Systems and Technology Group

Developing Code for Cell - DMA

Course Code: L3T2H1-39
Cell Ecosystem Solutions Enablement

Class Objectives – Things you will learn

- **How MFC commands were used to access main storage and maintain synchronization with other processors and devices in the system**
- **DMA commands and DMA-command tag groups**
- **DMA transfers and how to initiate a DMA transfer from an SPE**
- **DMA-list transfers and how to initiate a DMA-list**
- **Double buffering and multibuffering**

Class Agenda

- **MFC Commands**
- **DMA Commands**
  - MFC DMA Commands
  - MFC Synchronization Commands
  - MFC Atomic Commands
- **DMA-Command Tag Groups**
- **DMA Transfers**
  - How to initiate a DMA transfer from an SPE
  - Sample DMA Get Command (SPE)
- **DMA-List Transfers**
  - DMA List – SPU Get from Main Memory
  - DMA –List Transfers - *Creating the list*
  - Initiating the Transfers Specified in the List
- **DMA To/From Another SPE**
- **DMA Command Status**
- **DMA Transfers Example**
- **Double Buffering**
  - Moving Double-Buffered Data and DMA Transfers Using a Double-Buffering Method
  - How to Initiate a Buffer Transfer and Wait for a Buffer Transfer to Complete
  - Example Illustrates Double Buffering
- **Multibuffering**

Cell's primary Communication Mechanisms

- DMA transfers, mailbox messages, and signal-notification
- all three are implemented and controlled by the SPE's MFC

Mechanism	Description
DMA transfers	Used to move data and instructions between main storage and an LS. SPEs rely on asynchronous DMA transfers to hide memory latency and transfer overhead by moving information in parallel with SPU computation.
Mailboxes	Used for control communication between an SPE and the PPE or other devices. Mailboxes hold 32-bit messages. Each SPE has two mailboxes for sending messages and one mailbox for receiving messages.
Signal notification	Used for control communication from the PPE or other devices. Signal notification (also called *signaling)* uses 32-bit registers that can be configured for one-sender-to-one-receiver signalling or many-senders-to-one-receiver signalling.

MFC Commands

MFC Commands

- Main mechanism for SPUs to
  - access main storage
  - maintain synchronization with other processors and devices in the system
- Can be issued either SPU via its MFC by PPE or other device, as follows:
  - Code running on the SPU issues an MFC command by executing a series of writes and/or reads using **channel instructions**
  - Code running on the PPE or other devices issues an MFC command by performing a series of stores and/or loads to **memory-mapped I/O** (MMIO) registers in the MFC
- MFC commands are queued in one of two independent MFC command queues:
  - MFC SPU Command Queue — For channel-initiated commands by the associated SPU
  - MFC Proxy Command Queue — For MMIO-initiated commands by the PPE or other device

DMA Commands

- MFC commands that transfer data are referred to as DMA commands
- Transfer direction for DMA commands referenced from the SPE
  - Into an SPE (from main storage to local store) → **get**
  - Out of an SPE (from local store to main storage) → **put**

MFC Command Suffixes

	Suffix	Description
Start SPU	s	Starts the execution of the SPU at the current location indicated by the SPU Next Program Counter Register after the data has been transferred into or out of the local store.
Fenced	f	Tag-specific fence. Commands with a tag-specific fence are locally ordered with respect to all previously-issued commands within the same tag group and command queue.
Barrier	b	Tag-specific barrier. Commands with a tag-specific barrier are locally ordered with respect to all previously-issued commands within the same tag group and command queue and all subsequently-issued commands to the same command queue with the same tag.
List	l	List command. Executes a list of DMA transfer elements located in local store. The maximum number of elements is 2,048, and each element describes a transfer of up to 16 KB.

MFC DMA Commands

Mnemonic	Supported By[1]	Description
Put Commands		
put	PPE, SPE	Moves data from local store to the effective address.
puts	PPE	Moves data from local store to the effective address and starts the SPU after the DMA operation completes.
putf	PPE, SPE	Moves data from local store to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
putb	PPE, SPE	Moves data from local store to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
putfs	PPE	Moves data from local store to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after the DMA operation completes.
putbs	PPE	Moves data from local store to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after the DMA operation completes.
putl	SPE	Moves data from local store to the effective address using an MFC list.
putlf	SPE	Moves data from local store to the effective address using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
putlb	SPE	Moves data from local store to the effective address using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

MFC DMA Commands (Cont'd)

Get Commands		
get	PPE, SPE	Moves data from the effective address to local store.
gets	PPE	Moves data from the effective address to local store, and starts the SPU after the DMA operation completes.
getf	PPE, SPE	Moves data from the effective address to local store with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
getb	PPE, SPE	Moves data from the effective address to local store with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
getfs	PPE	Moves data from the effective address to local store with fence (this command is locally ordered with respect to all previously issued commands within the same tag group), and starts the SPU after the DMA operation completes.
getbs	PPE	Moves data from the effective address to local store with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue), and starts the SPU after the DMA operation completes.
getl	SPE	Moves data from the effective address to local store using an MFC list.
getlf	SPE	Moves data from the effective address to local store using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
getlb	SPE	Moves data from the effective address to local store using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

MFC Synchronization Commands

MFC synchronization commands

- Used to control the order in which DMA storage accesses are performed
  - Four atomic commands (**getllar**, **putllc**, **putlluc**, and **putqlluc**),
  - Three send-signal commands (**sndsig**, **sndsigf**, and **sndsigb**), and
  - Three barrier commands (**barrier**, **mfcsync**, and **mfceieio**).

Command	Supported By[1]	Description
barrier	PPE, SPE	Barrier type ordering. Ensures ordering of all preceding, nonimmediate DMA commands with respect to all commands following the barrier command within the same command queue. The barrier command has no effect on the immediate DMA commands: **getllar**, **putllc**, and **putlluc**.
mfceieio	PPE, SPE	Controls the ordering of get commands with respect to put commands, and of get commands with respect to get commands accessing storage that is caching inhibited and guarded. Also controls the ordering of **put** commands with respect to **put** commands accessing storage that is memory coherence required and not caching inhibited.
mfcsync	PPE, SPE	Controls the ordering of DMA **put** and **get** operations within the specified tag group with respect to other processing units and mechanisms in the system.
sndsig	PPE, SPE	Update SPU Signal Notification Registers in an I/O device or another SPE.
sndsigb	PPE, SPE	Update SPU Signal Notification Registers in an I/O device or another SPE with barrier.
sndsigf	PPE, SPE	Update SPU Signal Notification Registers in an I/O device or another SPE with fence.

1. There is a channel (for SPEs) and/or MMIO register (for PPE) to support the operation.

MFC Atomic Commands

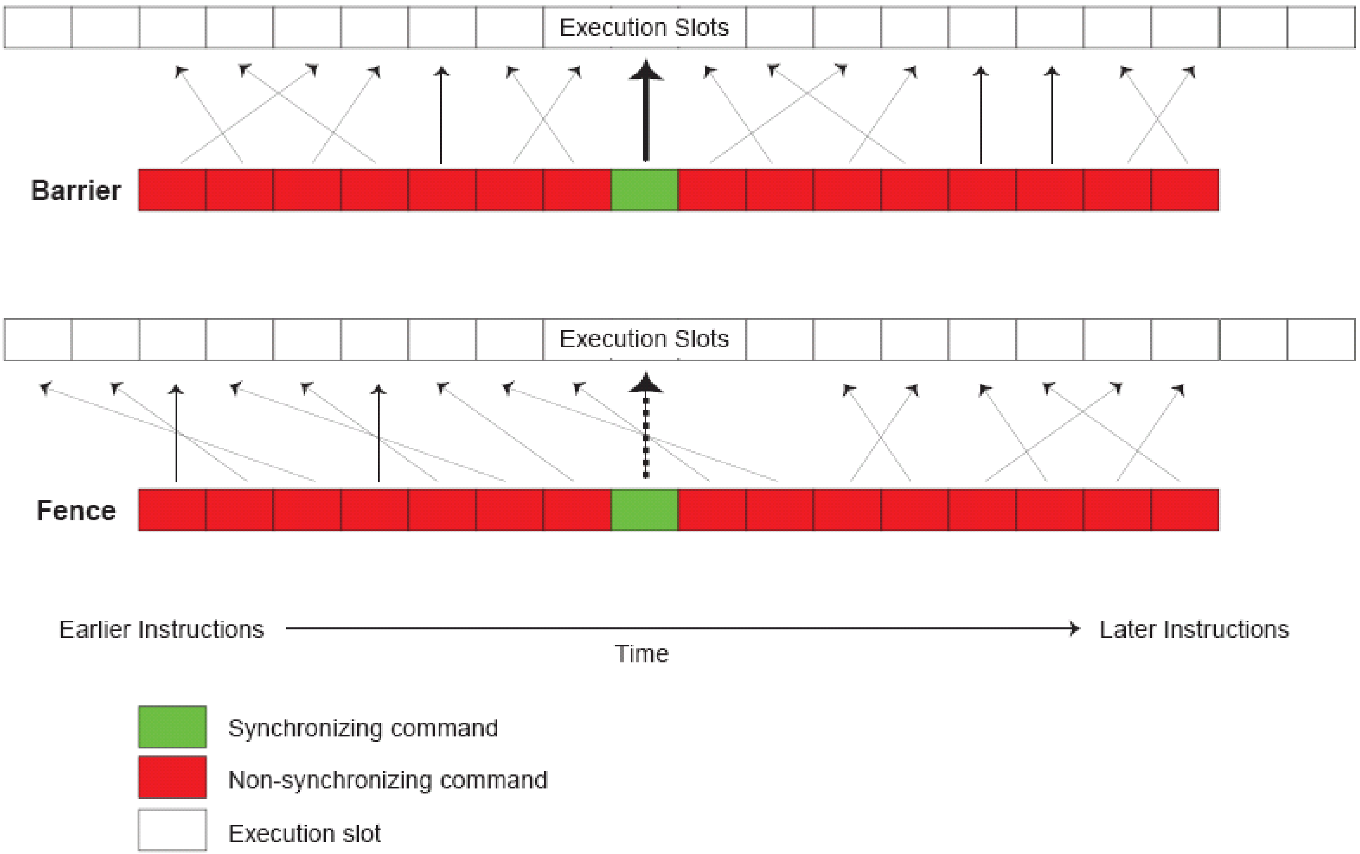
Command	Supported By[1]	Description
getllar	SPE	Get lock line and create a reservation (executed immediately).
putllc	SPE	Put lock line conditional on a reservation (executed immediately).
putlluc	SPE	Put lock line unconditional (executed immediately).
putqlluc	SPE	Put lock line unconditional (queued form).

1. There is a channel to support the operation.

DMA-Command Tag Groups

- 5-bit DMA Tag for all DMA commands (except **getllar**, **putllc**, and **putlluc)**
- Tag can be used to
  - **determine status for entire group or command**
  - **check or wait on the completion of all queued commands in one or more tag groups**
- Tagging is optional but can be useful when using barriers to control the ordering of MFC commands within a single command queue.
- Synchronization: fences and barriers
  - **Execution of a fenced command option is delayed until all previously issued commands within the same tag group have been performed.**
  - **Execution of a barrier command option and all subsequent commands is delayed until all previously issued commands in the same tag group have been performed.**

Barriers and Fences



DMA Transfers

DMA

- **DMA transfers**
  - transfer sizes can be 1, 2, 4, 8, and n*16 bytes (n integer)
  - maximum is 16KB per DMA transfer
  - 128B alignment is preferable
- **DMA command queues per SPU**
  - 16-element queue for SPU-initiated requests
  - 8-element queue for PPE-initiated requests
  - ➢ SPU-initiated DMA is always preferable
- **DMA tags**
  - each DMA command is tagged with a 5-bit identifier
  - same identifier can be used for multiple commands
  - tags used for polling status or waiting on completion of DMA commands
- **DMA lists**
  - a single DMA command can cause execution of a list of transfer requests (in LS)
  - lists implement scatter-gather functions
  - a list can contain up to 2K transfer requests

DMA Transfers

- **Addressing**
  - Main storage: effective address (EA) operand in a DMA command
  - Local Store: local store address (LSA) operand in a DMA command.
- **LS data is accessed sequentially with a minimum step of a quadword**
- **SPE accesses its MFC's DMA-transfer facilities through the channels**
- **To enqueue a DMA command, SPE software writes the MFC Command Parameter Channel Registers with the wrch instruction**
  - 1. Write the LS address to the MFC_LSA channel.
  - 2. Write the EA-high (EAH) to the MFC_EAH channel.
  - 3. Write the EA-low (EAL) to the MFC_EAL channel.
  - 4. Write the transfer size to the MFC_Size channel.
  - 5. Write the tag ID to the MFC_TagID channel.
  - 6. Write the class ID and command opcode to the MFC_Cmd channel.

How to initiate a DMA transfer from an SPE

```
extern void dma_transfer(volatile void *lsa, // local store address
                         unsigned int eah, // high 32-bit effective address
                         unsigned int eal, // low 32-bit effective address
                         unsigned int size, // transfer size in bytes
                         unsigned int tag_id, // tag identifier (0-31)
                         unsigned int cmd); // DMA command
```

An ABI-compliant assembly-language implementation of the subroutine is:

```
.text
.global dma_transfer
dma_transfer:
wrch $MFC_LSA, $3
wrch $MFC_EAH, $4
wrch $MFC_EAL, $5
wrch $MFC_Size, $6
wrch $MFC_TagID, $7
wrch $MFC_Cmd, $8
bi $0
```

How to initiate a DMA transfer from an SPE (Cont'd)

A comparable C implementation using the SPU composite intrinsic, spu_mfcdma64, is:

```
#include <spu_intrinsics.h>
void dma_transfer(volatile void *lsa, unsigned int eah, unsigned int eal,
unsigned int size, unsigned int tag_id, unsigned int cmd)
{
      spu_mfcdma64(lsa, eah, eal, size, tag_id, cmd);
}
```

- The performance of a DMA data transfer is best when the source and destination addresses have the same quadword offsets within a PPE cache line.
- Quadword-offset-aligned data transfers generate full cache-line bus requests for every unrolling, except possibly the first and last unrolling.
- Transfers that start or end in the middle of a cache line transfer a partial cache line (less than 8 quadwords) in the first or last bus request, respectively.

DMA Commands for SPU

Assembler Inst
wrch

↓

Channel Control Intrinsics
spu_writech

↓

Composite Intrinsics
spu_dmfcdma32

↓

MFC Commands
mfc_get

defined as macros in spu_mfcio.h

For details see: SPU C/C++ Language Extensions

Sample DMA Get Command (SPE)

- **DMA get from main memory**

  mfc_get(lsaddr,ea,size,tag_id,tid,rid);

  – lsaddr = target address in SPU local store for fetched data (SPU local address)
  – ea = effective address from which data is fetched (global address)
  – size = transfer size in bytes
  – tag_id = tag-group identifier
  – tid = transfer-class id
  – rid = replacement-class id

- **Also available via "composite intrinsic":**

  spu_mfcdma64(lsaddr, eahi, ealow, size, tag_id, cmd);

DMA-List Transfers

DMA-List Transfers

- A DMA list is a sequence of *transfer elements* (or list elements)
  - initiating DMA-list command
  - sequence of DMA transfers between a **single area of LS** and possibly discontinuous areas in main storage
- DMA lists are stored in an SPE's LS on 8 Byte boundary
- The sequence of transfers is initiated by **getl** or **putl**.
- DMA-list commands can only be issued by SPE
- PPE or other devices can create and store the lists in an SPE's LS
- DMA lists can be used to implement scatter-gather functions between main storage and the LS.

DMA –List Transfers - *Creating the list*

- **List sizes**
  - Each DMA transfer can transfer up to 16 KB
  - the list can have up to 2,048 (2 K) transfer elements.
- **The form of a transfer element is {LTS, EAL}.**
  - LTS: list transfer size
    - the most-significant bit of which serves as an optional stall-and-notify flag
  - EAL: is the low-order 32-bits of an EA
- **Transfer elements are processed sequentially, in the order they are stored.**
- **Stall and Notify Flag**
  - If set for an transfer element, the MFC will stop processing the DMA list after performing the transfer for that element until the SPE program clears the DMA List Command Stall-And-Notify Event from the SPU Read Event Status Channel.
  - This gives programs an opportunity to modify subsequent transfer elements before they are processed by the MFC.

Initiating the Transfers Specified in the List

- List transfer is started by **getl** or **putl** from the SPE whose LS contains the list
- A DMA-list command requires two different types of parameters than those required by a single-transfer DMA command:
  - *MFC_EAL:* This parameter must be written with the *starting local store address (LSA) of the list*, rather than with the EAL. (The EAL is specified in each transfer element.)
  - *MFC_Size:* This parameter must be written with the *size of the list*, rather then the transfer size. (The transfer size is specified in each transfer element.) The list size is equal to the number of transfer elements, multiplied by the size of the transfer-element structure (8 bytes).
- The starting LSA and the EA-high (EAH) are specified only once, in the DMA-list command that initiates the transfers. The LSA is internally incremented based on the amount of data transferred by each transfer element. However, if the starting LSA for each transfer element in a list does not begin on a 16-byte boundary, then hardware automatically increments the LSA to the next 16-byte boundary.
- The EAL for each transfer element is in the 4-GB area defined by EAH. Although each EAL starting address is in a single 4-GB area, individual transfers may cross the 4-GB boundary.

DMA List – SPU Get from Main Memory

- **Provides a gather function**
- **List of source effective addresses created in SPU local store as array of list elements**
  - each array element has 8 bytes, nominally as:

```
struct spu_dma_list_elem {
unsigned int size;
unsigned int ea_low;
};
```

- **List-oriented DMA get:**

  `mfc_getl(lsaddr,ea,list,size,tag_id,tid,rid);`

  - lsaddr = target address in SPU local store for fetched data (SPU local address)
  - ea = effective (high) address that is target of first list element
  - list = address of list element array in SPU local store (must be 8-byte aligned)
  - size = size of list array (must be a multiple of 8 bytes)

DMA To/From Another SPE

- **Address in the other SPE's local store is represented as a 32-bit effective address (global address)**
- **SPE issuing the DMA command needs a pointer to the other SPE's local store as a 32-bit effective address (global address)**
- **PPE code can obtain effective address of an SPE's local store:**

```
#include <libspe.h>
speid_t speid;
void *spe_ls_addr;
spe_ls_addr = spe_get_ls(spuid);
```

- **Effective address of an SPE's local store can then be made available to other SPEs (e.g. via DMA or mailbox)**

DMA Status

DMA Command Status (SPE)

- **DMA read and write commands are non-blocking**
- **Tags, tag groups, and tag masks used for:**
  - checking status of DMA commands
  - waiting for completion of DMA commands
- **Each DMA command has a 5-bit tag**
  - commands with same tag value form a “tag group”
- **Tag mask is used to identify tag groups for status checks**
  - tag mask is a 32-bit word
  - each bit in the tag mask corresponds to a specific tag id:

```
tag_mask = (1 << tag_id)
```

DMA Status Checking (SPE)

- **Set tag mask**

  ```
  unsigned int tag_mask;
  mfc_write_tag_mask(tag_mask);
  ```

  – tag mask remains set until changed

- **Fetch tag status**

  ```
  unsigned int result;
  result = mfc_stat_tag_status();
  ```

  – tag status Is logically ANDed with current tag mask

  – tag status bit of ‘1’ indicates that no DMA requests tagged with the specific tag id (corresponding to the status bit location) are still either in progress or in the DMA queue

Waiting for DMA Completion (SPE)

- **Wait for any tagged DMA:**

  mfc_read_tag_status_any():

  – wait until **any** of the specified tagged DMA commands is completed

- **Wait for all tagged DMA:**

  mfc_read_tag_status_all():

  – wait until **all** of the specified tagged DMA commands are completed

➢ **Specified tagged DMA commands = commands specified by current tag mask setting**

DMA Transfers Example

DMA List Sample

```
#include <spu_mfcio.h>

struct dma_list_elem {
        unsigned int size;
        unsigned int ea_low;
};

struct dma_list_elem list[16] __attribute__
((aligned (8)));

void get_large_region(void *dst, unsigned
int ea_low, unsigned int nbytes)
{
unsigned int i = 0;
unsigned int tagid = 0;
unsigned int listsize;
if (!nbytes)
   return;
```

```
while (nbytes > 0) {
   unsigned int sz;
   sz = (nbytes < 16384) ? nbytes : 16384;
   list[i].size = sz;
   list[i].ea_low = ea_low;
   nbytes -= sz;
   ea_low += sz;
   i++;
}

listsize = i * sizeof(struct dma_list_elem);

spu_mfcdma32((volatile *)dst, (unsigned int)
&list[0], listsize, tagid, MFC_GETL_CMD);
}
```

This C-language sample program creates a DMA list and, in the last line, uses an spu_mfcdma32 intrinsic to issue a single DMA-list command (**getl**) to transfer a main-storage region into LS.

Double Buffering

Moving Double-Buffered Data

Consider an SPE program that requires large amounts of data from main storage. The following is a simple scheme to achieve that data transfer:

1. Start a DMA data transfer from main storage to buffer *B* in the LS.
2. Wait for the transfer to complete.
3. Use the data in buffer *B*.
4. Repeat.

- A lot of time is wasted waiting for DMA transfers to complete.
- We can speed up the process significantly by
  - allocating two buffers, B0 and B
  - overlapping computation on one buffer with data transfer in the other

"Double Buffering"

- Double buffering is a form of *multibuffering,* which is the method of using multiple buffers in a circular queue to overlap processing and data transfer.

Double Buffering

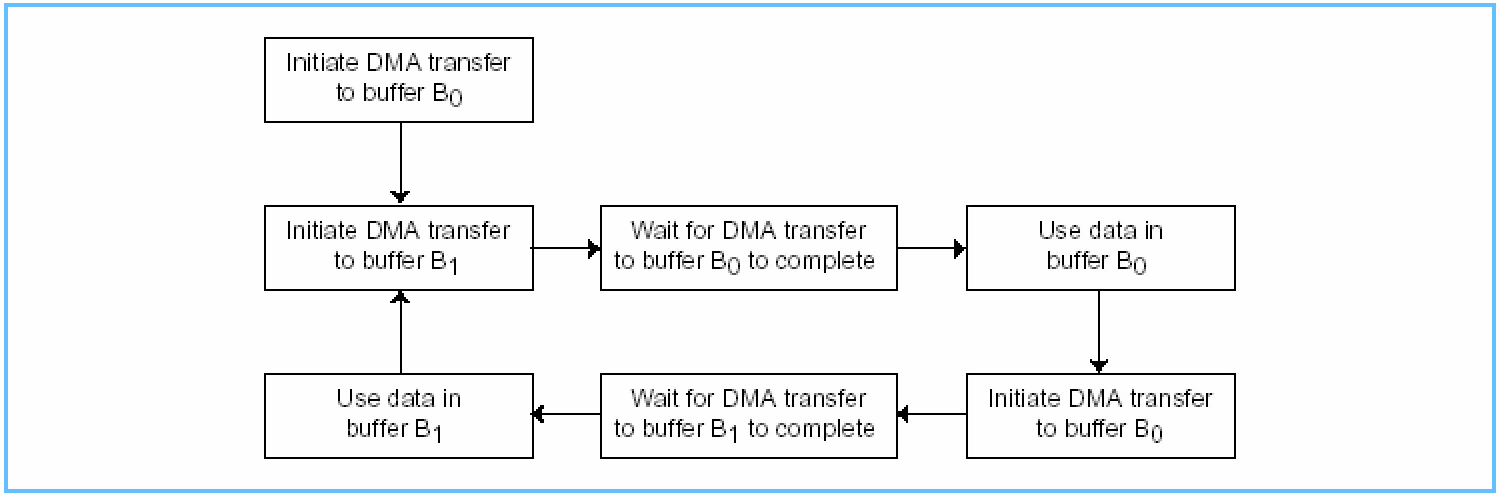
- The purpose of double buffering is to
  - maximize the time spent in the compute phase of a program
  - minimize the time spent waiting for DMA transfers to complete

- To use double buffering effectively, follow these rules for DMA transfers (SPE):
  - Use multiple LS buffers.
  - Use unique DMA tag IDs, one for each LS buffer.
  - Use *fenced* command options to order the DMA transfers within a tag group.
  - Use *barrier* command options to order DMA transfers within the MFC's DMA controller.

DMA Transfers Using a Double-Buffering Method

The double-buffering sequence is:

1. Initiate DMA transfer of incoming data from EA to LS buffer B0.
2. Initiate DMA transfer of incoming data from EA to LS buffer B1.
3. Wait for transfer of buffer B0 to complete.
4. Compute on data in buffer B0.
5. Initiate DMA transfer of incoming data from EA to LS buffer B0.
6. Wait for transfer of buffer B1 to complete.
7. Compute on data in buffer B1.
8. Repeat steps 2 through 7 as necessary.



How to Initiate a Buffer Transfer

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
volatile void *B[2]; /* Pointers to LS Buffers */
/* Initiate transfer using LS buffer B[idx] */
static inline void xfer(unsigned int ea, unsigned int size, unsigned
   int idx)
{
   spu_mfcdma32(B[idx], ea, size, idx, MFC_GET_CMD);
}
```

Index used as tag id

How to Wait for a Buffer Transfer to Complete

```
/* Wait for B[idx] transfer to complete. */
static inline void wait_xfer(unsigned int idx)
{
   unsigned int tag_mask = (1 << idx);
   spu_writech(MFC_WrTagMask, tag_mask);
   spu_mfcstat(MFC_TAG_UPDATE_ALL);
}
```

Example Illustrates Double Buffering

```
/* Example C code demonstrating double buffering using buffers B[0] and B[1].
 * In this example, an array of data starting at the effective address eahi|ealow is DMAed
 * into the SPU's local store in 4 KB chunks and processed by the use_data subroutine.
 */
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#define BUFFER_SIZE 4096
volatile unsigned char B[2][BUFFER_SIZE] __attribute__ ((aligned(128)));

void double_buffer_example(unsigned int ea, int buffers)
{
   int next_idx, idx = 0;
   // Initiate first DMA transfer
   spu_mfcdma32(B[idx], ea, BUFFER_SIZE, idx, MFC_GET_CMD);
   ea += BUFFER_SIZE;
   while (--buffers) {
      next_idx = idx ^ 1;  // toggle buffer index
      spu_mfcdma32(B[next_idx], ea, BUFFER_SIZE, idx, MFC_GET_CMD);
      ea += BUFFER_SIZE;
      spu_writech(MFC_WrTagMask, 1 << idx);
      (void)spu_mfcstat(MFC_TAG_UPDATE_ALL); // Wait for previous transfer done
      use_data(B[idx]); // Use the previous data
      idx = next_idx;
   }
   spu_writech(MFC_WrTagMask, 1 << idx);
   (void)spu_mfcstat(MFC_TAG_UPDATE_ALL); // Wait for last transfer done
   use_data(B[idx]); // Use the last data
}
```

Multibuffering

Multibuffered data transfers on the SPU

1. Allocate multiple LS buffers, B0..Bn.
2. Initiate transfers for buffers B0..Bn. For each buffer Bi, apply tag group identifier i to transfers involving that buffer.
3. Beginning with B0 and moving through each of the buffers in round robin fashion:
   - Set tag group mask to include only tag i, and request conditional tag status update.
   - Compute on Bi.
   - Initiate the next transfer on Bi.

This algorithm waits for and processes each Bi in round-robin order, regardless of when the transfers complete with respect to one another. In this regard, the algorithm uses a **strongly ordered** transfer model. Strongly ordered transfers are useful when the data must be processed in a known order.

IBM Microelectronics Division
1580 Route 52, Bldg. 504
Hopewell Junction, NY 12533-6351

The IBM home page is http://www.ibm.com
The IBM Microelectronics Division home page is
http://www.chips.ibm.com