.
- EDCA0_CTRL[BS] = `00` to compare to XABA address.

Use the EOnCE Configurator tool to configure the EOnCE registers as follows:

1. Select Debug →EOnCE →**EOnCE Configurator** to open the configuration window.
2. Select the **EDCA0** tab:
   - Bus Selection: XABA
   - Access Type: Read
   - Comparator A (Hex 32 bits): 0x14
   - Comparators Selection: A only
   - Enable After Event On: Enabled
3. Select the **Selector** tab:
   - Event(s) to Enter DEBUG Exception Mode: OR
   - DEBUG Exception Mode Mask: EDCA0
4. Click **OK**.

The EDCA registers are configured the same way as in **Example 6** on page 18. EDCA0 is programmed to detect a read access from memory location `0x14`. The code shown on the right implements a loop that is executed 0x100 times to read data from a memory location. A debug exception (`p:I_DEBUG` at location `VBA+0xC0`) is generated after data is read from memory location 0x14 since EDCA0 has detected the event. In this example, the debug exception interrupt service routine located at p:dbgexcp moves the value to which address register r0 points into data register d1. After the debug exception service routine executes, the value in `d1` is `0x18`, which is the value of r0 after the move.w (r0)+,d0 instruction is executed when r0=0x14.

Example 8. Generating a Debug Exception Using the ESEL

```
org     p:0
jmp     $1000

org     p:I_DEBUG       ;debug exception
jsr     dbgexcp
rte
```

```
                org     p:$1000
                dosetup3 START
        doen3           #$100
                move.w  #0,r0
                loopstart3
START
                move.w  (r0)+,d0
                loopend3
                jmp     *

dbgexcp         ;debug exception isr
                move.w  (r0),d1
                rts
```

10 Trace Unit

The DSP56300 core trace logic tracks program flow and consists of the following components:

- OnCE PAB Register for Fetch (OPABFR). A 16-bit read-only register that stores the address of the last instruction fetched before the system enters Debug mode.
- OnCE PAB Register for Decode (OPABDR). A 16-bit read-only register that stores the address of the last instruction decoded before the system enters Debug mode.
- OnCE PAB Register for Execute (OPABEX). A 16-bit read-only register that stores the address of the last instruction executed before the system enters Debug mode.
- A trace buffer that stores the addresses of the last 12 change of flow instructions that executed and the address of the last executed instruction

The SC140 core trace unit includes a 32-bit circular trace buffer. The buffer size is derivative-specific. For example, the size of the MSC8101 trace buffer is 2k words. When the end of memory is reached, the trace buffer wraps around to address zero and continues unless EMCR[TBFDM] is set. When the trace buffer is full, you can read the contents of the TB_BUFF. The ESR[TBFULL] flag is set when the trace buffer is full. Disabling the trace buffer by clearing TB_CTRL[TEN] also allows you to read the TB_BUFF. Due to the pre-fetch mechanism, a three-cycle delay must occur from the time the trace buffer is disabled until the first read-access to the trace buffer is issued. The EOnCE trace unit traces the following addresses:

- Normal execution
- Change-of-flow instructions
- Interrupts
- Hardware loops
- **mark** instruction

It operates during real-time processing. The debugging hardware can read the trace buffer during normal execution or in Debug mode when the trace buffer is disabled. It is enabled by the host, core software, or an EOnCE event.

10.1 Trace Buffer Register Set

The trace unit has a control register (TB_CTRL), two pointer registers (TB_WR and TB_RD), and a virtual register (TB_BUFF). **Table 9** shows the trace unit register set.

Table 9. Trace Unit Register Set

Register	Description		
TB_CTRL	Trace Buffer Control Register. Controls the operation of the Trace Unit.		
	Bit 7	TCNTEX	0 The value of the Extension Counter register is not placed into the trace buffer. 1 The value of the Extension Counter register is placed into the trace buffer.
	Bit 6	TCOUNT	0 Destination address put into the trace buffer is not followed by the value of the Event Counter register. 1 Destination address put into the trace buffer is followed by the value of the Event Counter register.
	Bit 5	TLOOP	0 Disable tracing of addresses of hardware loops. 1 Enable tracing of addresses of hardware loops.
	Bit 4	TEN	0 Trace buffer is disabled. 1 Trace buffer is always operational.
TB_CTRL Cont.	Bit 3	TMARK	0 Disable tracing of **mark** instruction. 1 Enable tracing of **mark** instruction.
	Bit 2	TEXEC	0 Disable tracing of addresses of every issued execution set. 1 Enable tracing of addresses of every issued execution set.
	Bit 1	TINT	0 Disable tracing of addresses of interrupt vectors. 1 Enable tracing of addresses of interrupt vectors.
	Bit 0	TCHOF	0 Disable tracing of addresses of execution sets with change-of-flow instructions. 1 Enable tracing of addresses of execution sets with change-of-flow instructions
TB_WR	Trace Buffer Write Pointer Register. Contains address of the next location available for writing into the trace buffer.		
TB_RD	Trace Buffer Read Pointer Register. Contains address of the next location available for reading from the trace buffer.		
TB_BUFF	Trace Buffer Register. Reads the contents of the trace buffer.		

10.2 Example: Tracing of Execution Sets

Example 9 shows how the trace buffer traces execution sets. The EOnCE registers are configured as follows:

- TB_CTRL[TEN] = `1` to enable the trace buffer.
- TB_CTRL[TEXEC] = `1` to trace the addresses of every execution set.
- ECNT_CTRL[ECNTEN] = `1` to enable the Event Counter.
- ECNT_CTRL[ECNTWHAT] = `1011` to count executed execution sets.
- ECNT_VAL = `0x7FFFFFFF` to initialize the counter value.

Since TB_RD and TB_WR are reset to zero when the trace buffer is enabled, it is not necessary to initialize these pointers. The ECNT_VAL is decremented for every executed execution set. In **Example 9**, there are six execution sets, so the final ECNT_VAL is `0x7FFFFFFF` –6 = `0x7FFFFFF9`. The addresses of the execution sets are written to the trace buffer as shown in **Table 10**.

Table 10. Trace Buffer Contents

TB_RD	TB_BUFF
0x0002	0x2000
0x0003	0x2002
0x0004	0x200A
0x0005	0x200C
0x0006	0x200E
0x0007	0x2008

Due to a pre-fetch mechanism, when the TB_BUFF location to which TB_RD points is read, the TB_RD pointer is already three stages ahead, so the first valid TB_BUFF value is located at TB_RD = `0x0002`.

Example 9. Tracing of Execution Sets

```
org     p:$2000
move.w  #0,d0           ;p:$2000
jsr     add             ;p:$2002
debug                   ;p:$2008
add     move.w  #1,d1   ;p:$200a
add2    d1,d0           ;p:$200c
rts                     ;p:$200e
```

11 Breakpoint Logic

In the DSP56300 core, breakpoints can be enabled to occur when a memory access is performed on P, X, or Y address space. These breakpoints occur when a memory address access is performed for read, write, or both operations. Breakpoints occur under one of the following conditions:

- Current memory address is not equal to the memory address in the OnCE Memory Limit Register (OMAL0 or OMAL1).
- Current memory address is equal to the memory address in the OMAL0 or OMAL1.
- Current memory address is less than the memory address in the OMAL0 or OMAL1.
- Current memory address is greater than the memory address in the OMAL0 or OMAL1.

In the SC140 core, breakpoints are enabled via the event selector (ESEL). For example, the ESEL is used with the EDU to detect reading/writing data from/to memory. The EDCD detects the data and the EDCA detects the address. Both events must occur for the EOnCE event to occur (see **Figure 12**).

Figure 12. Breakpoint Example 1

The ESEL is also used to with the EDU to detect reading/writing data from/to memory that is executed at a certain PC. For example, the EDCA0 can be programmed to detect the desired PC and upon detection of this PC, EDCA0 enables EDCA1 to detect the address and enables EDCD to detect the data that is read or written. When these events happen, the ESEL can be programmed to generate an EOnCE event. **Figure 13** shows a diagram of this breakpoint example.

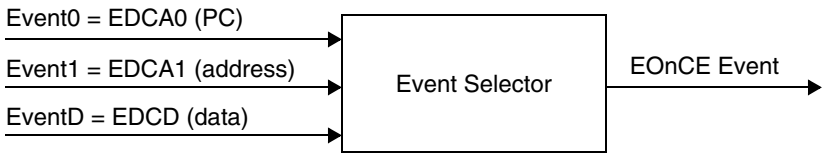

Figure 13. Breakpoint Example 2

The EOnCE has four registers for storing the PC address:

- PC_EXCP. Stores the PC of the instruction that caused an internal exception.
- PC_DETECT. Stores the PC of the last execution set that caused a watchpoint.
- PC_NEXT. Stores the PC of the execution set that would be executed next.
- PC_LAST. Stores the PC of the last executed instruction set.

12 Example: Cycle Count Profiling

The example discussed in this section implements the concepts learned from the previous examples. It shows how ECNT, EDCA, and ESEL perform cycle-count profiling to give a real-time cycle count between a start and a final address. The EOnCE modules can be programmed to perform the following:

- ECNT counts the number of cycles between a start and a final address.
- EDCA detects the start and final addresses.
- ESEL generates a debug exception when an address event detection channel detects the final address.

Cycle count profiling proceeds in the following stages:

- Detection of the start address, which enables the counter to start counting core cycles.
- Detection of the final address that generates a debug exception.
- Generation of a debug exception that disables the counter, reads the counter contents, and subtracts the interrupt service routine overhead.

12.1 ECNT Configuration

The ECNT_CTRL register is configured to do the following:

- Operate in normal mode.
- Enable the event counter when EDCA0 detects an event.
- Count core clocks.

The ECNT_VAL register is initialized with `0x7FFFFFFF`. When the counter is enabled, ECNT_VAL is decremented for each executed cycle.

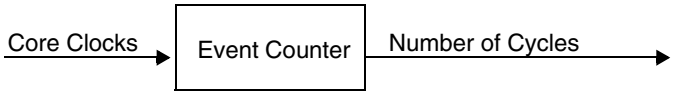

Figure 14. ECNT Configuration

12.2 EDCA Configuration

The EDCA0_CTRL register is configured to do the following:

- Enable EDCA0 to detect the start address.
- Compare the PC to the program start address in EDCA0_REFA.

The EDCA1_CTRL is configured to do the following:

- Enable EDCA1 to detect the final address.
- Compare the PC to the program final address in EDCA1_REFA.

The EDCA0_REFA and EDCA1_REFA registers are programmed with the start and final addresses.

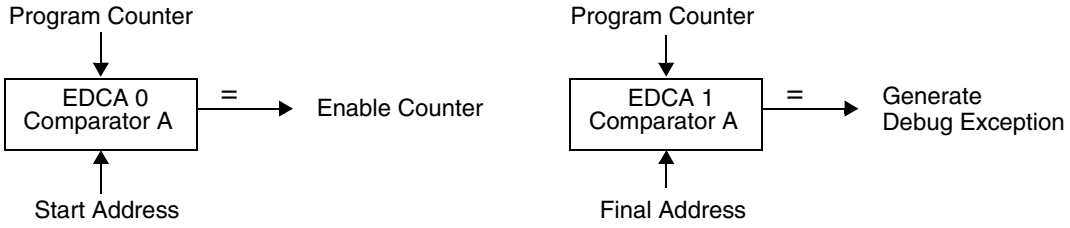

Figure 15. EDCA Configuration

12.3 ESEL Configuration

The ESEL_CTRL is configured to issue a debug exception upon detection of the final address. The ESEL_DI is configured to select EDCA1 as the source to cause a debug exception.

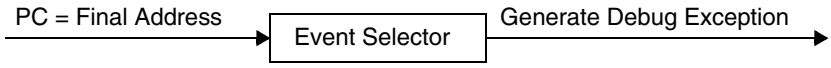

Figure 16. ESEL Configuration

12.4 Example Code

When EDCA0 detects the starting address `0x1000`, the counter is enabled. ECNT_VAL is decremented for each executed clock cycle. The end of the code is reached when the PC jumps to itself. When EDCA1 detects the final address (`0x1018`), a debug exception is generated. The debug exception service routine disables the timer and moves the final ECNT_VAL to data register `d1`. The number of cycles executed from the start to the final address is indicated by the new ECNT_VAL minus the new ECNT_VAL of 0x7FFFFFEB and the number of cycles to turn off the event counter, which gives a value of `0x7FFFFFFF` – `0x7FFFFFEB-2=12` cycles.

Example 10. Cycle Count Profiling Example

```
include 'eonce_regs.asm'
include 'intequ.asm'
input   ds      8
coeff   ds      8
org     p:0
jmp     $1000
```

```
                org     p:I_DEBUG
                jsr     dbgexcp
                rte
                org     p:$1000
                clr     d0      move.l  #input,r0
                move.f  #coeff,r1
                move.f  (r0)+,d2
                move.f  (r1)+,d1
                mac     d2,d1,d0                move.f   (r0)+,d2move.f
(r1)+,d1

                jmp     *

dbgexcp
                move.w  #0,d0
                move.w  d0,ECNT_CTRL
                move.l  ECNT_VAL,d1
                debug
                rts
```

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Order No.: AN2073
Rev. 1
1/2005