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)

Differences Between the EOnCE and OnCE Ports

By Barbara Johnson

CONTENTS

1. External Pins ...........................................................2
1.1 EE Signals Control Register ...................................2
1.2 Example: EE0 As an Output to Indicate Detection by EDCA0 ..............................................3
1.3 Example: EE2 As an Input to Enable the Event Counter.................................................... 4
2. Entering Debug Mode .............................................5
3. Dedicated Instructions .............................................5
4. Register Access ........................................................6
5. Real-Time Data Transfer ..............................................6
6. Executing Instructions in Debug Mode ..................7
6.1 CORE_CMD Instruction Format ............................8
6.2 Example: Generating the CORE_CMD Value .....8
6.3 Software Downloading ...........................................9
7. Event Counter ...........................................................10
7.1 Event Counter Register Set ..................................11
7.2 Example: Counting Core Clocks Using the ECNT .................................................................12
7.3 Event Detection Unit .............................................13
8.1 Address Event Detection Channel ......................13
8.2 Data Event Detection Channel ............................18
9. Event Selector ..........................................................20
9.1 ESEL Register Set .............................................21
9.2 Example: Generating a Debug Exception Using the ESEL ...................................................22
10. Trace Unit ..............................................................23
10.1 Trace Buffer Register Set ..................................23
10.2 Example: Tracing of Execution Sets .....................24
11. Breakpoint Logic ......................................................25
12. Example Code ..........................................................25
12.1 ECNT Configuration ..........................................26
12.2 EDCA Configuration ............................................27
12.3 ESEL Configuration .............................................27
12.4 Example Code ..........................................................27
1 External Pins

The only dedicated OnCE pin on the DSP56300 core is the Debug Event 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 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.

| 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. |

### Table 1. OnCE and EOnCE External Pins

| OnCE Pin | DE | Input: Debug request.  
Output: DSP acknowledge. |

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

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

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 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

<table>
<thead>
<tr>
<th>OnCE Dedicated Instructions</th>
<th>EOnCE Dedicated Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug  Enter Debug mode.</td>
<td>debug  Enter Debug mode.</td>
</tr>
<tr>
<td>debugcc Enter Debug mode conditionally.</td>
<td>debugev Generate a debug event.</td>
</tr>
<tr>
<td>mark  Write the PC into the trace buffer.</td>
<td></td>
</tr>
</tbody>
</table>

## 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

<table>
<thead>
<tr>
<th>Host</th>
<th>SC140 Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write into ECR: Write, no Go, ERCV</td>
<td>ERCV register is selected.</td>
</tr>
<tr>
<td>Shift in 64-bit ERCV data on TDI.</td>
<td>The ESR[RCV] bit is set to indicate that the host has finished writing to the ERCV register.</td>
</tr>
<tr>
<td></td>
<td>The SC140 core can now read the RCV. The LSB is read first. The RCV bit is cleared after the MSB is read.</td>
</tr>
</tbody>
</table>

**Figure 2.** Reading EOnCE Registers via JTAG

<table>
<thead>
<tr>
<th>Host</th>
<th>SC140 Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write into ECR: Read, no Go, ETRSMT</td>
<td>ETRSMT register is selected.</td>
</tr>
<tr>
<td>Shift out 64-bit ETRSMT data on TDO.</td>
<td>The ESR[TRSMT] bit is cleared to indicate that the core has finished writing the MSB of the ETRSMT register.</td>
</tr>
<tr>
<td></td>
<td>The host can now read the ETRSMT. TRSMT bit is cleared after ETRSMT is read.</td>
</tr>
</tbody>
</table>

---

**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)+
Executing Instructions in Debug Mode

- **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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>47–34</td>
<td>ImmB[0–13]</td>
</tr>
<tr>
<td>33–20</td>
<td>ImmA[0–13]</td>
</tr>
<tr>
<td>19–4</td>
<td>Opcode[0–15]</td>
</tr>
<tr>
<td>3–2</td>
<td>Prefix1[5]:Prefix1[7]</td>
</tr>
<tr>
<td>1–0</td>
<td>Length Control</td>
</tr>
<tr>
<td></td>
<td>00 Not supported</td>
</tr>
<tr>
<td></td>
<td>01 1-word</td>
</tr>
<tr>
<td></td>
<td>10 2-words</td>
</tr>
<tr>
<td></td>
<td>11 3-words</td>
</tr>
</tbody>
</table>

**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: 0x3820
- Prefix2: 0xA000
- Opcode: 0x30E0
- ImmB: 0x3FEE
- ImmA: 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>
<thead>
<tr>
<th>ImmA</th>
<th>ImmB</th>
<th>Opcode</th>
<th>Prefix1</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80C0</td>
<td>0x3FEE</td>
<td>0x30E0</td>
<td>0x3820</td>
<td>3 words</td>
</tr>
</tbody>
</table>

**Table 4. CORE_CMD Example**

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 (0xEFEE08), 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>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECNT_CTRL</td>
<td>ECNT Control Register. Controls the operation of the ECNT.</td>
</tr>
<tr>
<td>Bits 15–9</td>
<td>Reserved</td>
</tr>
<tr>
<td>Bit 8</td>
<td>EXT 0  ECNT operates in normal mode.</td>
</tr>
<tr>
<td></td>
<td>1  ECNT operates in extended mode.</td>
</tr>
<tr>
<td>Bits 7–4</td>
<td>ECNTEN 0000 = ECNT is disabled.</td>
</tr>
<tr>
<td></td>
<td>0001 = ECNT is disabled but enabled when EDCA0 detects an event.</td>
</tr>
<tr>
<td></td>
<td>0010 = ECNT is disabled but enabled when EDCA1 detects an event.</td>
</tr>
<tr>
<td></td>
<td>0011 = ECNT is disabled but enabled when EDCA2 detects an event.</td>
</tr>
<tr>
<td></td>
<td>0100 = ECNT is disabled but enabled when EDCA3 detects an event.</td>
</tr>
<tr>
<td></td>
<td>0101 = ECNT is disabled but enabled when EDCA4 detects an event.</td>
</tr>
<tr>
<td></td>
<td>0110 = ECNT is disabled but enabled when EDCA5 detects an event.</td>
</tr>
<tr>
<td></td>
<td>1000 = ECNT is disabled but enabled when EDCD detects an event.</td>
</tr>
<tr>
<td></td>
<td>1010 = ECNT is disabled but enabled when an EE2 is asserted and EE2 is an input.</td>
</tr>
<tr>
<td></td>
<td>1111 = ECNT is enabled.</td>
</tr>
<tr>
<td></td>
<td>All other settings are reserved.</td>
</tr>
</tbody>
</table>
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](image)

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

```assembly
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](image)

**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**

```assembly
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
    move.w d0,(r0) ; p:$100e
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>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDCD_REF</td>
<td>EDCD Reference Value Register. Contains reference value used by the comparator to compare to sampled data.</td>
</tr>
</tbody>
</table>
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[SELDMM] = 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.

![EDCD comparator](image)

**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
- **debug** 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.
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

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

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

```assembly
org p:0
jmp $1000
org p:I_DEBUG ;debug exception
jsr dbgexcp
rte
```
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TB_CTRL</td>
<td>Trace Buffer Control Register. Controls the operation of the Trace Unit.</td>
</tr>
<tr>
<td>TB_CTRL</td>
<td>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.</td>
</tr>
<tr>
<td>TB_CTRL</td>
<td>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.</td>
</tr>
<tr>
<td>TB_CTRL</td>
<td>Bit 5 TLOOP 0 Disable tracing of addresses of hardware loops. 1 Enable tracing of addresses of hardware loops.</td>
</tr>
<tr>
<td>TB_CTRL</td>
<td>Bit 4 TEN 0 Trace buffer is disabled. 1 Trace buffer is always operational.</td>
</tr>
<tr>
<td>TB_CTRL</td>
<td>Bit 3 TMARK 0 Disable tracing of mark instruction. 1 Enable tracing of mark instruction.</td>
</tr>
<tr>
<td>TB_CTRL</td>
<td>Bit 2 TEXEC 0 Disable tracing of addresses of every issued execution set. 1 Enable tracing of addresses of every issued execution set.</td>
</tr>
<tr>
<td>TB_CTRL</td>
<td>Bit 1 TINT 0 Disable tracing of addresses of interrupt vectors. 1 Enable tracing of addresses of interrupt vectors.</td>
</tr>
<tr>
<td>TB_CTRL</td>
<td>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</td>
</tr>
<tr>
<td>TB_WR</td>
<td>Trace Buffer Write Pointer Register. Contains address of the next location available for writing into the trace buffer.</td>
</tr>
<tr>
<td>TB_RD</td>
<td>Trace Buffer Read Pointer Register. Contains address of the next location available for reading from the trace buffer.</td>
</tr>
<tr>
<td>TB_BUFF</td>
<td>Trace Buffer Register. Reads the contents of the trace buffer.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>TB_RD</th>
<th>TB_BUFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0002</td>
<td>0x2000</td>
</tr>
<tr>
<td>0x0003</td>
<td>0x2002</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x200A</td>
</tr>
<tr>
<td>0x0005</td>
<td>0x200C</td>
</tr>
<tr>
<td>0x0006</td>
<td>0x200E</td>
</tr>
<tr>
<td>0x0007</td>
<td>0x2008</td>
</tr>
</tbody>
</table>

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).

![Event Selector Diagram](image)

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.
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.
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.

![EDCA Configuration Diagram]

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.

![ESEL Configuration Diagram]

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 0x7FFFFFFEB and the number of cycles to turn off the event counter, which gives a value of 0x7FFFFFFF - 0x7FFFFFFEB - 2 = 12 cycles.

Example 10. Cycle Count Profiling Example

```assembly
include 'eonce_regs.asm'
include 'intequ.asm'
input ds 8
coeff ds 8
org p:0
jmp $1000
```
Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized use, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. StarCore is a trademark of StarCore LLC. All other product or service names are the property of their respective owners.