PPC Core initiated DMAs

1 Introduction

BE (Broadband Engine) chip is an implementation of BPA (Broadband Processor Architecture) consisting of 1 PowerPC core with two threads (thread 0 and thread 1) and 8 SPUs (Synergistic Processing Unit) -- for a total of 10 processors -- as well as MFC (memory flow controller) interacting with the ten processors and external components like system memory and IO processors. One of the primary means by which PPC Cores and SPUs interact is using DMA transfers which can be initiated by either and managed by the MFC. This white paper discusses how to setup, initiate, monitor, and properly complete DMA transactions, initiated by PowerPC code running on a PPC core, between system memory and SPU local store using the MFC. Section 3 of this white paper describes (at a high level) the BE Implementation of PPC Core initiated DMAs and provides pointers to more detailed information. Section 4 of this white paper references and describes sample software implementations (i.e. the SDK associated with this white paper) of PPC Core initiated DMA function.

2 Background and Assumptions

This white paper assumes the reader has at least a very basic understanding of the BPA architecture as can be obtained by reading section *"1. Introduction to Broadband Processor Architecture"* in the *Broadband Processor Architecture - Books I-III* document.

3 BE Implementation

DMA transfers are performed between system memory and SPU local stores using MFC DMA command requests. The MFC maintains and processes an 8 entry queue for DMA command requests initiated by a PPC Core for each SPU in a system. You could potentially have 64 MFC DMA command requests enqueued simultaneously (8 requests per each of 8 SPUs) in a single BE system. The PPC Core controls an MFC using memory-mapped registers. There is a distinct memory-mapped register region for each SPU in a system. Since these registers are referenced using a translated effective address, it is possible for any PPC Core in the system to control any MFC, and to place MFC DMA command requests on any of its SPU queues.

3.1 Enqueue DMA commands

DMA command requests are made by first writing parameters to the DMA Command Parameter Registers. These parameters describe the transfer and must be submitted in a particular sequence (i.e. the order listed in the following table) in order to initiate a DMA. These registers are described in the following table.

Table 3-1. DMA Command Parameter Registers

Register	Description
DMA_LSA	DMA Local-Storage Address
DMA_EAH	DMA Effective Address (High 32bits)
DMA_EAL	DMA Effective Address (Low 32bits)
DMA_Size, DMA_Tag	DMA Transfer Size & Command Tag
DMA_ClassID, DMA_CMD & DMA_CMDStatus	DMA Class ID & Command Opcode for write & DMA Command-Status for read

Each of these 5 32-bit registers can be written using 32-bit stores or you can write 1 32-bit store (containing the DMA_LSA), followed by 1 64-bit store (containing both the DMA_EAH and DMA_EAL), followed by another 64-bit store (containing the DMA_Size, DMA_Tag, DMA_ClassID, and DMA_CMD). Writing the DMA_EAH register is optional and it will be set to zero if not written.

After the DMA Command Parameter Registers are written, a read of the DMA Command-Status Register (DMA_CMDStatus) causes a DMA command to be enqueued. You must check the value returned (least significant 2 bits) to insure that the enqueue was successful. If bit 31 was set, then the enqueue failed due to a sequencing error. If bit 30 was set, the enqueue failed due to insufficient space in the command queue. Note that DMA_ClassID/DMA_CMD share the same register address as DMA_CMDStatus.

For more detailed information about various topics in this section, see the noted sections of the *Broadband Processor Architecture - Books I-III* document in the following table.

Table 3-2. DMA command/parameter/enqueue info

Topic	Section in BPA Books I-III
DMA command opcodes	*"5.1 DMA Commands"*
DMA command parameters	*"9.1 DMA Command Parameter Registers"*
DMA command enqueue sequence	*"10.2 PPC Core DMA-Command Issue Sequence"*

3.2 Control/Monitor DMA commands

In addition to the DMA Command-Status Register, which causes a command to be enqueued, there are additional Command-Queue Control Registers for determining the number of command slots available in the PPC Core DMA queue as well as

registers for determining when a tag group, in the PPC Core DMA queue, is complete. These registers are listed in the table which follows.

Table 3-1. DMA Command-Queue Control Registers

Register	Description
DMA_CMDStatus	DMA Command-Status for read
DMA_QStatus	DMA-Queue Status
Prxy_QueryMask	Proxy Tag-Group Query-Mask Register
Prxy_TagStatus	Proxy Tag-Group Status Register
Prxy_QueryType	Prxy_QueryType

To determine if there are any commands in the PPC Core DMA queue or more importantly how much space is available in this queue, you read the DMA_QStatus register. If the top bit (i.e. bit 0) is set, then the queue contains no commands (i.e. is empty). The number of available slots in the queue can be determined from the lower 16-bits of this register (i.e. bits 16-31). A value of zero in this field indicates that the queue is full. Software can use this field to set a loop count for the number of DMA commands to enqueue.

Each DMA command is tagged with a 5-bit identifier, which can be used for multiple DMA commands. A set of commands with the same identifier is defined as a tag group. Software can use the identifier to determine when a command of group of commands have completed. In addition, an interrupt may be presented to a processor or device upon completion of one or more tag groups, if enabled by privileged software.

To perform either a poll for, or setup for an interrupt on, completion of one or more tag groups, you need to set the Prxy_QueryMask register to select the DMA tag groups you are interested in. The valid tag group identifiers range from 0 to 31. Setting the most significant bit of this register (i.e. bit 0) indicates you are interested in tag group 31. Likewise, setting the least significant bit of this register (i.e. bit 31) indicates you are interested in tag group 0.

There are two methods which can be used to poll for the completion of one or more tag groups.

The first involves reading the Prxy_TagStatus register. You must first enqueue a DMA command, described earlier, and set the Prxy_QueryMask register appropriately. If you wish to wait for only one tag group or any tag group in multiple groups to complete, you can simply poll reading the Prxy_TagStatus register until it is non-zero. The value set when non-zero will correspond to the tag identifier(s) of the specific tag group(s) that have completed. If you wish to wait for all tag groups selected in the Prxy_QueryMask, you must set up a variable to accumulate the values read from the Prxy_TagStatus register. Then, after each poll read of this register you must compare this accumulated value to the value previously written to the Prxy_QueryMask register. When the values match (e.g. XOR of values is zero), then all the tag groups are complete.

The second method for polling involves using the Prxy_QueryType register. Again, you must first enqueue a DMA command, described earlier, and set the Prxy_QueryMask register appropriately. Then, you write a specific condition to the Prxy_QueryType register. Finally, you poll reading this same register until it returns a value of zero, which indicates the condition has been meet. If you wish to wait for only one tag group or any tag group in multiple groups to complete, you could specify the condition of "upon completion of ***any*** enabled tag groups." This requires writing a binary value of "01" to the least significant bits of the Prxy_QueryType register. If you wish to wait for all tag groups selected in the Prxy_QueryMask, you should specify the condition of "only upon completion of ***all*** enabled tag groups." This requires writing a binary value of "10" to the least significant bits of the Prxy_QueryType register.

If you wish to generate an interrupt upon completion of tag groups, you must first insure that the Tag_Group_Completion interrupt is enabled by privileged software. Then, like the polling methods, you must first enqueue a DMA command, described earlier, and set the Prxy_QueryMask register appropriately. Next, you simply write the specific condition you desire to the Prxy_QueryType register and wait for the interrupt to occur. The condition requested must be either "any" or "all", as described earlier. You do not need to read from the Prxy_QueryType register to determine status since an interrupt will occur when the condition is met. However, you need to insure that the appropriate signal handling or SPU event handling software is present in your application.

When using either polling or interrupts to determine DMA command completion one must insure that the tag groups specified in the Prxy_QueryMask register actually match the tag groups which were used when the DMA commands were submitted.

For more detailed information about various topics in this section, see the noted sections of the *Broadband Processor Architecture - Books I-III* document in the following table.

Table 3-2. DMA command-queue control info

Topic	Section in BPA Books I-III
DMA command-queue control registers	*"9.2 DMA Command-Queue Control Registers"*

4 Sample SW Implementation (SDK)

A sample software implementation of this topic has been developed/tested and is available in the SDK associated with this white paper. The table which follows indicates the various

files/routines containing the software constructs used to implement the functions described in this section.

Table 4-1. SDK files for PPC Core initiated DMAs

Topic	(file)/routine
Various defines, macros, and structure definitions for the BPA memory mapped registers	(include/bpa_map.h) / typedef struct SPU_Problem
Various defines and macros used for describing & initiating DMA commands	(src/include/bpa_mfc.h)
Error checking macros, convenience macros, and routines to perform DMA command enqueue and control	(src/include/pu/mfc.h) / MFC DMA routines _mfc_cmd, _mfc_cmd_int, _mfc_cmd_large, _mfc_cmd_large_int, _get_mfc_freespace, _wait_mfc_freespace, _set_mfc_tagmask, _stat_mfc_tagmask, _wait_mfc_tags_all, _int_mfc_tags_all, _wait_mfc_tags_any, _int_mfc_tags_all These routines are also described in more detail in the ***MFC Library*** section of the ***STI Design Center Cell Processor Libraries Strategy, Plan, and Users Guide*** document (sdk/docs/lib/libraries.pdf)
Source code containing example usage of the various MFC DMA routines	(src/tests/mfc/pu/dma_pu.c, src/tests/mfc/pu/dma_pu.h)

5 Conclusions and Future Work

This white paper has attempted to describe, at a high level, the software interfaces and algorithms required to perform PPC Core initiated DMAs on the BE Implementation of the BPA.

6 Glossary

BE - Broadband Engine (an implementation of BPA)
BPA - Broadband Processor Architecture
MFC - Memory Flow Controller
PPC - PowerPC Core. For BE, there two hardware threads
SDK - Software Development Toolkit
SPU - Synergistic Processing Unit
SPC - Synergistic Processing Core is a combination of SPU and MFC
SW - Software

7 References

1. PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-Bit Microprocessors Version 2.0, June 10, 2003 (includes 970 -- that is, Apple G5 -- information). Found on the web at:
   http://www-3.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_970_Microprocessor
2. PowerPC User Instruction Set Architecture Books I,II,III, Version 2.01, December 2003. Found on the web at:
   http://www-106.ibm.com/developerworks/eserver/articles/archguide.html
3. Broadband Processor Architecture - Books I,II,III
4. STIDC Software Development Toolkit source code for BE