eSi-SG-DMA

1 Contents

1 Contents 2
2 Overview 3
3 Hardware Interface 4
3.1 Flow Control Interface 5
4 Software Interface 7
4.1 Register Map 7
4.2 Descriptor Format 12
4.3 Interrupts 15
4.4 Error Handling 15
4.5 Stopping a Transfer 15
4.6 Channel Priority 16
4.7 Examples 16
5 Revision History 23

2 Overview

The eSi-SG-DMA core can be used to implement 1D and 2D memory-to-memory, memory-to-peripheral, peripheral-to-memory and peripheral-to-peripheral data transfers, with scatter and gather functionality. It supports the following features:

- Memory-based, linked-list transfer descriptors.
- 3 descriptor sizes to balance functionality and setup overhead.
- Configurable number of channels.
- Configurable number of peripherals (up to 64).
- Programmable X and Y count, increment, access size and burst length.
- CRC and IP checksum calculation.
- AMBA 3 AHB-lite slave interface for control register access.
- Dual AMBA 3 AHB-lite master interfaces for simultaneous read and write.

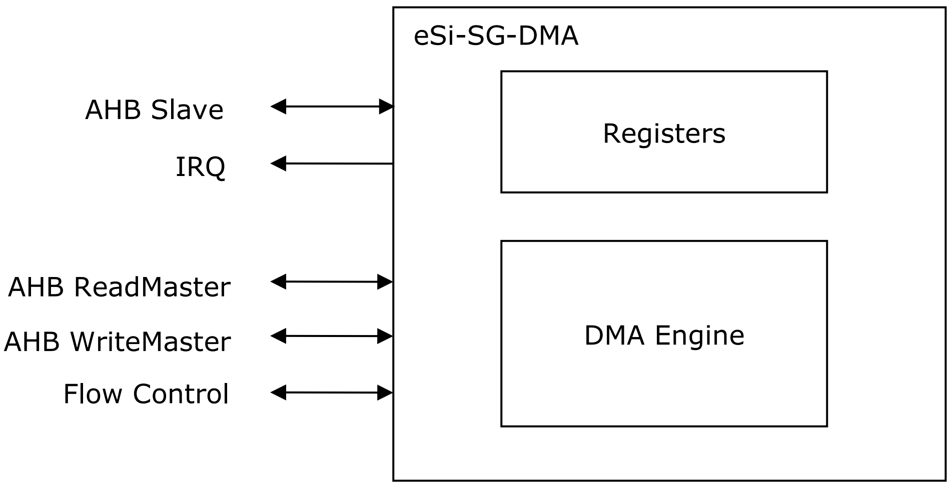

Figure 1: eSi-SG-DMA

3 Hardware Interface

Module Name	cpu_ahb_sg_dma
HDL	Verilog
Technology	Generic
Source Files	cpu_ahb_sg_dma.v, cpu_fifo.v

Configuration Option	Values	Description
CRC_ENABLED	TRUE, FALSE	Determines whether CRC calculation is supported
IP_CHECKSUM_ENABLED	TRUE,FALSE	Determines whether IP checksum calculation is supported

Table 1: Configuration Options

Port	Type	Description
channels	Integer	Specifies the number of channels implemented. 1-16
peripherals	Integer	Specifies the number of peripherals connected. 1-64
fifo_depth	Integer	Specifies the depth of the FIFO used for holding data during transfers. Must be 4 or greater and a power of 2
address_width	Integer	Specifies the width of src/dst address registers. Min of 10 bits
count_width	Integer	Specifies the width of count registers. Min of 8 bits
inc_width	Integer	Specifies the width of increment registers. Min of 4 bits
ahb_data_width	Integer	Specifies the width of AHB data busses. 32 or 64

Table 2: Parameters

Port	Direction	Width	Description
s_hclk	Input	1	Slave interface, AHB clock
s_hresetn	Input	1	Slave interface, AHB reset, active-low
s_haddr	Input	32	Slave interface, AHB address
s_hburst	Input	3	Slave interface, AHB burst type
s_hmastlock	Input	1	Slave interface, AHB locked transfer
s_hprot	Input	4	Slave interface, AHB protection
s_hsize	Input	3	Slave interface, AHB size
s_htrans	Input	2	Slave interface, AHB transfer type
s_hwdata	Input	ahb_data_width	Slave interface, AHB write data
s_hwrite	Input	1	Slave interface, AHB write
s_hready	Input	1	Slave interface, AHB ready
s_hsel	Input	1	Slave interface, AHB select
s_hready	Output	1	Slave interface, AHB ready
s_hrdata	Output	ahb_data_width	Slave interface, AHB read data
s_hresp	Output	1	Slave interface, AHB response
r_hclk	Input	1	Master read interface, AHB clock. Must be the same frequency and synchronous to s_hclk
r_hresetn	Input	1	Master read interface, AHB reset, active-low
r_hready	Input	1	Master read interface, AHB ready
r_hrdata	Input	ahb_data_width	Master read interface, AHB read data
r_hresp	Input	1	Master read interface, AHB response
r_haddr	Output	32	Master read interface, AHB address
r_hburst	Output	3	Master read interface, AHB burst type
r_hmastlock	Output	1	Master read interface, AHB locked transfer
r_hprot	Output	4	Master read interface, AHB protection

r_hsize	Output	3	Master read interface, AHB size
r_htrans	Output	2	Master read interface, AHB transfer type
r_hwdata	Output	ahb_data_width	Master read interface, AHB write data
r_hwrite	Output	1	Master read interface, AHB write
w_hclk	Input	1	Master write interface, AHB clock. Must be the same frequency and synchronous to s_hclk
w_hresetn	Input	1	Master write interface, AHB reset, active-low
w_hready	Input	1	Master write interface, AHB ready
w_hrdata	Input	ahb_data_width	Master write interface, AHB read data
w_hresp	Input	1	Master write interface, AHB response
w_haddr	Output	32	Master write interface, AHB address
w_hburst	Output	3	Master write interface, AHB burst type
w_hmastlock	Output	1	Master write interface, AHB locked transfer
w_hprot	Output	4	Master write interface, AHB protection
w_hsize	Output	3	Master write interface, AHB size
w_htrans	Output	2	Master write interface, AHB transfer type
w_hwdata	Output	ahb_data_width	Master write interface, AHB write data
w_hwrite	Output	1	Master write interface, AHB write
tx_ready	Input	peripherals	Indicates peripheral can accept new data
rx_ready	Input	peripherals	Indicates peripheral has data to be read
tx_ack	Output	peripherals	Acknowledges tx_ready after transfer complete
rx_ack	Output	peripherals	Acknowledges rx_ready after transfer complete
interrupt_n	Output	1	Interrupt request, active-low
w_hclk_cactive	Output	1	Clock active. When deasserted, w_hclk can be gated
r_hclk_cactive	Output	1	Clock active. When deasserted, r_hclk can be gated
debug_active	Input	1	Indicates when debugger is active

Table 3: I/O Ports

For complete details of the AHB signals, please refer to the AMBA 3 AHB-Lite Protocol v1.0 Specification available at:

http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

The DMA does not include internal synchronizing flip-flops. These should be implemented externally for the rx_ready and tx_ready ports if the transmitting clock domain is asynchronous to w/r_hclk.

3.1 Flow Control Interface

The flow control interface allows peripherals to indicate to the DMA when they have data available to be read or are able to accept new write data.

- The tx_ready signal indicates the peripheral can accept new data.
- The rx_ready signal indicates the peripheral has data to be read.

A simple handshaking mechanism is employed to ensure that the DMA will not generate an underflow or overflow in the peripheral.

- The tx_ack signal acknowledges the tx_ready signal

- The `rx_ack` signal acknowledges the `rx_ready` signal

The DMA will only perform a single transaction (which may consist of multiple beats, as controlled by the `BURST` register) after the `ready` signal is asserted. It will then assert the corresponding `ack` signal. This will be held high until the `ready` signal is cleared. The peripheral should only then reassert the `ready` signal after the `ack` has cleared and it is ready to proceed with another transaction.

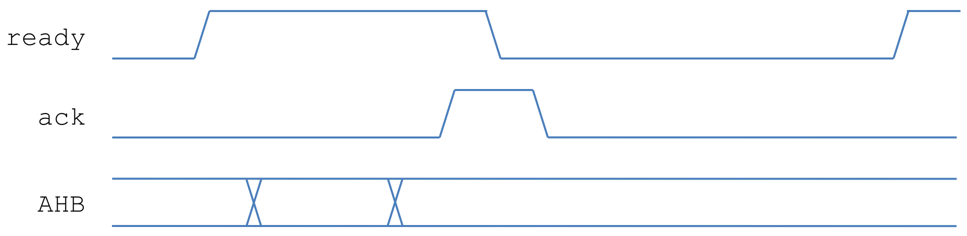

Figure 2: Flow Control Interface Handshaking

4 Software Interface

4.1 Register Map

Each SG-DMA channel has its own set of registers, as illustrated in Table 4: Register Map. In this table, `N`, indicates the channel number, which ranges from 0 to `channels`-1.

Register	Address offset	Access	Description
`next_address[N]`	0x80*N+0x00	-	Address of next descriptor
`src_address[N]`	0x80*N+0x04	R	Source address register
`dst_address[N]`	0x80*N+0x08	R	Destination address register
`src_control[N]`	0x80*N+0x0c	-	Channel N source control register
`dst_control[N]`	0x80*N+0x10	-	Channel N destination control register
`count_x[N]`	0x80*N+0x14	R	Count X register
`count_y[N]`	0x80*N+0x18	R	Count Y register
`src_inc_x[N]`	0x80*N+0x1c	-	Source increment X register
`src_inc_y[N]`	0x80*N+0x20	-	Source increment Y register
`dst_inc_x[N]`	0x80*N+0x24	-	Destination increment X register
`dst_inc_y[N]`	0x80*N+0x28	-	Destination increment Y register
`status[N]`	0x80*N+0x40	R/W	Status register
`control[N]`	0x80*N+0x44	R/W	Control register
`current[N]`	0x80*N+0x48	R/W	Address of current descriptor
`calc_control[N]`	0x80*N+0x50	R/W	Calculation control
`crc[N]`	0x80*N+0x54	R/W	CRC
`polynomial[N]`	0x80*N+0x58	R/W	CRC polynomial
`ip_checksum[N]`	0x80*N+0x5c	R/W	IP checksum

Table 4: Register Map

The registers that are not writable are programmed via memory based descriptors.

4.1.1 Next Address Register

The next address register contains the address of the next descriptor to process.

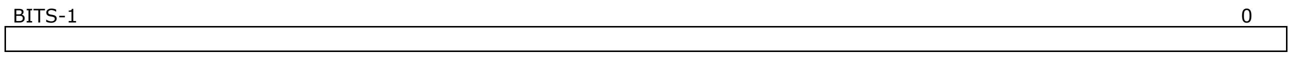

Figure 3: Format of the `next_address` register

4.1.2 Source Address Register

The source address register contains the address to copy from.

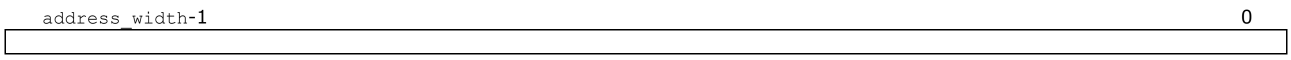

Figure 4: Format of the `src_address` register

4.1.3 Destination Address Register

The destination address register contains the address to copy to.

address_width-1	0

Figure 5: Format of the `dst_address` register

4.1.4 Source Control Register

The source control register contains a selection of flags that control the operation of the SG-DMA channel with respect to the source data.

15	10	9	7	6	4	3	2	1	0
PIDX		-		BURST		SIZE		T	

Figure 6: Format of the `src_control` register

Register	Values	Description
`T`	0 – Memory 1 – Peripheral 2 – None	Source address type. When set to peripheral type, the `rx_ready[PIDX]` signal from the peripheral must be asserted before a transfer will take place. When set to memory, the transfer will occur unconditionally. When set to none, `src_address` is not used and source data will just be zeros
`SIZE`	0 – 1 byte 1 – 2 bytes 2 – 4 bytes 3 – 8 bytes	Size of each access when T=1. This drives the AHB `hsize` signal. `src_address` must be aligned according to the transfer size. When T=0, this should be set to 0, and access size will be determined automatically depending upon count and alignment
`BURST`	0 – Undefined (`INCR`) 1 – 1 beat (`SINGLE`) 2 – 4 beats (`INCR4`) 3 – 8 beats (`INCR8`) 4 – 16 beats (`INCR16`)	Maximum length of burst. This drives the AHB `hburst` signal. The actual burst length may be shorter, depending upon the FIFO size and count of elements to transfer
`PIDX`	0 – `peripherals`-1	Peripheral index. Determines which of the `rx_ready` signals should be used for flow control when source address type T=1

Table 5: Fields of the `src_control` register

4.1.5 Destination Control Register

The destination control register contains a selection of flags that control the operation of the SG-DMA channel with respect to the destination data.

15	10	9	7	5	4	3	2	1	0
PIDX		-		BURST		SIZE		T	

Figure 7: Format of the `dst_control` register

Register	Values	Description
`T`	0 – Memory 1 – Peripheral 2 – None	Destination address type. When set to peripheral type, the `tx_ready[PIDX]` signal from the peripheral must be asserted before a transfer will take place. When set to memory, the transfer will occur unconditionally. When set to none,

		dst_address is not used and the data will be discarded
SIZE	0 – 1 byte 1 – 2 bytes 2 – 4 bytes 3 – 8 bytes	Size of each access when T=1. This drives the AHB hsize signal. dst_address must be aligned according to the transfer size. When T=0, this should be set to 0, and access size will be determined automatically depending upon count and alignment
BURST	0 – Undefined (INCR) 1 – 1 beat (SINGLE) 2 – 4 beats (INCR4) 3 – 8 beats (INCR8) 4 – 16 beats (INCR16)	Maximum length of burst. This drives the AHB hburst signal. The actual burst length may be shorter, depending upon the FIFO size and count of elements to transfer
PIDX	0 – peripherals-1	Peripheral index. Determines which of the tx_ready signals should be used for flow control when destination address type T=1

Table 6: Fields of the dst_control register

4.1.6 Count X Register

The count X register contains the number of transfers that should occur in the X dimension. The number of bytes transferred in the X dimension is (1 << src_control.SIZE) * count_x.

If src_control.SIZE and dst_control.SIZE differ, the count of transfers to the destination, is count_x * (1 << src_control.SIZE) / (1 << dst_control.SIZE).

For transfers where the address type is memory (src_control.T or dst_control.T = 0), reads or writes can be coalesced, providing that the address is appropriately aligned, count is sufficiently high, and the address increment is the same as the size. For example, specifying a copy where control.T=0, control.SIZE=0, address=0, count_x=4, inc_x=1 can result in a single word accesses, instead of 4-byte accesses. Coalescing is not performed when the address type is peripheral.

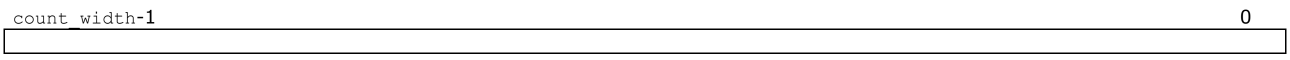

Figure 8: Format of the count_x register

4.1.7 Count Y Register

The count Y register contains the number of iterations, minus 1, of transfers in the X direction (the Y dimension). So, for a 1-dimensional transfer, set count_y to 0. The total number of bytes transferred is ((1 << src_control.SIZE) * count_x) * (count_y + 1).

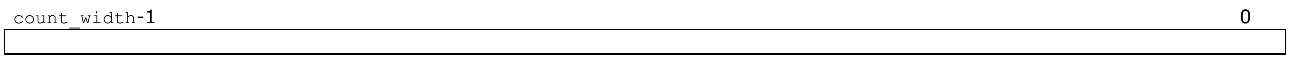

Figure 9: Format of the count_y register

4.1.8 Source Increment X Register

The source increment X register contains a signed integer that is added to the `src_address` register after each transfer in the X dimension.

inc_width-1	0

Figure 10: Format of the `src_inc_x` register

4.1.9 Source Increment Y Register

The source increment Y register contains a signed integer that is added to the `src_address` register after each transfer in the Y dimension.

inc_width-1	0

Figure 11: Format of the `src_inc_y` register

4.1.10 Destination Increment X Register

The destination increment X register contains a signed integer that is added to the `dst_address` register after each transfer in the X dimension.

inc_width-1	0

Figure 12: Format of the `dst_inc_x` register

4.1.11 Destination Increment Y Register

The destination increment Y register contains a signed integer that is added to the `dst_address` register after each transfer in the Y dimension.

inc_width-1	0

Figure 13: Format of the `dst_inc_y` register

4.1.12 Status Register

The status register contains a selection of flags that indicate the current status of the SG-DMA channel. The `AC` flag is read-only. To clear the `DC`, `DCO` or `ER` flags, write a 1 to it. Writing 0 will leave it unchanged.

	3	2	1	0
-	ER	DCO	DC	AC

Figure 14: Format of the `status` register

Register	Values	Description
`AC`	0 – Not complete 1 – Complete	All descriptors complete
`DC`	0 – Not complete 1 – Complete	Descriptor complete
`DCO`	0 – No overflow	Descriptor complete overflow. This indicates the

	1 – Overflow	DC flag was already set when a descriptor was completed
`ER`	0 – No error 1 – Error	Indicates an error occurred during (such as hresp not OK during a read or write or an unsupported value in a descriptor).

Table 7: Fields of the `status` register

4.1.13 Control Register

The control register contains a selection of flags that control the operation of the SG-DMA channel.

	5	4	3	2	1	0
	DD	SP	ERIE	DCIE	ACIE	E

Figure 15: Format of the `control` register

Register	Values	Description
`E`	0 – Disabled 1 – Enabled	Enables the SG-DMA channel
`ACIE`	0 – Disabled 1 – Enabled	All descriptors complete interrupt enable
`DCIE`	0 – Disabled 1 – Enabled	Descriptor complete interrupt enable
`ERIE`	0 – Disabled 1 – Enabled	Error interrupt enable
`SP`	0 – Continue 1 – Stop	Stop transfer immediately. Data may be lost
`DD`	0 – Enable during debug 1 – Disable during debug	Disable channel (after current burst) when debugger is active

Table 8: Fields of the `control` register

4.1.14 Current Address Register

The current address register contains the address of the current descriptor being processed. When the SG-DMA is idle, the `current` register should be written with the address of the first descriptor in order to start processing. Writing the `current` register with 0, will set `status.AC` to 1.

BITS-1 … 0

Figure 16: Format of the `current` register

4.1.15 Calculation Control Register

The calculation control register contains a selection of flags that control calculations (CRC/IPChecksum) that can be performed on data passed through the SG-DMA channel.

	2	1	0
-	IPE	BO	CE

Figure 17: Format of the `calc_control` register

Register	Values	Description
CE	0 – Disabled 1 – Enabled	Enables CRC calculation. Only implemented if CRC_ENABLED is TRUE
BO	0 – LSB first 1 – MSB first	Bit ordering for CRC. Only implemented if CRC_ENABLED is TRUE
IPE	0 – Disabled 1 – Enabled	Enable IP checksum calculation. Only implemented if IP_CHECKSUM_ENABLED is TRUE

Table 9: Fields of the `control` register

4.1.16 CRC Register

The CRC register contains the computed CRC value. It should be initialised before a CRC operation is started to the initialisation value specified by the corresponding CRC standard (typically all zeros or all ones). The CRC register is only implemented if CRC_ENABLED is TRUE.

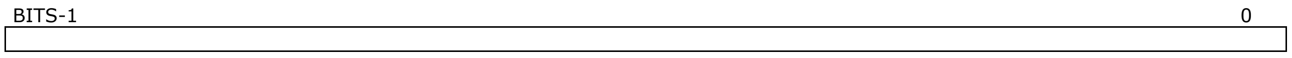

Figure 18: Format of the `crc` register

4.1.17 Polynomial Register

The CRC polynomial register contains the CRC polynomial coefficients. The most significant bit of the polynomial is implicit and does not need to be written into this register. For example, CRC-32 has a 33-bit polynomial, but only the lower 32-bits are required. If a polynomial has fewer than 32-bits, it should be left aligned in this register, with the least-significant bits being set to 0. The CRC polynomial register is only implemented if CRC_ENABLED is TRUE.

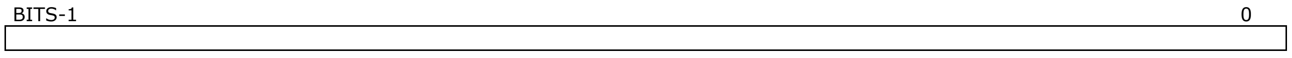

Figure 19: Format of the `polynomial` register

4.1.18 IP Checksum Register

The IP checksum register contains a 16-bit IP checksum. It should be initialised to 0. The IP checksum register is only implemented if IP_CHECKSUM_ENABLED is TRUE.

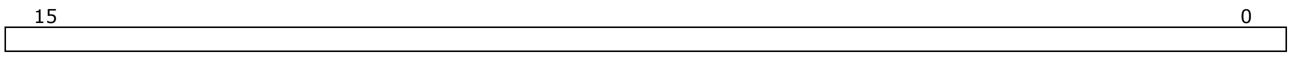

Figure 20: Format of the `ip_checksum` register

4.2 Descriptor Format

The SG-DMA transfer descriptors are stored in memory and are read by the SG-DMA. Three descriptor formats are supported:

- a tiny descriptor for basic transfers that minimizes memory footprint and descriptor read overhead
- a small descriptor that supports the same functionality as the tiny descriptors with the addition of the `next_address` field for chaining descriptors
- a full descriptor that has a larger memory footprint but offers full use of all of the SG-DMA's features.

Small and full descriptors can be chained together via the `next_address` field. When the SG-DMA completes one descriptor, if the `next_address` field is non-`NULL`, the descriptor at that address is read and the corresponding transfer takes place. Chained descriptors can be a mixture of small and full descriptors.

An interrupt can be raised after a descriptor is completed by setting its `desc_control.DCIE` field to 1.

With the exception of the `desc_control` field, the format of the descriptor fields corresponds to the format of the SG-DMA's registers.

4.2.1 Tiny and Small Descriptors

Register	Address offset	Description
`desc_control`	0x00	Descriptor control
`src_address`	0x04	Source address
`dst_address`	0x08	Destination address
`count`	0x0c	Count

Table 10: Tiny Descriptor Format

Register	Address offset	Description
`desc_control`	0x00	Descriptor control
`next_address`	0x04	Address of next descriptor
`src_address`	0x08	Source address
`dst_address`	0x0c	Destination address
`count`	0x10	Count

Table 11: Small Descriptor Format

15 10	9 8	7 6	5 3	2	1 0
PIDX	BURST	SIZE	MODE	DCIE	DT

Figure 21: Format of the `desc_control` field for Tiny and Small Descriptors

Register	Values	Description
`DT`	1 – Small 2 – Tiny	Descriptor type
`DCIE`	0 – Disable interrupt 1 – Enable interrupt	Descriptor complete interrupt enable
`MODE`	0 – Memory to memory 1 – Memory to peripheral 2 – Peripheral to memory 3 – Memory to FIFO 4 – FIFO to memory 5 – None to memory 6 – Memory to none	Transfer mode
`SIZE`	0 – 1 byte	Size of each access

	1 – 2 bytes 2 – 4 bytes 3 – 8 bytes	
BURST	0 – 1 beat 1 – 4 beats 2 – 8 beats 3 – 16 beats	Length of burst
PIDX	0 – `peripherals`-1	Peripheral index. Determines which peripheral to use for flow control, if `MODE` is not 0

Table 12: Fields of the `desc_control` field for Tiny and Small Descriptors

When a tiny or small descriptor is read, the SG-DMA's registers are set according to the following logic:

```
next_address = tiny ? NULL : next_address
src_address = src_address
dst_address = dst_address
src_control.T = (desc_control.MODE == 2) || (desc_control.MODE == 4)
src_control.SIZE = desc_control.SIZE
src_control.BURST = desc_control.BURST
src_control.PIDX = desc_control.PIDX
dst_control.T = (desc_control.MODE == 1) || (desc_control.MODE == 3)
dst_control.SIZE = desc_control.SIZE
dst_control.BURST = desc_control.BURST
dst_control.PIDX = desc_control.PIDX
count_x = count
count_y = 0
src_inc_x = desc_control.MODE == 4 ? 0 : (1 << desc_control.SIZE)
src_inc_y = 0
dst_inc_x = desc_control.MODE == 3 ? 0 : (1 << desc_control.SIZE)
dst_inc_y = 0
control.DCIE = desc_control.DCIE
```

The total number of bytes copied is `(1 << desc_control.SIZE) * count`.

4.2.2 Full Descriptor

Register	Address offset	Description
`desc_control`	0x00	Descriptor control
`next_address`	0x04	Address of next descriptor
`src_address`	0x08	Source address
`dst_address`	0x0c	Destination address
`src_control`	0x10	Source control
`dst_control`	0x14	Destination control
`count_x`	0x18	Count X
`count_y`	0x1c	Count Y
`src_inc_x`	0x20	Source address X increment
`src_inc_y`	0x24	Source address Y increment
`dst_inc_x`	0x28	Destination address X increment
`dst_inc_y`	0x2c	Destination address Y increment

Table 13: Full Descriptor Format

	11 10	9 8	7 3	2	1 0
	IPE	CE	-	DCIE	T

Figure 22: Format of the `desc_control` field for Full Descriptors

Register	Values	Description
DT	0 – Full	Descriptor type
DCIE	0 – Disable interrupt 1 – Enable interrupt	Descriptor complete interrupt enable
CE	0 – No change 1 – Enable CRC 2 – Disable CRC	CRC enable
IPE	0 – No change 1 – Enable IP checksum 2 – Disable IP checksum	IP checksum enable

Table 14: Fields of the `desc_control` field for Full Descriptors

4.3 Interrupts

The SG-DMA supports the following interrupts.

- Per-channel all descriptors complete interrupt
- Per-channel descriptor complete interrupt
- Per-channel error interrupt

The all descriptors complete interrupt will be raised when the SG-DMA has finished the transfer specified by the current descriptor and the `next_address` register is `NULL`. The `AC` flag in the `status` register will be set 1 to indicate this. When the `AC` flag in the `status` register is set to 1 and the `ACIE` flag in the `control` register is set to 1, the all descriptors complete interrupt will be asserted. The `AC` flag is cleared when the `current` register is next written.

The descriptors complete interrupt will be raised when the SG-DMA has finished the transfer specified by the current descriptor. The `DC` flag in the `status` register will be set 1 to indicate this. When the `DC` flag in the `status` register is set to 1 and the `DCIE` flag in the `control` register is set to 1, the descriptors complete interrupt will be asserted.

The error interrupt will be raised then the `ER` flag in the `status` register is 1 and the `ERIE` flag in the `control` register is set to 1. This indicates an error was detected during either the reading of a descriptor or the transfer specified by the descriptor.

4.4 Error Handling

When the SG-DMA detects a read or write error on the AHB bus, either the reading of the descriptor will be terminated, or the transfer will be terminated. The termination may not be immediate, as any burst in progress will be allowed to complete.

The SG-DMA will set the `status.ER` flag to indicate the error, and no further transfers will take place on the corresponding channel until the flag is cleared. Before clearing the flag, the `current` register should be set to `NULL`, otherwise the descriptor will be re-read. If chained descriptors are being used, the current register should contain the address of the descriptor that caused the error.

4.5 Stopping a Transfer

A transfer on DMA channel can be stopped by writing to the channel's `control.SP` register. The transfer may not stop immediately, as a burst that is in progress may need to be

completed. When the transfer is stopped, the `control.E` register will be cleared to indicate this.
To start a new transfer, the `current` register should first be written with 0, to set the `status.AC` flag. The channel should be reenabled by setting `control.E` to 1, and then the address of the new descriptor can be written to the `current` register.

4.6 Channel Priority

Transfers on the lowest valid channel have priority. For example, if the transfer specified by channel 0 is ready to be processed as the same time as the transfer on channel 1, channel 0 will be completed first. A higher channel (with lower priority) may be switched to part way through the transfer on a lower channel (with higher priority); if the transfer on the lower channel cannot be completed due to a peripheral indicating it is not ready.

4.7 Examples

4.7.1 Memory to Memory Copy

To implement a `memcpy(dest, src, n)` like memory copy, to copy `n` bytes from `dest` to `src`, a tiny descriptor can be initialised as follows:

```
esi_sg_dma_tiny_desc_t desc;

desc.desc_control = ESI_SG_DMA_DESC_MODE_M_TO_M | ESI_SG_DMA_DESC_TYPE_TINY;
desc.src_address = src;
desc.dst_address = dest;
desc.count = n;
```

4.7.2 Memory Set

To implement a `memset(dest, c, n)` like memory fill operation, that fills the first `n` bytes of memory at the address `dest` with the value `c`, a full descriptor should be used, where the source increment is set as 0, so that the same data is repeatedly copied:

```
esi_sg_dma_full_desc_t desc;

char c = 0x00;

desc.desc_control = ESI_SG_DMA_DESC_TYPE_FULL;
desc.next_address = NULL;
desc.src_address = &c;
desc.dst_address = dest;
desc.src_control = ESI_SG_DMA_MEMORY;
desc.dst_control = ESI_SG_DMA_MEMORY;
desc.count_x = n;
desc.count_y = 0;
desc.src_inc_x = 0;
desc.src_inc_y = 0;
desc.dst_inc_x = 1;
desc.dst_inc_y = 0;
```

4.7.3 Memory Zero

To implement a `bzero (dest, n)` like memory zero operation, that zeros the first `n` bytes of memory at the address `dest`, a tiny descriptor should be used, where the source type is set as NONE. This halves the bus bandwidth compared to the memory set example above, as no data has to be read, only written:

```
esi_sg_dma_tony_desc_t desc;

desc.desc_control = ESI_SG_DMA_DESC_MODE_N_TO_M | ESI_SG_DMA_DESC_TYPE_TINY;
desc.src_address = NULL;
desc.dst_address = dest;
desc.count = n;
```

4.7.4 Memory to FIFO

To copy a one-dimensional array of data from memory to a peripheral's FIFO, a tiny descriptor can be used. In this example, we copy a string to a UART's transmit FIFO:

```
esi_sg_dma_small_desc_t desc;
char src_data[] = "hello";
esi_device_info_t *uart_device;
esi_uart_t *uart;

desc.desc_control = (uart_device->dma << 10)
                  | ESI_SG_DMA_DESC_BURST_1
                  | ESI_SG_DMA_DESC_SIZE_BYTE
                  | ESI_SG_DMA_DESC_MODE_M_TO_F
                  | ESI_SG_DMA_DESC_TYPE_TINY;
desc.src_address = src_data;
desc.dst_address = &uart->tx_data;
desc.count = strlen(src_data);
```

The burst length should be set so that it is smaller than the destination FIFO's depth. The size of the access should be set to match the width of the destination FIFO.

4.7.5 Double Buffering (Ping-Pong Buffers)

When receiving data from a peripheral, double buffering can be used to ensure that a receive buffer is always available to avoid data loss. Two buffers are used: buffer 0 for receiving data, and buffer 1 for holding data that is being processed. When the receive buffer becomes full, the buffers are swapped, and buffer 1 is used for receiving data, while buffer 0 is processed, and so on.

To implement this, two small descriptors are used, and chained together in a circular loop. The descriptor complete interrupt enable is set, so that after a buffer becomes full, an interrupt handler can swap the buffers.

```
esi_sg_dma_small_desc_t desc0, desc1;
char buffers[2][BUF_SIZE];
int processing_buffer;
esi_device_info_t *uart_device;
esi_uart_t *uart;

desc0.desc_control = (uart_device->dma << 10)
                  | ESI_SG_DMA_DESC_BURST_1
```

```
                      | ESI_SG_DMA_DESC_SIZE_BYTE
                      | ESI_SG_DMA_DESC_MODE_F_TO_M
                      | ESI_SG_DMA_DESC_DC_INT_ENABLE
                      | ESI_SG_DMA_DESC_TYPE_SMALL;
desc0.next_address = &desc1;
desc0.src_address = &uart->rx_data;
desc0.dst_address = &buffers[0][0];
desc0.count = BUF_SIZE;

desc1.desc_control = (uart_device->dma << 10)
                      | ESI_SG_DMA_DESC_BURST_1
                      | ESI_SG_DMA_DESC_SIZE_BYTE
                      | ESI_SG_DMA_DESC_MODE_F_TO_M
                      | ESI_SG_DMA_DESC_DC_INT_ENABLE
                      | ESI_SG_DMA_DESC_TYPE_SMALL;
desc1.next_address = &desc0;
desc1.src_address = &uart->rx_data;
desc1.dst_address = &buffers[1][0];
desc1.count = BUF_SIZE;

processing_buffer = 1;

void dma_interrupt_handler(void) {
      …
      /* Each time we get an descriptor complete interrupt, swap buffers. */
      processing_buffer ^= 1;
      process_data(buffers[processing_buffer]);
      …
}
```

If it is possible that processing a buffer may take longer than it does to fill one up with data, triple buffering can be used. This simply adds a third buffer and descriptor.

4.7.6 Gather

Gather operations use multiple descriptors to combine multiple blocks of memory distributed at random memory addresses in to one contiguous block. For example, this might be used in a networking application, where different parts of a packet to be transmitted need to be gathered together for transmission.

To achieve this, simply use one small descriptor per block of memory that is to be combined, and chain them together:

```
esi_sg_dma_small_desc_t desc0, desc1, desc2;
char *src0, *src1, *src2;
int src0_size, src1_size, src2_size;
char *dst;

desc0.desc_control = ESI_SG_DMA_DESC_MODE_M_TO_M | ESI_SG_DMA_DESC_TYPE_SMALL;
desc0.next_address = &desc1;
desc0.src_address = src0;
desc0.dst_address = &dst[0];
desc0.count = src0_size;

desc1.desc_control = ESI_SG_DMA_DESC_MODE_M_TO_M | ESI_SG_DMA_DESC_TYPE_SMALL;
desc1.next_address = &desc2;
desc1.src_address = src1;
desc1.dst_address = &dst[src0_size];
desc1.count = src1_size;
```

```
desc2.desc_control = ESI_SG_DMA_DESC_MODE_M_TO_M | ESI_SG_DMA_DESC_TYPE_SMALL;
desc2.next_address = NULL;
desc2.src_address = src2;
desc2.dst_address = &dst[src0_size+src1_size];
desc2.count = src2_size;
```

4.7.7 Scatter

Scatter operations are the opposite of gathers. They can be used to split up a contiguous block of data to multiple random addresses.

```
esi_sg_dma_small_desc_t desc0, desc1, desc2;
char *dst0, *dst1, *dst2;
int dst0_size, dst1_size, dst2_size;
char *src;

desc0.desc_control = ESI_SG_DMA_DESC_MODE_M_TO_M | ESI_SG_DMA_DESC_TYPE_SMALL;
desc0.next_address = &desc1;
desc0.src_address = &src[0];
desc0.dst_address = dst0;
desc0.count = dst0_size;

desc1.desc_control = ESI_SG_DMA_DESC_MODE_M_TO_M | ESI_SG_DMA_DESC_TYPE_SMALL;
desc1.next_address = &desc2;
desc1.src_address = &src[dst0_size];
desc1.dst_address = dst1;
desc1.count = dst1_size;

desc2.desc_control = ESI_SG_DMA_DESC_MODE_M_TO_M | ESI_SG_DMA_DESC_TYPE_SMALL;
desc2.next_address = NULL;
desc2.src_address = &src[dst0_size+dst1_size];
desc2.dst_address = dst2;
desc2.count = dst2_size;
```

4.7.8 Endian Conversion

2D transfers can be used for reordering data, such as in an endian conversion, where the most-significant bytes need to be swapped with the least significate bytes. For example, to endian swap the array:

```
unsigned long src[4] = {0x11223344, 0x55667788, 0x99aabbcc, 0xddeeff00};
```

So that it becomes:

```
unsigned long dst[4] = {0x44332211, 0x88776655, 0xccbbaa99, 0x00ffeedd};
```

A full descriptor should be used, where the source X increment is 1, but the destination X increment is -1, to reverse the data. The destination address therefore needs to start offset by 3, with a Y increment of 8 to jump to the next word.

```
esi_sg_dma_full_desc_t desc;

desc.desc_control = ESI_SG_DMA_DESC_TYPE_FULL;
desc.next_address = NULL;
desc.src_address = &src[0];
desc.dst_address = ((char)&dst[0]) + 3;
desc.src_control = ESI_SG_DMA_SIZE_BYTE | ESI_SG_DMA_MEMORY;
desc.dst_control = ESI_SG_DMA_SIZE_BYTE | ESI_SG_DMA_MEMORY;
desc.count_x = 4;
desc.count_y = 3;
desc.src_inc_x = 1;
desc.src_inc_y = 0;
desc.dst_inc_x = -1;
desc.dst_inc_y = 8;
```

4.7.9 Deinterleave

2D transfers can be used to deinterleave data. For example, in an audio application, 16-bit left and right samples may be stored interleaved, but need to be deinterleaved for individual channel processing. For example, to deinterleave:

```
#define SAMPLES    6
#define CHANNELS   2

short src[SAMPLES*CHANNELS] = {
  0x0, 0x1000, 0x1, 0x1001, 0x2, 0x1002, 0x3, 0x1003, 0x4, 0x1004, 0x5, 0x1005
};
```

To:

```
short dest[SAMPLES*CHANNELS] = {
  0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x1000, 0x1001, 0x1002, 0x1003, 0x1004, 0x1005
};
```

We use a full descriptor where the X increment steps through the samples for each channel, and the Y increment steps through channels:

```
esi_sg_dma_full_desc_t desc;

desc.desc_control = ESI_SG_DMA_DESC_TYPE_FULL;
desc.next_address = NULL;
desc.src_address = &src[0];
desc.dst_address = &dst[0];
desc.src_control = ESI_SG_DMA_SIZE_BYTE | ESI_SG_DMA_MEMORY;
desc.dst_control = ESI_SG_DMA_SIZE_BYTE | ESI_SG_DMA_MEMORY;
desc.count_x = SAMPLES;
desc.count_y = CHANNELS-1;
desc.src_inc_x = BYTES_PER_SAMPLE * CHANNELS;
desc.src_inc_y = -BYTES_PER_SAMPLE * CHANNELS * SAMPLES + BYTES_PER_SAMPLE;
desc.dst_inc_x = BYTES_PER_SAMPLE;
desc.dst_inc_y = 0;
```

4.7.10 Matrix Transpose

2D transfers can be used to transpose the sub-elements of a matrix:

```
#define FULL_WIDTH      8
#define FULL_HEIGHT     8
#define WIDTH           4
#define HEIGHT          4
#define X_OFFSET        2
#define Y_OFFSET        2

char src[FULL_WIDTH*FULL_HEIGHT] = {
      8, 8, 8, 8, 8, 8, 8, 8,
      0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 1, 2, 3, 4, 0, 0,
      0, 0, 1, 2, 3, 4, 0, 0,
      0, 0, 1, 2, 3, 4, 0, 0,
      0, 0, 1, 2, 3, 4, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0,
      8, 8, 8, 8, 8, 8, 8, 8,
};
```

To:

```
char dst[FULL_WIDTH*FULL_HEIGHT] = {
      8, 8, 8, 8, 8, 8, 8, 8,
      0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 1, 1, 1, 1, 0, 0,
      0, 0, 2, 2, 2, 2, 0, 0,
      0, 0, 3, 3, 3, 3, 0, 0,
      0, 0, 4, 4, 4, 4, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0,
      8, 8, 8, 8, 8, 8, 8, 8,
};

esi_sg_dma_full_desc_t desc;

desc.desc_control = ESI_SG_DMA_DESC_TYPE_FULL;
desc.next_address = NULL;
desc.src_address = &src[FULL_WIDTH*Y_OFFSET+X_OFFSET];
desc.dst_address = &dst[FULL_WIDTH*Y_OFFSET+X_OFFSET];
desc.src_control = ESI_SG_DMA_SIZE_BYTE | ESI_SG_DMA_MEMORY;
desc.dst_control = ESI_SG_DMA_SIZE_BYTE | ESI_SG_DMA_MEMORY;
desc.count_x = WIDTH;
desc.count_y = HEIGHT-1;
desc.src_inc_x = 1;
desc.src_inc_y = FULL_WIDTH-WIDTH;
desc.dst_inc_x = FULL_WIDTH;
desc.dst_inc_y = -FULL_WIDTH*WIDTH + 1;
```

4.7.11 Memory to Peripheral with Wrapping Address

Some eSi-Crypto APB cores require the data to be processed to be written to and then read back from registers at successive addresses. These crypto cores typically work on small blocks of data at a time (E.g. 128-bits / 16-bytes). In order to stream in and out a larger block of data, a 2D transfer is required, as while the memory address may simply need to increment, the peripheral address needs to wrap every 16 bytes.

The following example streams data in to the eSi-AES core using word sized accesses, in blocks of 16-bytes, for a 128-bit AES operation. `count_x` is set to 16 / 4, as we need four 32-bit word accesses to read 16 bytes. `count_y` is set to the number of 16-byte transfers needed for the complete source data, minus 1. For the source data, the source address is set to increment by 4 after each word is read. The destination address also increments by 4 after each word is written, but then wraps backwords by 16 bytes, after every 16 bytes.

```
esi_sg_dma_full_desc_t desc;
esi_aes_t *aes;
char src[LENGTH];

desc.desc_control = ESI_SG_DMA_DESC_TYPE_FULL;
desc.next_address = NULL;
desc.src_address = src;
desc.dst_address = &aes->data_in[0];
desc.src_control = ESI_SG_DMA_BURST_4
                  | ESI_SG_DMA_SIZE_WORD
                  | ESI_SG_DMA_MEMORY;
desc.dst_control = ESI_SG_DMA_BURST_4
                  | ESI_SG_DMA_SIZE_WORD
                  | ESI_SG_DMA_PERIPHERAL;
desc.count_x = 16 / 4;
desc.count_y = (LENGTH / 16) - 1;
desc.src_inc_x = 4;
desc.src_inc_y = 0;
desc.dst_inc_x = 4;
desc.dst_inc_y = -16;
```

4.7.12 Circular Buffer

A circular buffer can be implemented with a single descriptor that is chained to itself. So each time the buffer is filled up, it immediately refills from the start.

```
esi_sg_dma_small_desc_t desc;
char buffer[BUF_SIZE];
esi_device_info_t *uart_device;
esi_uart_t *uart;

desc.desc_control = (uart_device->dma << 10)
                  | ESI_SG_DMA_DESC_BURST_1
                  | ESI_SG_DMA_DESC_SIZE_BYTE
                  | ESI_SG_DMA_DESC_MODE_F_TO_M
                  | ESI_SG_DMA_DESC_TYPE_SMALL;
desc.next_address = &desc;
desc.src_address = &uart->rx_data;
desc.dst_address = &buffer[0];
desc.count = BUF_SIZE;
```

How a circular buffer is read, depends on the speed of the data being received. Generally, it would be unsafe to wait for the descriptor complete interrupt to be raised before trying to read from the buffer, as if another element of data is received before the interrupt is handled, the first element in the buffer would be lost. Instead, the data can be read from the buffer as it is being filled up, by reading the `count_x` register as it is filled, to work out how much valid data the buffer contains.

4.7.13 CRC

For any transfer, a CRC can be calculated on the data transferred. The CRC is calculated across chained descriptors. Before starting a transfer, the `crc` register should be initialised with the initial value appropriate for the desired CRC and the `polynomial` register with the polynomial coefficients for the CRC. The `calc_control.BO` set to indicate whether data should be processed LSB or MSB first. After the transfer is compete, the computed CRC value can be read from the `crc` register.

If a CRC needs to be calculated on some data without copying it to a destination, the `dst_control.T` field can be set to `NONE`, so that the data is only read, not written.

If only some parts of the data should have the CRC calculated for, the `CE` field in the full descriptor can be used to disable or enable the calculation on a per descriptor basis.

5 Revision History

Hardware Revision	Software Release	Description
1	4.0.3	Initial release.
2	4.1.14	Add CRC support.
3	6.0.3	Added `debug_active` input. Added `control.DD` field.
4	6.0.4	Added `address_width` parameter. Added `count_width` parameter. Added `inc_width` parameter.
4	8.0.0	Added `ahb_data_width` parameter.

Table 15: Revision History