SPE User's Manual

Generated for organization: gsc-game

Table of Contents

1. Preface

This document contains implementation-specific information about the Synergistic Processor Element. The Synergistic Processor Element (SPE) consists of the Synergistic Processor Unit (SPU) core and the Synergistic Memory Flow Controller (MFC) core.

1.1. Reference Materials

This document is intended for use in conjunction with the other supporting documents listed below.

- CBE Overview
- PPE User's Manual
- Cell Broadband Engine™ Architecture
- Synergistic Processor Unit Instruction Set Architecture
- Cell Broadband Engine™ Registers

2. Introduction

A Synergistic Processor Element (SPE) is a processor optimized for running compute-intensive applications. Typically, an SPE is a component of a system compliant with the *Cell Broadband Engine™ Architecture*; such a system is referred to as a Cell Broadband Engine™ (CBE) in the rest of this document. A system with a significant number of SPEs managed by a PowerPC Processor Element (PPE) allows for cost-effective and power-efficient processing over a wide range of applications. The SPE includes two types of functional components: the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC).

Each SPU is an independent compute-optimized processor. The SPU processor executes on its local storage (LS) only. The local storage is used to store both code and data. With respect to accesses by its SPU, the local storage is unprotected and untranslated storage.

MFC components are principally data-transfer engines that move data between system storage and the local storage of the SPE. Attributes of system storage (address translation and protection) in the CBE are governed by the page and segment tables of the *PowerPC Architecture*. Direct memory access (DMA) transactions in the CBE are coherent with respect to system storage. To accomplish the data transfers, the MFC maintains and processes queues of DMA commands. The MFC can also autonomously fetch and execute a sequence of DMA commands from the local storage in response to a DMA list command.

In addition, the MFC implements some bus bandwidth-reservation and data-synchronization features. Logically, a single MFC command (CMD) queue is always associated with each SPU in a system. The architecture can share a single physical MFC between multiple SPUs. In such a case, multipath DMA capability is needed, as well as the ability to maintain and process multiple MFC CMD queues.

The SPU interacts with the MFC through SPU channels. Channel commands are used to enqueue MFC commands, query MFC tag-group status, interrogate processor status, perform MFC synchronization operations, and access auxiliary resources such as timers.

Local storage addresses and many of the MFC resources have aliases in system storage that enable the PPE or other SPUs in the system to access these resources and control the SPU.

2.1. Synergistic Processor Unit

The Synergistic Processor Unit (SPU) is an area and power efficient microprocessor that falls between general-purpose processors and special-purpose hardware. General purpose processors aim to achieve the best average performance on a broad set of applications. Special-purpose hardware aims to achieve the best performance on a single application. The SPU Instruction Set Architecture (ISA), however, aims to deliver maximum performance on a broad set of compute-intensive applications.

The SPU is a single-instruction, multiple-data (SIMD) reduced instruction set computer (RISC). Most operations are performed on a unified (128-entry by 128-bit) register file. For most operations, the 128b registers are interpreted as multiple data elements with an aggregate size of 128-bit (that is, as four 32-bit elements, eight 16-bit elements, and so on). Load and store instructions transfer data between the local storage and the register file.

The SPU delivers performance by issuing up to two instructions per cycle. Each of these instructions operates in a SIMD fashion on four 32-bit words in parallel. The SPU channel interface offers a high-bandwidth interface to a DMA engine that transfers data to and from the local storage. Instruction throughput is maximized by the large number of registers and by an instruction set that eliminates or helps predict branches. The SPU simplified architecture eliminates privileged states, address translation, and hardware-managed caches.

2.2. SPU Channels

The SPU channels are the primary interface between the SPU and the MFC. SPU channel instructions are architected to be 128 bits wide, but in this implementation, channels instructions only set and use the 32 bits from the preferred slot. The preferred slot is the left-most word (bytes 0, 1, 2, and 3) of a 128-bit register. Each channel has a corresponding count that indicates the remaining capacity in that channel. The channel capacity (that is, the maximum number of outstanding transfers) is implementation specific.

SPU channels are either read only or write only. Each channel is either blocking or nonblocking. Blocking channels cause the SPU to stall when reading or writing a channel with a count of zero.

Channel instructions defined in the SPU architecture are used by an SPU program to transfer information on the channel interface. (See the *Synergistic Processor Unit Instruction Set Architecture* document for more information.) In the *Cell Broadband Engine™ Architecture*, the SPU uses channels as the primary interface between the MFC facilities and other devices and processors. In some cases, a register is defined in the MFC for other processors or devices to transfer information on the channel interface via memory-mapped input/output (MMIO).

2.3. Memory Flow Controller (MFC)

The MFC provides two main functions for the SPU:

- The MFC moves data between the SPU local storage and the main storage.
- The MFC provides synchronization mechanisms between the SPU and the rest of the processing units in the system.

SPUs operate only on the local storage. Code and data must be transferred into the local storage for an SPU to execute it or operate on it. Local storage addresses are aliased in the CBE memory map; transfers to and from the local storage to memory at large (including other local storages) are coherent in the system. A pointer to a data structure created on the PPE can be passed to an SPU. The SPU can use this pointer to issue a DMA command to bring the data structure into its local storage to perform operations on it. If, after operating on this data structure, the SPU (or PPE) issues a DMA command to place it back in non-local storage memory, the transfer is again coherent in the system according to the normal PPE ordering rules. Equivalents of the PPE locking instructions, as well as a memory-mapped mailbox for each SPU, are used for synchronization and mutual exclusion.

2.4. MMIO Registers

SPE MMIO registers initialize and control of the SPU. The SPU accesses facilities outside of the SPU through channel commands. The other processors in the system or on chip access the SPU through SPE MMIO registers.

3. SPE Architecture Details

3.1. Synergistic Processor Unit (SPU)

Each Synergistic Processor Unit (SPU) is an independent processor with its own program counter, optimized to run under the control of a PPE in a CBE system. The SPU delivers performance by executing up to two instructions per cycle. Each of these instructions operates in a single-instruction, multiple-data (SIMD) fashion on four 32-bit words in parallel. The SPU offers a high-bandwidth interface to a direct memory access (DMA) engine that transfers data to and from the local storage. Instruction throughput is maximized by the large number of registers and by an instruction set that eliminates or helps predict branches. The SPU simplified architecture eliminates privileged states, address translation, and hardware-managed caches.

The SPUs implement a new instruction set architecture (ISA) specific to the *Cell Broadband Engine™ Architecture*. It is described in a separate document, the *Synergistic Processor Unit Instruction Set Architecture*.

The SPU architecture has the following main characteristics:

- Provides a load-and-store architecture with sequential semantics, using a large set of registers
- Provides a load-and-store access to an associated local storage.
- Provides channel input/output for Memory Flow Controller (MFC), used for external data access.

The SPU architecture has the following restrictions:

- There is no direct access to main storage. The SPU accesses main storage only by using the MFC.
- There is no distinction between privileged state and problem state.
- There is no access to critical system control such as page-table entries. This restriction is enforced by PPE privileged software.
- The PPE allows no synchronization facilities for shared local-storage access.

Figure 3-1 on page 10 is a block diagram of the SPU. The SPU has six functional units — hardware macros that contain the logic for several functions — and a transport unit.

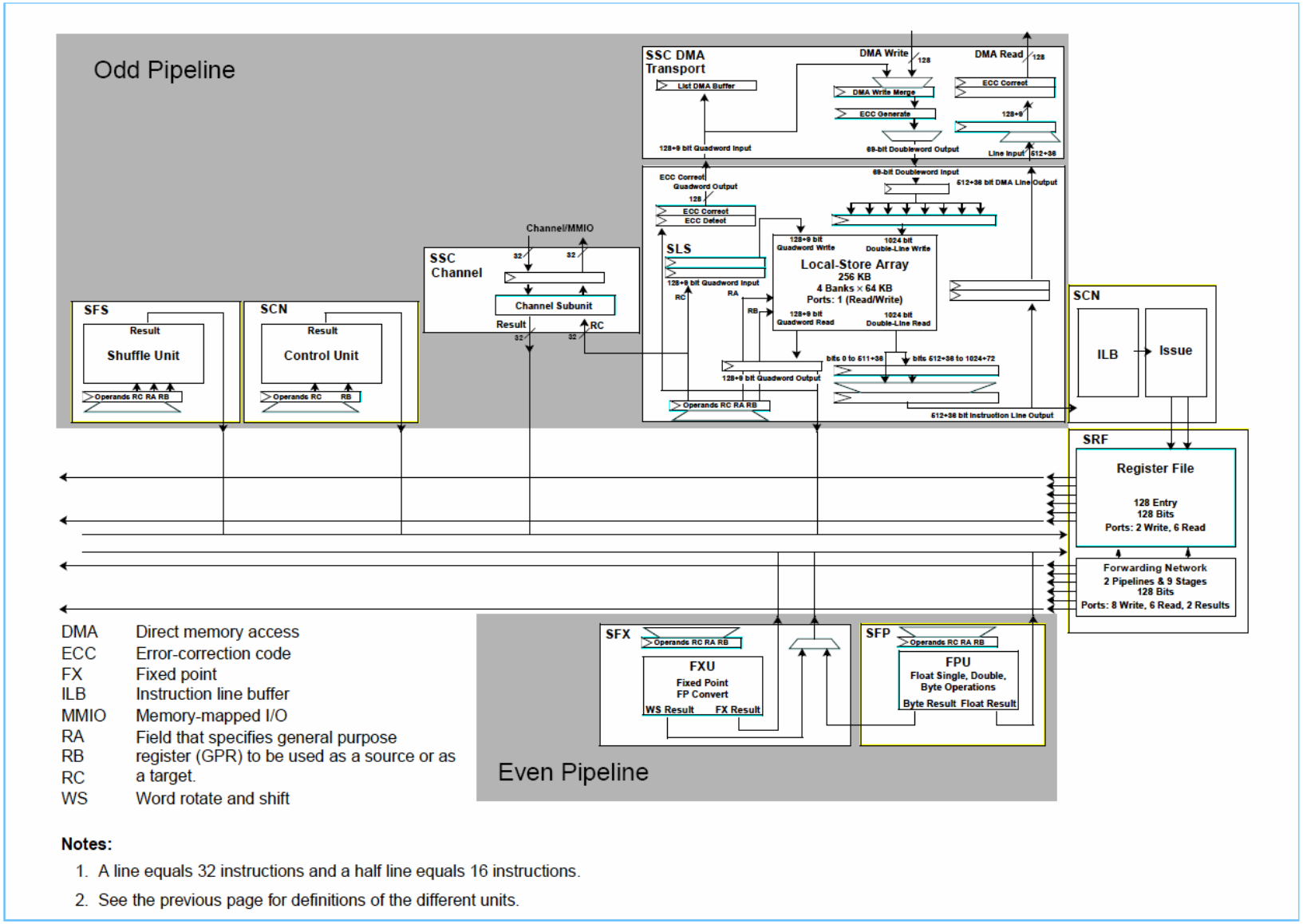
- SPU floating point unit (SFP) — Calculates single-precision, double-precision, and integer multiplies.
- SPU even fixed-point unit (SFX) — Performs arithmetic, logical operations, and word shifts.
- SPU odd fixed-point unit (SFS) — Performs shuffles and quadword rotations.
- SPU register file unit (SRF) — Responsible for storing and delivering register operands to functional units.
- SPU load/store unit (SLS) — Contains the local memory or local store (single-ported, 256-KB memory); executes load and store instructions. Supplies instructions to the SPU control unit.
- SPU control unit (SCN) — Responsible for branch execution and instruction sequencing.
- SPU channel/DMA transport (SSC) — Responsible for all input and output functions (There are two SSCs: SSC Channel and SSC DMA transport.)

Figure 3-1 on page 10 illustrates the SPU instruction flow, which begins with an instruction read request initiated by the SCN unit.
First, the SCN requests instruction reads from the SLS. Next, the SLS sends 32 instructions in two cycles back to the SCN's instruction line buffer (ILB). Then, the ILB sends the instructions, two at a time, to the issue logic. When the operands are ready, the issue logic sends the instructions to the read ports of the register file and forwarding network in program order. Later, the issue logic sends the instructions to the functional units for execution.
Functional unit pipelines vary in length from two to seven cycles. Actual instruction latencies are determined by instruction class. When the results are ready, they are sent to the forwarding network, where they are held until they are written back to the register file. Results are written to the register file in program order.

SFX, SFP, and SLS can send results directly to operands of subsequent instructions without incurring additional latency penalties through the forwarding unit.

(Note)
The term "local storage" is the architectural name, as specified by the *Cell Broadband Engine™ Architecture*.
The term "local store" is the name used for an implementation of an SPE's local storage.
However, the discrimination of the two terms may not be so strict in some context.

Figure 3-1. High-Level Block Diagram of SPU



Notes:

1. A line equals 32 instructions and a half line equals 16 instructions.
2. See the previous page for definitions of the different units.

3.1.1. SPU Pipeline

The SPU uses pipeline techniques to execute instructions at high speeds. Several instruction operations can be processed simultaneously by different pipeline stages, increasing instruction throughput. The SPU can issue and complete up to two instructions per cycle. One instruction can be issued on each of the two execution pipelines, the even pipeline or the odd pipeline. Whether an instruction goes to the odd or even pipeline depends on its instruction class. The instruction class is related to the execution unit that performs the function. Each execution unit is assigned to one of the two pipelines. The SFP and the SFX units are on the even pipeline while the rest of the functional units are on the odd pipeline. *Table 3-1* lists the instruction classes with their latencies, pipeline assignment, and the functional unit responsible for their execution.

Table 3-1. SPU Instruction Classes

Class	Description	Latency	Pipeline	Unit
LS	Loads and stores	6	odd	SLS
HB	Branch hints		odd	SLS
BR	Branch resolution Inline or correctly hinted branches have zero cycle delay. The mispredicted branch penalty is 17 or 18 clock cycles, based on the program counter.	4	odd	SCN
CH	Channel interface, special purpose registers	6	odd	SSC
SP	FPU single precision	6	even	SFP
DP	FPU double precision	13	even	SFP
FI	Floating-point integer	7	even	SFP
SH	Shuffle	4	odd	SFS
FX	Simple fixed point	2	even	SFX
WS	Word rotate and shift	4	even	SFX
BO	Byte operations	4	even	SFP
NOP	No operation (execute)		even	
LNOP	No operation (load)		odd	

The SPU issues all instructions in program order according to the pipeline assignment. Each instruction is part of a doubleword-aligned instruction pair called a fetch group. A fetch group can have one or two valid instructions, but they are aligned to doubleword boundaries. This means that the first instruction in the fetch group is from an even word address, and the second instruction from an odd word address. The SPU processes fetch groups one at a time, continuing to the next fetch group when the current instruction group becomes empty. An instruction becomes issueable when it is valid, its register dependencies are satisfied, and there is no structural hazard (resource conflict) with prior instructions, DMA activity, or Error-Correcting Code (ECC) activity.

Dual issue occurs when a fetch group has two issueable instructions where the first instruction of an issue group is executed by an execution unit on the even pipeline, and the second instruction is executed by a unit on the odd pipeline. There are no instruction alignment requirements. Instructions processed by units on the even pipeline can exist in odd locations, just as odd pipeline instructions can exist at even addresses. However, such instruction placement inhibits dual issue. If a fetch group cannot be dual-issued but the first instruction can be issued, then the first instruction is issued to the proper execution pipeline and the second instruction is held until it can be issued. A new fetch group is loaded after both instructions of the current fetch group are issued.

Table 3-2 shows how an SPU can issue a short program in the absence of resource conflicts and register dependencies. The program features four cases of pipeline assignment versus instruction alignment. The eight instructions are issued in seven cycles. Dual issue occurs only for the first instruction group. The first fetch group consists of an instruction executed by the even pipeline in address x'0' and an instruction executed by the odd pipeline at address x'4'.

Table 3-2. Instruction Issue in the Four Cases

Address	Fetch Group in Memory		Issue schedule		Core Clock (NClk) Cycles
	Even Address	Odd Address	Even Pipeline	Odd Pipeline	
0x0000	1:Even	2:Odd	1:Even	2:Odd	1
0x0008	3:Even	4:Even	3:Even	Penalty	2
0x0010	5:Odd	6:Odd	4:Even		3
0x0018	7:Odd	8:Even		5:Odd	4
				6:Odd	5
				7:Odd	6
			8:Even		7

dual issue
(no register dependency)

Instruction alignment can be optimized to minimize execution time through the insertion of NOP and LNOP instructions. For example, *Table 3-3* shows how an SPU can schedule another short program. These eight instructions issue in six cycles. This same program can be issued in five cycles when the instruction alignment is improved.

Table 3-3. Example of Instruction Issue without NOP and LNOP

Address	Fetch Group in Memory		Instruction Issue		Core Clock (NClk) Cycles
	Even Address	Odd Address	Even Pipe	Odd Pipe	
0x0000	1: Even	2: Odd	1: Even	2: Odd	1
0x0008	3: Odd	4: Even	—	3: Odd	2
0x0010	5: Odd	6: Even	4: Even	—	3
0x0018	7: Even	8: Odd	—	5: Odd	4
0x0020			6: Even	—	5
			7: Even	8: Odd	6

Table 3-4 shows that a NOP added between instruction 2 and instruction 3 maintains dual issue through the third instruction group, and an added LNOP continues dual issued for the remainder of the program.

Table 3-4. Example of Instruction Issue Using NOP and LNOP

Address	Fetch Group in Memory		Instruction Issue		Core Clock (NClk) Cycles
	Even Address	Odd Address	Even Pipe	Odd Pipe	
0x0000	1: Even	2: Odd	1: Even	2: Odd	1
0x0008	NOP	3: Odd	NOP	3: Odd	2
0x0010	4: Even	5: Odd	4: Even	5: Odd	3
0x0018	6: Even	LNOP	6: Even	LNOP	4
0x0020	7: Even	8: Odd	7: Even	8: Odd	5
			—	—	6

Table 3-5 shows how the various operations on the even and odd execution pipelines are related. Each pipeline stage has a unique alphabetical identifier so that a later stage has a letter that occurs later in the alphabet. Execution begins in stage M, where inputs to the execution units are launched and latched. In the first cycle, no unit produces a result; FX class instruction results do become available for use by subsequent instructions in the second cycle. When a result becomes available in a particular cycle, it is available for use by any subsequent instruction executed by any unit. Although shorter latency instructions make their results available for forwarding earlier, their results are staged out to latency 8(stage T) before the results are sent to the register file. The longest latency of a pipelined result is seven cycles, which allows seven cycles of result equalization to commit register-file results in program order. Double precision (DP) class results are not fully pipelined.

Table 3-5. Execution Pipelines and Result Latency

Pipeline	Pipeline Stage Name/Latency Cycles									
	M/1	N/2	O/3	P/4	Q/5	R/6	S/7	T	U	V
Even		FX		BO, WS		SP	FI	Pre-writeback	Register file write	Read bypass
Odd				BR, SH		LS, CH		Pre-writeback	Register file write	Read bypass

Register-file reads and writes are both 2-cycle operations. Result data is sent to the register file while the write addresses are decoded during the pre-writeback stage. In the next cycle, the register file is written. One final stage of forwarding, read bypass, is necessary before the written result is available from the register file.

Table 3-6 SPU Pipeline on page 15 provides an overview of SPU pipeline operations. Some of these operations are described in more detail in later sections. Early stages of instruction fetch are named with a two letter code. The first letter is the letter of the corresponding stage from a load operation; the second letter, "f," denotes its presence in the fetch activity. Operations are often described as "XX: yyy" where XX is a unit or subunit identifier and the yyy is a brief description of the action.

The SPU pipeline begins with the Kf cycle. In this cycle, the SPU starts a new stream of execution with a mispredict signal asserted in branch resolution. The mispredict signal is sent to the instruction-fetch (IF) unit to request an instruction fetch from the program counter.
IF control logic arbitrates for the local store and selects an address source, according to the next local-store transaction in stage Lf. If the fetch request is top priority, the pipeline continues with a local-store double half-line read transaction from stages Mf to Qf.
The line data is transferred from the read data buffer (RDB) to the instruction line buffer (ILB) in two cycles, starting in stage Rf. The even line drives first, followed in the next cycle by the odd line.

The ILB, shown in *Figure 3-2 Instruction-Fetch Dataflow Diagram* on page 17, holds the instructions until they are sent to issue control (IC).
In stage Sf, the line holding the mispredict data is read from one of the ILB entries. The instruction data is written to the ILB in two consecutive cycles with data from the even half line arriving first. If the mispredict data is in the odd half line of the double half-line fetch, the pipeline pauses for a cycle, waiting for the second write to occur.
When the half-line data is read from ILB storage, it is held in the predicted-path buffer while it is sequenced to issue control one instruction-pair at a time. Sixteen instructions are held in the predicted-path buffer and are accessed by the doubleword multiplexer (MUX) during the G stage.
The H stage is filled with simple decoding and instruction distribution.
The primary work of issue control begins in the I stage and is complete by the M stage. When all the required input operands are ready, issue control directs the register file (RF) and forwarding networks (FWD) to deliver them to the operand latches of the execution units. Later, issue control sends instructions to the execution units.

Register dependencies between the new instructions and instructions already in-flight are detected in stage I. Issue control uses the dependency information, resource conflict information, and instruction decode information to determine which instructions in stage J, if any, the fetch group can issue, and whether the issue

pipeline advances. If the fetch group is not dual issued, the SPU stalls until both instructions have issued.
In stage JJ, register file addresses are decoded, the instruction is sent to the proper units, and the forward priority encoders analyze the dependency matrix to determine the source of all input operands.

The register file is read in stage K, and the forward priority decoders send control to the forwarding network so that, in stage L, the forward MUX and operand latch MUXes produce the correct input operands for each units.

In stage M, the first execution cycle, the units begin processing their operands. Some results (load/store [LS], fixed-point [FX], single precision [SP], floating-point integer [FI], double-precision [DP] classes) are sent directly from the unit result to the operand latches when they are ready. All others are sent first to the forwarding network and are then distributed to the units at the cost of an extra cycle of latency.
Results are delayed by the forwarding network until stage V, when the values can be read from the register file. The commitment point is in stage N, where an issued packet is either marked as flushed or its calculation of the next instruction address to be committed is accepted as the next program counter. Just before the commitment point is the wait point, where the channel unit can request that the instruction be flushed and retried.

Table 3-6. SPU Pipeline

Stage	Activities That Take Place in Parallel				
Kf	BR: Drive mispredict				
Lf	IF: Control, local arbitration, load-store address selection				
Mf	SLS: Generate address				
Nf	SLS: Transfer address				
Of	SLS: Decode				
Pf	SLS: Access array				
Qf	SLS: Transfer (read latch)				
Rf	IB: Transfer from read data buffer	IB: Trigger compare			
Sf	IB: MUX ILB	IB: Line trigger, doubleword advance			
G	IB: Doubleword MUX (predicted-path buffer)				
H	IC: Decode source				
I	IC: Compare source and target				
J	IC: Issue				
JJ	RF: Decode address	IC: Forwarding network priority			
K	RF: Read	IC: Forwarding network decode			SSC: Decode
L	FWD: MUX	SFX: Decode	SFP: Decode		SSC: Arbitration
M	SLS: Generate address	SFX: Add (operand latch)	SFP: DP/FI formatter; SP first exponent stage	BR: Target address/ branch condition generator	SSC: Drive wait
N	SLS: Transfer address	SFX: FX result	SFP: DP/FI first exponent stage	BR: Mispredict; program counter MUX	
O	SLS: Decode	SFS: SH result to forwarding network	SFP: BO result to forwarding network	BR: Drive mispredict	
P	SLS: Access array	FWD: SH result	FWD: BO result		
Q	SLS: Bank MUX (read latch)				
R	SLS: Load/store result (bank mux)		SFP: SP result		
S			SFP: FI result		
T		FWD: Predict write back			
U	RF: Write	FWD: Register bypass writeback			
V		FWD: Register bypass read			

BR	Branch Resolution	IC	Issue Control	SSC	SPU Channel and DMA Transport
FWD	Forwarding Network	IF	Instruction Fetch	SFP	SPU Floating-Point Unit
IB	Instruction Buffer	ILB	Instruction Line Buffer	SLS	SPU Local-Store Unit
		RF	Register File	SFX	SPU Even Pipeline Fixed-Point Unit

3.1.2. Control Unit

The control unit is responsible for local-store arbitration, instruction fetch, instruction issue, register-file control, forward control generation, instruction completion, and the execution of branch instructions. This section describes these activities beginning at instruction fetch.

3.1.2.1. Instruction Fetch

An instruction fetch gets data from the local store. There are three types of instruction fetches: flush-initiated fetches, inline prefetches, and hint fetches.
Flush-initiated fetches occur after branch mispredicts and other pipeline flush events.
Inline prefetches are automatically generated in an attempt to keep the instruction line buffer full as execution continues in linear fashion through an instruction sequence.
Hint fetches are initiated by the execution of hint instructions to get branch target instructions into the instruction line buffer in advance of a predicted taken branch.

Because the local store is single ported, and load and store instruction frequency drives local-store occupancy to high levels, it is important that DMA and instruction-fetch activity transfer as much useful data as possible in each local-store request. Instruction fetch loads 32 instructions per local-store request by accessing all banks of the local store simultaneously.

The four banks of the local store are interleaved on the low-order bits of the address. To increase the average utility of the text returned by the local store, these four banks are grouped into two half-line wide groups, and the fetch address is aligned to a half-line address. The bank pair that is not used for the actual fetch is used to inline prefetch the next half line. For example, if the fetch address is instruction address 5, the first bank pair fetches instructions 5 to 15, while the second bank pair returns instructions 16 to 31. If the fetch requests instruction 18, the second bank pair returns instructions 18 to 31, while the first bank pair returns instructions 32 to 47.

Note: A line equals 32 instructions, and a half-line equals 16 instructions. In *Figure 3-2* on page 17, the data flow width is a half line.

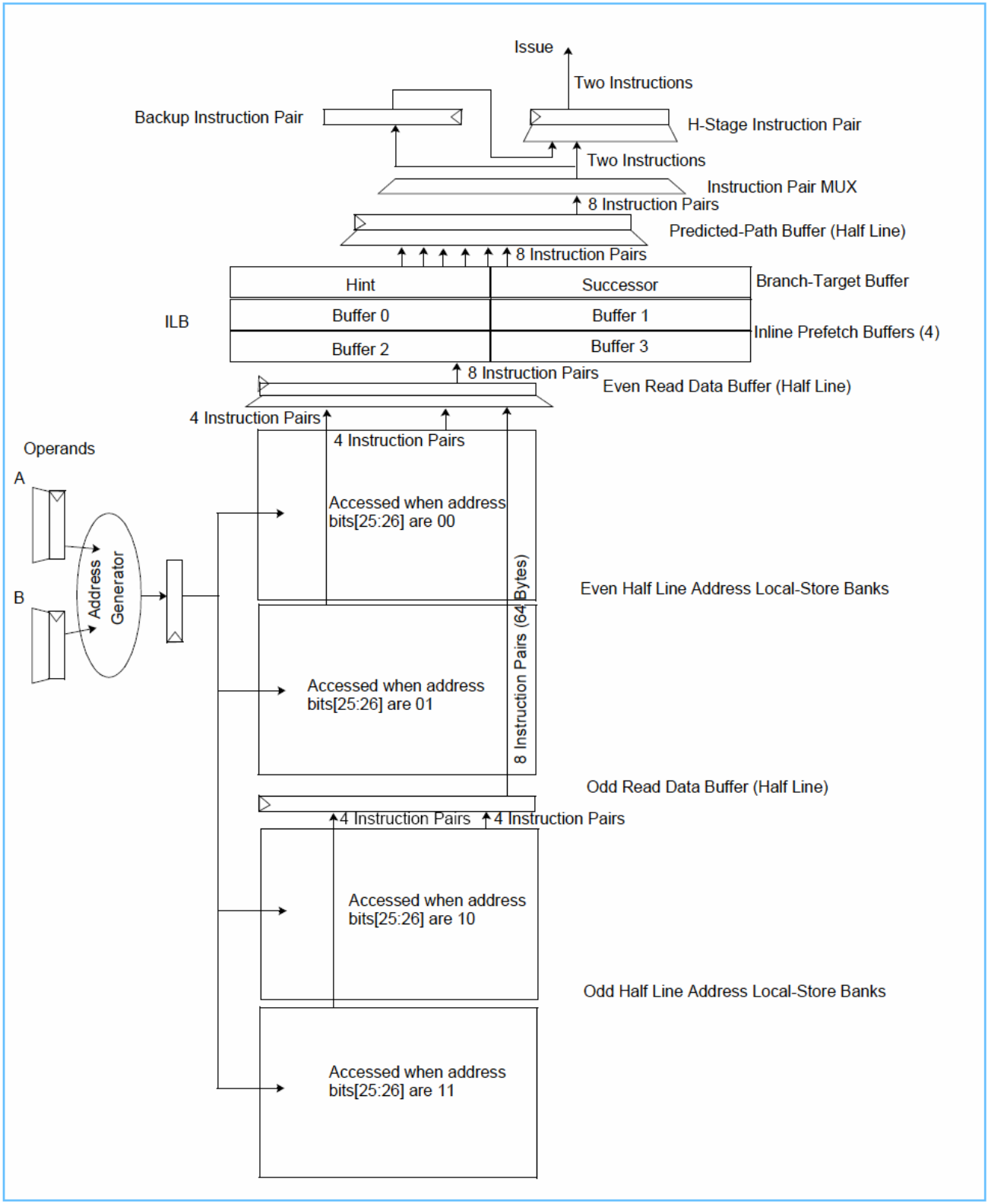
Figure 3-2 Instruction-Fetch Dataflow Diagram on page 17 illustrates the data flow of instruction fetch.

Table 3-7 shows the instruction-fetch pipeline labeled as though the fetch were a load instruction.

Table 3-7. Instruction-Fetch Execution Pipeline

<table>
<tr><th rowspan="2">Halfline</th><th colspan="10">Stage</th></tr>
<tr><th>Kf</th><th>Lf</th><th>Mf</th><th>Nf</th><th>Of</th><th>Pf</th><th>Qf</th><th>Rf</th><th>Sf</th><th>G</th></tr>
<tr><td>Even</td><td rowspan="2">Arbitrate</td><td rowspan="2">Select Address</td><td rowspan="2">Generate Address</td><td rowspan="2">Transfer</td><td rowspan="2">Decode</td><td rowspan="2">Access Array</td><td rowspan="2">To RDB</td><td>To ILB</td><td>To Predicted-Path Buffer</td><td>Pair MUX</td></tr>
<tr><td>Odd</td><td>To Even RDB</td><td>To ILB</td><td>To Predicted-Path Buffer</td></tr>
</table>

Figure 3-2. Instruction-Fetch Dataflow Diagram



3.1.2.2. Branch Hints

Branch hints are instructions software can use to facilitate efficient branch processing. The effects of branch hint instructions are not defined by the *Synergistic Processor Unit Instruction Set Architecture*. However, the branch hint instructions generally provide three kinds of advance knowledge about future branches:

- The target address
- The branch address (hint trigger address)
- The prefetch schedule

If a branch hint is provided, software speculates that the instruction branches to the target path. If a hint is not provided, software speculates that the instruction does not branch to a new location (that is, it stays inline). If either speculation is incorrect, the speculative branch is flushed by branch resolution and refetched from local store.

Branch Hint Fetch

For the SPU, the branch-hint instructions manage a branch target buffer (BTB) inside the ILB (See *Figure 3-2* on page 17). When the BTB is loaded with a branch target by a hint instruction, the hint trigger address is also loaded with the hint instruction's branch address. After loading, the BTB monitors the instruction stream as it goes into the issue stage of the pipeline. When the address of the instruction going into issue matches the hint trigger address, the hint is triggered and the SPU speculates to the target address in the hint buffer.

Branch hints arbitrate for the local store as though they were loads, except that they require three idle DMA cycles before they can issue. Three cycles are required because the even read data buffer is used in two consecutive cycles and is shared between DMA reads and instruction fetch. After the instruction is issued, it arrives at the local store in stage L. By stage T, the target instruction is loaded into the hint buffer and is ready to be merged into the instruction stream when the hint trigger occurs.

Table 3-8 shows the execution of the branch hint starting in stage J.

Table 3-8. Branch Hint Pipeline

Stage	Activity
J	Issue or Stall depending upon resource conflicts with DMA or ECC activity or register dependency
JJ	Decode Register File
K	Read Register File
L	Select address source. Assert local-store controls.
M	Generate addresses.
N	Transfer address.
O	Decode local-store array.
P	Access local-store array.
Q	Data is launched from Read latch inside bank, sent to read data buffers (RDB)
R	Even half-line is launched from even RDB; sent to ILB. Odd half-line sent from Odd RDB to even RDB.
S	Even half-line in ILB. Odd half-line sent from even RDB to ILB.
T	Odd half-line in ILB. (Worst case: this is the hint text, Best case: this stage is in parallel with stage U)
U	Calculate hint trigger for instruction in G stage. If a hit, branch is in predicted-path buffer and it is time to move target there.
G	Target in the predicted-path buffer (G-stage), branch in H stage
H	Target in H stage, branch in I stage.
I	Target in I stage, branch in J stage.
J	Target in J stage, branch in JJ stage.

Hint Trigger

Starting in stage U, the hint must be loaded and is ready for trigger. *Table 3-8* on page 18 shows the relationship between the branch and the target, assuming the trigger occurs immediately. If the objective is to load the hint early enough that the target is issued in the cycle after the branch, the branch hint should precede the branch by at least eleven cycles plus four instruction pairs. The branch hint buffer is invalid at SPU program start. After the branch hint buffer is loaded, the hint remains valid until it is either replaced by another hint or is invalidated by a **sync** instruction.
Ideally, a hint trigger occurs when the address of the instruction entering issue is equal to the hint trigger address. However, this is approximated by dividing the address matching process into two parts. Two elements are required for the hint trigger to occur:

- Line-address match: The half-line aligned address of the hint trigger must match the half-line aligned address of the data in the predicted-path buffer.
- Index match: The doubleword index of the trigger within a half-line must match the index of the instruction pair leaving the instruction pair multiplexer.

A line-address match can occur when the predicted-path buffer is loaded or when the hint buffer is loaded. An index match can occur when the hint buffer is loaded, when the predicted-path buffer is loaded, or when a sequential advance moves the instruction pair MUX to the trigger point.

There are a few cases in which hints may not apply and the SPU speculates to the inline path.
Simultaneous loading of the hint and predicted-path buffer to the trigger point is not supported in pipeline-hint mode, so there is no speculation to target.
Since index matching is done on a doubleword basis, hints that indicate that the branch target is in the same instruction pair as the branch are problematic. If such a hint triggers, the target is loaded but the trigger is inhibited to prevent a second consecutive trigger.
Hints are disabled for the first instruction pair sent to issue after a flush.

Hint Stall and Pipelined Hint Mode

Table 3-8 Branch Hint Pipeline on page 18 shows how a critically scheduled hint arrives just in time for the target to follow the branch into issue. This section explores several sequencing alternatives. It is expected that most hint usage either takes a hint for a loop-ending branch out of the loop or pairs a hint with an upcoming branch, although it is possible to sequence multiple hints in advance of multiple branches.

After the BTB and the trigger address are loaded, they remain available for triggering until either another hint is loaded or a synchronizing event occurs (**sync** or start). In particular, the BTB is not cleared if an instruction-sequencing error occurs. Therefore, it may be appropriate to hoist hint instructions from certain simple loops into loop-initialization code. Loops that execute in interrupt-enabled mode, that contain **sync** instructions, or that feature a likely-taken branch in the loop body are probably not candidates for hint hoisting.

Whenever possible, a hint instruction should precede the branch by at least eleven cycles plus four instruction pairs. Such separation between the hint and branch can improve the performance of applications on future SPU implementations.
Hints too close to the branch do not affect the speculation after the branch. Hint stall has been added to reduce the number of instructions required between the hint and the branch so that the hinted target follows the branch through issue. If there are at least four instruction pairs between the hint and the branch (based on the branch address index in the hint instruction), the SPU enters hint stall. Hint stall does not stall the hint instruction; rather, it holds the branch instruction in or before the G stage of the pipeline (where triggering occurs). The branch is held there until the hint trigger address is loaded. When the hint is ready for trigger, hint stall is released, proper address matching occurs, and the target follows the branch through the issue pipeline.

Hint stall is set up by hints with branches that are at least 8 but less than 32 instructions ahead. These hints are said to be stallable. Issue stalls if the hinted branch advances to or just before the G stage between the time when the stallable hint issues and when the BTB is loaded.

If another stallable hint is issued in the window between the first stallable hint and the issuance of the original hinted branch, the SPU disables the hint stall and enters pipelined hint mode. When pipelined hint mode is entered, the hint stall is negated and no further hint stall is generated for the duration of the pipelined hint mode. Pipelined hint mode lasts for 16 cycles after the last stallable hint is issued in pipelined hint mode. Hints executed during pipelined hint mode are written to the hint buffer in order; they are available for trigger from the time they are written until the time the next hint is written. Performance enhancement obtained by using pipelined hint mode may not be portable to future implementations of the SPU.

Table 3-9 shows the tightest code generation that enables a hint stall. Four instruction pairs and one cycle must separate the hint from the branch in order for the branch to be predicted to the taken path. If the hint is closer to the branch than four instruction pairs and one cycle, hint stall does not occur. To use hint stall, the instruction sequence must be padded with no operation (**nop**) and no operation-load (**lnop**) instructions. During hint stall, the instructions in stages H though J sequence as usual. If hint stall occurs, the speculation may still be incorrect. If that is the case, the incorrect speculation is flushed when it gets to branch resolution.

Table 3-9. Hint Stall Trigger Pipeline

Stage	Activity
G	Instruction pair containing the branch to be predicted to the taken path. This instruction pair is held while leaving the instruction pair MUX.
H	Instruction pair
I	Instruction pair
J	Instruction pair that may be issued
JJ	Instruction routing stage; hint stall generated
K	Hint does register-file read (for indirect target); hint stall effective

3.1.2.3. Inline Prefetch

An Inline prefetch is similar to a branch hint, except that the prefetcher uses empty slots on the local-store schedule to perform the fetch rather than a specific instruction. It is possible to prevent prefetch with long sequences of load and store instructions. The prefetcher starts looking for slots when one of the inline prefetch buffer pairs becomes empty.

At this point, there should be 16 instructions in the predicted-path buffer and another 32 instructions in the other prefetch buffer. If the SPU sustains dual issue, there should be 24 cycles before the prefetched instructions are needed for inline speculation. Prefetch requests require 15 cycles to load the ILB. This means that the prefetcher has nine cycles in which it must succeed in securing a slot. If the local store is busy during this interval and the prefetch is not started, instruction runout may occur (that is, there may be no instructions in the pipeline).

Runout occurs after the last valid instruction pair has been sequenced from the predicted-path buffer into the issue logic, and the line into which the SPU speculates has not been filled. The predicted-path buffer then becomes invalid, the pipelines drain, and the SPU becomes idle. When the fetch does occur, the instructions are loaded into the ILB and are then sent to the predicted-path buffer and into the instruction-issue stage (J).

The minimum runout delay to reset the fetch state machine is three cycles if the prefetch was late by only one or two cycles. Prefetches late enough to cause runout for more than three cycles incrementally delay the issue of the next instruction by one cycle for every cycle that the prefetch is delayed. As the pipeline drains, loads and stores cease, the prefetcher can schedule the prefetch, and execution resumes.

Long sequences of instructions that access the local store every cycle can cause instruction runout. Instruction runout can be prevented by breaking up these sequences with instructions that do not access the local store. The hit for branch (**hbr**[P]) instruction allows programs to ensure that the cycles needed for instruction fetch are not consumed by DMA transport. In principle, any instruction that does not use the local store and does not dual issue with an instruction that uses the local store is sufficient. Keep in mind the following facts:

- The empty slot in the local store schedule may be adjacent to a slot used by a line-read DMA transport or branch hint, thereby preventing the prefetch from using this empty slot.
- The issue-control unit can schedule non-local-store instructions during DMA transport, but doing so causes the slot that the schedule intended for prefetch to be lost.

The hint for branch (**hbr**) instruction has a P feature bit in its encoding that alters its behavior. For more information, see the *Synergistic Processor Unit Instruction Set Architecture* document. When this bit is set, **hbr**[P] arbitrates for the local store as though it were a hint. However, after it is scheduled, it becomes a **nop** instruction. This leaves a hole in the schedules of the local store and the even RDB that is wide enough for an instruction prefetch.

3.1.2.4. Instruction Issue

After fetch, instructions are processed by the issue pipeline before they are routed to one or more execution units. *Figure 3-3* on page 23 illustrates the issue pipeline. After a group of 16 instructions (8 pairs) is in the predicted-path buffer, they are sent in two instruction-fetch groups to the issue logic. The instruction-fetch address is broken into two pieces: the 16-instructions line-address and the index to a pair within the line. The pair index is used to select the instruction pair MUX and the quadword MUX (not shown). Quadword data is routed to the instruction error-correction code (ECC) checker.

The instruction pair either is sent directly to the H stage of the issue pipeline or is routed to the backup latch. An instruction pair is sent to the backup latch when the predicted-path buffer loads but the issue pipeline is not advancing. The fetch address is loaded when the predicted-path buffer is loaded; it increments when the issue pipeline advances.

In the H stage, some simple decoding of the instructions occurs as preparation for the source-to-target comparisons in the I stage.
The I stage compares the register-source addresses of the two new instructions (I0 and I1, the first and second instructions of the issue group in the I stage of the pipeline) to the register-target addresses of older instructions. If the source-to-target comparisons find matches, they have found register dependencies in the program, and it is possible that new instructions have to wait in the issue pipeline until the older instructions are far enough ahead that their results are ready when needed. The source-to-target comparisons are done in three steps.
The I-stage compares sources to targets in the following order:

1. Even pipeline
2. Odd pipeline
3. Instructions that have not yet been issued to an execution pipeline

Targets in the pipelines have a register address and a latency code. The latency code is decoded from the instruction and is related to the instruction class. The latency code allows the source-to-target comparators to decide if a result is ready when needed or, if not, how long a dependent instruction must wait in the issue stage. The source-to-target comparators produce a summary for each instruction. The summary is used to decide whether the instruction should stall or advance when it gets to the issue control stage. The source target comparators also produce a source versus target matrix that is used by forward control to derive the forwarded source selects.

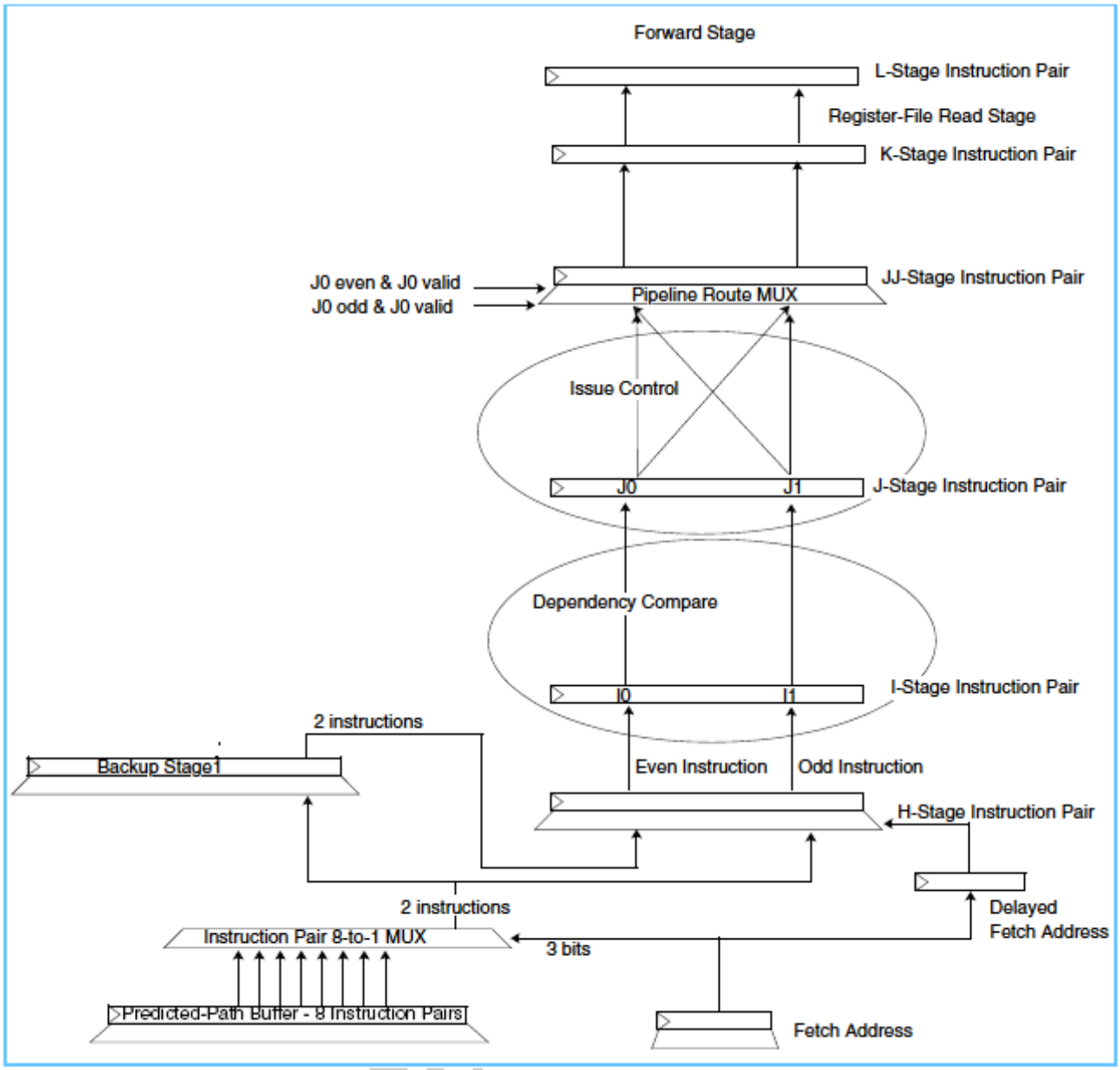
Inside the even and odd comparators, there is a target pipeline that accepts issued instruction targets at the beginning and advances every cycle to track the instruction's progress through execution. These targets must be compared with the sources of the instruction in the I-stage during every cycle. These comparators compute two functions:

- Forward match (fm) = (Source == Target) && Source Valid && Target Valid
- Stall match (sm) = (Source == Target) && Source Valid && Target Valid but not Available according to latency code

The forward matches are processed in the next stage by forward control. The stall matches are ORed together to produce two dependency signals: I0 depends on some unavailable target and I1 depends on some unavailable target. These signals are sent to issue control.

In the J stage, issue control makes the issue/stall decision for both J0 and J1 and advances the issue pipeline. The issue/stall decision is based on the stall match signals, local store, and floating point pipeline hazard detection. (J0 is the first instruction of an issue group in the J stage and J1 is the second instruction in the J stage.)

Figure 3-3. Instruction Issue Data Flow



3.1.2.5. Forward Control and the Operand Pipeline

Instruction routing, register-file read-address decode, and forward control begin during the JJ stage.
The J0 and J1 instructions are sent to the even and odd execution pipelines.
If the J0 instruction is valid, it is sent to the pipeline containing the unit needed for execution.
If the J0 instruction is not valid or if it is sent to the even pipeline, then the J1 instruction, if it is valid and for the odd pipeline, is sent to the odd pipeline and executed by an odd-pipeline unit.
If the J0 instruction is valid and is to be sent to the odd pipeline, then the J1 instruction is not issued.
If the J0 instruction is not valid, then the J1 instruction will be issued to the unit that will execute it.

The forward priority encoders analyze the match bits generated in the I stage to produce an encoded operand location code so that the following occurs:

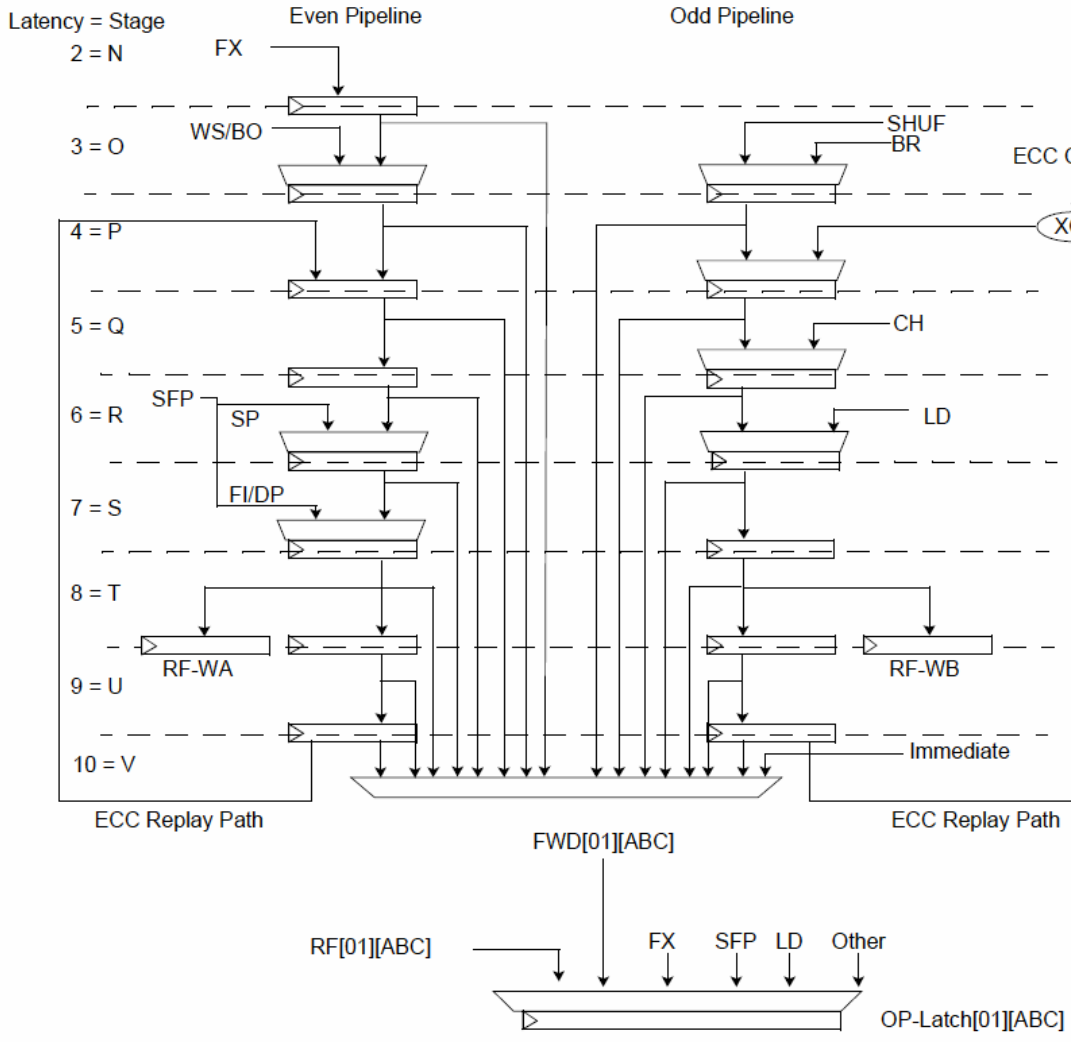
- Results in earlier stages of execution (more recent instructions) have priority over results in later stages of execution (older instructions).
- Results from functional units on the odd pipeline have priority over results from functional units on the even pipeline in the same stage.
- All results have priority over operands from the register file.

The outputs of the priority encoders are routed like the instructions and sent to the forward-select decoders. The forward-select decoders expand the encoded operand selectors to fully decoded MUX selects during the K stage. These selects are latched and used to select the correct results in the L stage of the forwarding network.

Figure 3-4. Forwarding Data Flow on page 25 illustrates the result equalization network and L stage forwarding operation, which starts in the forwarding network and finishes on the operand latch within each unit.
Results enter the network at various points, depending on the unit latency and pipeline assignment.
For example, the FX class instructions enter the network during the N stage on the even pipeline.

After a result is entered into the result equalization network, it is staged to the T stage, where it is driven to the register file's writeback latch. While the result is in the network, it can be forwarded to operand latches for dependent operations. From stages U and V, the result continues to advance in the equalization network to bypass the register file.

Figure 3-4. Forwarding Data Flow



ABC	A, B, C operands for instruction execution in the even (0) and odd (1) pipelines
	[01][ABC] = A0, B0, C0, A1, B1, C1
BR	Branch resolution
BO	Byte operations
CH	Channel interface
DP	Double precision
ECC	Error-correction code
FI	Floating-point integer
SFP	Floating-point unit
FWD	Forwarding network

FX	Fixed point instruction
LD	Load
OP-Latch	Operand latch
RF	Register file
SHUF	Shuffle
SP	Single-precision
WA	Reg file write port A
WB	Write port B
WS	Word rotate and shift
XOR	Exclusive OR

3.1.2.6. Branch Resolution and Instruction Commit

The branch subunit executes the BR class instructions. These instructions execute on the odd pipeline, and the link setting instructions have a latency of 4 cycles. The branch unit processes every issue packet in order to maintain the program counter as part of the instruction commit process. The branch resolution unit processes all instruction packets and ensures the instructions are correctly sequenced. Each fetch group flows through the issue pipeline with its fetch address. When an issue group is issued, the fetch address of the first instruction of the issue group is determined and sent through the branch resolution pipeline. Instruction fetch and issue are entirely speculative, but satisfy two properties that ensure forward progress and allow branch resolution to guarantee correct instruction sequencing:

- All issue packets are correctly labeled with the address from which they were fetched.
- A fetch after a pipeline flush returns at least the first instruction group from the requested address.

Instructions are sequenced through the program counter (SPU Next Program Counter Register (SPU_NPC)), which is always the address of the next instruction to be committed. When the SPU becomes enabled, the program counter and interrupt enable status are loaded from the channel unit. After the program counter is loaded, the branch subunit initiates a pipeline flush and an instruction fetch from the program counter. It then moves the program counter into the next inline register. As instructions flow, each instruction packet computes, from its fetch address, the address of the instruction packet that follows it. The branch subunit then verifies the address of the issue packet that follows. If an instruction packet is committed, its next address is moved to the program counter. If the address of the subsequent issue packet matches the next address, the subsequent packet is committed. Otherwise, there is an instruction sequencing error. The instruction packet is flushed, and an instruction fetch is requested for the address in the program counter. *Table 3-10 Branch Resolution Pipeline* on page 27 describes the branch resolution pipeline starting in stage K, just after issue.

In stage K, both the next inline instruction address and relative target address are computed.
Stage L is the forwarding stage for branches. Branches use two different operands from the register file. Indirect branches need the target address from operand A. Conditional branches need the condition from operand C that goes to an indirect target. The actual target address is formed on the operand A latch. Branch indirects receive data from the register file or forwarding networks, while branch absolute and branch relative override the forwarding control and select either the relative or the absolute target, based on the decode of the branch instruction performed at the K stage.

During the M stage, the conditional operand is evaluated for its zero or nonzero status. To reduce branch resolution latency, the address of the next instruction packet (in the L stage) is compared with the two possible next program counter values: the next inline address and the next target address. These comparisons are performed during the M stage of the current instruction and the L stage of the next instruction packet.

In the N stage, branch control decides (based on the state of the SPU, the decode of the current instruction, the address comparators, and the condition status) whether to hold the program counter or to load the program counter with the next inline address or the next target address. If the sampled address comparator has not matched, the speculative execution is in error and a mispredict is declared. The branch resolution unit then initiates a pipeline flush and a restart.

Instructions are tracked, starting in the K stage through register-file writeback, with a set of valid bits. Each bit in the living vector indicates whether an instruction that has been issued but not yet cancelled is present in the corresponding stage of the pipeline. If a live bit corresponding to a particular instruction is off, the instruction does not write the register file, advance the program counter, update the local store, or have side effects in the channel unit.

The pipeline flush is driven in the O stage of the branch and must cancel the instruction packets in and before stage N. These cancelled instructions must not modify any architectural state. Important examples of architectural state are the register file and the local store. The next packet does not update the register file until

stage T; therefore merely negating the live bits for stage N prevents write back. Local-store writes are canceled in stage N of the store.

After the flush is driven in the O stage, fetch activity for the restart begins. The fetch must arbitrate for the local store in stage Kf (the branch's O stage). Instructions before the stage O are flushed and do not impede the arbitration for the fetch. The fetch progresses as shown in *Table 3-7 Instruction-Fetch Execution Pipeline* on page 16. The fetch ends with the fetched instruction in stage G if its address is in an even half line. The fetch ends with the fetched instruction about to enter stage G if its address is in an odd half line. Therefore, the mispredict penalty is 17 or 18 cycles, measured from stage O of the branch to stage N of the fetched instruction, depending upon the address of the fetched instruction.

Table 3-10. Branch Resolution Pipeline

Stage	Load with ECC Error	Stage	Branch Packet	Stage	Next Packet
		K	Compute next inline address. Compute relative target address.		
Q	Bank MUX	L	Decode indirect condition. Forward indirect target. Compute hint trigger address.	K	
R	Forward ECC check 1	M	Decode instruction. Evaluate branch condition. Target MUX.	L	
S	ECC check 2	N	Branch control. Packet address compare. Program counter/mispredict MUX.	M	
T	Logic	O	Drive program counter/mispredict	N	Drive store inhibit
U	ECC error known. Kill stage O.	P	Stage Lf of *Table 3-7*	O	Store address decode
		Q		P	Store array access is inhibited

3.1.2.7. Commit Priority

Branch control prioritizes events, chooses the source for the next program counter, and generates the flush signal when necessary. When the flush is asserted, branch control enters the flush state, moves the program counter to the inline address register in the M stage, and waits for the refetch. While in the flush state, all additional flush requests unrelated to interrupt acknowledgment will be rejected.

The following events can cause a pipeline flush:

1. When there is a rising edge on the run bit, a power-on-reset (POR) pulse is generated. This event has highest priority.
2. A data ECC error is the next event in priority order. The ECC error is for a load instruction in the U stage. If live instructions are being processed, the program counter is backed up one cycle, and a flush is generated.
3. When there are no valid instructions to be committed, the program counter is held from the previous cycle but no flush is generated. The most likely cause is an issue stall, but instruction runout is also a possibility.
   Although the N-stage instruction is not valid, the M-stage instruction may be valid.
   If the M-stage instruction is valid, branch control is in the flush state, and the inline address matches, branch control leaves the flush state. If, while branch control is in the flush state, the M-stage instruction is valid but the inline address does not match the M-stage instruction address, the M-stage instruction is rejected and another pipeline flush is initiated and branch control remains in the flush state.

4. Instruction ECC error status is propagated alongside the instructions it applies to. When an instruction packet from a quadword with an ECC error is committed, the pipeline is flushed and the erroneous instruction is refetched.
5. If the N-stage instruction packet is valid, it might be a channel operation or branch on event that can request wait status. Wait is a low-power alternative to programmatic polling. The instruction that requests wait is flushed and fetched. It is held in the issue pipeline until the channel unit indicates that the instruction completes without delay.
6. The final event is a normal instruction packet commit. If the instruction is a **sync** instruction, the pipeline after the **sync** is flushed and refetch is initiated, perhaps in preparation for self-modifying code. Otherwise, if the next program counter does not match the address of the next valid instruction packet, a mispredict has occurred. That instruction packet is flushed and refetch is initiated.

The SPU interrupt acknowledge occurs in parallel with the above priority encoder. If an SPU interrupt is asserted and interrupts are enabled, a flush is generated and any outstanding flush is cancelled. Interrupts are disabled and the value in the program counter is moved to Save and Restore Register 0 (SRR0) while the program counter is cleared. An instruction-fetch request is made for address 0. Additional information about the interrupt facility is available in the *Interrupt Facility* chapter of *Synergistic Processor Unit Instruction Set Architecture* document and in *Section 3.1.7 SPU Interrupt Facility* on page 39.

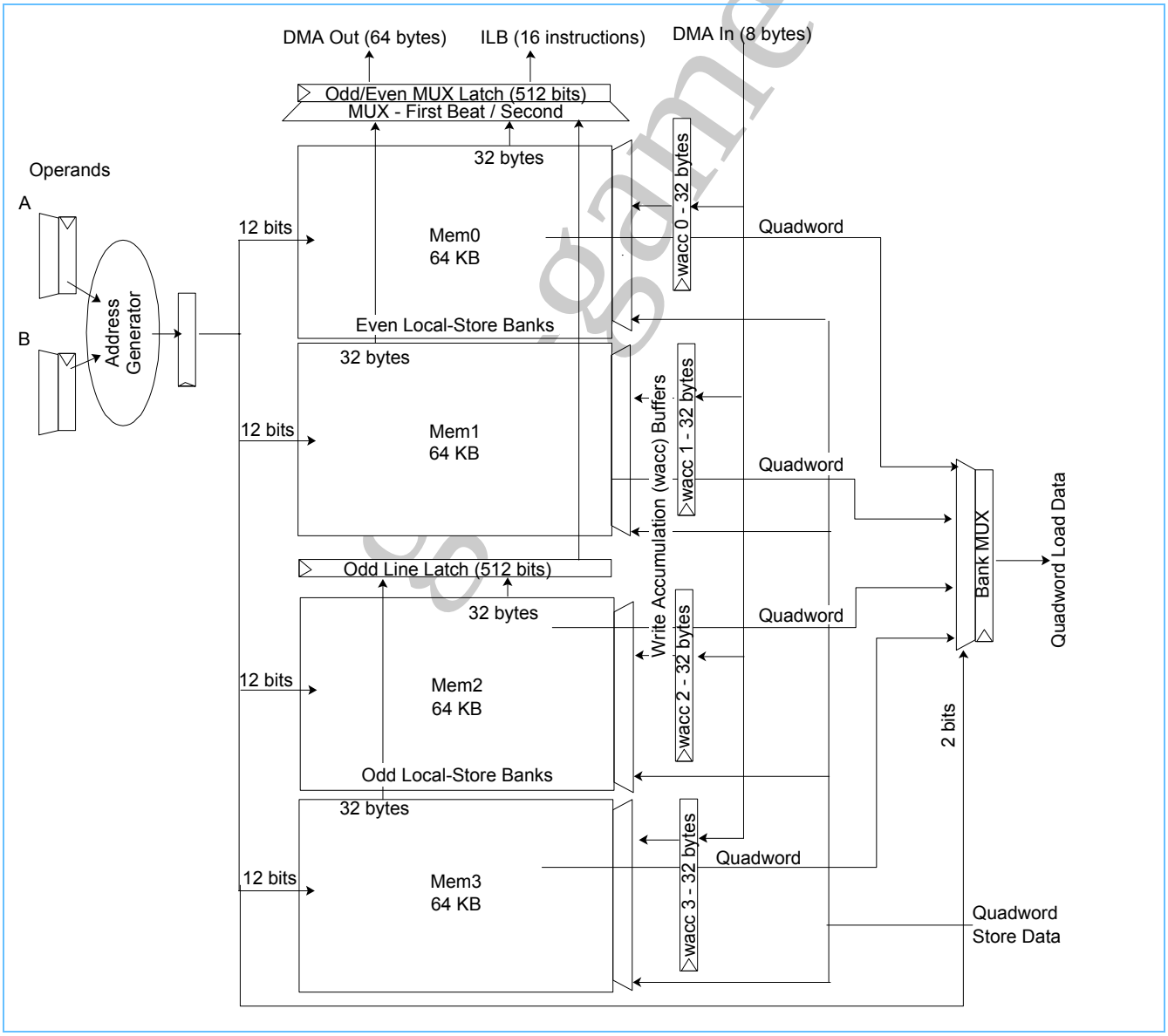
3.1.3. SPU Load and Store Unit

The SPU load and store unit has the following key features:

- 256 KB unified instruction and data storage
- 6-stage pipeline
- Four individual arrays
- 128-bit, quadword interface for load and store data
- 1024-bit line transfer for DMA read and write to provide low DMA occupancy
- 1024-bit line, line-aligned transfer for instruction read to provide low instruction-fetch occupancy and high fetch efficiency
- 512-bit interface for instruction data and DMA read
- 64-bit interface for DMA write with 1024-bit write accumulation buffer
- Clock gating for power efficiency

Figure 3-5 provides a diagram of the load and store unit.

Figure 3-5. Load and Store Unit



The load and store unit serves three purposes: executing LS and HB class instructions, fetching instructions, and processing external DMA requests. The local store is single-ported and can support only a single access per cycle. The SPU arbitrates the local store, according to the priority levels described in *Table 3-11*.

Table 3-11. Local-Store Access Arbitration Priority and Transfer Size

Transaction	Transfer Size (Bytes)	Priority	Maximum Local-Store Occupancy (SPU Cycle)	Access Path
MMIO	≤ 16	1 (Highest)	1/8	Line Interface
DMA	≤ 128	1		
DMA List Fetch	6	1	1/4	Quadword Interface
ECC Scrub	16	2	1/10	
SPU Load and Store	16	3	1	
Hint Fetch	128	3	1	Line Interface
Inline Fetch	128	4 (Lowest)	1/16 for inline code	

Load and store instructions flow through the pipeline as shown in *Table 3-12*. The address operands for the load become available in the M stage, where an add is performed. The load data is available in the forwarding network in stage R. The remainder of the pipeline is associated with the ECC check.

Table 3-12. Load Execution

Load/Store	Load and Store Stage Name / Latency Cycles									
	M/1	N/2	O/3	P/4	Q/5	R/6	S/7	T/8	U	V
Load	Generate address	Transfer	Decode	Access array	Bank MUX	Forward ECC1	Stage ECC2	Pre-write back. Error status known. ECC3	Register-file write. ECC4	Read bypass. Correct ECC error.
Store	Generate address. Generate ECC.	Transfer. Generate ECC.	Decode	Access array	—	—	—	—	—	—

The SPU uses an error checking and correcting (ECC) mechanism to protect the SPU local store memory from soft errors. A soft error is a bit that changes in memory without being written.
For each quadword (128 bits wide), the ECC implementation detects and corrects single-bit (correctable) errors, and detects (uncorrectable) 2-bit errors. When data is written to the local store, a 9-bit ECC code is generated and stored with each quadword of data. When the data is read, the 9-bit code and the data are processed using detection logic to determine if there are errors.

Correctable ECC errors from load instructions and DMA reads are corrected inline without correcting the local store; the data remains in error in the local store. When an ECC error is detected, the read address is no longer available. An ECC scrub state machine exists in the DMA logic that accesses every quadword in local store to find and correct every ECC error in the local store. This state machine is initialized after ECC errors are detected during load instructions or DMA reads.

Loads that generate ECC errors flow through the pipeline, as described in *Table 3-12*.
The result is available for forwarding in the R stage. The R stage is also the first of stage of ECC checking.

In stage T, the ECC error status becomes known, and ECC correction can begin. Results (uncorrected) are written to the register file in stage U during the second cycle of ECC correction, when branch resolution starts reacting to the ECC error. In stage V, corrected load data is recycled into the P stage of the forwarding network. (See *Figure 3-4. Forwarding Data Flow* on page 25 for details of the forwarding network.)

The SPU ECC recovery strategy temporarily stops program execution between two instructions, cleans up the incorrect state, and then continues with the next instruction. To obtain a correct state consistent with stopping between the independent and the corrupted instructions, the first ECC error and the next five cycles of register file writes are replayed through the ECC data corrector and returned to the forwarding network, replacing the first corrupted results, to be rewritten in the original order.

3.1.4. SPU Floating-Point Unit

3.1.4.1. FPU Instruction Classes

The floating-point unit (SFP) is responsible for computing instructions grouped into the four SPU instruction classes shown in *Table 3-13*. These classes of instructions comprise single-precision floating-point operations, double-precision floating-point operations, integer multiply operations, integer converts, and byte operations. Floating-point compare instructions are computed in the fixed-point unit (SFX), not in the floating-point unit (SFP). The same holds for the floating-point reciprocal and reciprocal square-root estimate operations.

Table 3-13. FPU Instruction Classes

Class	Description	Output Bus	Latency	SP First-Stage Cycle
FI	Floating-point integer	FP	7	1
SP	FPU single-precision	FP	6	1
DP	FPU double-precision (partially pipelined)	FP	13	2
BO	Byte operations	BO	4	1 (operand latch)

3.1.4.2. FPU Instruction Execution

Figure 3-6 on page 33 shows a high-level block diagram of the FPU. The FPU consists of three units, the byte unit, the double-precision unit, and the single-precision unit.
These three units are fed by two sets of operand latches. The first set feeds the byte unit, the double-precision pipeline, and the formatter of the single-precision pipeline. It is used by DP, BO, and FI instructions.
SP instructions bypass this set; their operands go directly to the second set of operand latches, which is used by SP and FI instructions. This set of operand latches feeds the core of the SP pipeline.

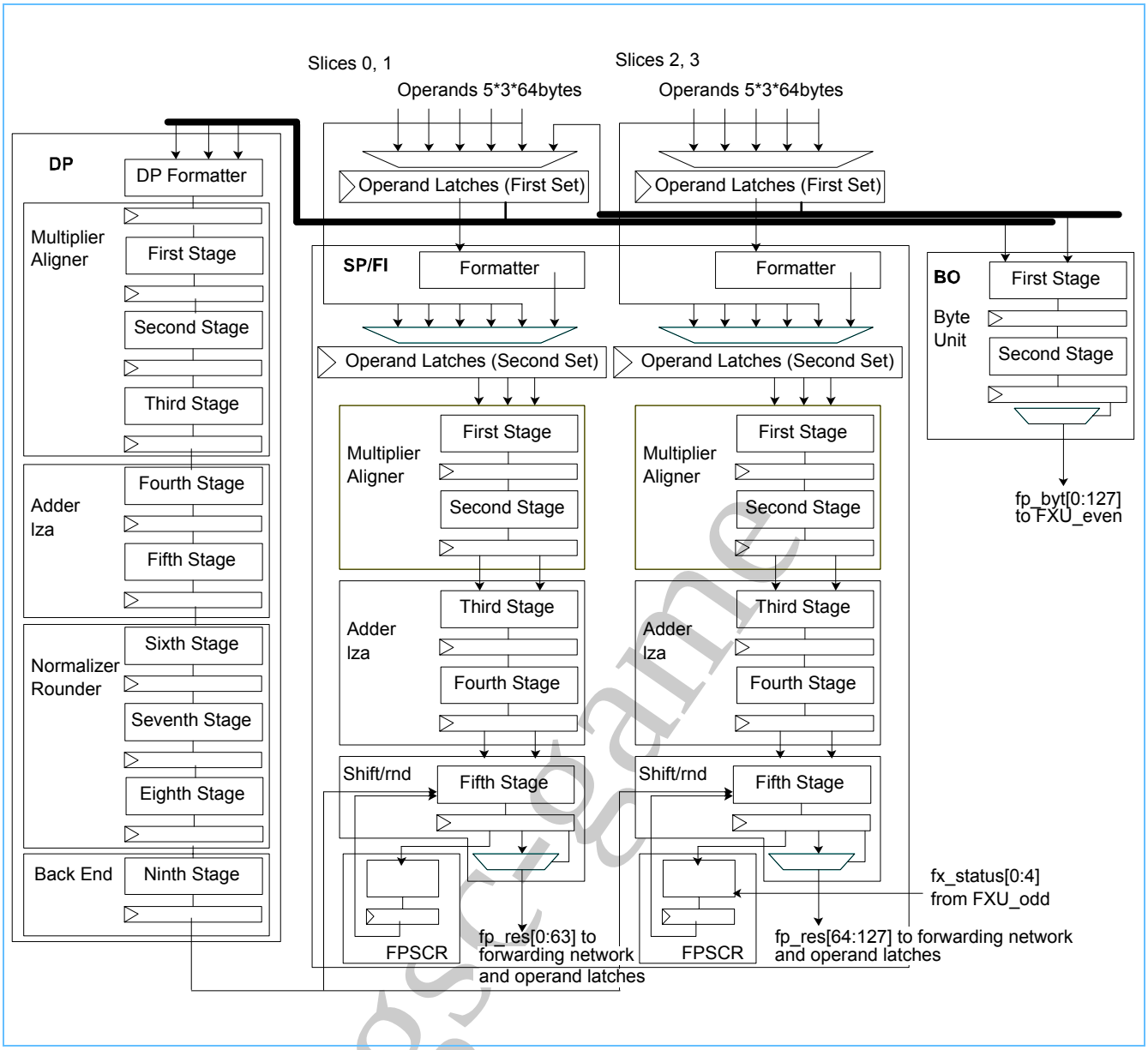
3.1.4.3. Structural Hazards

Byte operations, single-precision, and floating-point integer instructions are computed in a fully pipelined manner with a latency of 4, 6, and 7 respectively. The floating-point integer pipeline reuses most of the single-precision floating-point pipeline hardware and shares the same result bus. However, there is an extra stage inserted before the single-precision pipeline for operand formatting. The first set of operand latches provides the data inputs for this extra stage, and its outputs are captured in the second set of operand latches. This extra stage and sharing most of the SP hardware cause structural hazards for both the SP operand latch and the SFP result bus that prevent an SP instruction from issuing directly after an FI instruction.

Byte operations (BO) share the operand latches (first set) with the FI instructions. To keep the FPU control simple, the four-cycle byte operations share the first two stages of the FPU control automaton with the FI instructions. This sharing causes the same structural hazards as for the FI instructions, which prevent an SP instruction from issuing directly after a BO instruction. The BO instructions also share the result bus, not with the single-precision pipeline but with the word rotate and shift (WS) class instructions in the SFX unit. Because both kinds of instructions are executed in the even pipeline, and because both have the same latency, the result bus can be shared without causing any structural hazard.

The SPU DP class instructions process two double precision operations in 2-way SIMD fashion. However, the double precision unit is capable of performing only one double precision operation per cycle. Thus, the FPU executes the DP class instructions by breaking up the SIMD operands and executing the two operations in consecutive cycles. The SPU does not manage enough instruction states to fully pipeline the DP class instructions. Although these DP class instructions have 13-cycle latencies, only the final seven cycles are pipelined. No instructions are dual issued with these instructions, and no instructions of any kind are issued for six cycles after a DP class instruction is issued.

Figure 3-6. Functional Block Diagram of the FPU



Note: lza: leading-zero anticipator

3.1.4.4. Precision of Divide and Square Root

The reciprocal and reciprocal square root of a floating-point number are computed by a sequence of instructions with increasing precision after each step. The sequence starts with a table lookup, executed in the odd fixed-point unit. The result of the table lookup can either be interpreted as a first approximation of the result, or as a base and step, which serve as inputs for the floating-point interpolate instruction. The interpolate instruction, executed in the SFP, produces a better approximation of the result. This value is then used as the seed for a Newton Raphson iteration. After one Newton Raphson iteration, the error is, at most, one unit in last place (ULP) except for some corner cases.

The single-precision FPU implements IEEE arithmetic with the following deviations:

- It only supports one rounding mode: round towards zero, also known as truncation.
- Denormal operands are treated as zero, and denormal results are forced to zero.
- Numbers with an exponent of all ones are interpreted as normalized numbers and not as infinity or not a number (NaN).

Thus, the representable, nonzero numbers are in magnitude in the range of `Xmin` = 2^{-126} to `Xmax` = $(2-2^{-23}) \cdot 2^{128}$. If the exact result overflows (that is, if it is larger in magnitude than `Xmax`), the rounded result is set to `Xmax` with the appropriate sign. If the exact result underflows (that is, if it is smaller in magnitude than `Xmin`), the rounded result is forced to zero.

Reciprocal Estimate

The goal is to calculate `y = 1/x`. The following code sequence computes approximations `y0`, `y1`, and `y2` of `1/x` with increasing precision:

```
FREST       y0,x              // table lookup
FI          y1,x,y0           // interpolation
FNMS        t1,x,y1,ONE       // t1= -(x*y1 - 1.0)
FMA         y2,t1,y1,y1       // y2= t1*y1 + y1
```

For some operands `x`, the reciprocal requires infinite precision (for example, `x = 3`). Because the floating-point numbers have a finite precision, the results have to be truncated to the next representable number. For some operands `x`, the exact result `1/x` is outside the range of representable numbers. In case of an overflow, the result is forced to `Xmax` with the appropriate sign; in case of an underflow, it is forced to zero.

With truncation rounding, `truncate(-1/x)` equals `-truncate(1/x)`. Therefore, the following analysis is restricted to positive operands `x`. Let `x` be a positive floating-point number, with `1/x` in the representable range; let `Y = 2^e (1.f)` be the rounded value of `1/x`. Thus, Y is the largest floating-point number with `(xY ≤ 1.0)`, or:

$$\left(Y \le \frac{1}{x} < Y + 1\text{ulp} = Y + 2^{(e-23)}\right)$$

Precision of the Approximations

For zero and denormal inputs, the correctly rounded result is `Xmax (x'7FFF FFFF')`. The approximations `y0`, `y1`, and `y2` are as follows:

$y_0 \ge$ x'7F84 1D07'

$y_1 =$ x'7F80 000'

$y_2 =$ x'7FFF FFFF'

For normalized numbers, $x < 2^{126}$, the reciprocal $1/x$ is a normalized number. The three approximations are within the following absolute and relative error bounds:

$$|Y - y_0| \le 956084 \text{ ulp} \qquad |1 - xy_0| < 1/16$$
$$|Y - y_1| \le 2164 \text{ ulp} \qquad |1 - xy_1| < 1/4765$$
$$|Y - y_2| \le 1 \text{ ulp} \qquad |1 - xy_2| < 1/6635785$$

Value y2 approximates Y from below; it is either exact or it is one ULP too small in magnitude.

For large normalized numbers, $x > 2^{126}$, the reciprocal $1/x$ is smaller than Xmin. The rounded result, therefore, underflows to $Y = 0$. Approximation y0 is a denormalized number, which the SFP treats as zero. Approximations y1 and y2 are a true zero.

Corner Case $x = 2^{126}$

For this operand, the exact reciprocal is Xmin. Most iterative methods approximate the result from below. Any such approximation of Xmin is a denormalized number, which, in the SFP, is treated as zero. The approximations y0, y1, and y2 are, therefore, zeros as in the previous case.
The absolute error is larger than one ULP. If this is a concern, the following code sequence produces the IEEE answer:

```
maxnounderflow=x'7e800000
min=x'00800000
msb=x'80000000
fcmeq selmask,x,maxnounderflow
and s1,x,msb
or smin,s1,min
selb y3,selmask,y2,smin
```

Reciprocal Absolute Square-Root Estimate

The goal is to calculate y = 1/sqrt(abs(x)). The following code sequence computes approximations y0, y1, and y2 of 1/sqrt(abs(x)) with increasing precision:

```
mask = x'7fff ffff
HALF = 0.5
ONE = 1.0
FRSQEST      y0,x              // table lookup
AND          ax,x,mask         // ax= abs(x)
FI           y1,ax,y0          // interpolation
FM           t1,x,y1           // t1= x * y1
FM           t2,y1,HALF        // t2= y1 * 0.5
FNMS         t1,t1,y1,ONE      // t1= -(t1 * y1 - 1.0)
FMA          y2,t1,t2,y1       // y2= t1 * t2 + y1
```

For some operands x, the reciprocal square root requires infinite precision. Since the floating-point numbers have a finite precision, the results have to be truncated to the next representable number. For zero inputs, the exact result overflows to Xmax.

Let x be a normalized floating-point number, and let $Y = 2^e(1.f)$ be the rounded value of 1/sqrt(abs(x)). Thus, Y is the largest floating-point number with $xY^2 \le 1.0$, or:

$$Y \le 1/(\sqrt{|x|}) < Y + 1\text{ulp} = Y + 2^{(e-23)}$$

Precision of the Approximations

For normalized numbers `x`, the reciprocal `1/x` is a normalized number. The three approximations are within the following absolute and relative error bounds:

$$|Y - y_0| \le 489115 \text{ ulp} \qquad |1 - xy_0| < 1/16$$
$$|Y - y_1| \le 2243 \text{ ulp} \qquad |1 - xy_1| < 1/2918$$
$$|Y - y_2| \le 1 \text{ ulp} \qquad |1 - xy_2| < 1/3797410$$

Unlike the reciprocal estimate, the value `y2` can equal `Y`, or be one ULP too large or too small. All three cases occur.

For zero and denormal inputs, the correctly rounded result is `Xmax (x'7FFF FFFF')`. The approximations `y0` and `y1` are as follows:

$$y_0 \ge x'7F82\ 0C83'$$
$$y_1 \ge x'7F80\ 0000'$$

The result of the Newton Raphson step depends on the representation of the zero operand. For a true zero and for a denormalized number with a fraction smaller than or equal to `x'FF53C'` (that is, $|x| \le 1045820 \cdot 2^{(-126-23)}$) approximation `y2` equals `Xmax`.
For denormalized numbers with a fraction larger than `x'FF53C'`, approximation `y2` is `x'7F00 0000'`.
In the latter case, the absolute error is larger than one ULP. The following sequence could be used to correct the answer:

```
zero=0.0
mask=x‘7fffffff
fcaeq z,x,zero
and zmask,z,mask
or y3,zmask,y2
```

3.1.5. Fixed-Point Unit

3.1.5.1. FXU Instruction Classes

The fixed-point unit (FXU) is responsible for computing instructions, which are grouped into the three SPU instruction classes shown in *Table 3-14*. These classes of instructions comprise word shifts and rotates (WS), fixed-point arithmetic and logical operations (FX), and shuffle operations (SH).
Floating-point compare instructions belong to the FX instruction class and are computed in the SFX unit, not in the FPU. The same holds for the floating-point reciprocal and reciprocal square-root estimate operations, which belong to SH instruction class.

Table 3-14. Fixed-Point Unit Instruction Classes

Class	Description	Pipeline	Latency
WS	Word shifts and rotates	Even	4
FX	Fixed-point arithmetic and logical operations	Even	2
SH	Shuffle operations	Odd	4

3.1.5.2. FXU Instruction Execution

The fixed point unit consists of two units on two different pipelines:

- SPU even fixed-point unit (SFX) -- Executes class FX and class WS instructions.
- SPU odd fixed-point unit (SFS) -- Executes class SH instructions.

The class SH instructions are assigned to the odd pipeline, and class FX and WS instructions are assigned to the even pipeline. That being the case, a class SH instruction can be issue in parallel with a class FX or WS instruction, but a class FX instruction can never be issued in parallel with a class WS instruction.

Execution of FX Class Instructions

There are three subunits that execute the FX class instructions: the count leading-zero macro, the logical macro, and the add/compare macro.
The count leading-zero macro implements the **clz** instruction. The logical macro implements all the logical instructions. The add/compare macro implements the adds and subtracts as well as the integer and floating-point compare instructions.
For all class FX instructions, the actual calculation is done in the first cycle. The second cycle MUXes the add and the count leading-zero and logical macro results. Then, the second cycle distributes the results to all operand latches that need to receive the results (SFX, SFP, local store, forwarding network, and control unit).

Execution of WS Class Instructions

For class WS instructions, the actual execution is done by the word rotate macros.
The first two cycles do the actual computation, and the third cycle MUXes the word rotate result and the SFP byte unit. Then, the third cycle distributes the result to the forwarding network.
In the fourth cycle, the forwarding network can send the result to the operand latches.

Execution of SH Class Instructions

For class SH instructions, the actual implementation consists of two different parts: the table-lookup macro and the shuffle macro.
The table-lookup macro implements the reciprocal estimate instructions as well as some other instructions, such as the “form select mask”, “gather bits”, and “generate control for” instructions. The shuffle macro implements the quadword shuffle, rotate, and shift instructions.

The results of the table-lookup and the shuffle macro are MUXed in the third cycle and distributed to the forwarding network.

3.1.6. Synchronization

The SPU is a deeply pipelined processor with a large register file and some buffering. This section discusses how instruction execution is synchronized with run state changes, local-store updates, and channel operations.

Execution begins when the external interface causes the SPU to enter the run state and ends when the SPU exits the run state. The SPU can exit the run state for a number of reasons:

- The SPU executes a stop instruction.
- The SPU executes a halt instruction with a true condition.
- The SPU executes an illegal instruction
- The SPU detects an uncorrectable ECC error
- The SPU is in single-step mode and commits an instruction packet.
- The external interface forces the SPU out of the run state.

Regardless of how the SPU exits the run state, there is a point in the instruction stream before which execution has completed and after which it has not started.
When the SPU executes a stop instruction, execution stops after the stop instruction and before the instruction after the stop instruction.
Halt instructions can also stop the SPU; however, the SPU continues execution for some time after the halt before it stops. In particular, the halt instruction negates the run bit in the Q stage, which initiates the drain sequence. Two cycles after the run bit is negated, the instructions in the H stage of the issue pipeline are invalidated. Instructions after the H stage are allowed to run, unless one of these instructions causes a mispredict. If a mispredict occurs, the refetch is suppressed. Thus, the stop point is after the instructions in the I-stage, two cycles after the run is negated, or after the last instruction of the correctly sequenced instruction stream, whichever is earlier.
If the external interface forces an exit from the run state, the run bit is negated and the SPU begins the drain sequence.

If the SPU is in single-step mode and is put into the run state, it fetches an instruction packet from the start program counter. This instruction packet is committed, and the SPU stops after this instruction packet and before the next.

A primary source of potentially asynchronous interaction is the local store. Three different reference streams access the same storage: instruction fetch, load and store, and DMA activity. Two instructions can help software manage the visibility of results in the local store: synchronize (**sync**) and synchronize data (**dsync**). Instruction-fetch buffers and pipelines can be flushed by executing the **sync** instruction. The **sync** instruction must be used before attempting to execute new code that either arrives through DMA activity or is written with store instructions. DMA activity can also interfere with store instructions and the store buffer. Software can ensure all store buffers have been flushed to the local store by executing the **dsync** instruction. No **dysnc** is needed for DMA activity in this implementation.

Channel activity is viewed in the software through channel read, channel write, and channel count instructions. Data through a particular channel is never reordered. Channel instructions can either be blocking or nonblocking. If they are blocking, read data must be present or previous write data must have been consumed before the instruction can complete. Thus, the hardware synchronizes the channel streams. If nonblocking mode is used, then software must manage the synchronization based on the ordering of channel data. Some channel activity may, as a side effect, alter the SPU operating state. These side effects are not ordered with respect to the SPU pipeline. Such side effects must be synchronized with instruction execution with a channel synchronization. Channel synchronization is initiated by executing a **sync** instruction with the C feature bit set to cause channel synchronization to occur before instruction synchronization.

3.1.7. SPU Interrupt Facility

The SPU can monitor event status through the SPU Read Event Status (SPU_RdEventStat) Channel. The events monitored are unrelated to the external interrupts handled by the PPE. Software can process the event status in the SPU_RdEventStat Channel by polling using the **bisled** instruction or by enabling the SPU interrupt facility. When enabled, the interrupt facility causes the SPU to execute the interrupt handler located at local-storage address x'0' when the SPU_RdEventStat Channel has a nonzero count.

The SPU interrupt can be enabled in one of two ways:

- Setting the E feature bit in the indirect branch instruction (see the *Branch Indirect* section in the *Synergistic Processor Unit Instruction Set Architecture* manual)
- Setting bit [31] of the *SPU Next Program Counter Register (SPU_NPC)* prior to running the SPU

The SPU interrupt is taken if the interrupt is enabled and if the count in the SPU_RdEventStat Channel has transitioned from zero to one (edge triggered based on the ORing of events in the SPU_RdEventStat Channel, masked by bits in the SPU Write Event Mask (SPU_WrEventMask) Channel, and the acknowledge bits in the SPU Write Event Acknowledgment (SPU_WrEventAck) Channel). A logical representation of the SPU event support and the descriptions of these channels are described in the *Cell Broadband Engine™ Architecture* document. See *Section 5.4 SPU Interrupt Facility in SPU ISA* on page 73 for more information about using indirect branches for entering and returning from the interrupt handler.

An interrupt sequence might go through the following steps:

1. Enable interrupts as described above.
2. The program runs until conditions cause the channel count for the SPU_RdEventStat Channel to transition from 0 to 1; then, the interrupt is taken.
3. Interrupts are acknowledged through internal logic as follows:
    a. Internal logic disables interrupts.
    b. Any blocked channel access is preempted.
    c. The program counter goes to local-storage address 0.
    d. The return address is stored in SRR0.
4. Read the SPU_RdEventStat Channel to see the status of events (the channel count for the SPU_RdEventStat Channel transitions to 0).
5. Acknowledge all events to be processed (write to the SPU_WrEventAck Channel).
6. Process events (for example, mailbox events).
7. Execute an interrupt return (**iret**) instruction, which branches to the return address held in SRR0. The SPU interrupt can be re-enabled using the [E] feature bit of the **iret** instruction.

Nested interrupts are supported through reading the SPU Read State Save-and-Restore (SPU_RdSRR0) Channel and writing the SPU Write State Save-and-Restore (SPU_WrSRR0) Channel.

Notes:

1. The channel count for the SPU_RdEventStat Channel transition from 0 to 1 is delayed several cycles before it can cause an interrupt acknowledgment in stage N of the pipeline. One cycle later there is an instruction-fetch request in the Kf pipeline stage for the handler at address zero. If this fetch successfully arbitrates for the local store on the next cycle, the first instruction of the handler commits 18 cycles after the interrupt acknowledgment in stage N.
2. The value written to the SPU_WrSRR0 Channel is not immediately available to the **iret** instruction. This write must be synchronized by execution of a channel **sync** instruction (with the C bit set in the instruction) before executing an **iret** instruction.

3.1.8. Execution Behavior for Unused or Reserved Opcodes

Synergistic Processor Unit Instruction Set Architecture opcodes are divided into three general classes: valid instructions, opcodes reserved for future performance enhancement, and opcodes reserved for future functional enhancement.
All opcodes not assigned to valid instructions or reserved for future performance enhancement are reserved for future functional enhancement. If an opcode reserved for future functional enhancement is executed, the SPU halts and signals an illegal instruction. The halt does not result in a restartable state. However, the register file and local storage are consistent with execution stopping before the reserved instruction.
When an SPU executes an opcode reserved for future performance enhancement, there is no architectural effect other than advancing the program counter; the SPU does not halt. The opcode is treated as a no-op; it does not modify the local storage, register file, or the channel unit.

The following opcodes are reserved for future performance enhancement:

Temporary Name	Binary Opcode	Temporary Name	Binary Opcode
M_R00	00111000000	M_HBCNZ	00100010101
M_R01	00111000001	M_HBCHZ	00100010110
M_R02	00111000010	M_HBCHNZ	00100010111
M_R03	00111000011	M_R70	01000000000
M_R04	00111010000	M_R72	01000000010
M_R05	00111010001	M_R73	01000000011
M_R06	00111010010		
M_R07	00111010011		
M_R11	00101000101		
M_R12	00101000110		
M_R13	00101000111		
M_R14	00101010100		
M_R15	00101010101		
M_R16	00101010110		
M_R17	00101010111		
M_R20	00111101		
M_R50	0000001		
M_R60	000000010		
M_R80	01101111		
M_R31	00110101101		
M_R32	00110101110		
M_R33	00110101111		
M_HBII	00100101100		
M_R41	00100101101		
M_R42	00100101110		
M_R43	00100101111		
M_HBCZ	00100010100		

3.2. SPU Interface

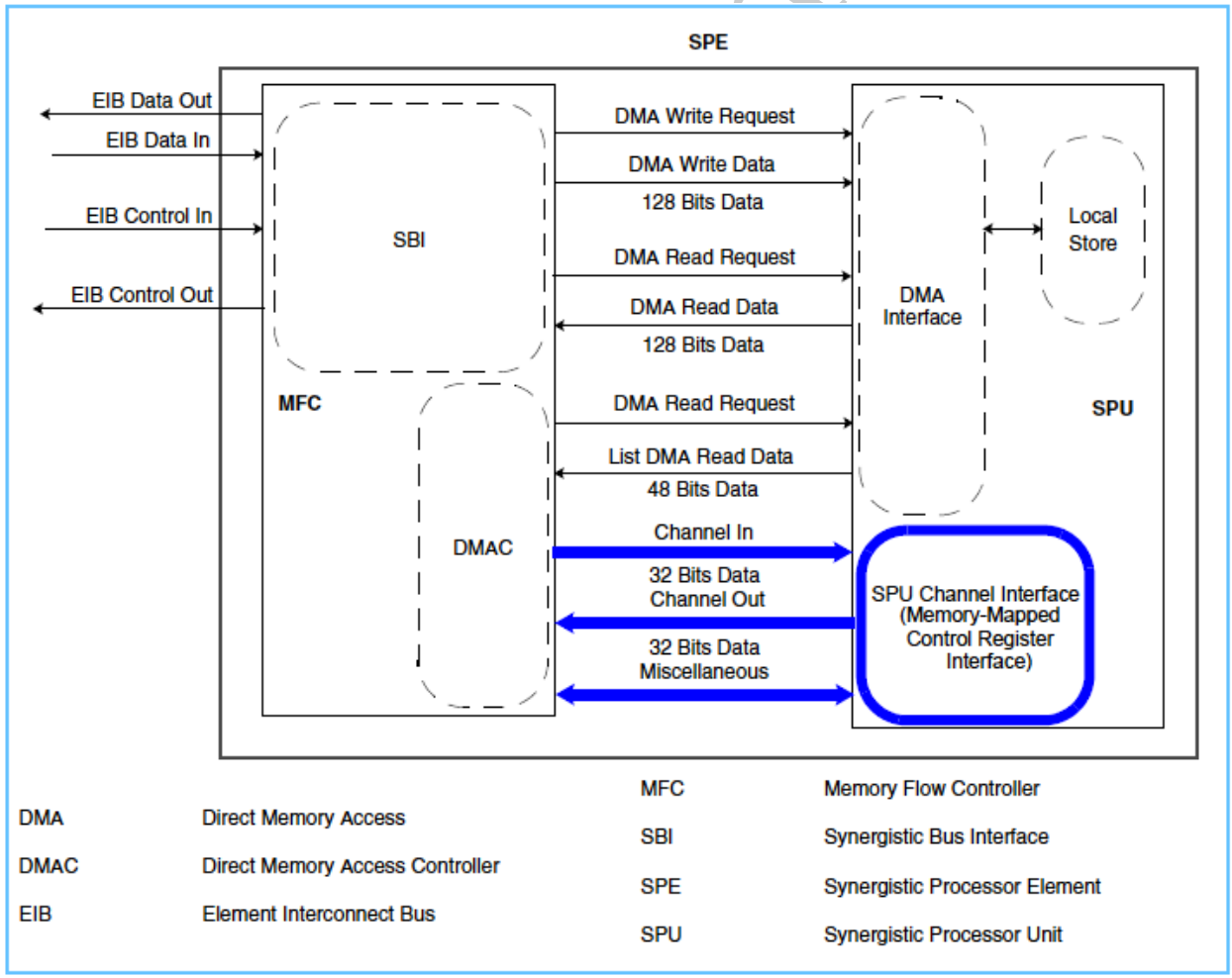
The SPU interface, illustrated in *Figure 3-7*, enables communication, data transfer, and control into and out of the SPU. In the Cell Broadband Engine™ (CBE), the SPU interface communicates with the memory flow controller (MFC).

The SPU interface consists of two parts:

- SPU channel interface. An input and output communication interface used for the following:
  - For interprocessor communication
  - To issue MFC commands and to determine the status of the commands (see *Section 3.3 Synergistic Memory Flow Controller (MFC)* on page 45 for a detailed discussion of MFC commands)
  - For external access to the SPU memory-mapped control registers
- DMA interface. Used to access the local SPU memory known as the local store. The DMA interface has three parts:
  - A DMA write interface to load data into the local store
  - A DMA read interface to read data from the local store
  - A special list DMA interface used for certain DMA commands to read information from the local store

The rest of this section describes the SPU channel interface.

Figure 3-7. Overview of the SPU Interface



The area above marked with the thick blue line is the area of the processor discussed in this section.

3.2.1. SPU Channels

The SPU uses channels for external communication. These channels are accessed by the processor using the channel instructions in *Table 3-15*. The purpose of a channel is determined by the implementation. *Table 6-1* on page 76 shows the channel definitions. The CBE (unidirectional) channels are assigned the following functions:

- Interprocessor-communications mailboxes and signal-notification registers
- MFC command issuance and status
- Decrementer (timer) access
- Event management

Table 3-15. SPU Channel Instruction

Channel Instruction	Mnemonic	Operational Description
Read Channel	**rdch**	Causes a read of data stored in the addressed channel to be loaded into the selected General Purpose Register (GPR).
Write Channel	**wrch**	Causes data to be read from the selected GPR and stored in the addressed channel.
Read Channel Count	**rchcnt**	Causes the count associated with the addressed channel to be stored in the selected GPR.

3.2.2. SPU Channel Bus Operations Overview

SPU channel operations include the following key features:

- All transactions on the channel bus are unidirectional.
- There is no stalling of channel transactions on the channel interface.
- Each channel transaction is independent of any other transaction.
- Back-to-back read and write transactions are supported.
- From the SPU, a read channel acknowledges the commitment of the read channel instruction. This means that the data is not valid, which is indicated by the data valid signal.
- From the SPU, a write channel commits the write channel instruction. The data is valid, which is indicated by the data valid signal.
- External actions to control registers have a higher priority than channel operations.
- Channel operations are done in program order.

3.2.2.1. Read and Write Channel Implementation Difference from the SPU ISA

The *Synergistic Processor Unit Instruction Set Architecture* (SPU ISA) specifies that a read channel (**rdch**) instruction or a write channel (**wrch**) instruction to an invalid channel causes the SPU to stop on or after these instructions. The SPU implementation differs from the SPU ISA in that channel read instructions to reserved channels or valid write channels return zeros, and channel writes to reserved channels or valid read channels have no effect. Read channel counts to reserved channels return zero.

3.2.2.2. SPU Memory-Mapped Access

The SPU memory-mapped registers can be written or read through the SPU-to-MFC channel interface. The SPU can be in any state when the memory-mapped control registers are read: running, stalled, or stopped. Also, there is no restriction against interleaving channel access with memory-mapped control register access on the SPU-to-MFC channel interface. If there is a conflict between a memory-mapped control register read access and the SPU issuing an **rdch** or **wrch** instruction on the SPU-to-MFC channel-out interface, the SPU provides read-access priority to the memory-mapped control register and retries the channel instruction. All transfers across the SPU-to-MFC memory-mapped control register interface take one cycle. There is no cycle-to-cycle dependency for this interface.

3.2.2.3. SPU Channel Ports

The SPU channel interface allows the SPU to communicate externally. All transactions across the SPU-to-MFC channel interface take one cycle. There is no cycle-to-cycle dependency.

The SPU-to-MFC channel-out port is used when the program being executed needs to communicate outside the SPU. The program uses **rdch** and **wrch** instructions. All channel commands issued through the channel- out port occur in strict program order. The channel-in port is used to communicate information into the SPU for access by the **rdch** instruction, or to acknowledge receipt and completion of **wrch** instructions.

Not all channels are accessible through the SPU-to-MFC channel interface. See *Table 6-1* on page 76 for channel definitions. Some channels function only within the SPU and do not output across the channel-out interface when the SPU issues channel instructions. Also, these channels are not accessible using the channel-in interface.

Write Channel

When the SPU needs to send information out, it uses a **wrch** instruction.

If the channel is defined as a blocking channel, the SPU does not issue a **wrch** instruction unless the count associated with that channel is greater than zero. When a channel is defined as blocking, a count of zero means that the channel is full, and the SPU stalls until the PPE has read at least one entry from the equivalent MMIO register. The program determines the channel count by either of two methods. It can poll the channel count for that register using the read channel count (**rchcnt**) instruction, or it can issue a **wrch** instruction. If the program issues a **wrch** instruction, the SPU stalls, waiting until an acknowledgement is received from the write channel.

If the write channel is defined as a nonblocking channel, then the **wrch** instruction is issued independently of the value of the channel count associated with that channel.

A write channel operation stores 32 bits of data from the preferred slot (bits [0:31] of the 128-bit register) of the SPU GPR specified by the instruction. See *Section 4.2 Register Layout* on page 59 for a more complete definition of the preferred slot in the register layout.

Read Channel

When the SPU program needs to receive information, it uses an **rdch** instruction. Typically, there is a register inside the SPU that holds this information. The information can be loaded into this register through the channel-in interface using a read-data-load transaction.

In the SPU, if the channel is a blocking channel, the SPU processor does not read from this register until the channel count for that register indicates that the data is valid (that is, when the count is greater than zero). If the count is zero, then there is no data in the channel and the SPU stalls until actions associated with that channel occur. These actions can include the updating of the MFC_RdTagStat Channel; the PPE writing data to the corresponding MMIO register, such as a mailbox channel; or other actions. The program determines the channel count by either of two methods. It can poll the channel count for that register using the **rchcnt** instruction, or it can issue the **rdch** instruction. If the program issues an **rdch** instruction, the SPU stalls, waiting until valid data is loaded.

A read channel operation loads 32 bits of data into the preferred slot (bits [0:31] of the 128-bit register) of the SPU GPR specified by the instruction.

Read Channel Count

Certain SPU channels have an implementation-specific count associated with them. This count decrements whenever a channel instruction is issued to a specific channel, and the count increments whenever an action associated with that channel occurs. If the count is zero for a particular channel, and that channel is

configured as a blocking channel, then a channel instruction issued to that channel causes the SPU to stall (that is, to not complete the channel instruction), and to stop issuing additional instructions until an action associated with that channel occurs for that channel. Channel counts for channels with a count can be accessed through the **rchcnt** instruction. For channels implemented without a count, the return value is one. The channel count can be initialized externally through writes to channel-count-configuration memory-mapped registers. See *Cell Broadband Engine™ Registers* for more information.

3.2.2.4. Channel Performance Note

All SPU channel instructions (the **rdch**, **wrch**, or **rchcnt** instructions described in *Table 3-15* on page 42) that are issued within five SPU cycles to the same channel, or to the SPU_RdEventStat Channel, the SPU_WrEventMask Channel, or the SPU_WrEventAck Channel, cause a retry of the second channel instruction. This may significantly impact performance.

For example, a **wrch** instruction to the MFC_Cmd / MFC_ClassID Channel followed within five cycles (5–10 instructions) by a **rchcnt** instruction to the MFC_Cmd / MFC_ClassID Channel may cause the SPU to retry the **rchcnt** instruction. This may delay completion of the **rchcnt** instruction by the same delay as a branch mispredict.
If the branch indirect and set link if external data (**bisled**) instruction occurs within eight cycles after a **rdch**, **wrch**, or **rchcnt** instruction, completion of the **bisled** instruction is delayed by the same delay as a branch mispredict.
See *Section 3.1.1 SPU Pipeline* on page 11 for more information about the branch mispredict.

3.3. Synergistic Memory Flow Controller (MFC)

The Synergistic Processor Element (SPE) consists of the Synergistic Processor Unit (SPU) and the Synergistic Memory Flow Controller (MFC). This section provides details of the MFC.

3.3.1. Overview

The MFC, as illustrated in *Figure 3-8. SPE Block Diagram* on page 46, provides two main functions for the SPU:

- The MFC moves data between SPU Local Storage (LS) and the main storage.
- The MFC provides synchronization mechanisms between the SPU and the rest of the processing units in the system.

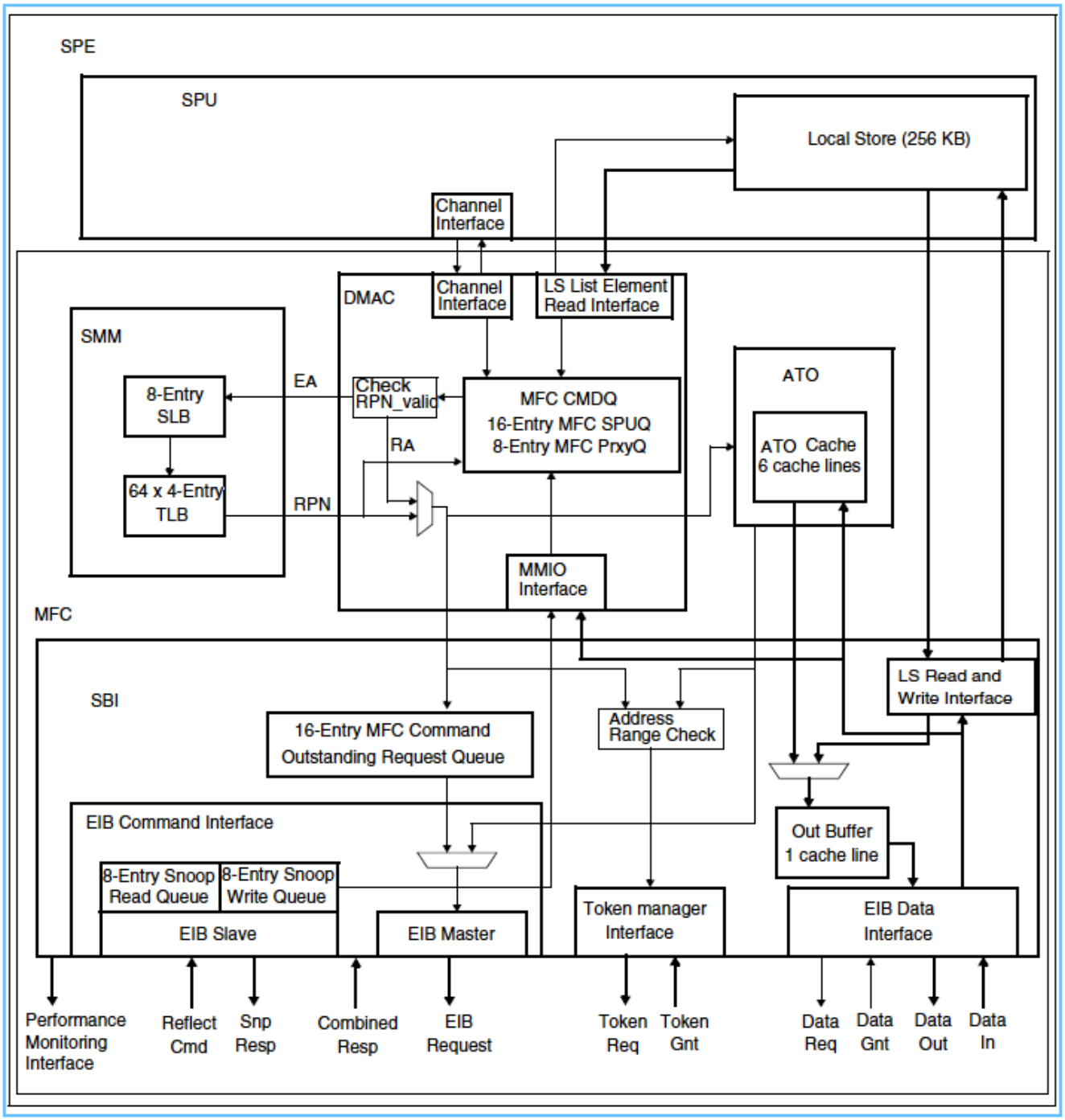
The MFC has four functional units described in detail in this section:

- Direct Memory Access Controller (DMAC)
  The DMAC maintains and processes the MFC command queues (MFC CMDQ), which consist of an MFC SPU Command queue (MFC SPUQ) and an MFC proxy command queue (MFC PrxyQ). The 16-entry MFC SPUQ contains MFC commands that come from the SPU channel interface. The 8-entry MFC PrxyQ contains MFC commands that come from other processing units through Memory Mapped Input and Output (MMIO) load and store.
  The DMAC handles all MFC commands, including the MFC DMA command that moves data between the LS and the main storage. The main storage is addressed by the Effective Address (EA) MFC command operand. The LS is addressed by the Local Storage Address (LSA) MFC command operand.
- Synergistic Memory Management (SMM) unit
  When in relocate mode (MFC_SR1[R] equals 1), the SMM unit provides address translation and memory protection facilities to handle EA translation requests from the DMAC and sends back the translated 4-KB Real Address Page Number (RPN). The SMM maintains a Segment Lookaside Buffer (SLB) and a Translation Lookaside Buffer (TLB). The SLB translates an EA to a Virtual Address (VA), and the TLB translates the VA coming out of the SLB to a Real Address (RA).
- Atomic (ATO) unit
  The ATO unit maintains a 6-cache-line buffer. The ATO dedicates four of six cache lines to provide synchronization functions to other processing units in the system by processing MFC atomic command requests from the DMAC and by maintaining cache coherency when it receives snoop requests forwarded from the SBI unit. One of the remaining cache lines is dedicated to supporting page table entry (PTE) tablewalks or Reference or Change bit (RC) update requests from the SMM. The last cache line is dedicated to supporting the castout operation.
- Synergistic Bus Interface (SBI) unit
  The SBI unit provides the Element Interconnect Bus (EIB) interface to the SPE. The SBI accepts data transfer and memory coherency requests from the DMAC and the ATO units. It snoops and decodes EIB requests and forwards them to the ATO, SMM, DMAC, and SPU. The SBI also provides a direct interface to read and write data to the LS of the SPU.

Currently, with the exception of the Channel Interface, the MFC runs at half (NClk/2) of the SPU frequency (NClk). The Channel Interface runs at the same frequency as the SPU to convert the interface signals between the MFC and SPU clock domains.

See *Figure 3-8 SPE Block Diagram* on page 46 for detailed information.

Figure 3-8. SPE Block Diagram



3.3.2. Direct Memory Access Controller (DMAC)

The primary function of the direct memory access controller (DMAC) is to manage MFC commands from the PPU and the SPU. MFC commands include DMA commands that transfer data to and from the local storage (LS), MFC parameter commands that control the set up of MFC operations, storage control commands to provide storage control for the SPU Level 1 cache (SL1), and synchronization commands.

- DMA Data Transfer Commands
  Fifteen types of **put** commands move data from local storage to main storage. Nine types of **get** commands move data into local storage from main storage.
  See *MFC DMA Data-Transfer Commands* on page 87 for more information on these commands.
- Storage Control Commands
  Storage control commands manage the SL1 cache. SL1 is a first-level cache for DMA transfers between local storage and main storage. The five SL1 commands are **sdcrt**, **sdcrtst**, **sdcrz**, **sdcrst**, and **sdcrf**. The **sdcrt** and **sdcrtst** commands are implemented as no operations (no-ops) because this implementation has no SL1. However, **sdcrz**, **sdcrst**, and **sdcrf** commands are issued to the SBI unit because they affect the Atomic cache.
  See *SL1 Storage Control Commands* on page 88 for more information on these commands.
- Synchronization Commands
  Synchronization commands include four Atomic commands, three send signal commands, and three barrier commands. Three Atomic commands (**getllar**, **putllc**, and **putlluc**) are decoded and issued immediately to the ATO unit. One Atomic command, **putqlluc**, is queued and issued to the ATO unit following the rules for DMA Issue mechanism. The three send signal commands (**sndsig**, **sndsigf**, and **sndsigb**) are implemented in the same manner as the **put** commands. The dedicated barrier commands are **mfcsync**, **mfceieio**, and **barrier**.
  See *MFC Atomic Commands* and *MFC Synchronization Commands* on page 88 for more information on these commands.

3.3.2.1. Receiving MFC Commands

MFC commands originate from instructions in the SPU or the PPU.
Commands from the SPU are transferred to the DMAC by using the MFC Command Parameter channels. After a complete MFC command is received through the channel, it is partially decoded and placed in the MFC SPU command queue (MFC SPUQ) and issue logic.
MFC commands from the PPU are transferred to the DMAC by using the MMIO interface. After a complete MFC command is received through the MMIO interface, it is partially decoded and placed in the MFC proxy command queue (MFC PrxyQ) and issue logic.
The 16-entry MFC SPUQ and the 8-entry MFC PrxyQ are collectively known as the MFC command queue (MFC CMDQ). There are no dependency relationships between the commands in the MFC SPUQ and MFC PrxyQ.

3.3.2.2. Status of Tag Groups Containing DMA Commands

All MFC commands except **getllar**, **putllc**, and **putlluc** are associated with a tag group. By assigning a command or group of commands to different tag groups, the status of the entire tag group can be determined.
The SPU uses the MFC_WrTagMask Channel, the MFC_WrTagUpdate Channel, and the MFC_RdTagStat Channel to query the MFC SPUQ tag groups, while the PPU uses MMIO to query MFC PrxyQ tag groups. The MFC SPUQ has tag groups 0 to 31, which are different from the MFC PrxyQ tag groups 0 to 31.
When one or more commands belonging to an MFC SPUQ tag group are queued in the DMAC, the status of that MFC SPUQ tag group is not empty. When all commands belonging to a tag group in the MFC SPUQ are processed and completed, the status of that MFC SPUQ tag group becomes empty.
When one or more commands belonging to an MFC PrxyQ tag group are queued in the DMAC, the status of that MFC PrxyQ tag group is not empty. When all commands belonging to a tag group in the MFC PrxyQ are processed and completed, the status of that MFC Prxy tag group becomes empty.

3.3.2.3. MFC Commands with Tag-Group Dependencies

The issue mechanism tracks tag-group dependencies between commands in the MFC SPUQ and MFC PrxyQ separately. These dependencies are determined by the opcode of the command when it is received. Dependencies on prior commands are cleared as each prior command is completed.

The **get**, **put**, and **sndsig** commands with f (fence) or b (barrier) modifier bits create tag-group-specific dependency for the command against other commands in the same tag group in the same queue (MFC SPUQ or MFC PrxyQ).

The **putqlluc** command has a tag-group-specific dependency against other commands in the MFC SPUQ and a non-tag specific dependency against other **putqlluc** commands in the MFC SPUQ. (In the *Cell Broadband Engine™ Architecture*, **putqlluc** has an implied tag-specific fence only; this implementation also adds a fence against all other **putqlluc** commands.)

In the current implementation, both **mfceieio** and **mfcsync** are treated as not tag-specific. (In the *Cell Broadband Engine™ Architecture*, both the **mfceieio** and **mfcsync** commands are defined as tag-specific **barrier** commands, and the MFC **barrier** command is defined as not tag-specific.)

3.3.2.4. MFC Command Issue

The DMAC supports out-of-order execution of independent MFC commands. MFC commands that have no dependencies are eligible to be issued. Each MFC command is assigned to either Slot 0 or Slot 1 in the issue mechanism. Slot 0 commands are all **put**, **sndsig**, **putqlluc**, **sdcrz**, **sdcrst**, and **sdcrf** commands. Slot 1 commands are all **get**, **sdcrt**, **sdcrtst**, **barrier**, **mfcsync**, and **mfceieio** commands. The issue mechanism alternates issuing commands in Slot 0 and Slot 1 every other cycle (at the EIB frequency, or half the SPU frequency). When multiple commands in the Slot are eligible, the oldest command in the MFC SPUQ has priority; if no commands in the MFC SPUQ are eligible, then the oldest eligible command in the MFC PrxyQ is selected. Only 16 requests to the SBI can be outstanding.

Issuing an MFC command creates a request. In real-mode, requests are sent to the SBI or the ATO. When translation mode is turned on, a request can go through the SMM for translation first, if needed, before being sent to the SBI or the ATO. Some commands complete in the DMAC without a request to the SBI or the ATO, including **barrier**, **sdcrt**, **sdcrtst**, **get**, **put**, and SL1 commands with a Transfer Size of 0 bytes. Enabling Transfer Class ID adds additional arbitration rules to the issue mechanism.

As soon as the **mfcsync** or **mfceieio** command is issued to the SBI, the DMAC issue mechanism stops. The DMAC issue mechanism resumes when **mfcsync** or **mfceieio** is completed by the SBI.

Atomic commands **getllar**, **putllc**, and **putlluc** are immediately issued to the ATO unit in either a Slot 0 or Slot 1 cycle. If the command requires address translation and the translation is unsuccessful, the command waits in the queue until the translation condition is resolved in the SMM and the DMAC is notified. Then the Atomic command is reissued immediately. The queued Atomic command **putqlluc** must wait in the MFC SPUQ until it is free of dependencies. The ATO unit can have one immediate form command outstanding and one queued form command outstanding.

When the DMAC issue mechanism is stopped due to **mfceieio** or **mfcsync** command, any new MFC atomic command received by the MFC SPUQ is held in the queue. However, the MFC atomic command is issued as soon as the **mfceieio** or **mfcsync** command is completed.

DMA data transfer commands that span multiple cache lines generate multiple bus requests through the process of unrolling. Each bus request for the command points to the next cache line or partial cache line in the data block. After each unrolling of the command, the transfer size, local-storage address, and effective address are updated in the MFC CMDQ to point to the start of the next unrolling of the data block.

Performance of Quadword-Aligned Data Transfers

The performance of a DMA data transfer is best when the source and destination addresses have the same quadword offsets within a 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 in the middle of a cache line transfer a partial cache line (less than 8 quadwords) in the first bus request; transfers that end in the middle of a cache line transfer a partial cache line (less than 8 quadwords) in the last bus request.

Performance of Quadword-Unaligned Data Transfers

The performance of a DMA data transfer when the source and destination addresses have different quadword offsets within a cache line is about half that of transfers that are quadword offset-aligned. Every bus request is a partial cache-line transfer; in effect, there are two bus requests for each cache line of data.

For example, assume the quadword offset for the source address is **qw1**, the quadword offset for the destination is **qw2**, and the remaining transfer size in number of quadwords is **rqw**. The transfer size of the unrolled bus request is **tqw = min** (**8–qw1**, **8–qw2**, **rqw**). After the unrolling, the **qw1** is updated to **qw1=mod**$_8$(**qw1 +tqw**), the **qw2** is updated to **qw2=mod**$_8$(**qw2 +tqw**), and the **rqw** is updated to **rqw=rqw–tqw**. The updated values are then written back to the MFC CMDQ.

3.3.2.5. Transfer Class ID

The Cell Broadband Engine™ (CBE) supports the Transfer Class ID for MFC commands. The Transfer Class ID manages request streams to system storages with different characteristics. For more information, see the MFC Transfer Class ID (MFC_TClassID) Register section in the *Cell Broadband Engine™ Registers* document.

TClassID Disabled

By default, Transfer Class ID is disabled and the issue mechanism does not use the Transfer Class ID. However, TClassID0 Issue Quota (IQ0) and Slot Alternation (SA0) in MFC_TClassID Register affect all commands in the DMAC. IQ0 is 16, by default, to match the depth of the SBI Queue. Lowering the value of IQ0 reduces the maximum number of outstanding DMAC requests to the SBI. SA0 is 0, by default, to enable the issue mechanism to alternate issues between commands in Slot 0 and Slot 1. Setting SA0 to 1 disables this alternation of slots so that all MFC commands are issued only in Slot 0 every fourth cycle (at the EIB frequency).

TClassID Enabled

Performance of the DMAC can be improved by enabling TClassID in MFC_TClassID Register to manage request streams to system storages with different characteristics, such as on-chip storage, off-chip memory, and I/Os.

When TClassID is enabled, the issue mechanism enables round-robin selection between TClassID0-, TClassID1-, and TClassID2-eligible MFC commands for both Slot0 and Slot1. Slot 0 and Slot 1 have separate round-robin tokens that track the class of the last command issued in that slot. When Slot Alternation is disabled for a class, that class is skipped by the Slot 1 round-robin mechanism, and all entries are issued by the Slot 0 round-robin mechanism.

Each TClassID has a quota (IQ0, IQ1, and IQ2) for the number of outstanding requests to the SBI. The sum of these quotas should not exceed 16. The quota for each class prevents the issue mechanism from selecting an MFC command in that class when the quota is reached. When the number of outstanding requests for a class is below the quota for that class, the issue mechanism can select eligible commands from the class.

Alternation of issues between Slot 0 and Slot 1 can be enabled or disabled separately for each TClassID using SA0, SA1, and SA2. Enabling Slot Alternation for a class allows commands from that class to issue alternately in Slot 0 and Slot 1 every other cycle (at the EIB frequency). Disabling Slot Alternation for a class forces commands from that class to issue only in Slot 0 every 4th cycle (at the EIB frequency).

The suggested use of Transfer Class ID 0 is for DMA commands that bypass the token request, such as local storage (LS) to LS (EA translated to LS address) transfer or on-chip MMIO transfer with Slot Alternation enabled (SA0 equals 0) to allow **put** and **get** commands to execute at the same time.

The suggested use of Transfer Class ID 1 is for off-chip memory-access DMA commands with Slot Alternation disabled (SA1 equals 1) to reduce the penalties associated with switching the direction of the bi-directional memory interface.

The suggested use of Transfer Class ID 2 is for I/O-access DMA commands with Slot Alternation enabled (SA2 equals 0) to allow **put** and **get** commands to execute at the same time.

3.3.2.6. SMM Interface

The DMAC makes address translation requests to the SMM for MFC commands when translation mode is turned on. The Effective Address (EA) of the MFC command is sent to the SMM for translation and the SMM returns either the Real Page Number (RPN) if successful or a miss if unsuccessful. The DMAC stores the RPN in the corresponding MFC CMDQ entry and reuses this RPN for each unrolling of the MFC command on the same 4-KB page. When the EA of the unrolled MFC command crosses a 4-KB page boundary, another translation request is made to the SMM to get a new RPN to store in the MFC CMDQ for the next page.
The DMAC sets the MMU dependency bit for the corresponding entry in the MFC CMDQ for a translation miss. MFC CMDQ entries with MMU dependency bits set are blocked from issue until the SMM sends a miss clear to the DMAC that resets all MMU dependency bits. The SMM sends a miss clear when it resolves an outstanding miss or when software sends the MFC_CNTL Restart bit. (For details of the MFC_CNTL Register, see the MFC Privileged Facilities section of the *Cell Broadband Engine™ Architecture* document.)

3.3.2.7. DMAC Error Handling

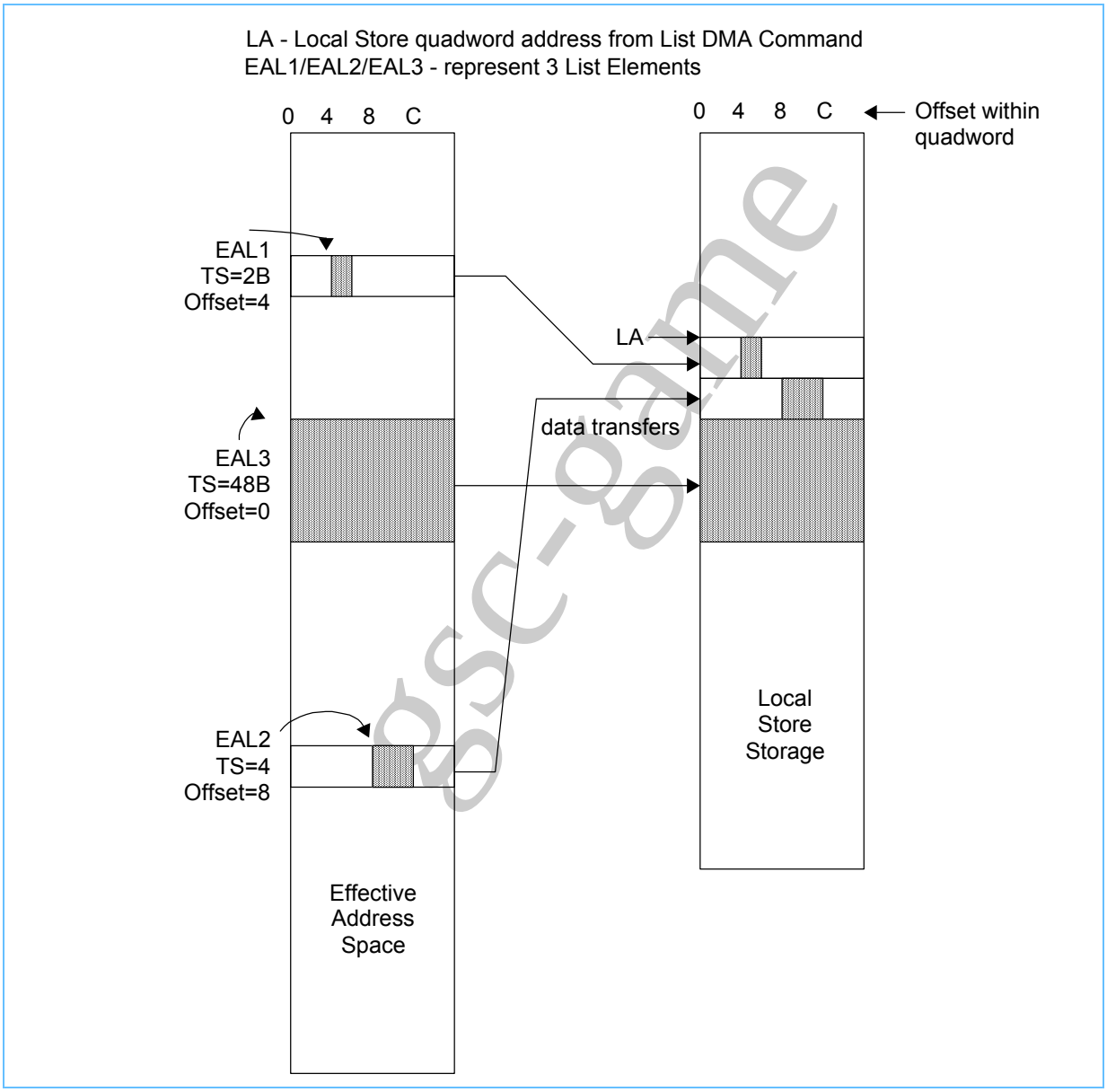
MFC command and alignment errors causes the DMAC to suspend by setting the Sc bit in the MFC_CNTL Register and generating a Class 0 interrupt.

3.3.2.8. List DMA Command Address Alignment

The **getl** and **putl** types of DMA commands are used to gather data from and scatter data to the local storage. The data in the local storage is accessed sequentially with a minimum step of one quadword. Therefore, even when a list element contains a Transfer Size of 1, 2, 4, or 8 bytes, the local storage address increments by 16 bytes to start the next list element on a quadword boundary. For transfers of multiple quadwords, the local storage address increments by the same amount.

DMA list element transfers cannot cross the 4-GB area defined by the EAH. If a DMA list element contains an effective address and a transfer size that would result in violation of this rule, DMA is halted at the 4-GB boundary, and an MFC exception is signaled to the PPE.

Figure 3-9. List DMA Command Address Alignment Diagram



In this example of List DMA Command, the first List Element moves 2 bytes of data from EAL1 to LA+4, the second List Element moves 4 bytes of data from EAL2 to LA+16+8, and the third List Element moves 3 quadwords of data from EAL3 to LA+16+16.

The 4 LSBs of the LA are ignored and replaced with the 4 LSBs of the EAL in all operations.

3.3.2.9. DMA Command Completion

When a DMA command from the SPU is complete, an acknowledgement is sent to the SPU on the MFC_Cmd / MFC_ClassID Channel. When a DMA command from the PPU is complete, the MFC_QStatus Register is updated. Command completion also updates Tag-Group Status if the command's tag group is empty.

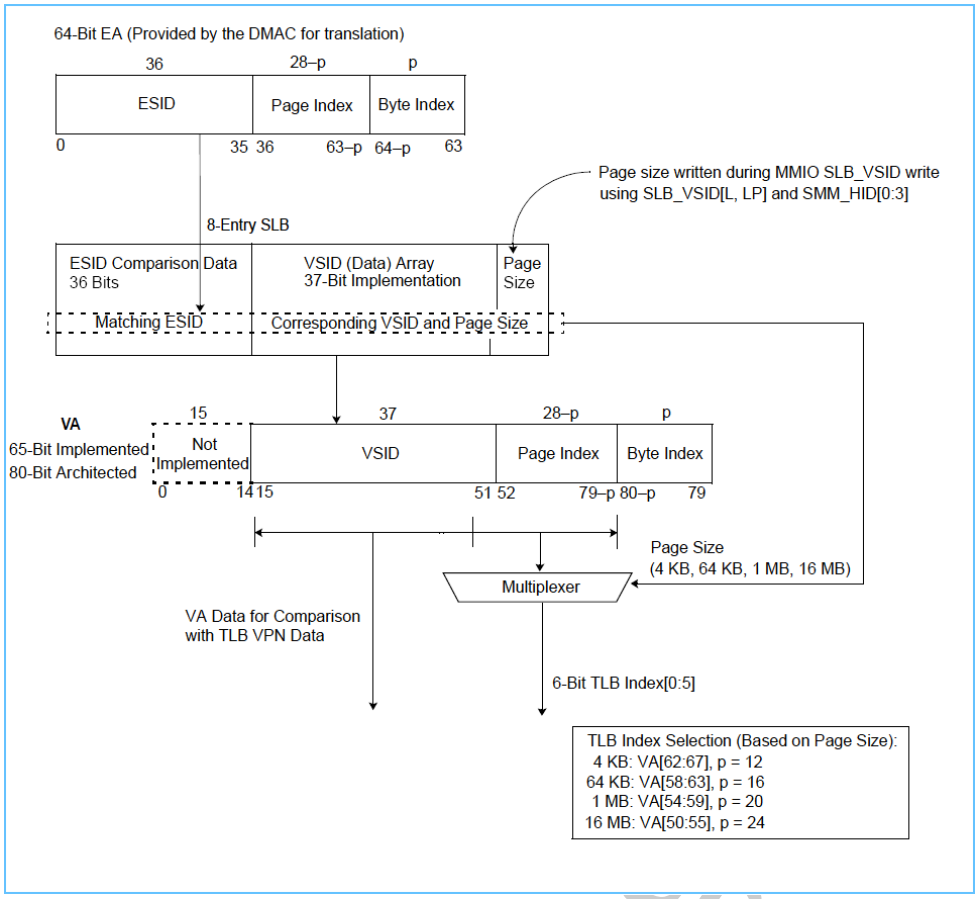
3.3.3. Synergistic Memory Management (SMM) Unit

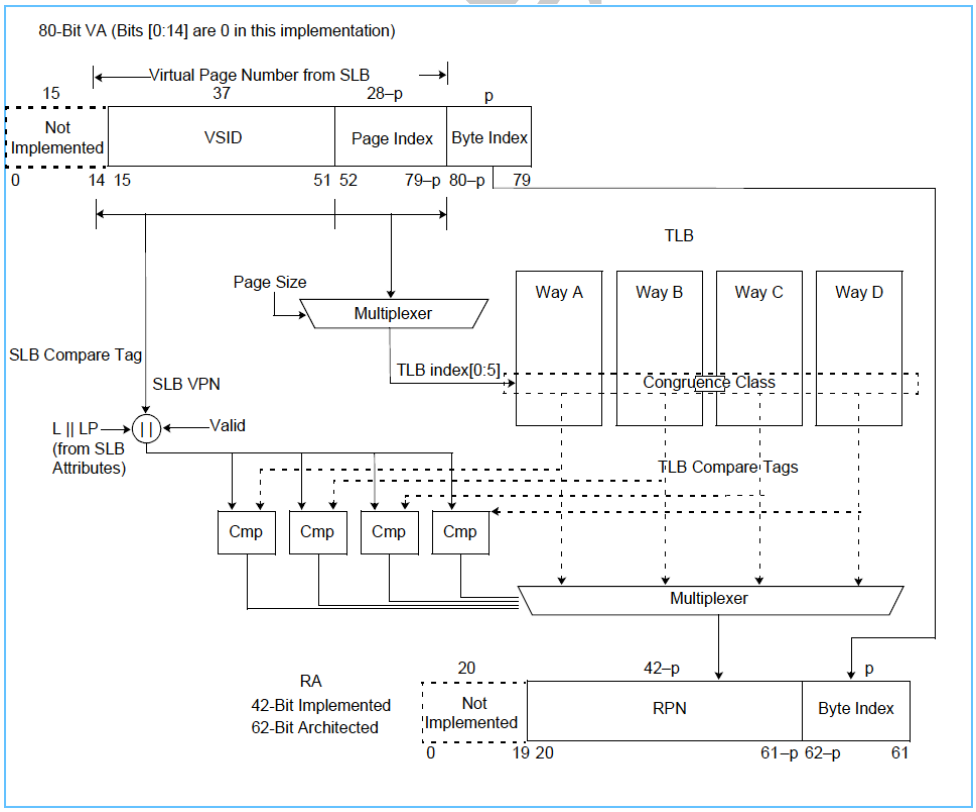
The primary function of the SMM is to translate a MFC effective address to a real address for memory access in virtual mode addressing. The SMM is a 64-bit memory management model in an SPE partition of the CBE chip. It is based on the PowerPC memory management model and is very similar to the MMU unit in the CBE PPU.

In virtual mode, the SMM unit address translation and memory protection facilities handle EA translation requests from the DMAC, and then send back the translated Real Address (RA). The SMM maintains a Segment Lookaside Buffer (SLB) and a Translation Lookaside Buffer (TLB). The SLB translates an EA to a Virtual Address (VA), and the TLB translates this VA to a Real Address (RA). The 8-entry SLB is mapped into MMIO space and the 256-entry, four-way set-associative Translation Lookaside Buffer (TLB) is loaded by the hardware tablewalk machine and optionally handled by software TLB miss management. There is also a 4-entry Replacement Management Table (RMT) for the TLB (used for hardware TLB miss handling) with four set enables.

- All arrays are invalidated on power-up.
- Address translation (PowerPC architecture)
  - 64-bit effective address (EA)
  - 65-bit virtual address (VA)
  - 42-bit real address (RA)
  - Pipelined, hit-under-miss supported
- SLB
  - 8 fully associative entries
  - Software management via MMIO: SLB Index (W), ESID (CAM) (R/W), VSID (Data) (R/W), SLBIE (W), and SLBIA(W) registers
  - Parity protected data array
  - No multiple-hit detection on content addressable memory (CAM) array
- TLB
  - 256 total entries
  - 4-way set-associative
  - Hardware tablewalk (can be disabled via MFC_SR1[TL] bit)
  - Software management via MMIO: TLB Index (W), VPN (R/W), RPN (R/W), and TLBIE (W) registers.
  - Full parity protection
  - Pseudo-least recently used (LRU) default replacement policy. The binary tree pseudo-LRU is 64 entries x 3 bits.
  - The RMT (4 classes x 4 bits) table allows software to override the LRU algorithm.
- Small and large page support: There are three concurrent page sizes: one small (4KB) and two large pages (64KB, 1MB, or 16MB)
- Reference- and change-bit management
- Coherency maintained via snooped **tlbie** (through ATO and SBI units)
- Interrupt handling
  - Segment fault
  - SLB/TLB parity errors on DMA lookups or MMIO reads
  - Atomic cache inhibited access (real and virtual address modes)
  - Page fault (software and hardware tablewalk modes)
  - Data address compare (DAC)
  - Page protection violation
- Performance and Trace Monitors

Figure 3-10. Effective-to-Virtual-Address Mapping and Virtual-to-Real-Address Mapping





3.3.4. Atomic (ATO) Unit

The Atomic Unit handles semaphore operations for the SPU and provides Page Table Entry (PTE) data to the SMM for hardware tablewalks. The Atomic cache stores four lines of data for semaphore operations and a cache line of page table entry. In addition, one cache line is used for reloading data from cache miss loads like **getllar**. Memory coherency is maintained by supporting snoop operations from other processors.

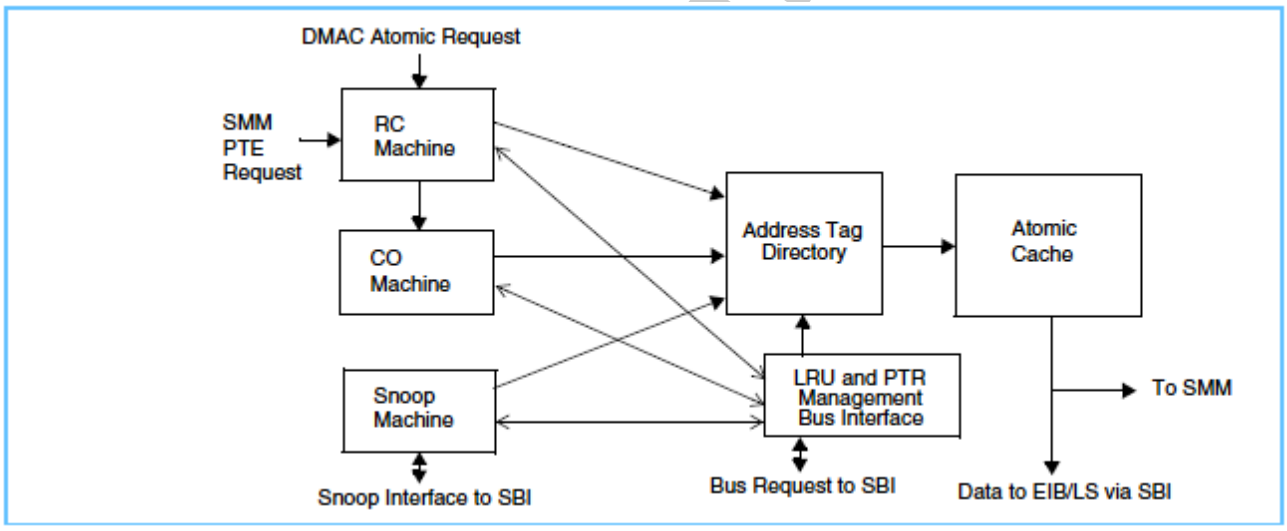
The ATO is a small 6-line cache in the SPE that performs the following functions:

- Provides Atomic operations for the **getllar**, **putllc**, **putlluc**, and **putqlluc** MFC atomic commands
- Provides PTEs to the SMM for a hardware tablewalk, and updates the referenced (R) and changed (C) bits (an RC update)
- Maintains cache coherency by supporting snoop operations

MFC atomic commands (**getllar**, **putllc**, **putlluc**, or **putqlluc**) are issued from the DMAC to the ATO. These requests are executed one at a time. The ATO can have up to two outstanding requests: one immediate form command (**getllar**, **putllc**, or **putlluc**) and one queued form command (**putqlluc**). The SMM's PTE request can occur at the same time as MFC atomic requests. However, the ATO handles this situation by alternating SMM and MFC atomic requests.

SMM PTE requests and MFC atomic requests are executed in the Read and Claim (RC) machine of the ATO. The Castout (CO) machine is used by the RC machine to kick out the least-recently-used (LRU) cache line to memory when space is needed in the cache. The snoop machine handles snoop requests from the bus to maintain memory coherency in the Atomic cache, as well as other cache memory in the system.

Figure 3-11. Atomic Unit Block Diagram



3.3.5. Synergistic Bus Interface (SBI)

The SBI provides an interface between the EIB, the DMAC, and the ATO unit. As the bus master, the SBI receives DMAC and Atomic requests and forwards them to the EIB. The SBI also receives all of the interrupt requests from the rest of the SPE units and forwards these requests to the EIB. As the bus slave, the SBI receives data coherency traffic for the ATO unit, MMIO requests, or local storage access requests from its own or other SPEs and PPE.

The master side of the SBI has a 16-entry, outstanding DMA bus request command queue that holds commands from the DMAC that are sent to the EIB. Each entry represents one EIB transaction. Because the maximum transfer size of the EIB is 128 bytes, DMAC unrolls the transfer into multiple EIB transactions if the transfer size exceeds 128 bytes. See *Section 3.3.2.4 MFC Command Issue* on page 48 for a description of unrolling. Commands from the ATO unit are not enqueued in the outstanding DMA bus request command queue.

After the outstanding DMA bus request command queue is full, the DMAC does not send new commands to the SBI and waits until one of the entries in the outstanding DMA bus request command queue is free. Each entry in the outstanding DMA bus request command queue is freed whenever a **put** operation sends data to the EIB or a **get** operation acquires data from the EIB.

The slave side of the SBI has a read queue and a write queue. These queues have read and write access to MFC MMIO resources (read queue) or Local Storage (write queue).

The read queue has a dedicated entry for MMIO reads and seven entries for local-store reads. Each entry is freed when the requested data is sent to the EIB. If an MMIO read request arrives while the MMIO read entry is occupied, the request is retried. If the local-store read request arrives while all seven entries are occupied, the request is retried.

The write queue has eight entries shared with MMIO write and local-store write. Each entry is freed when data for write arrives. If an MMIO write or local-store write request arrives while all eight entries are occupied, the request is retried.

4. Data Format

The SPU architecture includes 128 registers, each of which contains 128 bits. Registers are used to hold fixed point and floating-point format data. Instructions operate on the full width of the register, treating it as multiple operands of the same format.

The SPU supports halfword (16-bit) and word (32-bit) integers in signed format, and provides limited support for 8-bit unsigned integers. The number representation is 2s complement.

The SPU supports word (32-bit) and doubleword (64-bit) floating point data in IEEE 754 format. However, full single-precision IEEE 754 arithmetic is not implemented.

4.1. Data Representation

4.1.1. Byte Ordering

The architecture defines:

- An 8-bit byte
- A 16-bit halfword
- A 32-bit word
- A 64-bit doubleword
- A 128-bit quadword

Byte ordering defines how the bytes that make up halfwords, words, doublewords, and quadwords are ordered in memory. The SPU supports most significant byte (MSB) ordering. With MSB ordering, also called *Big Endian*, the most significant byte is located in the lowest addressed byte position in a storage unit (byte 0).

The conventions for bit and byte numbering within the various width storage units are shown in the figures listed in the following table:

For a figure that shows...	See...
Bit and Byte Numbering of Halfwords	*Figure 4-1* on page 57
Bit and Byte Numbering of Words	*Figure 4-2* on page 58
Bit and Byte Numbering of Doublewords	*Figure 4-3* on page 58
Bit and Byte Numbering of Quadwords	*Figure 4-4* on page 58
Register Layout of Data Types	*Figure 4-5* on page 59

These conventions apply to integer and floating-point data (where the most significant byte holds the sign and at a minimum the start of the exponent). The figures show byte numbers on the top and bit numbers in the lower corners.

Figure 4-1. Bit and Byte Numbering of Halfwords

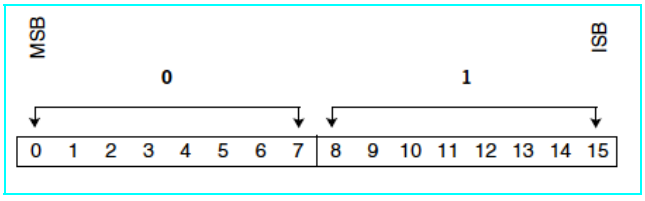


Figure 4-2. Bit and Byte Numbering of Words

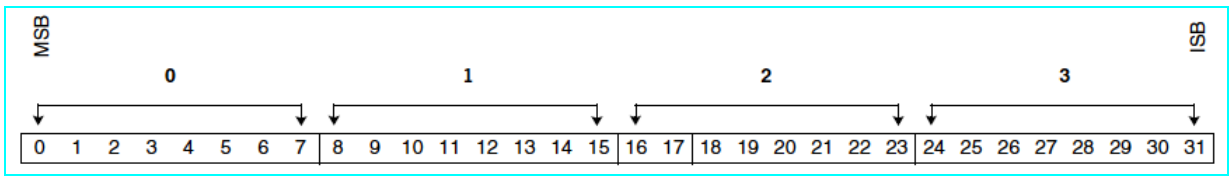


Figure 4-3. Bit and Byte Numbering of Doublewords

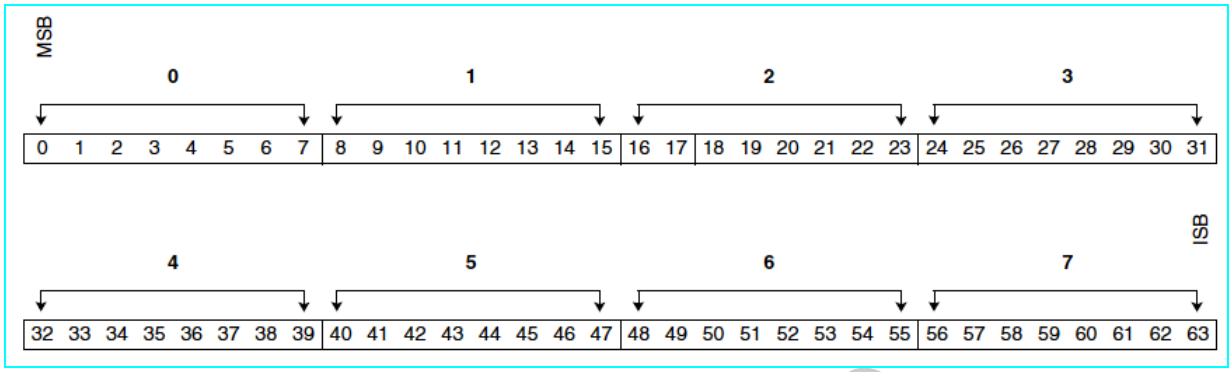
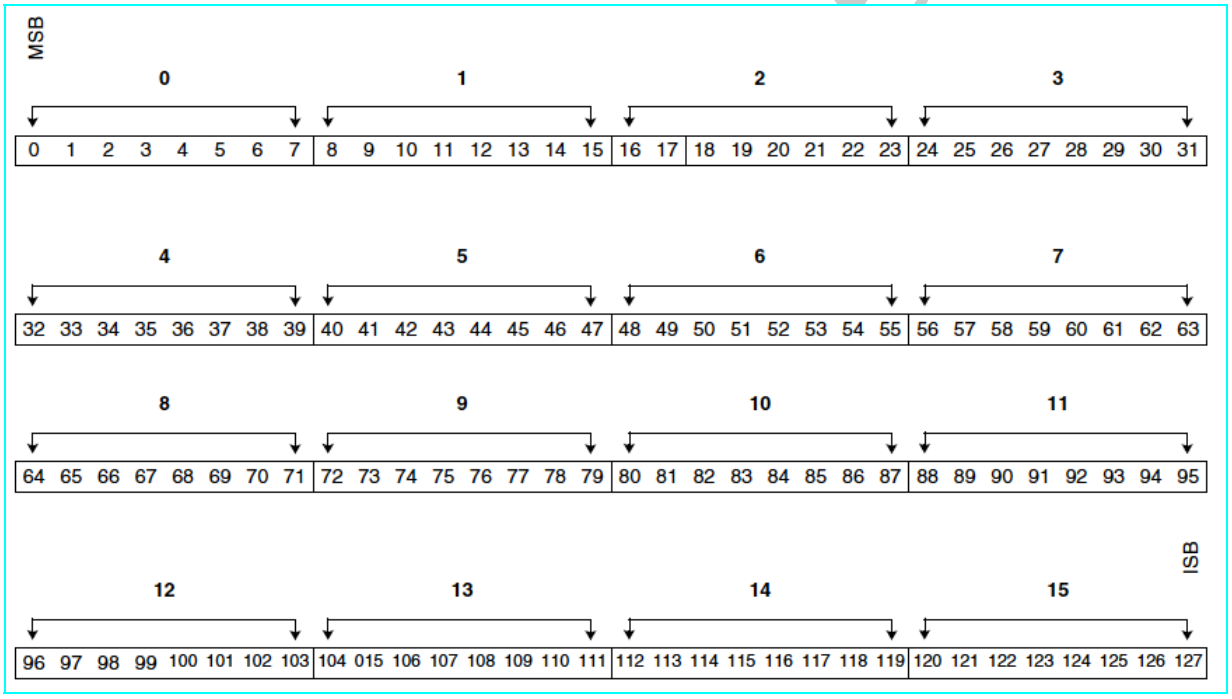


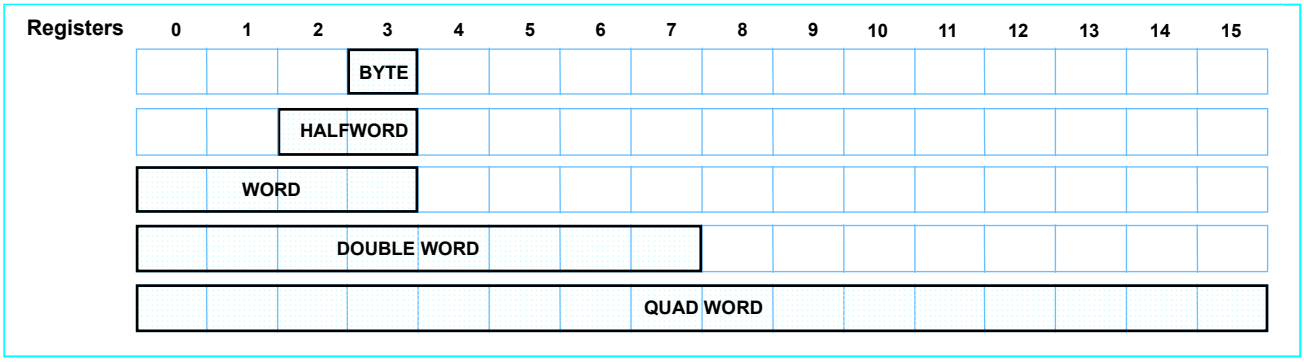
Figure 4-4. Bit and Byte Numbering of Quadwords



4.2. Register Layout

All general purpose registers are 128-bits wide. Data types less than 128-bits are placed within the register in a well defined location called the *preferred slot*. *Figure 4-5* illustrates how data types are laid out in a general purpose register.

Figure 4-5. Register Layout of Data Types



5. Instruction Set Overview

5.1. Instruction Formats

There are six basic instruction formats. These instructions are all 32-bits long. Minor variations of these formats are also used. Instructions in memory must be aligned on word boundaries. The instruction formats are shown in the following figures:

For...	See...
RR Instruction Format	*Figure 5-1* on page 60
RRR Instruction Format	*Figure 5-2* on page 60
RI7 Instruction Format	*Figure 5-3* on page 60
RI10 Instruction Format	*Figure 5-4* on page 60
RI16 Instruction Format	*Figure 5-5* on page 60
RI18 Instruction Format	*Figure 5-6* on page 60

Note: The OP code field is presented throughout this document in binary format.

Figure 5-1. RR Instruction Format

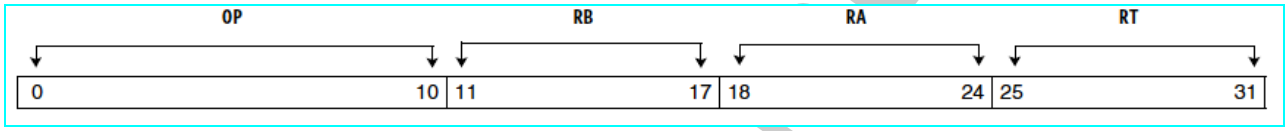


Figure 5-2. RRR Instruction Format

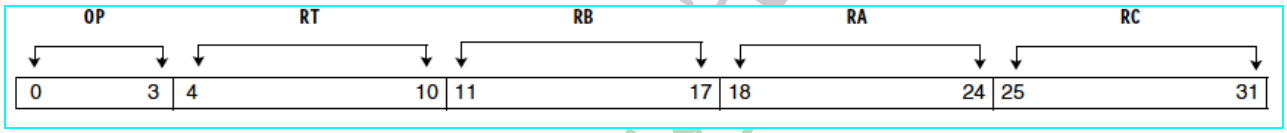


Figure 5-3. RI7 Instruction Format

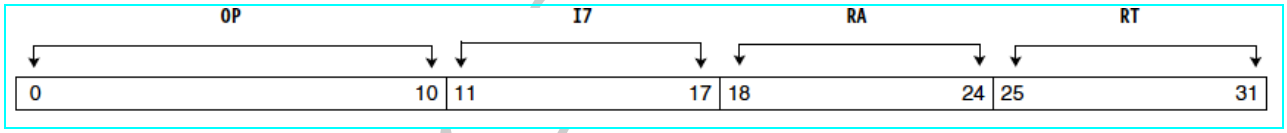


Figure 5-4. RI10 Instruction Format

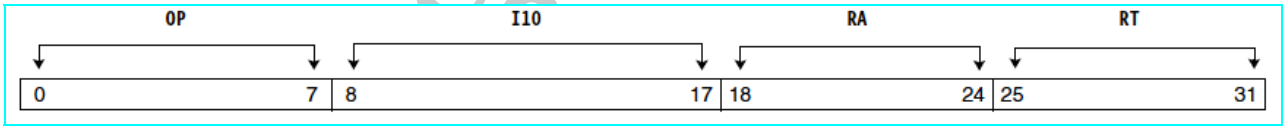


Figure 5-5. RI16 Instruction Format

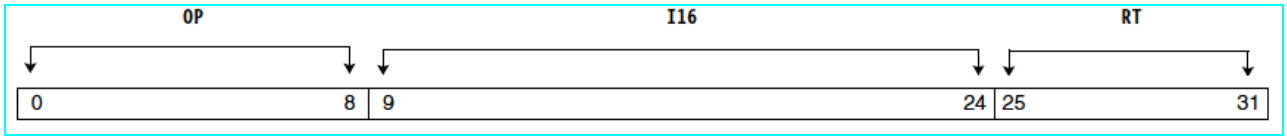
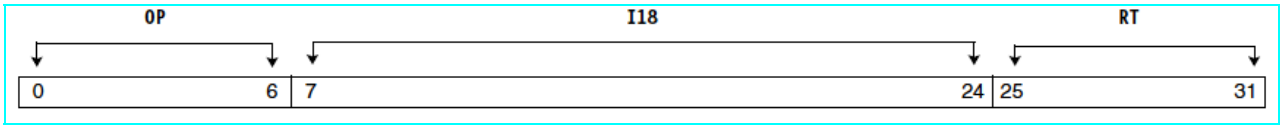


Figure 5-6. RI18 Instruction Format



The leftmost word of a register is called the *preferred slot*. When an instruction references a register as a scalar operand or result, it is always referenced to the preferred slot. For example, when an address is computed and a register is used as a base, the preferred slot of that register is used.

The SPU architecture is big-endian. This means that the most significant bit (the "big end") of an integer or floating point value is stored first, which is to say at the lowest-numbered memory location. Instructions are described in this document as they appear in memory, with successively higher addressed bytes appearing toward the right.

The architecture does not use a condition register. Instead, comparisons set a result which is either 0 (false) or -1 (true), and which is the same width as the operands compared.

A number of additional registers, mapped into system memory, provide a means of controlling some aspects of the architecture, and provide means for checking and handling of error conditions.

Additionally, a set of channels is used to interface with an external device such as the MFC.

Specific instructions are provided for interacting with SPRs and channels.

5.2. Instruction Summary

The following tables are the summary of SPU Instructions. To get the details, please refer *Synergistic Processor Unit Instruction Set Architecture*.

Memory – Load/Store Instructions

Instruction	Description
lqd	Load Quadword (d-form)
lqx	Load Quadword (x-form)
lqa	Load Quadword (a-form)
lqr	Load Quadword Instruction Relative (a-form)
stqd	Store Quadword (d-form)
stqx	Store Quadword (x-form)
stqa	Store Quadword (a-form)
stqr	Store Quadword Instruction Relative (a-form)
cbd	Generate Controls for Byte Insertion (d-form)
cbx	Generate Controls for Byte Insertion (x-form)
chd	Generate Controls for Halfword Insertion (d-form)
chx	Generate Controls for Halfword Insertion (x-form)
cwd	Generate Controls for Word Insertion (d-form)
cwx	Generate Controls for Word Insertion (x-form)
cdd	Generate Controls for Doubleword Insertion (d-form)
cdx	Generate Controls for Doubleword Insertion (x-form)

Constant-Formation Instructions

Instruction	Description
ilh	Immediate Load Halfword
ilhu	Immediate Load Halfword Upper
il	Immediate Load Word
ila	Immediate Load Address
iohl	Immediate Or Halfword Lower
fsmbi	Form Select Mask for Bytes Immediate

Integer and Logic Instructions

Instruction	Description
ah	Add Halfword
ahi	Add Halfword Immediate
a	Add Word
ai	Add Word Immediate
sfh	Subtract From Halfword
sfhi	Subtract From Halfword Immediate
sf	Subtract From Word
sfi	Subtract From Word Immediate
addx	Add Extended
cg	Carry Generate
cgx	Carry Generate Extended
sfx	Subtract From Extended
bg	Borrow Generate
bgx	Borrow Generate Extended
mpy	Multiply
mpyu	Multiply Unsigned
mpyi	Multiply Immediate
mpyui	Multiply Unsigned Immediate
mpya	Multiply and Add

Integer and Logic Instructions

Instruction	Description
mpyh	Multiply High
mpys	Multiply and Shift Right
mpyhh	Multiply High High
mpyhha	Multiply High High and Add
mpyhhu	Multiply High High Unsigned
mpyhhau	Multiply High High Unsigned and Add
clz	Count Leading Zeros
cntb	Count Ones in Bytes
fsmb	Form Select Mask for Bytes
fsmh	Form Select Mask for Halfwords
fsm	Form Select Mask for Words
gbb	Gather Bits from Bytes
gbh	Gather Bits from Halfwords
gb	Gather Bits from Words
avgb	Average Bytes
absdb	Absolute Differences of Bytes
sumb	Sum Bytes into Halfwords
xsbh	Extend Sign Byte to Halfword
xshw	Extend Sign Halfword to Word
xswd	Extend Sign Word to Doubleword
and	And
andc	And with Complement
andbi	And Byte Immediate
andhi	And Halfword Immediate
andi	And Word Immediate
or	Or
orc	Or with Complement
orbi	Or Byte Immediate
orhi	Or Halfword Immediate
ori	Or Word Immediate
xor	Exclusive Or
xorbi	Exclusive Or Byte Immediate
xorhi	Exclusive Or Halfword Immediate
xori	Exclusive Or Word Immediate
nand	Nand
nor	Nor
eqv	Equivalent
selb	Select Bits
shufb	Shuffle Bytes
orx	Or Across (DD 2.0 and later)

Shift and Rotate Instructions

Instruction	Description
shlh	Shift Left Halfword
shlhi	Shift Left Halfword Immediate
shl	Shift Left Word
shli	Shift Left Word Immediate
shlqbi	Shift Left Quadword by Bits
shlqbii	Shift Left Quadword by Bits Immediate
shlqby	Shift Left Quadword by Bytes
shlqbyi	Shift Left Quadword by Bytes Immediate
shlqbybi	Shift Left Quadword by Bytes from Bit Shift Count
roth	Rotate Halfword

Shift and Rotate Instructions

Instruction	Description
rothi	Rotate Halfword Immediate
rot	Rotate Word
roti	Rotate Word Immediate
rotqby	Rotate Quadword by Bytes
rotqbyi	Rotate Quadword by Bytes Immediate
rotqbybi	Rotate Quadword by Bytes from Bit Shift Count
rotqbi	Rotate Quadword by Bits
rotqbii	Rotate Quadword by Bits Immediate
rothm	Rotate and Mask Halfword
rothmi	Rotate and Mask Halfword Immediate
rotm	Rotate and Mask Word
rotmi	Rotate and Mask Word Immediate
rotqmby	Rotate and Mask Quadword by Bytes
rotqmbyi	Rotate and Mask Quadword by Bytes Immediate
rotqmbybi	Rotate and Mask Quadword Bytes from Bit Shift Count
rotqmbi	Rotate and Mask Quadword by Bits
rotqmbii	Rotate and Mask Quadword by Bits Immediate
rotmah	Rotate and Mask Algebraic Halfword
rotmahi	Rotate and Mask Algebraic Halfword Immediate
rotma	Rotate and Mask Algebraic Word
rotmai	Rotate and Mask Algebraic Word Immediate

Compare, Branch and Halt Instructions

Instruction	Description
heq	Halt If Equal
heqi	Halt If Equal Immediate
hgt	Halt If Greater Than
hgti	Halt If Greater Than Immediate
hlgt	Halt If Logically Greater Than
hlgti	Halt If Logically Greater Than Immediate
ceqb	Compare Equal Byte
ceqbi	Compare Equal Byte Immediate
ceqh	Compare Equal Halfword
ceqhi	Compare Equal Halfword Immediate
ceq	Compare Equal Word
ceqi	Compare Equal Word Immediate
cgtb	Compare Greater Than Byte
cgtbi	Compare Greater Than Byte Immediate
cgth	Compare Greater Than Halfword
cgthi	Compare Greater Than Halfword Immediate
cgt	Compare Greater Than Word
cgti	Compare Greater Than Word Immediate
clgtb	Compare Logical Greater Than Byte
clgtbi	Compare Logical Greater Than Byte Immediate
clgth	Compare Logical Greater Than Halfword
clgthi	Compare Logical Greater Than Halfword Immediate
clgt	Compare Logical Greater Than Word
clgti	Compare Logical Greater Than Word Immediate
br	Branch Relative
bra	Branch Absolute
brsl	Branch Relative and Set Link
brasl	Branch Absolute and Set Link
bl	Branch Indirect

Compare, Branch and Halt Instructions

Instruction	Description
iret	Interrupt Return
bisled	Branch Indirect and Set Link if External Data
bisl	Branch Indirect and Set Link
brnz	Branch If Not Zero Word
brz	Branch If Zero Word
brhnz	Branch If Not Zero Halfword
brhz	Branch If Zero Halfword
biz	Branch Indirect If Zero
binz	Branch Indirect If Not Zero
bihz	Branch Indirect If Zero Halfword
bihnz	Branch Indirect If Not Zero Halfword

Branch-Hint Instructions

Instruction	Description
hbr	Hint for Branch (r-form)
hbra	Hint for Branch (a-form)
hbrr	Hint for Branch Relative

Floating-Point Instructions

Instruction	Description
fa	Floating Add
dfa	Double Floating Add
fs	Floating Subtract
dfs	Double Floating Subtract
fm	Floating Multiply
dfm	Double Floating Multiply
fma	Floating Multiply and Add
dfma	Double Floating Multiply and Add
fnms	Floating Negative Multiply and Subtract
dfnms	Double Floating Negative Multiply and Subtract
fms	Floating Multiply and Subtract
dfms	Double Floating Multiply and Subtract
dfnma	Double Floating Negative Multiply and Add
frest	Floating Reciprocal Estimate
frsqest	Floating Reciprocal Absolute Square Root Estimate
fi	Floating Interpolate
csflt	Convert Signed Integer to Floating
cflts	Convert Floating to Signed Integer
cuflt	Convert Unsigned Integer to Floating
cfltu	Convert Floating to Unsigned Integer
frds	Floating Round Double to Single
fesd	Floating Extend Single to Double
fceq	Floating Compare Equal
fcmeq	Floating Compare Magnitude Equal
fcgt	Floating Compare Greater Than
fcmgt	Floating Compare Magnitude Greater Than
fscrwr	Floating Point Status and Control Register Write
fscrrd	Floating-Point Status and Control Register Read

Control Instructions

Instruction	Description
stop	Stop and Signal
stopd	Stop and Signal with Dependencies
lnop	No Operation (Load)
nop	No Operation (Execute)
sync	Synchronize
dsync	Synchronize Data
mfspr	Move from Special Purpose Register
mtspr	Move to Special Purpose Register

Channel Instructions

Instruction	Description
rdch	Read Channel
rchcnt	Read Channel Count
wrch	Write Channel

5.3. Instruction Table Sorted by Opcode

ISA book section	instr. class	pipeline	ISA instr. Name	ISA mnemonic & operands	instruction format			
					opcode	operands		
10	BR	odd	Stop and Signal	stop	0 0 0 0 0 0 0 0 0 0 0	///	Stop_and_Signal_Type	
10	LNOP	odd	No Operation (Load)	lnop	0 0 0 0 0 0 0 0 0 0 1	///	///	RT
10	–		Synchronize	sync	0 0 0 0 0 0 0 0 0 1 0	C / / / / / /	///	///
10	–		Synchronize Data	dsync	0 0 0 0 0 0 0 0 0 1 1	///	///	///
10	CH	odd	Move from Special Purpose Register	mfspr rt, sa	0 0 0 0 0 0 0 1 1 0 0	///	SA	RT
11	CH	odd	Read Channel	rdch rt, ca	0 0 0 0 0 0 0 1 1 0 1	///	CA	RT
11	CH	odd	Read Channel Count	rchcnt rt, ca	0 0 0 0 0 0 0 1 1 1 1	///	CA	RT
5	FX	even	Or Word Immediate	ori rt, ra, value	0 0 0 0 0 1 0 0	I10	RA	RT
5	FX	even	Or Halfword Immediate	orhi rt, ra, value	0 0 0 0 0 1 0 1	I10	RA	RT
5	FX	even	Or Byte Immediate	orbi rt, ra, value	0 0 0 0 0 1 1 0	I10	RA	RT
5	FX	even	Subtract From Word	sf rt, ra, rb	0 0 0 0 1 0 0 0 0 0 0	RB	RA	RT
5	FX	even	Or	or rt, ra, rb	0 0 0 0 1 0 0 0 0 0 1	RB	RA	RT
5	FX	even	Borrow Generate	bg rt, ra, rb	0 0 0 0 1 0 0 0 0 1 0	RB	RA	RT
5	FX	even	Subtract From Halfword	sfh rt, ra, rb	0 0 0 0 1 0 0 1 0 0 0	RB	RA	RT
5	FX	even	Nor	nor rt, ra, rb	0 0 0 0 1 0 0 1 0 0 1	RB	RA	RT
5	BO	even	Absolute Differences of Bytes	absdb rt, ra, rb	0 0 0 0 1 0 1 0 0 1 1	RB	RA	RT
6	WS	even	Rotate Word	rot rt, ra, rb	0 0 0 0 1 0 1 1 0 0 0	RB	RA	RT
6	WS	even	Rotate and Mask Word	rotm rt, ra, rb	0 0 0 0 1 0 1 1 0 0 1	RB	RA	RT
6	WS	even	Rotate and Mask Algebraic Word	rotma rt, ra, rb	0 0 0 0 1 0 1 1 0 1 0	RB	RA	RT
6	WS	even	Shift Left Word	shl rt, ra, rb	0 0 0 0 1 0 1 1 0 1 1	RB	RA	RT
6	WS	even	Rotate Halfword	roth rt, ra, rb	0 0 0 0 1 0 1 1 1 0 0	RB	RA	RT
6	WS	even	Rotate and Mask Halfword	rothm rt, ra, rb	0 0 0 0 1 0 1 1 1 0 1	RB	RA	RT
6	WS	even	Rotate and Mask Algebraic Halfword	rotmah rt, ra, rb	0 0 0 0 1 0 1 1 1 1 0	RB	RA	RT
6	WS	even	Shift Left Halfword	shlh rt, ra, rb	0 0 0 0 1 0 1 1 1 1 1	RB	RA	RT
5	FX	even	Subtract From Word Immediate	sfi rt, ra, value	0 0 0 0 1 1 0 0	I10	RA	RT
5	FX	even	Subtract From Halfword Immediate	sfhi rt, ra, value	0 0 0 0 1 1 0 1	I10	RA	RT
6	WS	even	Rotate Word Immediate	roti rt, ra, value	0 0 0 0 1 1 1 1 0 0 0	I7	RA	RT
6	WS	even	Rotate and Mask Word Immediate	rotmi rt, ra, value	0 0 0 0 1 1 1 1 0 0 1	I7	RA	RT
6	WS	even	Rotate and Mask Algebraic Word Immediate	rotmai rt, ra, value	0 0 0 0 1 1 1 1 0 1 0	I7	RA	RT
6	WS	even	Shift Left Word Immediate	shli rt, ra, value	0 0 0 0 1 1 1 1 0 1 1	I7	RA	RT
6	WS	even	Rotate Halfword Immediate	rothi rt, ra, value	0 0 0 0 1 1 1 1 1 0 0	I7	RA	RT
6	WS	even	Rotate and Mask Halfword Immediate	rothmi rt, ra, value	0 0 0 0 1 1 1 1 1 0 1	I7	RA	RT
6	WS	even	Rotate and Mask Algebraic Halfword Immediate	rotmahi rt, ra, value	0 0 0 0 1 1 1 1 1 1 0	I7	RA	RT
6	WS	even	Shift Left Halfword Immediate	shlhi rt, ra, value	0 0 0 0 1 1 1 1 1 1 1	I7	RA	RT

ISA book section	instr. class	pipeline	ISA instr. Name	ISA mnemonic & operands	instruction format			
					opcode	operands		
8	HB	odd	Hint for Branch (a-form)	hbra brinst,brtarg	0 0 0 1 0 0 0 ROH	I16		ROL
8	HB	odd	Hint for Branch Relative	hbrr brinst,brtarg	0 0 0 1 0 0 1 ROH	I16		ROL
5	FX	even	And Word Immediate	andi rt,ra,value	0 0 0 1 0 1 0 0	I10	RA	RT
5	FX	even	And Halfword Immediate	andhi rt,ra,value	0 0 0 1 0 1 0 1	I10	RA	RT
5	FX	even	And Byte Immediate	andbi rt,ra,value	0 0 0 1 0 1 1 0	I10	RA	RT
5	FX	even	Add Word	a rt,ra,rb	0 0 0 1 1 0 0 0 0 0 0	RB	RA	RT
5	FX	even	And	and rt,ra,rb	0 0 0 1 1 0 0 0 0 0 1	RB	RA	RT
5	FX	even	Carry Generate	cg rt,ra,rb	0 0 0 1 1 0 0 0 0 1 0	RB	RA	RT
5	FX	even	Add Halfword	ah rt,ra,rb	0 0 0 1 1 0 0 1 0 0 0	RB	RA	RT
5	FX	even	Nand	nand rt,ra,rb	0 0 0 1 1 0 0 1 0 0 1	RB	RA	RT
5	BO	even	Average Bytes	avgb rt,ra,rb	0 0 0 1 1 0 1 0 0 1 1	RB	RA	RT
5	FX	even	Add Word Immediate	ai rt,ra,value	0 0 0 1 1 1 0 0	I10	RA	RT
5	FX	even	Add Halfword Immediate	ahi rt,ra,value	0 0 0 1 1 1 0 1	I10	RA	RT
7	BR	odd	Branch If Zero Word	brz rt,symbol	0 0 1 0 0 0 0 0 0	I16		RT
3	LS	odd	Store Quadword (a-form)	stqa rt,symbol	0 0 1 0 0 0 0 0 1	I16		RT
7	BR	odd	Branch If Not Zero Word	brnz rt,symbol	0 0 1 0 0 0 0 1 0	I16		RT
10	CH	odd	Move to Special Purpose Register	mtspr sa, rt	0 0 1 0 0 0 0 1 1 0 0	///	SA	RT
11	CH	odd	Write Channel	wrch ca,rt	0 0 1 0 0 0 0 1 1 0 1	///	CA	RT
7	BR	odd	Branch If Zero Halfword	brhz rt,symbol	0 0 1 0 0 0 1 0 0	I16		RT
7	BR	odd	Branch If Not Zero Halfword	brhnz rt,symbol	0 0 1 0 0 0 1 1 0	I16		RT
3	LS	odd	Store Quadword Instruction Relative (a-form)	stqr rt,symbol	0 0 1 0 0 0 1 1 1	I16		RT
3	LS	odd	Store Quadword (d-form)	stqd rt,symbol(ra)	0 0 1 0 0 1 0 0	I10	RA	RT
7	BR	odd	Branch Indirect If Zero	biz rt,ra	0 0 1 0 0 1 0 1 0 0 0	/ D E / / / /	RA	RT
7	BR	odd	Branch Indirect If Not Zero	binz rt,ra	0 0 1 0 0 1 0 1 0 0 1	/ D E / / / /	RA	RT
7	BR	odd	Branch Indirect If Zero Halfword	bihz rt,ra	0 0 1 0 0 1 0 1 0 1 0	/ D E / / / /	RA	RT
7	BR	odd	Branch Indirect If Not Zero Halfword	bihnz rt,ra	0 0 1 0 0 1 0 1 0 1 1	/ D E / / / /	RA	RT
10	BR	odd	Stop and Signal with Dependencies	stopd	0 0 1 0 1 0 0 0 0 0 0	RB	RA	RC
3	LS	odd	Store Quadword (x-form)	stqx rt,ra,rb	0 0 1 0 1 0 0 0 1 0 0	RB	RA	RT
7	BR	odd	Branch Absolute	bra symbol	0 0 1 1 0 0 0 0 0	I16		///
3	LS	odd	Load Quadword (a-form)	lqa rt,symbol	0 0 1 1 0 0 0 0 1	I16		RT
7	BR	odd	Branch Absolute and Set Link	brasl rt,symbol	0 0 1 1 0 0 0 1 0	I16		RT
7	BR	odd	Branch Relative	br symbol	0 0 1 1 0 0 1 0 0	I16		///
4	SH	odd	Form Select Mask for Bytes Immediate	fsmbi rt,symbol	0 0 1 1 0 0 1 0 1	I16		RT
7	BR	odd	Branch Relative and Set Link	brsl rt,symbol	0 0 1 1 0 0 1 1 0	I16		RT
3	LS	odd	Load Quadword Instruction Relative (a-form)	lqr rt,symbol	0 0 1 1 0 0 1 1 1	I16		RT
3	LS	odd	Load Quadword (d-form)	lqd rt,symbol(ra)	0 0 1 1 0 1 0 0	I10	RA	RT
7	BR	odd	Branch Indirect	bi ra	0 0 1 1 0 1 0 1 0 0 0	/ D E / / / /	RA	///
7	BR	odd	Branch Indirect and Set Link	bisl rt,ra	0 0 1 1 0 1 0 1 0 0 1	/ D E / / / /	RA	RT

ISA book section	instr. class	pipeline	ISA instr. Name	ISA mnemonic & operands	instruction format		
					opcode	operands	
7	BR	odd	Interrupt Return	iret ra	00110101010	/DE////	RA	///
7	BR	odd	Branch Indirect and Set Link if External Data	bisled rt,ra	00110101011	/DE////	RA	RT
8	HB	odd	Hint for Branch (r-form)	hbr brinst,brtarg	00110101100	P//// ROH	RA	ROL
5	SH	odd	Gather Bits from Words	gb rt,ra	00110110000	///	RA	RT
5	SH	odd	Gather Bits from Halfwords	gbh rt,ra	00110110001	///	RA	RT
5	SH	odd	Gather Bits from Bytes	gbb rt,ra	00110110010	///	RA	RT
5	SH	odd	Form Select Mask for Words	fsm rt,ra	00110110100	///	RA	RT
5	SH	odd	Form Select Mask for Halfwords	fsmh rt,ra	00110110101	///	RA	RT
5	SH	odd	Form Select Mask for Bytes	fsmb rt,ra	00110110110	///	RA	RT
9	SH	odd	Floating Reciprocal Estimate	frest rt,ra	00110111000	///	RA	RT
9	SH	odd	Floating Reciprocal Absolute Square Root Estimate	frsqest rt,ra	00110111001	///	RA	RT
3	LS	odd	Load Quadword (x-form)	lqx rt,ra,rb	00111000100	RB	RA	RT
6	SH	odd	Rotate Quadword by Bytes from Bit Shift Count	rotqbybi rt,ra,rb	00111001100	RB	RA	RT
6	SH	odd	Rotate and Mask Quadword Bytes from Bit Shift Count	rotqmbybi rt,ra,rb	00111001101	RB	RA	RT
6	SH	odd	Shift Left Quadword by Bytes from Bit Shift Count	shlqbybi rt,ra,rb	00111001111	RB	RA	RT
3	SH	odd	Generate Controls for Byte Insertion (x-form)	cbx rt,ra,rb	00111010100	RB	RA	RT
3	SH	odd	Generate Controls for Halfword Insertion (x-form)	chx rt,ra,rb	00111010101	RB	RA	RT
3	SH	odd	Generate Controls for Word Insertion (x-form)	cwx rt,ra,rb	00111010110	RB	RA	RT
3	SH	odd	Generate Controls for Doubleword Insertion (x-form)	cdx rt,ra,rb	00111010111	RB	RA	RT
6	SH	odd	Rotate Quadword by Bits	rotqbi rt,ra,rb	00111011000	RB	RA	RT
6	SH	odd	Rotate and Mask Quadword by Bits	rotqmbi rt,ra,rb	00111011001	RB	RA	RT
6	SH	odd	Shift Left Quadword by Bits	shlqbi rt,ra,rb	00111011011	RB	RA	RT
6	SH	odd	Rotate Quadword by Bytes	rotqby rt,ra,rb	00111011100	RB	RA	RT
6	SH	odd	Rotate and Mask Quadword by Bytes	rotqmby rt,ra,rb	00111011101	RB	RA	RT
6	SH	odd	Shift Left Quadword by Bytes	shlqby rt,ra,rb	00111011111	RB	RA	RT
5	BR*	odd	Or Across	orx rt,ra	00111110000	///	RA	RT
3	SH	odd	Generate Controls for Byte Insertion (d-form)	cbd rt,symbol(ra)	00111110100	I7	RA	RT
3	SH	odd	Generate Controls for Halfword Insertion (d-form)	chd rt,symbol(ra)	00111110101	I7	RA	RT
3	SH	odd	Generate Controls for Word Insertion (d-form)	cwd rt,symbol(ra)	00111110110	I7	RA	RT
3	SH	odd	Generate Controls for Doubleword Insertion (d-form)	cdd rt,symbol(ra)	00111110111	I7	RA	RT
6	SH	odd	Rotate Quadword by Bits Immediate	rotqbii rt,ra,value	00111111000	I7	RA	RT
6	SH	odd	Rotate and Mask Quadword by Bits Immediate	rotqmbii rt,ra,value	00111111001	I7	RA	RT
6	SH	odd	Shift Left Quadword by Bits Immediate	shlqbii rt,ra,value	00111111011	I7	RA	RT
6	SH	odd	Rotate Quadword by Bytes Immediate	rotqbyi rt,ra,value	00111111100	I7	RA	RT
6	SH	odd	Rotate and Mask Quadword by Bytes Immediate	rotqmbyi rt,ra,value	00111111101	I7	RA	RT
6	SH	odd	Shift Left Quadword by Bytes Immediate	shlqbyi rt,ra,value	00111111111	I7	RA	RT
10	NOP	even	No Operation (Execute)	nop	01000000001	///	///	RT
4	FX	even	Immediate Load Word	il rt,symbol	01000001	I16		RT

ISA book section	instr. class	pipeline	ISA instr. Name	ISA mnemonic & operands	instruction format			
					opcode	operands		
4	FX	even	Immediate Load Halfword Upper	ilhu rt,symbol	0 1 0 0 0 0 0 1 0	I16		RT
4	FX	even	Immediate Load Halfword	ilh rt,symbol	0 1 0 0 0 0 0 1 1	I16		RT
4	FX	even	Immediate Load Address	ila rt,symbol	0 1 0 0 0 0 1	I18		RT
5	FX	even	Exclusive Or Word Immediate	xori rt,ra,value	0 1 0 0 0 1 0 0	I10	RA	RT
5	FX	even	Exclusive Or Halfword Immediate	xorhi rt,ra,value	0 1 0 0 0 1 0 1	I10	RA	RT
5	FX	even	Exclusive Or Byte Immediate	xorbi rt,ra,value	0 1 0 0 0 1 1 0	I10	RA	RT
7	FX	even	Compare Greater Than Word	cgt rt,ra,rb	0 1 0 0 1 0 0 0 0 0 0	RB	RA	RT
5	FX	even	Exclusive Or	xor rt,ra,rb	0 1 0 0 1 0 0 0 0 0 1	RB	RA	RT
7	FX	even	Compare Greater Than Halfword	cgth rt,ra,rb	0 1 0 0 1 0 0 1 0 0 0	RB	RA	RT
5	FX	even	Equivalent	eqv rt,ra,rb	0 1 0 0 1 0 0 1 0 0 1	RB	RA	RT
7	FX	even	Compare Greater Than Byte	cgtb rt,ra,rb	0 1 0 0 1 0 1 0 0 0 0	RB	RA	RT
5	BO	even	Sum Bytes into Halfwords	sumb rt,ra,rb	0 1 0 0 1 0 1 0 0 1 1	RB	RA	RT
7	FX	even	Halt If Greater Than	hgt ra,rb	0 1 0 0 1 0 1 1 0 0 0	RB	RA	RT
7	FX	even	Compare Greater Than Word Immediate	cgti rt,ra,value	0 1 0 0 1 1 0 0	I10	RA	RT
7	FX	even	Compare Greater Than Halfword Immediate	cgthi rt,ra,value	0 1 0 0 1 1 0 1	I10	RA	RT
7	FX	even	Compare Greater Than Byte Immediate	cgtbi rt,ra,value	0 1 0 0 1 1 1 0	I10	RA	RT
7	FX	even	Halt If Greater Than Immediate	hgti ra,symbol	0 1 0 0 1 1 1 1	I10	RA	RT
5	FX	even	Count Leading Zeros	clz rt,ra	0 1 0 1 0 1 0 0 1 0 1	///	RA	RT
5	FX	even	Extend Sign Word to Doubleword	xswd rt,ra	0 1 0 1 0 1 0 0 1 1 0	///	RA	RT
5	FX	even	Extend Sign Halfword to Word	xshw rt,ra	0 1 0 1 0 1 0 1 1 1 0	///	RA	RT
5	BO	even	Count Ones in Bytes	cntb rt,ra	0 1 0 1 0 1 1 0 1 0 0	///	RA	RT
5	FX	even	Extend Sign Byte to Halfword	xsbh rt,ra	0 1 0 1 0 1 1 0 1 1 0	///	RA	RT
7	FX	even	Compare Logical Greater Than Word	clgt rt,ra,rb	0 1 0 1 1 0 0 0 0 0 0	RB	RA	RT
5	FX	even	And with Complement	andc rt,ra,rb	0 1 0 1 1 0 0 0 0 0 1	RB	RA	RT
9	FX	even	Floating Compare Greater Than	fcgt rt,ra,rb	0 1 0 1 1 0 0 0 0 1 0	RB	RA	RT
9	SP	even	Floating Add	fa rt,ra,rb	0 1 0 1 1 0 0 0 1 0 0	RB	RA	RT
9	SP	even	Floating Subtract	fs rt,ra,rb	0 1 0 1 1 0 0 0 1 0 1	RB	RA	RT
9	SP	even	Floating Multiply	fm rt,ra,rb	0 1 0 1 1 0 0 0 1 1 0	RB	RA	RT
7	FX	even	Compare Logical Greater Than Halfword	clgth rt,ra,rb	0 1 0 1 1 0 0 1 0 0 0	RB	RA	RT
5	FX	even	Or with Complement	orc rt,ra,rb	0 1 0 1 1 0 0 1 0 0 1	RB	RA	RT
9	FX	even	Floating Compare Magnitude Greater Than	fcmgt rt,ra,rb	0 1 0 1 1 0 0 1 0 1 0	RB	RA	RT
9	DP	even	Double Floating Add	dfa rt,ra,rb	0 1 0 1 1 0 0 1 1 0 0	RB	RA	RT
9	DP	even	Double Floating Subtract	dfs rt,ra,rb	0 1 0 1 1 0 0 1 1 0 1	RB	RA	RT
9	DP	even	Double Floating Multiply	dfm rt,ra,rb	0 1 0 1 1 0 0 1 1 1 0	RB	RA	RT
7	FX	even	Compare Logical Greater Than Byte	clgtb rt,ra,rb	0 1 0 1 1 0 1 0 0 0 0	RB	RA	RT
7	FX	even	Halt If Logically Greater Than	hlgt ra,rb	0 1 0 1 1 0 1 1 0 0 0	RB	RA	RT
7	FX	even	Compare Logical Greater Than Word Immediate	clgti rt,ra,value	0 1 0 1 1 1 0 0	I10	RA	RT
7	FX	even	Compare Logical Greater Than Halfword Immediate	clgthi rt,ra,value	0 1 0 1 1 1 0 1	I10	RA	RT

ISA book section	instr. class	pipeline	ISA instr. Name	ISA mnemonic & operands	instruction format			
					opcode	operands		
7	FX	even	Compare Logical Greater Than Byte Immediate	clgtbi rt, ra, value	0 1 0 1 1 1 1 0	I10	RA	RT
7	FX	even	Halt If Logically Greater Than Immediate	hlgti ra, symbol	0 1 0 1 1 1 1 1	I10	RA	RT
4	FX	even	Immediate Or Halfword Lower	iohl rt, symbol	0 1 1 0 0 0 0 0 1	I16		RT
5	FX	even	Add Extended	addx rt, ra, rb	0 1 1 0 1 0 0 0 0 0 0	RB	RA	RT
5	FX	even	Subtract From Extended	sfx rt, ra, rb	0 1 1 0 1 0 0 0 0 0 1	RB	RA	RT
5	FX	even	Carry Generate Extended	cgx rt, ra, rb	0 1 1 0 1 0 0 0 0 1 0	RB	RA	RT
5	FX	even	Borrow Generate Extended	bgx rt, ra, rb	0 1 1 0 1 0 0 0 0 1 1	RB	RA	RT
5	FI	even	Multiply High High and Add	mpyhha rt, ra, rb	0 1 1 0 1 0 0 0 1 1 0	RB	RA	RT
5	FI	even	Multiply High High Unsigned and Add	mpyhhau rt, ra, rb	0 1 1 0 1 0 0 1 1 1 0	RB	RA	RT
9	DP	even	Double Floating Multiply and Add	dfma rt, ra, rb	0 1 1 0 1 0 1 1 1 0 0	RB	RA	RT
9	DP	even	Double Floating Multiply and Subtract	dfms rt, ra, rb	0 1 1 0 1 0 1 1 1 0 1	RB	RA	RT
9	DP	even	Double Floating Negative Multiply and Subtract	dfnms rt, ra, rb	0 1 1 0 1 0 1 1 1 1 0	RB	RA	RT
9	DP	even	Double Floating Negative Multiply and Add	dfnma rt, ra, rb	0 1 1 0 1 0 1 1 1 1 1	RB	RA	RT
9	DP	even	Floating-Point Status and Control Register Read	fscrrd rt	0 1 1 1 0 0 1 1 0 0 0	///	///	RT
5	FI	even	Multiply Immediate	mpyi rt, ra, value	0 1 1 1 0 1 0 0	I10	RA	RT
5	FI	even	Multiply Unsigned Immediate	mpyui rt, ra, value	0 1 1 1 0 1 0 1	I10	RA	RT
9	FI	even	Convert Floating to Signed Integer	cflts rt, ra, scale	0 1 1 1 0 1 1 0 0 0	I8	RA	RT
9	FI	even	Convert Floating to Unsigned Integer	cfltu rt, ra, scale	0 1 1 1 0 1 1 0 0 1	I8	RA	RT
9	FI	even	Convert Signed Integer to Floating	csflt rt, ra, scale	0 1 1 1 0 1 1 0 1 0	I8	RA	RT
9	FI	even	Convert Unsigned Integer to Floating	cuflt rt, ra, scale	0 1 1 1 0 1 1 0 1 1	I8	RA	RT
9	DP	even	Floating Extend Single to Double	fesd rt, ra	0 1 1 1 0 1 1 1 0 0 0	///	RA	RT
9	DP	even	Floating Round Double to Single	frds rt, ra	0 1 1 1 0 1 1 1 0 0 1	///	RA	RT
9	FI	even	Floating Point Status and Control Register Write	fscrwr ra	0 1 1 1 0 1 1 1 0 1 0	///	RA	RT
7	FX	even	Compare Equal Word	ceq rt, ra, rb	0 1 1 1 1 0 0 0 0 0 0	RB	RA	RT
9	FX	even	Floating Compare Equal	fceq rt, ra, rb	0 1 1 1 1 0 0 0 0 1 0	RB	RA	RT
5	FI	even	Multiply	mpy rt, ra, rb	0 1 1 1 1 0 0 0 1 0 0	RB	RA	RT
5	FI	even	Multiply High	mpyh rt, ra, rb	0 1 1 1 1 0 0 0 1 0 1	RB	RA	RT
5	FI	even	Multiply High High	mpyhh rt, ra, rb	0 1 1 1 1 0 0 0 1 1 0	RB	RA	RT
5	FI	even	Multiply and Shift Right	mpys rt, ra, rb	0 1 1 1 1 0 0 0 1 1 1	RB	RA	RT
7	FX	even	Compare Equal Halfword	ceqh rt, ra, rb	0 1 1 1 1 0 0 1 0 0 0	RB	RA	RT
9	FX	even	Floating Compare Magnitude Equal	fcmeq rt, ra, rb	0 1 1 1 1 0 0 1 0 1 0	RB	RA	RT
5	FI	even	Multiply Unsigned	mpyu rt, ra, rb	0 1 1 1 1 0 0 1 1 0 0	RB	RA	RT
5	FI	even	Multiply High High Unsigned	mpyhhu rt, ra, rb	0 1 1 1 1 0 0 1 1 1 0	RB	RA	RT
7	FX	even	Compare Equal Byte	ceqb rt, ra, rb	0 1 1 1 1 0 1 0 0 0 0	RB	RA	RT
9	FI	even	Floating Interpolate	fi rt, ra, rb	0 1 1 1 1 0 1 0 1 0 0	RB	RA	RT
7	FX	even	Halt If Equal	heq ra, rb	0 1 1 1 1 0 1 1 0 0 0	RB	RA	RT
7	FX	even	Compare Equal Word Immediate	ceqi rt, ra, value	0 1 1 1 1 1 0 0	I10	RA	RT
7	FX	even	Compare Equal Halfword Immediate	ceqhi rt, ra, value	0 1 1 1 1 1 0 1	I10	RA	RT

ISA book section	instr. class	pipeline	ISA instr. Name	ISA mnemonic & operands	instruction format				
					opcode		operands		
7	FX	even	Compare Equal Byte Immediate	ceqbi rt,ra,value	0 1 1 1 1 1 1 0	I10	RA	RT	
7	FX	even	Halt If Equal Immediate	heqi ra,symbol	0 1 1 1 1 1 1 1	I10	RA	RT	
5	FX	even	Select Bits	selb rt,ra,rb,rc	1 0 0 0	RT	RB	RA	RC
5	SH	odd	Shuffle Bytes	shufb rt,ra,rb,rc	1 0 1 1	RT	RB	RA	RC
5	FI	even	Multiply and Add	mpya rt,ra,rb,rc	1 1 0 0	RT	RB	RA	RC
9	SP	even	Floating Negative Multiply and Subtract	fnms rt,ra,rb,rc	1 1 0 1	RT	RB	RA	RC
9	SP	even	Floating Multiply and Add	fma rt,ra,rb,rc	1 1 1 0	RT	RB	RA	RC
9	SP	even	Floating Multiply and Subtract	fms rt,rb,ra,rc	1 1 1 1	RT	RB	RA	RC

(Note)
The **orx** instruction was added in DD2.0, and intentionally implemented within the branch resolution logic.
Since the BR class and the SH class have the same characteristics (odd pipeline and 4-cycle latency), the **orx** instruction can be practically regarded as a SH class instruction.

5.4. SPU Interrupt Facility in SPU ISA

This section describes the SPU interrupt facility defined in the *Synergistic Processor Unit Instruction Set Architecture*.

External conditions are monitored and managed though the external facilities that are controlled through the channel interface. External conditions can affect SPU instruction sequencing through the following facilities:

1. The **bisled** instruction
   bisled instruction tests for the existence of the external condition, and branches to a target if one is present. **bisled** allows the SPU software to poll for external conditions and to call a handler subroutine if one is present. When polling is not desirable, the SPU can be enabled to interrupt normal instruction processing and to vector to a handler subroutine when an external condition appears.
2. The interrupt facility

The interrupt facility is enabled and disabled with indirect branch instructions:

- **bi**
- **bisl**
- **bisled**
- **biz**
- **binz**
- **bihz**
- **bihnz**

All of these branch instructions provide the [D] and [E] feature bits. When one of these branches is taken, the interrupt-enable status will change before the target instruction is executed. *Table 5-1* describes the feature bit settings and their results.

Table 5-1. Feature Bits [D] and [E] Settings and Results

Feature Bit Setting	Result
[D] = 0 [E] = 0	Status does not change.
[D] = 0 [E] = 1	Interrupt processing is enabled.
[D] = 1 [E] = 0	Interrupt processing is disabled.
[D] = 1 [E] = 1	Causes undefined behavior.

These branch instructions allow software to enable interrupts and to disable interrupts during critical subroutines.

5.4.1. SPU Interrupt Handler

The SPU supports a single interrupt handler. The entry point for this handler is address 0 in local storage. When a condition is present and interrupts are enabled, the SPU branches to address 0 and disables the interrupt facility. The address of the next instruction to be executed is saved in the SRR0 register. The **iret** instruction can be used to return from the handler. **iret** branches indirectly to the address held in the SRR0 register. **iret**, like the other indirect branches, has an [E] feature bit that can be used to re-enable interrupts.

5.4.2. SPU Interrupt Facility Channels

The interrupt facility uses several channels for configuration, state observation and state restoration. The current value of SRR0 can be read from SPU_RdSRR0 Channel, and SPU_WrSRR0 Channel provides write access to SRR0. When SRR0 is written by **wrch** SPU_WrSRR0, synchronization is required to ensure that this new value is available to the **iret** instruction. This synchronization is provided by execution of the **sync** instruction with the [C] or Channel Sync feature bit set. Without this synchronization, **iret** instructions executed after **wrch** SPU_WrSRR0 instructions branch to unpredictable addresses. SPU_WrSRR0 Channel and SPU_RdSRR0 Channel support nested interrupts by allowing software to save and restore SRR0 to a save area in local storage.

6. SPU Channels

6.1. SPU Channel Definition

Table 6-1 lists all the channels defined for the SPU in the *Cell Broadband Engine™ Architecture* by channel number and briefly describes each channel.

Table 6-1. Channel Definitions

Channel Numbers		Channel Name	Description	Read/Write	Type[1]	Maximum Count	POR Count[2]	POR Data[3]	Blocking (B) or Nonblocking (N)[4]	See	Comments
Dec	Hex										
SPU Event Channels											
0	x'0'	SPU_RdEventStat	SPU Read Event Status Channel	Read	P	1	0	0	B	*Section 6.3.1.1* on page 84	Accumulative. *SPU_WrEventAck* Channel clears per bit. The count increments on event, noncleared event acknowledgement, or mask update.
1	x'1'	SPU_WrEventMask	SPU Write Event Mask Channel	Write	A	1[5]	1[5]	0	N	*Section 6.3.1.2* on page 84	Masked status stored in hidden buffer.
2	x'2'	SPU_WrEventAck	SPU Write Event Acknowledgment Channel	Write	A	1[5]	1[5]	U	N	*Section 6.3.1.3* on page 84	Write sets *SPU_RdEventStat* Channel count to 1 for unmasked status not acknowledged.
SPU Signal Notification Channels											
3	x'3'	SPU_RdSigNotify1	SPU Signal Notification 1 Channel	Read	P	1	U	C	B	*Section 6.3.2.1* on page *84*	Configurable accumulative or overwrite on external loads, and clears on reads. Value after load function: x'00000000'
4	x'4'	SPU_RdSigNotify2	SPU Signal Notification 2 Channel	Read	P	1	U	C	B	*Section 6.3.2.1* on page 84	Configurable accumulative or overwrite on external loads, and clears on reads.
5	x'5'		Reserved	—	R	0	0	U	—		
6	x'6'		Reserved	—	R	0	0	U	—		
SPU Decrementer Channels											
7	x'7'	SPU_WrDec	SPU Write Decrementer Channel	Write	A	1[5]	1[5]	U	N	CBEA	Sends load value to decrementer.
8	x'8'	SPU_RdDec	SPU Read Decrementer Channel	Read	A	1[5]	1[5]	0	N	*Section 6.3.3.1* on page 84	The SPU requests a read of decrementer count and stalls until it is received. The channel is nonblocking, but stalls the SPU execution for at least 20 cycles.
SPE Multi Source Synchronization Channel (DD2.0 and later)											
9	x'9'	MFC_WrMSSyncReq	MFC Write Multisource Synchronization Request Channel	Write	A	1[5]	1[5]	U	B	CBEA	The *MFC_WrMSSyncReq* Channel is used to cause the MFC to start tracking outstanding transfers targeting the associated MFC. (DD2.0 and later)
SPU Reserved Channel											
10	x'A'		Reserved	—	R	1[5]	1[5]	U	N		

Table 6-1. Channel Definitions

Channel Numbers		Channel Name	Description	Read/Write	Type [1]	Maximum Count	POR Count [2]	POR Data [3]	Blocking (B) or Nonblocking (N) [4]	See	Comments
Dec	Hex										
SPU Mask Read Channels (DD2.0 and later)											
11	x'B'	SPU_RdEventMask	SPU Read Event Status Mask Channel	Read	P	1 [5]	1 [5]	0	N	CBEA	Event mask loaded by write to *SPU_WrEventMask* Channel request. (DD2.0 and later)
12	x'C'	MFC_RdTagMask	MFC Read Tag-Group Query Mask Channel	Read	P	1 [5]	1 [5]	0	N	CBEA	Tag mask loaded by write to *MFC_WrTagMask* Channel request. (DD2.0 and later)
SPU State Management Channels											
13	x'D'	SPU_RdMachStat	SPU Read Machine Status Channel	Read	P	1 [5]	1 [5]	0	N	CBEA	Contains the status of the SPU.
14	x'E'	SPU_WrSRR0	SPU Write State Save-and-Restore Channel	Write	P	1 [5]	1 [5]	U	N	*Section 6.3.4.1* on page 84	Loads the *SPU_RdSRR0* Channel address for return from interrupt.
15	x'F'	SPU_RdSRR0	SPU Read State Save-and-Restore Channel	Read	P	1 [5]	1 [5]	0	N	CBEA	Contains the SRR0 address for return from interrupt.
MFC Command Parameter Channels											
16	x'10'	MFC_LSA	MFC Local Storage Address Channel	Write	A	1 [5]	1 [5]	U	N	CBEA	Writes local storage address command parameter.
17	x'11'	MFC_EAH	MFC Effective Address High Channel	Write	A	1 [5]	1 [5]	0	N	CBEA	Writes high-order MFC effective address command parameter. If not written, defaults to 0.
18	x'12'	MFC_EAL	MFC Effective Address Low or List Address Channel	Write	A	1 [5]	1 [5]	U	N	CBEA	Writes low-order MFC effective address command parameter. Invalidated by an *MFC_Cmd / MFC_ClassID* Channel write.
19	x'13'	MFC_Size	MFC Transfer Size or List Size Channel	Write	A	1 [5]	1 [5]	U	N	CBEA	Specifies the size of the MFC transfer or the size of the MFC list. Invalidated by an *MFC_Cmd / MFC_ClassID* Channel write.
20	x'14'	MFC_TagID	MFC Command Tag Identification Channel	Write	A	1 [5]	1 [5]	U	N	CBEA	Writes TAG identifier command parameter. Invalidated by an *MFC_Cmd / MFC_ClassID* Channel write.
21	x'15'	MFC_Cmd / MFC_ClassID	MFC Command Opcode Channel MFC Class ID Channel	Write	A	16	16	U	B	CBEA	Loads an MFC command, and invalidates the MFC Command Parameter Channels. MFC acknowledgment required. The maximum count for the *MFC_Cmd / MFC_ClassID* Channel for this implementation of the CBE is 16.

Table 6-1. Channel Definitions

Channel Numbers		Channel Name	Description	Read/Write	Type [1]	Maximum Count	POR Count [2]	POR Data [3]	Blocking (B) or Nonblocking (N) [4]	See	Comments
Dec	Hex										
MFC Tag Status Channels											
22	x'16'	MFC_WrTagMask	MFC Write Tag-Group Query Mask Channel	Write	A	1 [5]	1 [5]	0	N	CBEA	Selects the tag groups to be included in the query or wait operations. Value after load function: x'FFFFFFFF'
23	x'17'	MFC_WrTagUpdate	MFC Write Tag Status Update Request Channel	Write	A	1	1	U	B	CBEA	Request update of tag status in *SPU_RdEventStat* Channel and *MFC_RdTagStat* Channel. MFC acknowledgment required.
24	x'18'	MFC_RdTagStat	MFC Read Tag-Group Status Channel	Read	P	1	0	U	B	CBEA	Tag status loaded by *MFC_WrTagUpdate* Channel request.
25	x'19'	MFC_RdListStallStat	MFC Read List Stall-and-Notify Tag Status Channel	Read	P	1	0	U	B	CBEA	Accumulative. Clear on read. DMA list prefetching is implemented by hardware.
26	x'1A'	MFC_WrListStallAck	MFC Write List Stall-and-Notify Tag Acknowledgment Channel	Write	A	1 [5]	1 [5]	U	N	CBEA	A DMA list command remains stalled until acknowledged by writing the tag value to the *MFC_WrListStallAck* Channel.
27	x'1B'	MFC_RdAtomicStat	MFC Read Atomic Command Status Channel	Read	P	1	0	U	B	CBEA	Overwrite.
SPU/PPU Communication Mailboxes											
28	x'1C'	SPU_WrOutMbox	SPU Write Outbound Mailbox Channel	Write	P	1	1	U	B	CBEA	
29	x'1D'	SPU_RdInMbox	SPU Read Inbound Mailbox Channel	Read	P	4	0	U	B	*Section 6.3.5.1* on page 85	
30	x'1E'	SPU_WrOutIntrMbox	SPU Write Outbound Interrupt Mailbox Channel	Write	P	1	1	U	B	*Section 6.3.5.2* on page 85	A write sends an interrupt.
SPU Reserved Channels											
31:68	x'1F':x'44'		Reserved	—	R	0	0	U	—		
Performance Monitor Channels (DD2.0 and later)											
69	x'45'	Set_Bkmk_Tag	Set Bookmark Tag Channel	Write	A	1	1	U	N	*Section 6.3.5.3* on page 86	A channel write causes an event that can be logged in the performance monitor logic if enabled in the SPU performance monitor control registers. Ignored (no-op) if not enabled.
70	x'46'	PM_Start_Ev	Performance Monitor Start Event Channel	Write	A	1	1	U	N		
71	x'47'	PM_Stop_Ev	Performance Monitor Stop Event Channel	Write	A	1	1	U	N		

Table 6-1. Channel Definitions

Channel Numbers		Channel Name	Description	Read/Write	Type [1]	Maximum Count	POR Count [2]	POR Data [3]	Blocking (B) or Nonblocking (N) [4]	See	Comments
Dec	Hex										
SPU Reserved Channels											
72:127	x'48':x'7F'		Reserved	—	R	0	0	U	—		

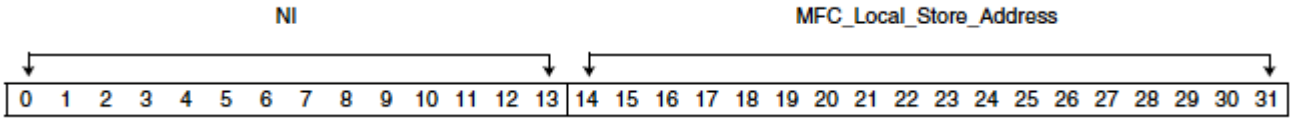
1. Type is defined as:
   A = Active. The SPU forces the completion of the channel operation within a predetermined number of cycles.
   P = Passive. Channel operation is dependent on a source outside the SPU to complete the action and is completed in an undetermined number of cycles.
   R = Reserved
2. Value at initial POR count is defined as:
   U = Undefined
   n = Number of count
3. Value at initial POR data is defined as:
   0 = x'0000 0000'
   U = Undefined
   C = Set by POR configuration ring..
4. A blocking channel is a channel where a **rdch** or **wrch** channel instruction to that channel with a count of zero causes the SPU to wait (see the W bit [bit 28] in the *SPU Status Register (SPU_Status)* on page 96) on that channel instruction until the count is no longer zero. A nonblocking channel does not cause the SPU to wait when the count on that channel is zero and a **rdch** or **wrch** channel instruction occurs to that channel.
5. Count is fixed at 1 and does not change

Note:

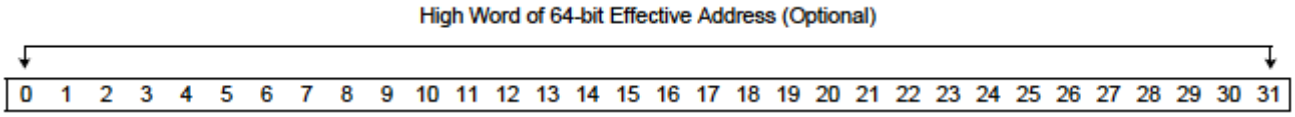
- Channels are unidirectional.
- There is no typical value or recommended value after POR since all of the channels contain application-specific data or status when later used by software.
- The value at initial POR is the value that was initialized during the scan initialization or configuration ring part of the POR sequence.
- The value after load function is an additional point during the POR sequence where the value may differ from the value at initial POR.

6.2. Quick Reference for Channel Fields

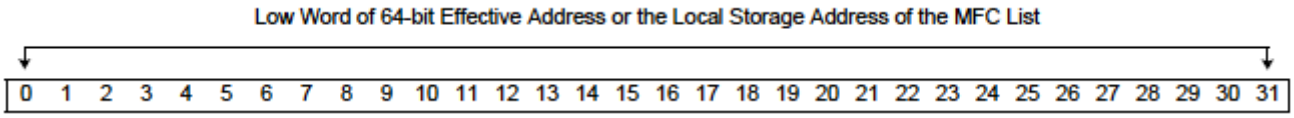
MFC Local Storage Address Channel (MFC_LSA)



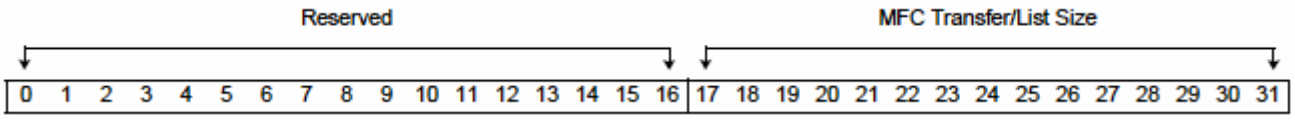
MFC Effective Address High Channel (MFC_EAH)



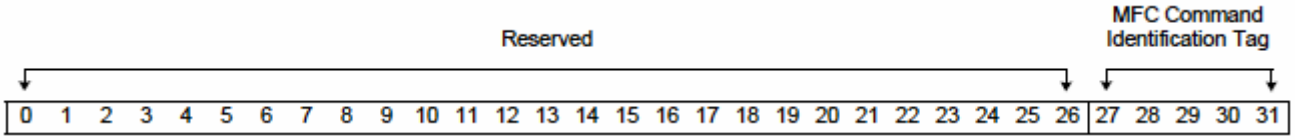
MFC Effective Address Low or List Address Channel (MFC_EAL)



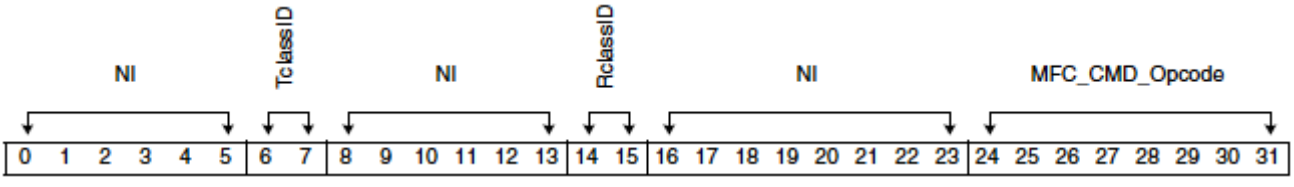
MFC Transfer Size or List Size Channel (MFC_Size)



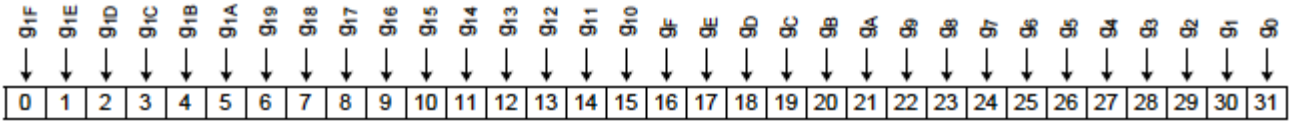
MFC Command Tag Identification Channel (MFC_TagID)



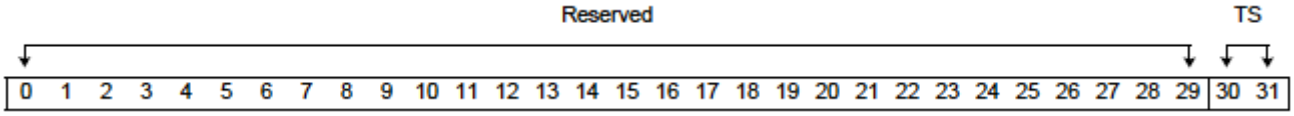
MFC Command Opcode and MFC Class ID Channel (MFC_Cmd / MFC_ClassID)



MFC Write Tag-Group Query Mask Channel (MFC_WrTagMask)
MFC Read Tag-Group Query Mask Channel (MFC_RdTagMask)
MFC Read Tag-Group Status Channel (MFC_RdTagStat)
MFC Read List Stall-and-Notify Tag Status Channel (MFC_RdListStallStat)

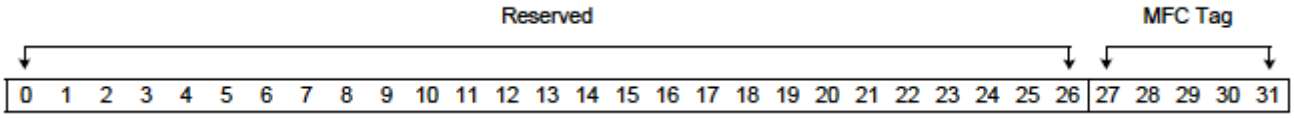


MFC Write Tag Status Update Request Channel (MFC_WrTagUpdate)



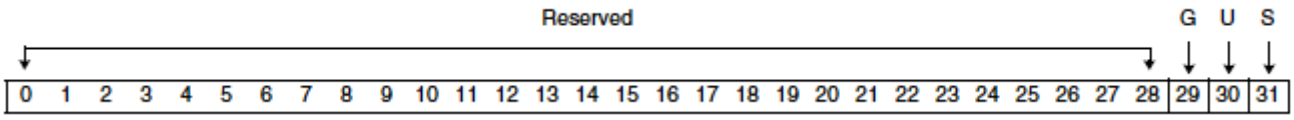
Bits	Field Name	Description
0:29	Reserved	Reserved.
30:31	TS	Tag-status update condition 00 Update immediately, unconditional. 01 Update tag status if or when *any* enabled tag group has “no outstanding operation” status. 10 Update tag status if or when *all* enabled tag groups have “no outstanding operation” status. 11 Reserved.

MFC Write List Stall-and-Notify Tag Acknowledgment Channel (MFC_WrListStallAck)



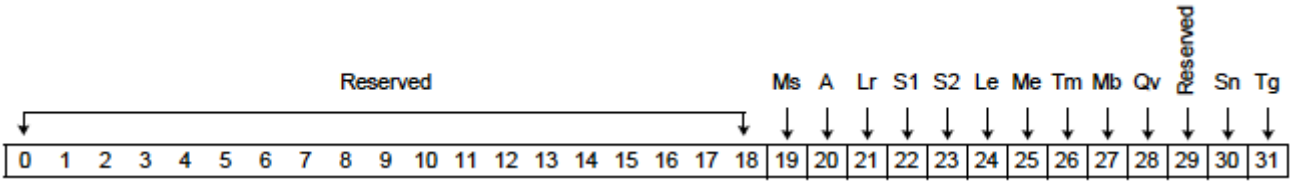
Bits	Field Name	Description
0:26	Reserved	Reserved.
27:31	MFC Tag	The tag can be any value between x‘0’ and x‘1F’.

MFC Read Atomic Command Status Channel (MFC_RdAtomicStat)



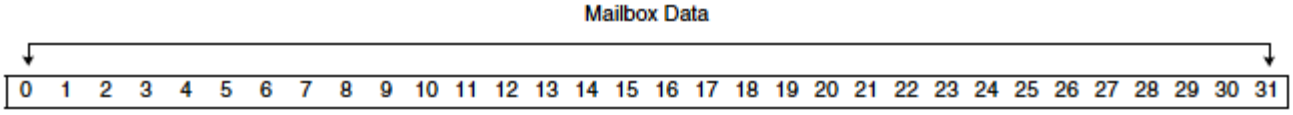
Bits	Field Name	Description
0:28	Reserved	Reserved.
29	G	Set if the get lock-line and reserve (**getllar**) command completed.
30	U	Set if the put lock-line unconditional (**putlluc**) command completed.
31	S	Put lock-line conditional command (**putllc**). 1 Put conditional unsuccessful. The reservation was lost. 0 Put conditional successful.

SPU Read Event Status Channel (SPU_RdEventStat)
SPU Write Event Mask Channel (SPU_WrEventMask)
SPU Read Event Status Mask Channel (SPU_RdEventMask)
SPU Write Event Acknowledgment Channel (SPU_WrEventAck)



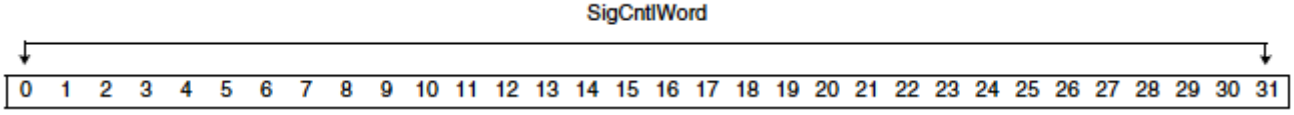
Bits	Field Name	Description
0:18	Reserved	Reserved.
19	Ms	Multisource Synchronization event
20	A	Privileged Attention event
21	Lr	Lock Line Reservation Lost event
22	S1	Signal Notification 1 Available event
23	S2	Signal Notification 2 Available event
24	Le	SPU Outbound Mailbox Available event
25	Me	SPU Outbound Interrupt Mailbox Available event
26	Tm	Decrementer event
27	Mb	SPU Inbound Mailbox Available event
28	Qv	MFC SPU Command Queue Available event
29	Reserved	Reserved.
30	Sn	MFC DMA List Stall-and-Notify event
31	Tg	MFC Tag-Group Status Update event

SPU Write Outbound Mailbox Channel (SPU_WrOutMbox)
SPU Read Inbound Mailbox Channel (SPU_RdInMbox)
SPU Write Outbound Interrupt Mailbox Channel (SPU_WrOutIntrMbox)



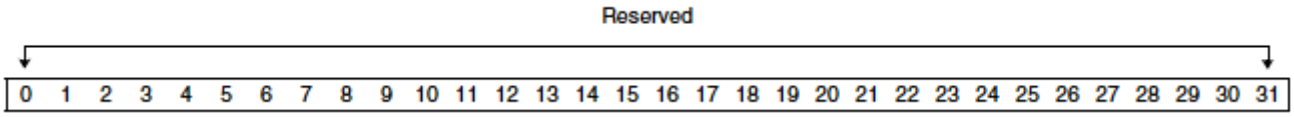
Bits	Field Name	Description
0:31	Mailbox Data	Application-specific mailbox data. Each application can uniquely define the mailbox data.

SPU Signal Notification 1 Channel (SPU_RdSigNotify1)
SPU Signal Notification 2 Channel (SPU_RdSigNotify2)



Bits	Field Name	Description
0:31	SigCntlWord	Signal-control word. The data is defined by the application. It can either be ORed with the previous value, or it can be overwritten.

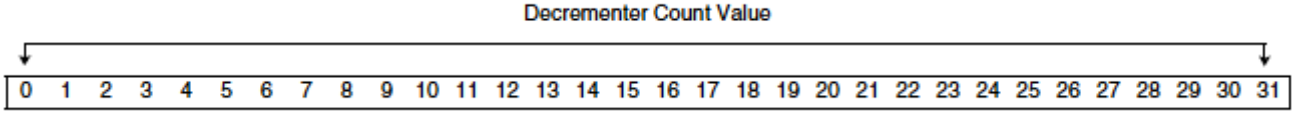
MFC Write Multisource Synchronization Request Channel (MFC_WrMSSyncReq)



Bits	Field Name	Description
0:31	Reserved	When the synchronization requested by a write of this channel is complete, the channel count is set back to 1. The data written to this channel is ignored; however, software should write a value of 0 for compatibility with future enhancements.

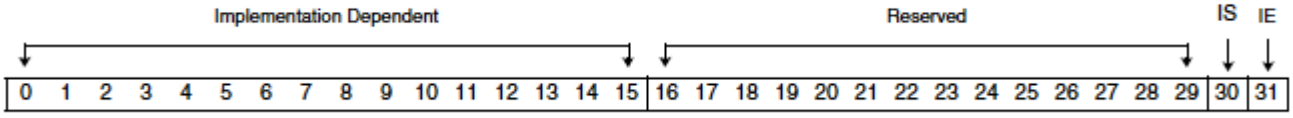
SPU Write Decrementer Channel (SPU_WrDec)

SPU Read Decrementer Channel (SPU_RdDec)



Bits	Field Name	Description
0:31	Decrementer Count Value	Decrementer count value.

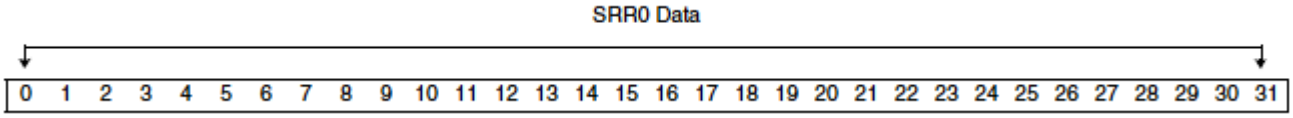
SPU Read Machine Status Channel (SPU_RdMachStat)



Bits	Field Name	Description
0:15	Implementation Dependent	
16:29	Reserved	Set to zeros.
30	IS	Isolation status. 0 The SPU is operating in a non-isolated state. 1 The SPU is operating in an isolated state.
31	IE	SPU interrupt enable status. Interrupts can be enabled by setting the SPU_NPC{IE} bit to '1' while the SPU is stopped or by an SPU instruction. See the *Synergistic Processor Unit Instruction Set Architecture* for more information on how to enable interrupts. 0 SPU interrupts disabled. 1 SPU interrupt enabled.

SPU Write State Save-and-Restore Channel (SPU_WrSRR0)

SPU Read State Save-and-Restore Channel (SPU_RdSRR0)



Bits	Field Name	Description
0:31	SRR0 Data	State save/restore register 0 data.

6.3. Channel Implementation Notes

This section includes implementation-specific information for each channel that differs from or is not covered in the *Cell Broadband Engine™ Architecture*. For a complete specification of channels, see the *Cell Broadband Engine™ Architecture* document.

6.3.1. SPU Event Channels

6.3.1.1. SPU Read Event Status (SPU_RdEventStat) Channel

The channel count for the *SPU_RdEventStat* Channel is also used to satisfy the branch on event and set link register (**bisled**) instruction. If the SPU interrupt is enabled and the channel count for this channel is not zero, then the SPU takes the interrupt and start executing at local-storage address zero. Events for the *SPU_RdEventStat* Channel are documented in the *Cell Broadband Engine™ Architecture* document. Also see *Section 3.1.7 SPU Interrupt Facility* on page 39 for more details.

6.3.1.2. SPU Write Event Mask (SPU_WrEventMask) Channel

If, after writing the *SPU_WrEventMask* Channel, there are pending (unacknowledged) events that could be read in the *SPU_RdEventStat* Channel, then the *SPU_RdEventStat* Channel count is set to 1.

6.3.1.3. SPU Write Event Acknowledgment (SPU_WrEventAck) Channel

Writing the *SPU_WrEventAck* Channel causes the channel count for the *SPU_RdEventStat* Channel to increment to 1 on a pending event that is unmasked and is not acknowledged. Writing this channel with no bits set causes the *SPU_RdEventStat* Channel to be reevaluated without clearing the pending events.
Clearing all pending events does not affect the channel count for the *SPU_RdEventStat* Channel. If the channel count is 1, it remains 1; if 0, it remains 0.

6.3.2. SPU Signal Notification Channels

6.3.2.1. SPU Signal Notification 1 (SPU_RdSigNotify1) Channel and SPU Signal Notification 2 (SPU_RdSigNotify2) Channel

The SPU_Cfg Register controls whether additional external writes on the channel-in interface to the register for this channel overwrite the value currently in the register (nonaccumulating mode), or if they add the new value to the current value stored in the register (accumulating mode). The contents of the register are cleared on a **rdch** instruction in accumulating and nonaccumulating modes.

6.3.3. SPU Decrementer Channel

6.3.3.1. SPU Read Decrementer (SPU_RdDec) Channel

At POR, the decrementer is stopped. Reads to the SPU_RdDec Channel require a minimum of a 20-cycle issue stall. That is, the SPU is stalled, waiting on completion of the read.

6.3.4. SPU State Management Channels

6.3.4.1. SPU Write State Save-and-Restore (SPU_WrSRR0) Channel

A channel synchronization instruction (**sync** with the C feature bit set) is needed after writing to the SPU_WrSRR0 Channel.

6.3.5. SPU Mailboxes

6.3.5.1. SPU Read Inbound Mailbox (SPU_RdInMbox) Channel

The implementation of this channel differs from the description in *Cell Broadband Engine™ Architecture*. The count for the SPU_RdInMbox Channel is initialized by firmware to zero. The channel is blocking, and the maximum channel count is four. A **rdch** instruction to the SPU_RdInMbox Channel performs the following steps:

1. Reads the oldest unread 32-bit value stored in the SPU_RdInMbox Channel registers in the channel logic.
2. Stores this value in the most-significant 32 bits of the GPR specified in the instruction.
3. Asserts the SPU mailbox threshold interrupt (see the *Cell Broadband Engine™ Architecture* document).

Each time the PowerPC Processor Unit (PPU) writes to the SPU Inbound Mailbox Register (SPU_In_Mbox), the channel count for the SPU_RdInMbox Channel increments. (See *Section 8.2.3.2* on page 94.) Each time the SPU reads the SPU_RdInMbox Channel, the channel count decrements. The SPU reads the oldest data out first. If the PPU writes more than four times before the SPU reads the data, then the channel count stays at four and the fourth location contains the last data written by the PPU. For example, if the PPU writes five times before the SPU reads the data, then the data read is the first, second, third, and fifth data elements. The fourth data element has been overwritten.

Context save and restore values are only valid when the SPU is idle. All four mailbox registers must be saved and restored to ensure correct operation. A memory-mapped control register read of this register returns one of the following values:

- The oldest valid value, if the count is greater than 1
- The last value written, if the channel count equals 1
- The oldest value written, if the channel count equals 0

The memory-mapped control register reads always read the same value until there is a **rdch** instruction for this channel, after which it steps to the next register value.

In this implementation, the SPU mailbox threshold for asserting an interrupt is four entries. When there are fewer entries available to write than the threshold, the mailbox threshold interrupt is deasserted.

6.3.5.2. SPU Write Outbound Interrupt Mailbox (SPU_WrOutIntrMbox) Channel

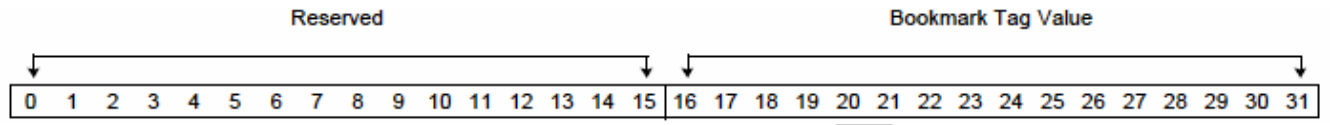
The channel is write-blocking enabled with a maximum count of one.

6.3.5.3. Performance Monitor Channels

The channels described in this section are Performance Monitor Channels that are not defined in the CBEA. The Set_Bkmk_Tag Channel, the PM_Start_Ev Channel, and the PM_Stop_Ev Channel are used in the Performance Monitor.

Set Bookmark Tag (Set_Bkmk_Tag) Channel

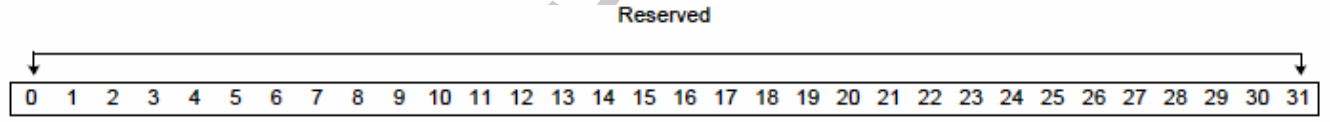
A channel write instruction to the Set_Bkmk_Tag Channel causes an event that can be logged in the performance monitor logic if enabled in one of the SPU Event Select MMIO Registers. If not enabled, writes to this channel are ignored (no-op).
Data used for this channel is the least significant 16 bits of data in the most significant word from the SPU register specified in the instruction.
This channel is nonblocking and does not have an associated count. If a read channel count is directed to this channel, the count is always returned as 1.



Bits	Field Name	Description
0:15	Reserved	Set to zero.
16:31	Bookmark Tag Value	Bookmark Tag Value

Performance Monitor Start Event (PM_Start_Ev) Channel

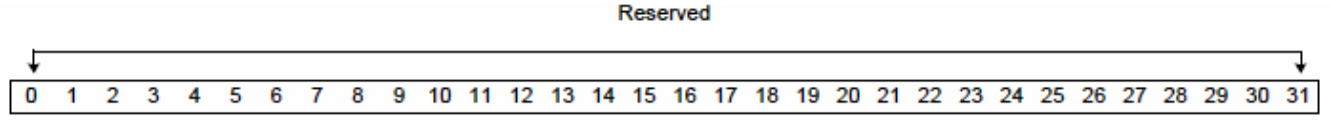
A channel write instruction to the PM_Start_Ev Channel causes an event that can be logged in the performance monitor logic if enabled in one of the SPU Trigger Select MMIO Registers. If not enabled, writes to this channel are ignored (no-op).
No data is used for this channel. This channel is nonblocking and does not have an associated count. If a read channel count is directed to this channel, the count is always returned as 1.



Bits	Field Name	Description
0:31	Reserved	Set to zero.

Performance Monitor Stop Event (PM_Stop_Ev) Channel

A channel write instruction to the PM_Stop_Ev Channel causes an event that can be logged in the performance monitor logic if enabled in one of the SPU Trigger Select MMIO Registers. If not enabled, writes to this channel are ignored (no-op).
No data is used for this channel. This channel is nonblocking and does not have an associated count. If a read channel count is directed to this channel, the count is always returned as 1.



Bits	Field Name	Description
0:31	Reserved	Set to zero.

7. MFC Commands

7.1. MFC Commands Quick Reference

Table 7-1. MFC DMA Data-Transfer Commands

Mnemonic	Opcode	Support		Description
Put Commands				
put	x‘20’	Proxy	Channel	Moves data from local storage to the effective address within the main storage domain.
putf	x‘22’	Proxy	Channel	Moves data from local storage to the effective address within the main storage domain with fence.
putb	x‘21’	Proxy	Channel	Moves data from local storage to the effective address within the main storage domain with barrier.
putl	x‘24’		Channel	Moves data from local storage to the effective address within the main storage domain using a DMA list.
putlf	x‘26’		Channel	Moves data from local storage to the effective address within the main storage domain using a DMA list with fence.
putlb	x‘25’		Channel	Moves data from local storage to the effective address within the main storage domain using a DMA list with barrier.
puts	x‘28’	Proxy		Moves data from local storage to the effective address within the main storage domain, and starts the SPU after the DMA operation completes.
putfs	x‘2A’	Proxy		Moves data from local storage to the effective address within the main storage domain with fence, and starts the SPU after the DMA operation completes.
putbs	x‘29’	Proxy		Moves data from local storage to the effective address within the main storage domain with barrier, and starts the SPU after the DMA operation completes.
putr	x‘30’	Proxy	Channel	Same as **put** with a PPE L2-cache scarf hint (used to send results to the PPE).
putrf	x‘32’	Proxy	Channel	Same as **putf** with a PPE L2-cache scarf hint (used to send results to the PPE).
putrb	x‘31’	Proxy	Channel	Same as **putb** with a PPE L2-cache scarf hint (used to send results to the PPE).
putrl	x‘34’		Channel	Same as **putl** with a PPE L2-cache scarf hint (used to send results to the PPE).
putrlf	x‘36’		Channel	Same as **putlf** with a PPE L2-cache scarf hint (used to send results to the PPE).
putrlb	x‘35’		Channel	Same as **putlb** with a PPE L2-cache scarf hint (used to send results to the PPE).
Get Commands				
get	x‘40’	Proxy	Channel	Moves data from the effective address within the main storage domain to local storage.
getf	x‘42’	Proxy	Channel	Moves data from the effective address within the main storage domain to local storage with fence.
getb	x‘41’	Proxy	Channel	Moves data from the effective address within the main storage domain to local storage with barrier.
getl	x‘44’		Channel	Moves data from the effective address within the main storage domain to local storage using a DMA list.
getlf	x‘46’		Channel	Moves data from the effective address within the main storage domain to local storage using a DMA list with fence.
getlb	x‘45’		Channel	Moves data from the effective address within the main storage domain to local storage using a DMA list with barrier.
gets	x‘48’	Proxy		Moves data from the effective address within the main storage domain to local storage, and starts the SPU after the DMA operation completes.
getfs	x‘4A’	Proxy		Moves data from the effective address within the main storage domain to local storage with fence, and starts the SPU after the DMA operation completes.
getbs	x‘49’	Proxy		Moves data from the effective address within the main storage domain to local storage with barrier, and starts the SPU after the DMA operation completes.

Command suffixes

s Starts the execution of the SPE at the current location indicated by the *SPU Next Program Counter Register (SPU_NPC)* after the data has been transferred into or out of the local storage.

r Performance hint for DMA put operations.
The hint is intended to allow another processor or device, such as PPE, to capture the data into its cache.

f Tag specific fence.
Commands with tag-specific fence are locally ordered
with respect to all previously-issued commands within the same tag group and command queue.

b Tag specific barrier.
Commands with 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.

l List command.
Executes a list of DMA-list elements located in local storage.

Table 7-2. SL1 Storage Control Commands

Mnemonic	Opcode	Support		Description
sdcrt	x'80'	Proxy	Channel	Brings a range of effective addresses into the SL1 (performance hint for DMA gets).[1]
sdcrtst	x'81'	Proxy	Channel	Brings a range of effective addresses into the SL1 (performance hint for DMA puts).[1]
sdcrz	x'89'	Proxy	Channel	Writes zeros to the contents of a range of effective addresses.
sdcrst	x'8D'	Proxy	Channel	Stores the modified contents of a range of effective addresses.
sdcrf	x'8F'	Proxy	Channel	Stores the modified contents of a range of effective addresses and invalidates the block.

1. These commands are treated as no-operations in implementations without an SL1.

Table 7-3. MFC Synchronization Commands

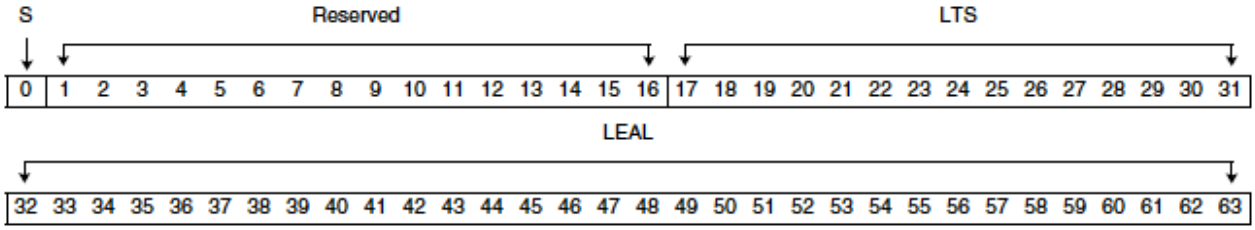
Command	Opcode	Support		Description
sndsig	x'A0'	Proxy	Channel	Update signal-notification registers in an I/O device or another SPU. [2]
sndsigf	x'A2'	Proxy	Channel	Update signal-notification registers in an I/O device or another SPU with fence. [2]
sndsigb	x'A1'	Proxy	Channel	Update signal-notification registers in an I/O device or another SPU with barrier. [2]
barrier	x'C0'	Proxy	Channel	Barrier type ordering. Ensures ordering of all preceding, non-immediate MFC commands with respect to all commands following the **barrier** command with the same command queue. The **barrier** command has no effect on the immediate MFC commands: **getllar**, **putllc**, and **putlluc**.
mfceieio	x'C8'	Proxy	Channel	Controls the ordering of **get** and **put** commands as shown below: • Orders **get** or **put** commands with respect to other **get** or **put** commands that access storage defined as caching inhibited and guarded (I=1,G=1). • Orders **put** commands that access storage defined as write through required (W=1) with respect to **put** or **get** commands that access storage defined as caching inhibited and guarded (I=1,G=1). • Orders **put** or **get** commands that access storage defined as caching inhibited and guarded (I=1,G=1) with respect to **put** commands that access storage defined as write through required (W=1). • Orders **put** commands with respect to other **put** commands that access storage that is defined as memory coherency required and is neither write through required nor caching inhibited (M=1,W=0,I=0). To ensure that the commands are correctly ordered, the commands must be in the same tag group as the mfceieio command, or a barrier command must be issued prior to the mfceieio command.
mfcsync	x'CC'	Proxy	Channel	Controls the ordering of DMA **put** and **get** operations within the specified tag group with respect to other processing units and devices in the system.

2. These commands are actually 4-byte DMA puts that can go to any address.

Table 7-4. MFC Atomic Commands

Command	Opcode	Support	Description
getllar	x'D0'	Channel	Get lock line and create a reservation (executed immediately).
putllc	x'B4'	Channel	Put lock line conditional on a reservation (executed immediately).
putlluc	x'B0'	Channel	Put lock line unconditional (executed immediately).
putqlluc	x'B8'	Channel	Put lock line unconditional (queued form).

DMA List Element



Bits	Field Name	Description
0	S	Stall-and-notify bit.
1:16	Reserved	Reserved.
17:31	LTS	List Element Transfer size (LTS).
32:63	LEAL	Low word of the 64-bit effective address (LEAL).

The maximum number of elements is 2048, and each element describes a transfer of up to 16K bytes.

8. SPE Memory Mapped Registers

8.1. Quick Reference for SPE Problem-State Memory Mapped Registers

Offset (Hexadecimal)	Register	Description	Access Type	See
Multi-source Synchronization Area				
x'00000'	MFC_MSSync	MFC Multi-source Synchronization Register	Read/Write	page 98
MFC Command Parameter Area				
x'03004'	MFC_LSA	MFC Local Storage Address Register	Write Only	page 91
x'03008'	MFC_EAH	MFC Effective Address High Register	Write Only	page 91
x'0300C'	MFC_EAL	MFC Effective Address Low Register	Write Only	page 91
x'03010'	MFC_Size	MFC Transfer Size Register (Upper 16-bits of register)	Write Only (must be written with using a single 32-bit store instruction)	page 91
	MFC_Tag	MFC Command Tag Register (Lower 16-bits of register)		
x'03014'	MFC_ClassID	MFC Class ID Register (Upper 16-bits of register, for write)	Write Only (must be written with using a single 32-bit store instruction)	page 92
	MFC_CMD	MFC Command Opcode Register (Lower 16-bits of register, for write)		
x'03014'	MFC_CMDStatus	MFC Command Status Register, (all 32-bits for read)	Read Only	page 92
MFC Command-Queue Control Area				
x'03104'	MFC_QStatus	MFC Queue Status Register	Read Only	page 93
x'03204'	Prxy_QueryType	Proxy Tag-Group Query Type Register	Read/Write	CBEA
x'0321C'	Prxy_QueryMask	Proxy Tag-Group Query Mask Register	Read/Write	CBEA
x'0322C'	Prxy_TagStatus	Proxy Tag-Group Status Register	Read Only	CBEA
SPU Control Area				
x'04004'	SPU_Out_Mbox	SPU Outbound Mailbox Register	Read Only	page 94
x'0400C'	SPU_In_Mbox	SPU Inbound Mailbox Register	Write Only	page 94
x'04014'	SPU_Mbox_Stat	SPU Mailbox Status Register	Read Only	page 94
x'0401C'	SPU_RunCntl	SPU Run Control Register	Read/Write	page 95
x'04024'	SPU_Status	SPU Status Register	Read Only	page 96
x'04034'	SPU_NPC	SPU Next Program Counter Register	Read/Write	page 97
Signal-Notification Area				
x'1400C'	SPU_Sig_Notify_1	SPU Signal Notification Register 1	Read/Write	CBEA
x'1C00C'	SPU_Sig_Notify_2	SPU Signal-Notification Register 2	Read/Write	CBEA

See the *Cell Broadband Engine™ Architecture* for the SPE privileged mode facilities.

8.2. Memory Mapped Registers Implementation Notes

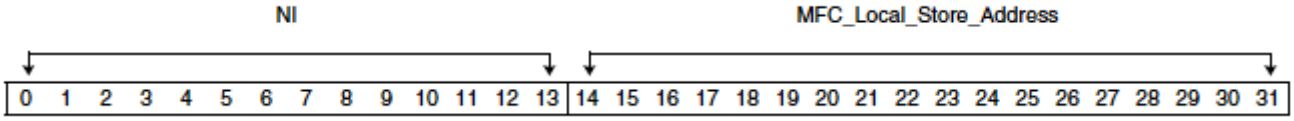
8.2.1. MFC Command Parameter Registers

These registers describe the MFC Command Parameter channels.

8.2.1.1. MFC Local Storage Address Register (MFC_LSA)

The MFC local storage address parameter stored in the MFC_LSA register is used to supply the SPU local storage address associated with a DMA command to be queued. This address is used as the source or destination of the DMA transfer as it is defined in the DMA command.
The contents of the MFC local storage address parameter are not persistent and must be written for each DMA command-enqueue sequence.
The validity of this parameter is checked asynchronous to the instruction stream. If the address is unaligned, MFC command queue processing is suspended, and an MFC DMA alignment exception is generated.

Note: Providing a local storage address above the implemented range of local store causes the local storage address to wrap around to a valid address, but no exception indicates that this condition has occurred.



Bit(s)	Field Name	Description
0:13	NI	Bits are defined in the *Cell Broadband Engine™ Architecture*, but are not implemented in the CBE. All bits read back zero.
14:31	MFC_Local_Store_Address	Bits [25:31] must always be aligned to a transfer-size boundary and the four least-significant bits of the local-storage address must match the four least-significant bits of the effective address.

8.2.1.2. MFC Effective Address High Register (MFC_EAH)

The validity of this parameter is checked asynchronous to the instruction stream. If a segment fault, mapping fault, or protection violation occurs, an MFC data segment exception is generated. If the address is not aligned, an MFC DMA alignment exception is generated.
See the *Cell Broadband Engine™ Architecture* for more information about this register.

8.2.1.3. MFC Effective Address Low Register (MFC_EAL)

The validity of this parameter is checked asynchronous to the instruction stream. If a segment fault, mapping fault, or protection violation occurs, an MFC data segment exception is generated. If the address is not aligned, an MFC DMA alignment exception is generated.
See the *Cell Broadband Engine™ Architecture* for more information about this register.

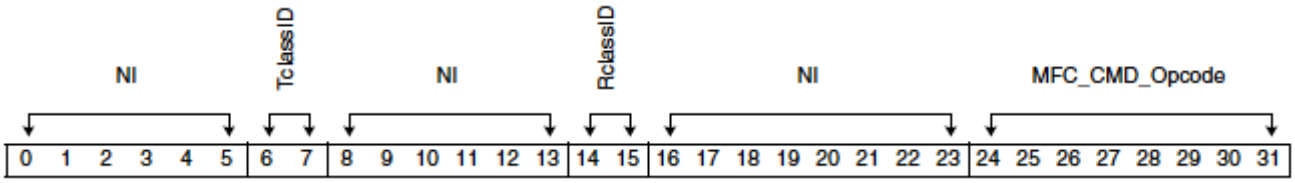
8.2.1.4. MFC Transfer Size Register (MFC_Size) and MFC Command Tag Register (MFC_Tag)

MFC_Size is the upper half of the MMIO word and MFC_Tag is the lower half of the same word. This word is written using a single 32-bit store instruction.
See the *Cell Broadband Engine™ Architecture* for more information about this register.

Programming Note: The architecture allows for a future increase in the MFC_Tag Register, along with the Prxy_QueryMask and Prxy_TagStatus Registers, to 7 bits.

8.2.1.5. MFC Class ID Command Opcode Register (MFC_ClassID_CMD)

This register is documented in the *Cell Broadband Engine™ Architecture* as two registers, MFC_ClassID and MFC_CMD. The MFC class ID parameter is used to specify the replacement class ID and the transfer class ID for each MFC command. The transfer class ID (TclassID) is used to identify access to storage with differing characteristics.
The MFC_ClassID_CMD Register, which is write only, is related to the MFC_CMDStatus Register, which is read only. See *MFC Command Status Register (MFC_CMDStatus)* for more information.



Bit(s)	Field Name	Description
0:5	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE. All bits read back zero.
6:7	TclassID	Transfer class identifier
8:13	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE. All bits read back zero.
14:15	RclassID	Replacement class identifier
16:23	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE. All bits read back zero.
24:31	MFC_CMD_Opcode	MFC command opcode

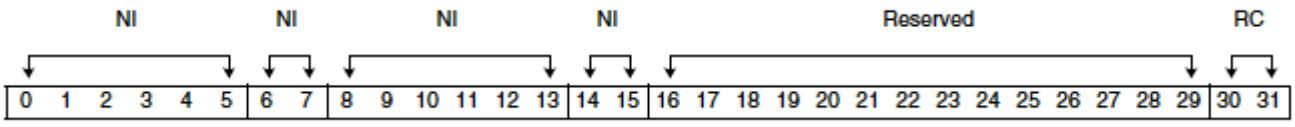
Programming Note: The total number of queue slots is implementation-specific and varies between implementations. For portability of an application, the enqueue sequence for MFC commands and the method to determine the number of queue slots available should be provided as a macro.

8.2.1.6. MFC Command Status Register (MFC_CMDStatus)

The MFC Command Status Register contains the return code from the last attempt to enqueue an MFC command. The return code is read from the same location as the command.

Note: The MFC Command Status Register is a read-only 32-bit register (the upper 16 bits are implementation specific). The MFC command return code in the least significant bits returns the command status when read.

The MFC_CMDStatus Register, which is read only, is related to the MFC_ClassID_CMD Register, which is write only. See *MFC Class ID Command Opcode Register (MFC_ClassID_CMD)* for more information.



Bit(s)	Field Name	Description
0:5	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE. All bits read back zero.
6:7	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE.. Bits return the last value written to this address (the TClass ID field, bits [6:7] of the MFC_ClassID_CMD Register).
8:13	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE. All bits read back zero.

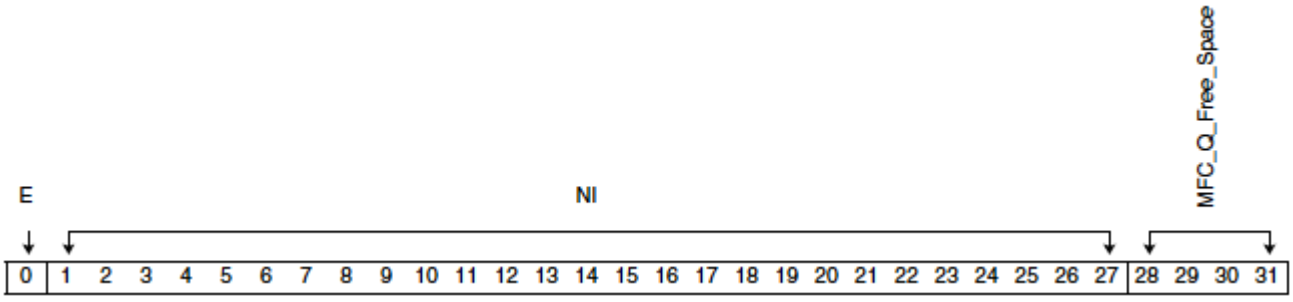
Bit(s)	Field Name	Description
14:15	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE.. Bits return the last value written to this address (the RClass ID field, bits [14:15] of the MFC_ClassID_CMD Register).
16:29	Reserved	All bits read back zero.
30:31	RC	MFC command return code '00' Command enqueue successful. '01' Command enqueue failed due to sequencing error. '10' Command enqueue failed due to insufficient space in the command queue (the free space in the command queue is zero). '11' Command enqueue failed due to sequencing error, and free space in the command queue is zero.

8.2.2. MFC Proxy Command Queue Control Registers

The registers in this section are used to control the MFC proxy command queue.

8.2.2.1. MFC Queue Status Register (MFC_QStatus)

The MFC Queue Status Register contains the current status of the MFC command queue. Bit zero of this register indicates whether the MFC proxy command queue is empty or contains valid commands that are not yet complete. The least-significant 4 bits of this register return the number of entries available in the MFC proxy command queue. A value of zero in this field indicates that the queue is full.



Bit(s)	Field Name	Description
0	E	MFC queue empty. All MFC operations are complete. 0 MFC Proxy Command queue contains commands 1 MFC Proxy Command queue does not contain commands
1:27	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE. All bits read back zero.
28:31	MFC_Q_Free_Space	MFC queue free space This field contains the number of queue entries available. Software can use this field to set a loop count for the number of MFC commands to enqueue. Software must not assume a command is enqueued based on the free space. Other conditions may cause the command issue sequence to fail. See the *Cell Broadband Engine™ Architecture* for more information.

8.2.2.2. Proxy Tag-Group Query Type Register (Prxy_QueryType)

The Proxy Tag-Group Query Type Register is used by software to request that the MFC detect a tag-group completion condition.
See the *Cell Broadband Engine™ Architecture* for more information about this register.

8.2.2.3. Proxy Tag-Group Query Mask Register (Prxy_QueryMask)

The Proxy Tag-Group Query Mask Register selects the tag groups to be included in the query operation.
See the *Cell Broadband Engine™ Architecture* for more information about this register.

8.2.2.4. Proxy Tag-Group Status Register (Prxy_TagStatus)

The Proxy Tag-Group Status Register contains the current status of the tag groups enabled in the Proxy Tag-Group Query Mask Register.
See the *Cell Broadband Engine™ Architecture* for more information about this register.

8.2.3. SPU Control Registers

8.2.3.1. SPU Outbound Mailbox Register (SPU_Out_Mbox)

Other processors or devices read the mailbox data from the SPU in the SPU_Out_Mbox Register. The SPU sends data to this mailbox by writing to the SPU_WrOutMbox Channel (see *SPU Write Outbound Mailbox Channel* (*SPU_WrOutMbox*) in the *Cell Broadband Engine™ Architecture* document for more information). When the SPU writes to the SPU_WrOutMbox Channel, the channel counter decrements from 1 to 0. Reading this register causes the SPU_WrOutMbox channel counter to increment to 1. While the channel count is 1, further reads to this register do not affect the count, and the read data is the last stored value in this register. The queue depth for this implementation is 1.

Note: For the SPU Write Outbound Mailbox Channel (SPU_WrOutMbox), the count is 1 at POR, and software reinitializes the count to 1.

The SPU Outbound Interrupt Mailbox Register (SPU_OutIntrMbox) has the same behavior as this register, except that it has a Mailbox Interrupt that allows the SPU to notify the PPU when the SPU has written data to the SPU_WrOutIntrMbox Channel.

See the *Cell Broadband Engine™ Architecture* for more information about this register.

8.2.3.2. SPU Inbound Mailbox Register (SPU_In_Mbox)

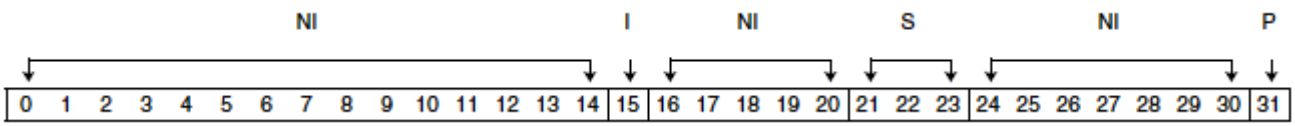
The PPE writes mailbox data to the SPU in the SPU_In_Mbox Register. This register corresponds to the SPU_RdInMbox Channel. If this register is full, then additional writes overwrite the last entry written. The channel count remains at 4. See the description of the *SPU Read Inbound Mailbox (SPU_RdInMbox) Channel* on page 85 for more information.

The SPU Inbound Mailbox Threshold interrupt is used to notify the PPE when the SPU has read all of the SPU_In_Mbox data. The SPU Inbound Mailbox Threshold interrupt is asserted when the SPU_RdInMbox channel count transitions from 1 to 0 (SPU_In_Mbox empty), and deasserted when the SPU_RdInMbox channel counter increments from 0 to 1. This interrupt is almost always asserted. The interrupt is taken when it is asserted and enabled.

See the *Class 2 Interrupt Mask Register (INT_Mask_class2)* in the *Cell Broadband Engine™ Architecture* for information about how to enable this interrupt. See the *Cell Broadband Engine™ Architecture* for more information about the SPU Inbound Mailbox Threshold interrupt.

8.2.3.3. SPU Mailbox Status Register (SPU_Mbox_Stat)

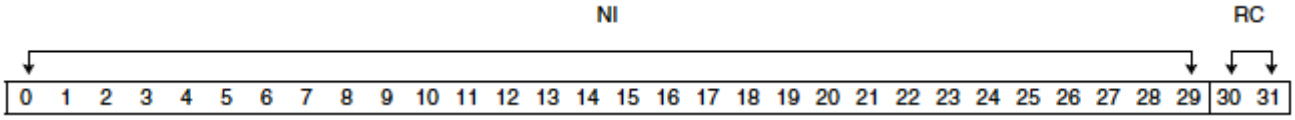
The SPU_Mbox_Stat Register contains the current state of the mailbox queues between the SPU and other processors and devices.



Bit(s)	Field Name	Description
0:14	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE. All bits read back zero.
15	I	SPU Outbound Interrupt Mailbox status is equal to 1 minus the SPU_WrOutIntrMbox channel count. This status bit is set when the SPU Outbound Interrupt Mailbox is written by the SPU, and reset when the PPU reads the SPU Outbound Interrupt Mailbox 0 SPU Outbound Interrupt Mailbox empty 1 SPU Outbound Interrupt Mailbox contains a new value
16:20	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE. All bits read back zero.
21:23	S	SPU Inbound Mailbox status is equal to 4 minus the SPU_RdInMbox channel count. This status decrements when the SPU Inbound Mailbox is written by the PPU, and increments when the SPU reads the SPU Inbound Mailbox. '000' SPU Inbound Mailbox full '001' SPU Inbound Mailbox has one location available to load '010' SPU Inbound Mailbox has two locations available to load '011' SPU Inbound Mailbox has three locations available to load '100' SPU Inbound Mailbox has four locations available to load '101' through '111' are unused.
24:30	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE. All bits read back zero.
31	P	SPU Outbound Mailbox status is equal to 1 minus the SPU_WrOutMbox channel count. This status bit is set when the SPU Outbound Mailbox is written by the SPU, and reset when the PPU reads the SPU Outbound Mailbox. 0 SPU Outbound Mailbox empty 1 SPU Outbound Mailbox contains a new value

8.2.3.4. SPU Run Control Register (SPU_RunCntl)

The SPU_RunCntl Register is used to start and stop the execution of instructions in the SPU. The SPU can dynamically change the state of the run bit. The current status of the SPU run state is available in the SPU Status Register (SPU_Status). When this register is read, it returns the last data written for the last valid write.



Bit(s)	Field Name	Description
0:29	NI	Bits are not implemented. All bits read back zero.
30:31	RC	SPU run control '00' SPU stop request. No instructions are issued. '01' SPU run request. Instruction is issued if not stalled on condition. '10' SPU isolate exit request. '11' SPU isolate load request. The current status of the SPU run state is available in the SPU_Status Register.

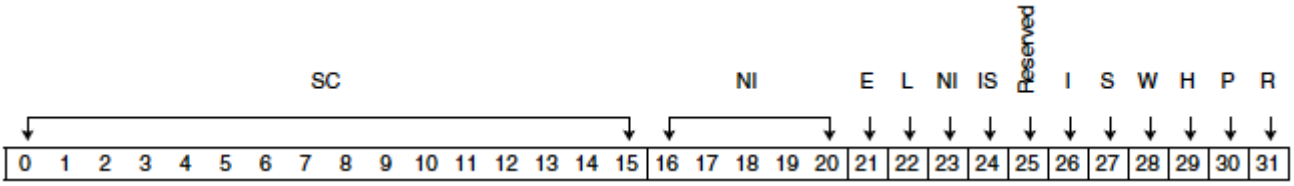
Programming Note: After SPU_RunCntl[30:31], the run control bit, is set to stop the SPU, the SPU is not stopped until SPU_Status[31], the run status bit, reads '0'. See *SPU Status Register (SPU_Status)* for more information. A write of '01' while the SPU is idle causes the SPU to restart from the Program Counter at which it stopped (including in isolation state), except if the SPU stopped from an error condition (illegal opcode or uncorrectable ECC) while in Isolation state. Then, if the SPU is stopped in isolation state because of an error, a write of '01' has no effect and a write of '10' or '11' is required to restart the SPU. The SPU ignores writes of '01', '10', or '11' unless the SPU is idle, as indicated when SPU_Status[31], the R bit, reads '0'.

8.2.3.5. SPU Status Register (SPU_Status)

Reading this register provides a snapshot of the current SPU state. The status read can be dynamically changing if the SPU is currently running. If the SPU is stopped or halted, the status remains static until the SPU state is changed by software. If the SPU was stopped under PPU control at the same time that the SPU was waiting on a blocked channel, the SPU Wait Status is set in conjunction with the SPU Stopped status. Multiple state bits ([26:27] and [29:30]) may be set based on the program design.

When SPU_Status[31] in this register transitions to '1', the states for SPU_Status[26:27] and SPU_Status[29:30] are reset to '0'.

Note: For more information on the bits below that reference the isolate load or isolate exit states, see the *SPU Isolation Facility* section in the *Cell Broadband Engine™ Architecture* document.



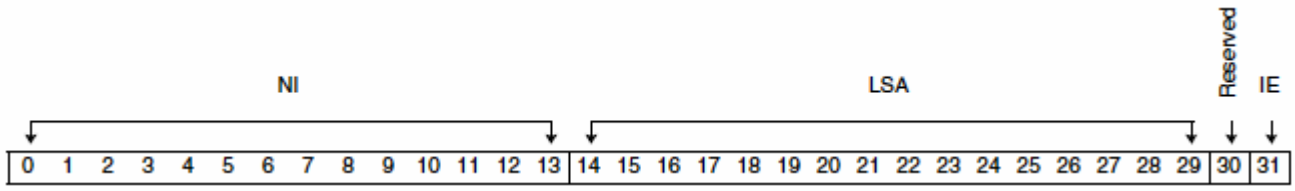
Bit(s)	Field Name	Description
0:15	SC	If the SPU_Status[30], the P bit, which is the stop and signal indication, is set to '1', this field provides a copy of bits[18-31] of the SPU stop and signal (**stop**) instruction that caused the SPU stop. Bits [0:1] of this field are always set to '0'. If SPU_Status[30] is not set, data in this field is not valid. A stop and signal with dependencies (**stopd**) instruction, used for debugging, always sets each of bits[2:15] to '1'.
16:20	NI	Bits are not implemented. All bits read back zero.
21	E	SPU isolate exit status 0 The SPU is not performing an isolate exit 1 The SPU is performing an isolate exit
22	L	SPU isolate load status 0 The SPU is not performing an isolate load 1 The SPU is performing an isolate load
23	NI	Bit is not implemented; bit reads back zero.
24	IS	SPU isolation status 0 The SPU is in a nonisolated state 1 The SPU is in an isolated state
25	Reserved	This bit is defined in the *Cell Broadband Engine™ Architecture*, but not implemented in the CBE; bit reads back zero. See *3.2.2.1 Read and Write Channel Implementation Difference from the SPU ISA* on page 42 for details.
26	I	Invalid Instruction Detected (The SPU does not stop precisely and the *SPU Next Program Counter Register (SPU_NPC)* may not indicate the instruction after the illegal instruction) 0 No illegal opcodes have been issued 1 An illegal opcode has been issued, and the SPU has been halted. An SPU error interrupt is also generated in INT_Class0[61] if enabled in SPU_ERR_Mask[63].
27	S	SPU Single-Step Status (The *SPU Next Program Counter Register (SPU_NPC)* points to the next instruction after the instructions committed for the single step operation) 0 SPU Not Stopped due to single-step mode 1 SPU Stopped after one completed instruction in single-issue mode or pair of instructions in dual-issue mode
28	W	SPU Wait Status 0 SPU is not waiting on a blocked channel 1 SPU is waiting on a blocked channel

Bit(s)	Field Name	Description
29	H	SPU Halt Status (The SPU does not stop precisely and the *SPU Next Program Counter Register (SPU_NPC)* may not indicate the instruction after the halt instruction). Is not set if SPU stops due to Single Step. 0 SPU Not Halted due to a halt instruction 1 SPU Halted due to a halt instruction
30	P	SPU Program Stop and Signal Status (The *SPU Next Program Counter Register (SPU_NPC)* points to the mext instruction after the commited stop-and-signal instruction) 0 SPU Not Stopped due to stop-and-signal instruction 1 SPU Stopped due to stop-and-signal instruction
31	R	SPU Run Status 0 SPU Stopped or Halted (idle) 1 SPU Running

8.2.3.6. SPU Next Program Counter Register (SPU_NPC)

The Local Store Address (LSA) value read from this register is limited by the value of the AMR field of the SPU Local Storage Limit Register (SPU_LSLR) Register.

A read from this register is only valid when the SPU is stopped (SPU_Status Register[31] is set to 0) and if the SPU is in non-isolate state. Otherwise, it returns meaningless data (all zeros). Values written to this register while the SPU is running or in Isolate State have no effect on the operation of the SPU and are ignored. SPU Interrupts can only be enabled in this register before starting the SPU and cannot be enabled while the SPU is running. That is, the internal enable bit gets its value from this register when the SPU starts running. When the SPU is stopped, the internal enable bit is loaded into this register and may have been changed during program execution by an indirect branch.



Bit(s)	Field Name	Description
0:13	NI	Bits defined in the *Cell Broadband Engine™ Architecture*, not implemented in the CBE. All bits read back zero.
14:29	LSA	Word-aligned local-store address (LSA).
30	Reserved	Bit is not implemented; bit reads back zero.
31	IE	SPU interrupt enable prior to start-up, or status of the enable when the SPU is stopped. 0 SPU interrupts disabled. 1 SPU interrupts enabled.

Programming Note: SPU_NPC is not valid for an illegal instruction.

8.2.4. SPU Signal Notification Registers

8.2.4.1. SPU Signal Notification Register 1 (SPU_Sig_Notify_1)

See the *Cell Broadband Engine™ Architecture* for more information.

8.2.4.2. SPU Signal Notification Register 2 (SPU_Sig_Notify_2)

See the *Cell Broadband Engine™ Architecture* for more information.

8.2.5. MFC Multisource Sync Register

8.2.5.1. MFC Multisource Synchronization Register (MFC_MSSync)

This register is the interface to the MFC Multisource Synchronization facility. See the *Cell Broadband Engine™ Architecture* document for more information on this facility. Writing any value to this register requests a synchronization. At the time of the write, the MFC starts to track all outstanding transfers targeting the corresponding SPE. When read, this register returns the current status of the last request. A value of zero will be returned when all transfers targeting the SPE and received before the last write of the MFC_MSSync Register are complete.

9. MFC Command Issue Sequence

9.1. MFC SPU Command Issue Sequence

To queue a MFC SPU command from the SPU, the MFC SPU command parameters must first be written to the MFC SPU command parameter channels. The following command parameters can be written in any order, except that step 6 must always be done last:

1. Write the local storage address parameter (32 bits).
2. Write the effective address high parameter (upper 32 bits).*1
3. Write the effective address low or list address parameter (lower 32 bits).
4. Write the MFC SPU transfer or list size parameter (16 bits).
5. Write the MFC SPU command tag parameter (16 bits).
6. Write the MFC SPU command opcode and class ID parameter (32 bits).*2

The MFC SPU command parameters are retained in the MFC SPU command parameter channels until a write of the MFC SPU command opcode and class ID parameter is processed by the MFC.

A write channel (**wrch**) instruction targeted to the MFC Command Opcode and Class ID Channel causes the parameters held in the MFC SPU command parameter channels to be sent to the MFC SPU command queue. The MFC SPU command parameters can be written in any order before the issue of the MFC SPU command itself. The values of the last parameters written to the MFC SPU command parameter channels are used in the enqueuing operation.

After an MFC SPU command has been queued, the values of the MFC parameters become invalid and must be respecified for the next MFC command queuing request. Not specifying all of the required MFC parameters (that is, all the parameters except for the optional EAH) can result in the improper operation of the MFC command queue.

The MFC SPU command parameter channels, with the exception of the MFC Command Opcode and Class ID Channel, are non-blocking. They do not have channel counts associated with them. A read channel count (**rchcnt**) instruction that targets these channels returns a count of 1.

The MFC Command Opcode and Class ID Channel has a maximum count configured by hardware to the number of MFC SPU queue commands supported by hardware. Software must initialize the channel count of the MFC Command Opcode and Class ID Channel to the number of empty MFC SPU command queue slots supported by the implementation after power on and after a purge of the MFC SPU command queue. The channel count of the MFC Command Opcode and Class ID Channel must also be saved and restored on a SPE preemptive context switch.

*1. This parameter is optional and is set to zero if not written.
*2. This write channel (**wrch**) instruction stalls the SPU until there is room in the MFC queue for the command.

9.2. MFC Proxy Command Issue Sequence

The CBEA requires software to follow a particular sequence to enqueue an MFC proxy command. If the correct sequence is not followed, an MFC proxy command is not queued and an invalid command sequence is posted in the *MFC Command Status Register (MFC_CMDStatus)* (see page 92) (that is, MFC_CMDStatus[RC] = '01' or '11').

The MFC Command Parameter Registers must be written in increasing address order. To enqueue an MFC proxy command from the PPE, the MFC parameters must first be written to the MFC proxy command address space in the following sequence:

1. Set the local storage address.
2. Set the MFC effective address. *1
3. Set the MFC size and the MFC tag.*2
4. Set the transfer class and replacement management class IDs and the MFC proxy command.*2
5. Read the MFC proxy command status.*3
   Reading the MFC proxy command status causes an attempt to enqueue the specified command and its parameters.
6. Restart the MFC proxy command issue sequence, starting at step 1, if the status indicates failure, or if there is no slot available in the MFC proxy command queue.

These parameters are held in the corresponding registers. The read of the MFC proxy command status causes the data held in these registers to be enqueued, if the sequence has not been interrupted and if there is a slot in the MFC proxy command queue.

*1. The effective address can be written using either one 64-bit store or two 32-bit stores. If written using 32-bit stores, the EAH must be written first.
*2. The *MFC Transfer Size Register (MFC_Size) and MFC Command Tag Register (MFC_Tag)* (see page 91) are a pair of registers. The *MFC Class ID Command Opcode Register (MFC_ClassID_CMD)* (see page 92) consists of a pair of registers. Each pair must be updated using a single store instruction.
*3. Reading the *MFC Command Status Register (MFC_CMDStatus)* (see page 92) causes the parameters to be enqueued if the sequence is correct.

Implementation Note:

When the local storage address is set in step 1, hardware should set an internal state bit, CMD_Pending. This bit indicates that a command is pending for the queue.
The CMD_Pending bit is reset if the required sequence for enqueuing an MFC proxy command is not followed. (If the local storage address is set again, then CMD_Pending is set again instead of being reset.)
An MFC proxy command fails if the CMD_Pending bit is reset when the MFC Command Status Register is read in step 5. (An MFC proxy command can also fail if there is insufficient room in the MFC proxy command queue.)
If the CMD_Pending bit is still set when the MFC Command Status Register is read in step 5, the command is enqueued. Once the command is enqueued, the CMD_Pending bit is reset.