

## PowerPC 405 APU Controller

---

This chapter only applies to the PowerPC 405 processor in the Virtex-4-FX family and covers the following topics:

- “FCM Instruction Processing”
- “APU Controller Configuration”
- “Interface Definition”
- “FCM Interface Timing Specification”

**Note:** The Auxiliary Processor Unit (APU) controller is not available in the Virtex-II Pro family.

### Introduction

The Auxiliary Processor Unit (APU) controller allows the designer to extend the native PowerPC 405 instruction set with custom instructions that are executed by an FPGA Fabric Co-processor Module (FCM). This enables a much tighter integration between an application-specific function and the processor pipeline than is possible using, for example, a bus peripheral. [Figure 4-1](#) shows the pipeline flow between the PowerPC 405 Core, the APU controller, and the Fabric Co-processor Module.

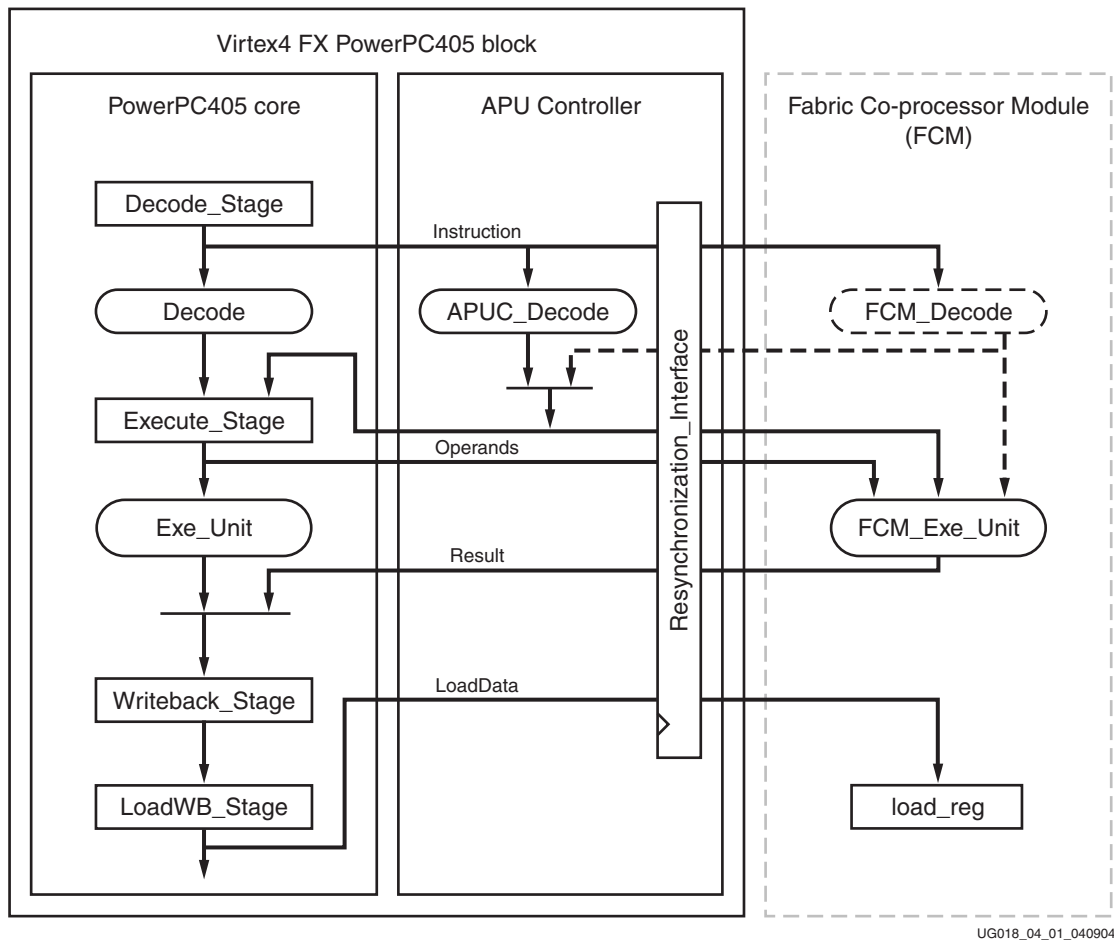


Figure 4-1: Pipeline Flow Diagram

The APU controller serves two purposes: to perform clock domain synchronization between the fast PowerPC clock and the slow FCM interface clock, and to decode certain FCM instructions and notify the CPU of the CPU resources needed by the instruction (for example source data from the CPU's GPR). Depending on the FCM application, the APU controller can decode all instructions or no instructions at all, or decode some while the FCM decodes others. A floating-point unit (FPU) is an example of a good FCM candidate. In the case of an FCM FPU, the APU controller is capable of decoding all PowerPC floating point instructions.

The FCM interface is a Xilinx adaptation of the native APU interface implemented on the IBM processor core. The hard core APU controller bridges the PowerPC 405 APU interface and the external FCM interface.

## FCM Instruction Processing

FCM instruction decoding can be done by the APU controller or by the FCM. APU controller decoding means the APU controller determines what CPU resources are needed for the instruction execution and passes this information to the CPU. For example, the APU controller will determine if an instruction is a load, a store, or if it needs source data from the GPR, etc. The FCM can also perform this part of decoding and pass the needed

information to the APU controller. However, in order to determine the complete function of the instruction, the FCM must perform a full decode as well as handle all instruction execution. There are two types of instructions that can be executed by an FCM: pre-defined and user-defined (UDI). A pre-defined instruction has its format defined by the PowerPC instruction set (for example, floating point), and the FCM is simply a co-processor performing the ISA-defined execution. A user-defined instruction has a configurable format and is a true extension of the PowerPC instruction set architecture (ISA).

## Enabling the APU Controller

The PowerPC MSR register must be configured before the processor can use the APU controller. Table 4-1 describes the APU controller-related bits in the MSR.

**Table 4-1: APU Controller-Related MSR Bits**

Bit(s) in MSR	Description
6	APU present (1=true, 0=false)
12	Enable APU exception (1=true, 0=false)
18	FCM floating point unit present (1=true, 0=false)
(20,23)	Floating point exception mode (FE0,FE1): <ul style="list-style-type: none"> <li>• (0,0) Ignore FP exceptions</li> <li>• (1,0) Imprecise recoverable mode</li> <li>• (0,1) Imprecise non-recoverable mode</li> <li>• (1,1) Precise mode</li> </ul>

## Instruction Classes

The ISA extensions to the PowerPC are defined by their interaction with the normal processor pipeline execution. This leads to three different instruction classes: autonomous, non-autonomous blocking, and non-autonomous non-blocking.

### Autonomous Instructions

Instructions in the autonomous class do not stall the pipeline of the PowerPC. They are typically fire-and-forget type instructions that are not expected to return any state (for example, overflow) or data to the processor pipeline. An example is a user-defined UDI\_FCM\_Read instruction<sup>(1)</sup>, where an FCM register is loaded with the contents of one of the PowerPC GPR registers without returning any data to the processor. Although autonomous instructions do not stall execution of native instructions, they can stall execution of subsequent FCM instructions in case the FCM is not done with an earlier instruction.

### Non-autonomous Instructions

A non-autonomous instruction will stall normal instruction execution in the PowerPC pipeline until the FCM instruction is done. This is typical for instructions that are expected to return some state (e.g. overflow) or data to the PowerPC. For example a user-defined

1. Note that this would not be the same as the “Load” instructions that operate on the storage hierarchy, such as caches, OCM, or PLB.

UDI\_FCM\_Write instruction that takes data from the FCM and writes it to a PowerPC GPR location.

### Blocking Instructions

Any non-autonomous instruction that cannot be predictably aborted and later re-issued must be blocking. Once a blocking instruction has completed the first Execute cycle in the PPC pipeline (the same cycle source operands are ready), all interrupts and exceptions to the PowerPC are blocked, so as not to prevent the instruction from completing. This is, for example, true of the UDI\_FCM\_Write instruction if the source of the data is a FIFO inside the FCM. If aborted after the FIFO pointer has been changed, but before the data has been stored in the PowerPC register file, such instruction could not be re-issued predictably.

### Non-blocking Instructions

Any non-autonomous instruction that can be aborted and predictably re-issued later can be defined as non-blocking. A non-blocking instruction allows the processor to terminate the FCM execution, service interrupts and exceptions, and subsequently re-issue the terminated instruction, with predictable results. If we replace the FIFO in the blocking example above with a traditional random access memory, the aborted UDI\_FCM\_Write instruction could be predictably re-issued (with no remaining side-effects associated with a FIFO read pointer).

## Instruction Format

All FCM instructions conform to the general format shown in Figure 4-2.

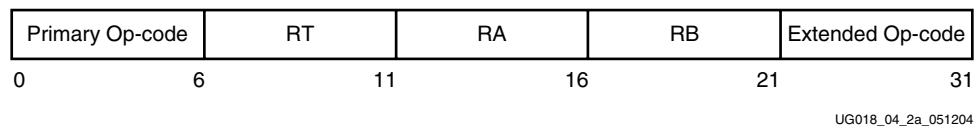


Figure 4-2: FCM Instruction Format

Generally speaking, the PowerPC uses both primary and extended op codes to identify potential FCM instructions. The op codes are decoded by the APU controller or the FCM to identify uniquely the specific FCM instruction. For all pre-defined instructions, the RA and RB fields specify operand registers, and the RT field the target register. User-defined instructions (UDI) can be configured to interpret these bit fields as, for instance, immediate values instead.

The primary and secondary op-codes shown in Table 4-2 can be used as APU instructions:

Table 4-2: APU Op-codes

Primary Op-code	Extended Op-code	Description
0 (= 0b0000000)	0b000000000000	Illegal
	all except above	Available for UDIs <sup>(1)</sup> that do not set PPC405(CR) bits

Table 4-2: APU Op-codes (Continued)

Primary Op-code	Extended Op-code	Description
4 (= 0b000100)	0b-----1--0-	MAcc and Xilinx reserved
	0b1----000110	Available for UDIs that need to set PPC405(CR) bits
	all except above	Available for UDIs that do not set PPC405(CR) bits
31 (= 0b011111)	0b-----001110	Pre-defined FCM Load/Store
	0b-111-010-1-	FCM integer divide
(= 0b110---)(2)	0b-----	Pre-defined FPU Load/Store
32 (= 0b011111)	0b1----1-111-	Pre-defined FPU Load/Store
59 (= 0b111011)	0b-----	Pre-defined PowerPC FPU instructions
62 (= 0b111110)	0b-----1--	Pre-defined FPU Load/Store
63 (= 0b111111)	0b-----	Pre-defined PowerPC FPU instructions

**Notes:**

1. User-defined Instruction. For details refer to the “[User-Defined Instruction Decoding](#)” section of this chapter.
2. In this case, the first three bits are defined and the last three will change depending on the FPU instruction.

## Instruction Decoding

FCM instructions can be decoded either by the APU controller or by the FCM itself. As mentioned earlier in the “[Introduction](#)”, the main purpose of APU controller decoding is to pass needed resource information to the CPU.

APU controller decoding benefits from the higher clock frequencies possible inside the hard core. This results in a minimum of latency overhead in the decode stage, improving overall performance. The APU controller can decode two types of FCM instructions: pre-defined instructions that are hard coded in the APU controller and a limited number of user-defined instructions.

FCM decoding, although slower than its counterpart, allows many more user-defined instructions to be implemented.

There are also several signals sent to the FCM when the APU controller decodes an instruction. The first signal is called APUFCMDECODED. When this signal is gated with APUFCMINSTRVALID it indicates the APU controller has notified the CPU of the instruction and the resources required. The signal is especially useful in the situation where the FCM receives an FCM instruction that it cannot execute (for example if only part of the FPU instruction set is implemented). The “[Floating Point Instructions](#)” section gives more details on this situation. There is one exception to APUFCMDECODED. Due to the

nature of store instructions, the user should not expect to receive APUFCMDECODED when the APU controller decodes a store. Instead the FCM must decode for a store instruction.

The two other signals sent to the FCM after an APU controller decode are APUFCMDECUDI[0:2] and APUFCMDECUDIVALID. These signals are explained further in the “User-Defined Instruction Decoding” section.

## APU Controller Pre-Defined Instruction Decoding

Two types of pre-defined instructions can be decoded by the APU controller: Floating point and FCM Load/Store.

### Floating Point Instructions

The APU controller can be enabled to decode all PowerPC floating-point instructions. In addition to this, three groups of floating point instructions can be selectively disabled: the complex arithmetic, conversion, and estimates groups.

#### Complex Arithmetic Group

- fdiv
- fdivs
- fsqrt
- fsqrts
- fdiv.
- fdivs.
- fsqrt.
- fsqrts.

#### Conversion Group

- fcfid
- fctidz
- fctiw.
- fctiwz.
- fctid
- fctiw
- fctiwz

#### Estimates Group

- fres
- fres.
- frsqрте
- frsqрте.

The decoded instructions require an FCM floating point unit to be used. FPU instructions that return results to the PowerPC will default to execute as non-autonomous, non-blocking. All other FPU instructions default to execute as autonomous. The user can force FPU instructions to be non-blocking in an APU controller configuration register.

If the FCM is designed such that there are FPU instructions that cannot be executed and are not included in one of the three groups users can selectively disable (listed above), the FCM must make use of the APUFCMDECODED signal. If the FPU receives an instruction it cannot execute and APUFCMDECODED is High (along with APUFCMINSTRVALID), the FPU must generate an exception by setting FCMAPUEXCEPTION High. It is assumed that all FPU store instructions can be executed and therefore the FPU should not expect to receive APUFCMDECODED along with a store instruction. Instead, if an FPU cannot execute an FPU store instruction the FPU is expected to decode the store instruction and generate an exception by setting FCMAPUEXCEPTION High.

**Note:** While the APU controller decodes these instructions, the FCM has to decode them independently for its own execution. The APU can send the 32-bit instruction, but it cannot tell the FCM which FPU instruction it decoded.

## FCM Load/Store Instructions

FCM Load/Store instructions transfer data between the PowerPC processor's data memory system (D-Cache or DSPLB/DSOCM addressable memory) and the FCM. An FCM Load transfers data from a memory location to a destination register in the FCM and vice-versa for an FCM Store. All Load/Store instructions are of indexed format, that is, RA stores the base address and RB the offset. Furthermore, RA can either be the contents of RA or the number 0. R0 cannot be used as RA for loads or stores.

FCM Load/Store should not be confused with user-defined FCM read/write instructions. A user-defined FCM read that transfers data from the PowerPC to the FCM will access data from the PowerPC GPR operand registers not from DSOCM or DSPLB memory. The same is true for a user-defined FCM write instruction.

The FCM Load/Store instructions behave somewhat differently in comparison with other FCM instructions. In a way, they are semi-autonomous because the PowerPC CPU is responsible for performing the necessary memory access involved. That is, the processor pipeline is executing, but it is executing a memory access related to the Load/Store instruction. Since FCM Load/Store instructions can be flushed from the CPU's pipeline, the APU controller is responsible for signaling the FCM when it is safe to commit internal state changes.

For details regarding instruction flushing refer to the [“FCM Instruction Flushing”](#) section of this chapter.

**Note:** While the APU controller decodes Load/Store instructions, the FCM has to decode them independently for its own execution. The APU can send the 32-bit instruction, but it cannot tell the FCM which FPU instruction it decoded.

The extended op-code for Load/Store operations are described in [Table 4-3](#).

**Table 4-3: Load/Store Extended Op-code**

Field	Bit position	Description
U	21	Update: If 1 then load RA with effective address: $RA \leftarrow (RA \mid 0) + (RB)$
W[0:2]	(22,24:25)	0b000 = Byte 0b001 = Half-word 0b010 = Word 0b-11 = Quad-word 0b100 = Double-word 0b101, 0b110 = illegal
L/S	23	0 = Load 1 = Store
-	(26:31)	hard coded 0b001110

APU controller Load/Store instruction decoding can be disabled in the APU controller configuration register.

The PowerPC405 native VMX instructions are a subset of the supported FCM Load/Store instructions.

## User-Defined Instruction Decoding

In addition to the pre-defined instructions described previously, the user can also define up to eight custom instructions to be decoded by the APU controller. The instructions conform to the same standard FCM format presented earlier, however, the interpretation of the RA, RB, and RT fields are up to the FCM. The UDI interaction with the PowerPC 405 pipeline is defined in the APU controller UDI configuration registers. When there are user instructions being decoded by the APU, the FCM will receive two important signals (along with the 32-bit instruction). The FCM will receive APUFCMDECUDI[0:2] (the bit encoded UDI register number) and APUFCMDECUDIVALID the same cycle as APUFCMINSTRVALID. APUFCMDECUDI[0:2] indicates which UDI register was just decoded in the APU controller and the APUFCMDECUDIVALID signal tells the FCM this bus is valid. These signals allow the FCM to simply decode 3 bits to determine the instruction instead of decoding all 32 bits of instruction. For details refer to the “[APU Controller Configuration](#)” section in this chapter.

## FCM Pre-Defined Instruction Decoding

There is one group of pre-defined PowerPC instructions that can be configured to be decoded in the FCM: integer divide instructions.

### Integer Divide Instructions

The PowerPC integer divide instruction constitutes a special case. While it would normally be executed in the PowerPC natively (consuming 35 cycles), the APU controller can be configured to give the FCM ownership of decoding and executing integer divide instructions (listed below). See the section “[APU Controller Configuration](#),” page 194, for details on enabling the FCM divide.

- |         |          |          |           |
|---------|----------|----------|-----------|
| • divd  | • divduo | • divwo. | • divwuo. |
| • divdo | • divw   | • divwu  |           |
| • divdu | • divw.  | • divwu. |           |
|         | • divwo  | • divwuo |           |

## FCM User-Defined Instruction Decoding

User-defined instructions that are not recognized (i.e., decoded) by the APU controller are passed to the FCM for decoding in fabric logic. While this allows for more custom instructions than the eight APU controller decoded UDIs to be defined, additional instructions come at an execution speed penalty. Decoding in the FCM is not as efficient as in the APU controller.

FCM decoded UDI instructions adhere to the same configuration rules as those decoded by the APU controller.

## FCM Exceptions

The FCM can signal an exception (FCMAPUEXCEPTION) to the APU controller while executing store, blocking, or non-blocking instructions. This causes the CPU to flush the instruction from the pipeline, and the APU controller to flush the FCM instruction (see “[FCM Instruction Flushing](#)”), and the PowerPC to launch the appropriate exception handler, provided the PowerPC MSR enables APU exceptions (see “[Enabling the APU Controller](#)”).



To execute the exception routine, the PowerPC saves the return program counter in its SSR0 register and the current value of MSR in the SSR1 register. The exception vector used for FCM exceptions is 0x700. When an exception occurs during the execution of a floating point instruction, bit 12 in the PowerPC ESR register is asserted. For exceptions during all other types of instructions, bit 13 in the ESR is asserted instead.

To return from the exception the FCM must provide the processor some way to strike down the FCMAPUEXCEPTION signal from the exception handler. This could be done using, for example, a UDI or an external DCR bus access.

In the FPU decode section it stated that if an FPU instruction was decoded by the APU controller but cannot be executed in the FPU, the FPU should respond by signaling FCMAPUEXCEPTION. This exception will only be precise if the instruction is blocking, non-blocking, or a store. In this situation the user can force all FPU non-storage instructions to be non-blocking (see [“APU Controller Configuration”](#) for more details on how to set this option). If an autonomous or load instruction causes an exception FCMAPUEXCEPTION can be set High, but the exception will not be precise. The CPU will not see the exception signal until the next FCM/FPU instruction is decoded.

## CPU Exceptions

An FCM instruction can cause several different CPU exceptions. The possible exceptions are shown in [Table 4-4](#).

**Table 4-4: CPU Exceptions Due to FCM Instructions**

Exception	Interrupt Type	Description
Illegal Instruction	Program	Attempted execution of an instruction not recognized by either the CPU or the FCM. Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 0700}, ESR[4] set.
Privileged Instruction	Program	Occurs when the processor is operating in the problem state (user mode MSR[17] = 1'b1) and the execution of a privileged instruction is attempted. Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 0700}, ESR[5] set.
APU Enabled	Program	Occurs when MSR[12] = 1'b1 and the FCM is asserting FCMAPUEXCEPTION. Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 0700}, ESR[13] set.
FPU Enabled Exception	Program	Occurs when MSR[20] and/or MSR[23] are set and the FPU instruction generated FCMAPUEXCEPTION. Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 0700}, ESR[12] set.
APU Unavailable	APU Unavailable	Occurs when attempting to execute a UDI while MSR[6] = 1'b0. Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 0F20}, no ESR bit set.

Table 4-4: CPU Exceptions Due to FCM Instructions

Exception	Interrupt Type	Description
FPU Unavailable	FPU Unavailable	Occurs when attempting to execute an FPU instruction while MSR[18] = 1'b0. Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 0800}, no ESR bit set.
Operand	Alignment	Occurs when there is any misaligned data access by FCM storage instruction. Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 0600}, DEAR loaded with data address that caused access violation, no ESR bit set.
Unsupported Endian	Alignment	Occurs when an FCM Load/Store attempts access with an endian attribute not supported by hardware (for example using option TrapLE or TrapBE). Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 0600}, DEAR loaded with data address that caused access violation, no ESR bit set.
Write Fault	Data Storage	Occurs when any store with an effective address with the WR bit clear and ZPR field != 11 while in the problem state with data translation enabled (MSR[27] = 1'b1).  Also occurs when in the supervisor state with data translation enabled (MSR[27] = 1'b1) and any store with an effective address with the WR bit clear and ZPR field other than 11 or 10. Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 0300}, DEAR loaded with data address that caused access violation, ESR[8] set.
Zone Fault	Data Storage	Occurs when in the problem state with data translation enabled (MSR[27] = 1'b1) and any user-mode Load/Store with an effective address with ZPR field = 00. Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 0300}, DEAR loaded with data address that caused access violation, ESR[9] set.
U0 Fault	Data Storage	Occurs on a Store with an effective address with the U0 bit set and CCR0[u0ex] = 1. Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 0300}, DEAR loaded with data address that caused access violation, ESR[16] set.
Data TLB Miss	Data TLB Miss	Occurs when desired data storage access (Load/Store) to the effective address does not match any valid TLB entry if the data translation is enabled in the CPU (MSR[27] = 1'b1). Current instruction pointer saved in SSR0, current MSR saved in SSR1, vector to 32'x{EVPR[0:15], 1100}, DEAR loaded with data address that caused access violation, ESR[8] set.

## FCM Instruction Flushing

The CPU pipeline can be flushed for various reasons. If an FCM instruction is in the CPU pipeline when this occurs, it is said to be flushed. The APU controller will notify the FCM of a flushed instruction by asserting `APUFCMFLUSH`. If the FCM instruction is flushed from the CPU pipeline, the FCM must be able to re-issue the same instruction without corrupting its internal state. For each FCM instruction, including blocking non-autonomous instructions, the APU controller signals when the point-of-no-return has been reached (`APUFCMWRITEBACKOK` asserted), after which the instruction cannot be flushed. The conditions where `APUFCMWRITEBACKOK` is asserted are as follows:

- The instruction is a non-blocking, multi-cycle operation and is currently in the last cycle of execution (two FCM clock cycles after `FCMAPUDONE` asserted).
- The instruction is a Blocking or Autonomous multi-cycle in the first cycle of execution (same cycle as `APUFCMOPERANDVALID` is asserted).
- Executing an FCM Load and the last word is in the PowerPC LoadWB stage.
- Executing an FCM Store with the APU controller configuration register bit `StoreWBOK` set, and return data has been committed to the PowerPC WriteBack stage.

If the APU controller configuration register bit `StoreWBOK` is not set, the `APUFCMWRITEBACKOK` will not be asserted when a Store is executed.

Normally `APUFCMWRITEBACKOK` is not asserted for store instructions for two reasons:

- In most solutions the FCM will not alter internal registers when executing a store and therefore it does not matter if the instruction is re-issued.
- Store instructions that do not wait for `APUFCMWRITEBACKOK` have a performance benefit.

However, if the FCM store instruction alters any internal registers and could cause errors if re-issued (for example if the FCM uses a FIFO), the user should set the `StoreWBOK` bit in the APU controller configuration register. This will force the APU controller to generate the `APUFCMWRITEBACKOK` signal for all store instructions.

## Execution Hazards

The APU controller ensures that there are no data or structural hazards with regard to the PowerPC405 pipeline execution.

FCM internal data hazards such as read-after-write (RAW) and write-after-write (WAW) are eliminated if the designer ensures that all FCM instructions complete in order. This can be done conservatively by asserting `FCMAPUDONE` only after each instruction has completed. This is, however, incompatible with execution pipelining. A pipelined FCM must handle all possible hazards internally.

# APU Controller Configuration

## General Configuration Register

The general configuration register defines the APU controller's behavior. The register is 32 bits wide. Individual bits are described in [Table 4-5](#). For reset values, refer to [Table 4-11](#), page 201.

**Table 4-5: APU Controller Configuration Register Bit Description**

Name	Bit	Description
RstUDICfg	0	Reset the UDI configuration registers by loading attribute interface signals (TIEAPUUDIn). Reset the APU controller configuration register by loading TIEAPUCONTROL.
-	(1:4)	Not used.
LdStDecDis	5	Disable Load/Store instruction decoding only in APU controller.
UDIDecDis	6	Disable UDI instruction decoding in APU controller. This bit also disables load store instruction decoding.
ForceUDINonB	7	Force all UDI instructions to execute as if Non-Blocking.
FPUDecDis	8	Disable FPU instruction decoding in APU controller.
FPUArithDis	9	Disable decoding of FPU complex arithmetic instruction group (see <a href="#">"Floating Point Instructions"</a> ).
FPUConvIDis	10	Disable decoding of FPU conversion instruction group (see <a href="#">"Floating Point Instructions"</a> ).
FPUEstimIDis	11	Disable decoding of FPU estimation instruction group (see <a href="#">"Floating Point Instructions"</a> ).
-	(12:14)	Not used.
ForceFPUNonB	15	Force all FPU instructions to execute as if they are non-blocking.
StoreWBOK	16	Enable generation of the APUFcmWriteBackOk signal for FCM Store operations (see <a href="#">"FCM Instruction Flushing"</a> ).
LdStPrivOp	17	Execute Load/Store operations only in privileged mode.
-	(18:19)	Not used.
ForceAlign	20	Force address alignment for FCM Load/Store data. Forces alignment on the natural boundary depending on the transfer size (word boundary for words, quad word boundary for quad words, etc.) Used for load/store with Update.

Table 4-5: APU Controller Configuration Register Bit Description (*Continued*)

Name	Bit	Description
LETrap	21	Enable little-endian Traps for FCM Load/Store. If the accessed memory is little-endian (APUFMENDIAN=1), an alignment exception will be cast.
BETrap	22	Enable big-endian Traps for FCM Load/Store. If the accessed memory is big-endian (APUFMENDIAN=0), an alignment exception will be cast.
BESSteer	23	Forces big-endian steering of FCMAPURESULT for FCM Store. PowerPC internally byte-flips little-endian results.
APUDiv	24	Perform PPC integer divide operations in FCM.
-	(25:30)	Not used.
FCMEn	31	Enable FCM usage.

## UDI Configuration Registers

The APU controller includes eight UDI configuration registers. This allows the user to define as many custom instructions and have them decoded in the fast APU controller, rather than out in the slower FCM. The 32-bit-wide registers define the PowerPC related behavior of the UDI execution. The individual bits are described in [Table 4-6](#).

Table 4-6: UDI Configuration Register Bit Description

Name	Bit	Description
PriOpCodeSel	0	Select primary op-code for instruction: 0b0 select 0 (= 0b0000000) 0b1 select 4 (= 0b000100)
ExtOpCode	(1:11)	Extended op-code of instruction.
PrivOp	12	Execute only in privileged mode.
RaEn	13	Requires operand from GPR(RA).
RbEn	14	Requires operand from GPR(RB).
GPRWrite	15	Write back result to GPR(RT) .
XerOVEn	16	Enable return of overflow status.
XerCAEn	17	Enable return of carry status.
CRFieldEn	(18:20)	Select which field in the PowerPC CR the instruction should affect (only applies to UDI op-codes that can set CR bits, see table <a href="#">Table 4-2, page 186</a> ).
-	(21:25)	Hard coded 0b0000.

Table 4-6: UDI Configuration Register Bit Description (Continued)

Name	Bit	Description
Type	(26:27)	Instruction class definition, and reserved DCR use: 0b00 = Blocking 0b01 = Non-blocking 0b10 = Autonomous 0b11 = reserved for UDI register selection for DCR read operations (see “ <a href="#">DCR Access to the Configuration Registers</a> ”).
DCRRegPtr	(28:30)	reserved for DCR UDI register addressing (see “ <a href="#">DCR Access to the Configuration Registers</a> ”)
UDIEEn	31	Enable APU controller decoding of this UDI configuration.

The reset value of the individual UDI registers can be defined using attribute inputs to the APU controller. For details see the “[APU Controller Attributes](#)” section in this chapter.

## DCR Access to the Configuration Registers

The APU controller general configuration register has its own DCR address and can be read and written using normal DCR accesses. Refer to the section “[Internal Device Control Register \(DCR\) Interface](#)” in [Chapter 2](#) for address mapping.

The eight UDI registers share a single DCR address for accessing. A UDI register pointer allows individual access to the different registers.

When performing a DCR write to the UDI configuration register address, the DCRRegPtr field of the write data is used to select which UDI register to write, that is, if DCRRegPtr=3, then the DCR write will affect the configuration register associated with UDI number 3. For this DCR write operation, the Type field should be one of the following: autonomous, blocking or non-blocking.

A DCR read from the UDI configuration register address uses a 3-bit read pointer register in the APU controller to select which specific UDI configuration to return. This pointer auto-increments after each DCR read operation. To load the read pointer with a specific value, the user must perform a “ghost” write to the UDI configuration DCR address. This write will not affect the contents of any UDI configuration registers, only the read pointer. The data used for a “ghost” write has two significant fields: the Type field and the DCRRegPtr field. All other data fields are ignored. The Type field must be set to 0b11, and the DCRRegPtr should be set to the desired read pointer value. A DCR read performed to the UDI configuration address after such “ghost” write will return the contents of the desired UDI configuration register.

## FCM/APU Controller Clocking

The APU controller internally runs at the PowerPC clock speed. It receives as an input the FCM clock, CPMFCMCLK, in order to synchronize signals to/from the FCM. The APU controller will automatically detect the clock ratio between the two clocks. The APU controller to FCM interface clock ratio can be any integer between 1:1 and 16:1. The clocks must be rising-edge aligned. The actual FCM frequency depends on the complexity of the logic designed in the user's FCM.

## Interface Definition

The tables below describe all I/O ports related to the APU controller. They connect the APU controller in the PowerPC 405 block to the FCM in the FPGA fabric. The naming convention implies the direction of the data flow: "APUFCM" signifies "from APU controller to FCM", and "FCMAPU" represents "from FCM to APU controller".

### APU Controller Input Signals

All APU controller input signals should be synchronized on the FCM clock (CPMFCMCLK).

Table 4-7: FCM Interface Input Signals

Signal	Function
FCMAPUINSTRACK	Valid instruction decoded in FCM. Must be asserted the first cycle in which FCMAPUDECODEBUSY is Low, after APUFCMINSTRVALID has been asserted. All instruction decode signals from the FCM to APU controller must be valid when asserted. If the instruction is decoded by the APU controller, this signal should remain Low.
FCMAPURESULT[0:31]	FCM execution result being passed to the CPU through the APU controller.
FCMAPUDONE	Indicates the completion of the instruction in the FCM to the APU controller. In the case of an autonomous instruction, FCMAPUDONE simply means that the FCM can receive another instruction.
FCMAPUSLEEPNOTREADY	Indicates to the APU controller that the FCM is still executing. It is used to determine when the CPU is allowed to enter sleep mode.
FCMAPUDECODEBUSY	Allows FCM to do a multi-cycle instruction decode before returning FCMAPUINSTRACK. Two modes: with or without instruction hold. If this signal is Low when APUFCMINSTRVALID asserts, the APUFCMINSTRUCTION data is only valid for that cycle; if, on the other hand, FCMAPUBUSYDECODE is High then APUFCMINSTRUCTION is held until FCMAPUDECODEBUSY is lowered.
FCMAPUDCDGPRWRITE	FCM decoded instruction must write back to the GPR.
FCMAPUDCDRAEN	FCM decoded instruction need data from GPR(Ra).
FCMAPUDCDRBEN	FCM decoded instruction need data from GPR(Rb).
FCMAPUDCDPRIVOP	FCM decoded instruction executes in privileged mode.
FCMAPUDCDFORCEALIGN	FCM decoded load/store instruction with forced address alignment.
FCMAPUDCDXEROVEN	FCM decoded instruction returns overflow status.
FCMAPUDCDXERCAEN	FCM decoded instruction returns carry status.
FCMAPUDCDCREN	FCM decoded instruction sets condition register (CR) bits.
FCMAPUEXECRFIELD[0:2]	FCM decoded instruction selects which of the eight PowerPC CR field to update: 0=CR0, 1=CR1, etc.
FCMAPUDCDLOAD	FCM decoded load instruction.



Table 4-7: FCM Interface Input Signals (Continued)

Signal	Function
FCMAPUDCDSTORE	FCM decoded store instruction.
FCMAPUDCDUPDATE	FCM decoded load/store instruction should update Ra with effective address.
FCMAPUDCDLDSTBYTE	FCM decoded load/store instruction does byte transfer.
FCMAPUDCDLDSTHW	FCM decoded load/store instruction does half word transfer.
FCMAPUDCDLDSTWD	FCM decoded load/store instruction does word transfer.
FCMAPUDCDLDSTDW	FCM decoded load/store instruction does double word transfer.
FCMAPUDCDLDSTQW	FCM decoded load/store instruction does quad word transfer.
FCMAPUDCDTRAPLE	FCM decoded load/store instruction will cause alignment exception if the storage endian attribute is 1'b1.
FCMAPUDCDTRAPBE	FCM decoded load/store instruction will cause alignment exception if the storage endian attribute is 1'b0.
FCMAPUDCDFORCEBESTEERING	FCM decoded store instruction will force big-endian steering.
FCMAPUFPUOP	FCM decoded FPU instruction.
FCMAPUEXEBLOCKINGMCO	FCM decoded instruction for multi cycle operation of blocking class.
FCMAPUEXENONBLOCKINGMCO	FCM decoded instruction for multi cycle operation of non-blocking class.
FCMAPULOADWAIT	FCM is not yet ready to receive next load data.
FCMAPURESULTVALID	Values on the FCMAPURESULT[0:31], FCMAPUXEROV, FCMAPUXERCA and FCMAPUCR[0:3] are valid.
FCMAPUXEROV	FCM execution overflow status bit.
FCMAPUXERCA	FCM execution carry status bit.
FCMAPUCR[0:3]	Condition result bits to set in the PowerPC CR field selected by FCMAPUEXECRFIELD: <ul style="list-style-type: none"> <li>• Bit 0 = set LT-bit, meaning result is less than zero</li> <li>• Bit 1 = set GT-bit, meaning result is greater than 0</li> <li>• Bit 2 = set EQ-bit, meaning result is zero</li> <li>• Bit 3 = set SO-bit, meaning Summary Overflow</li> </ul>
FCMAPUEXCEPTION	FCM generate program exception on the processor (vector 0x0700). Exception must be enabled by processor to trap.
CPMFCMCLK	FCM interface clock. Used for synchronizing FCM signals to/from APU controller. See <a href="#">“CPMFCMCLK (Input, Virtex-4-FX Only)”</a> in Chapter 2.



## APU Controller Output Signals

All APU controller output signals are synchronous with the FCM clock (CPMFCMCLK).

Table 4-8: FCM Interface Output Signals

Signal	Function
APUFCMINSTRUCTION[0:31]	Instruction being presented to the FCM. Is valid as long as APUFCMINSTRVALID is High.
APUFCMINSTRVALID	<p>This signal is asserted on two conditions:</p> <ul style="list-style-type: none"> <li>• A valid APU instruction was decoded by the APU controller</li> <li>• An undecoded instruction passed to FCM for decoding</li> </ul> <p>The signal will remain High for one FCM clock cycle, unless FCMAPUDECODEBUSY is High when it asserts. In that case it stays High until FCMAPUDECODEBUSY goes Low.</p>
APUFCMRADATA[0:31]	Instruction operand from GPR(RA).
APUFCMRBDATA[0:31]	Instruction operand from GPR(RB).
APUFCMOPERANDVALID	Instruction operand valid.
APUFCMFLUSH	Flush APU instruction in the FCM. If asserted no APUFCMWRITEBACKOK signal will be generated
APUFCMWRITEBACKOK	<p>Safe for FCM to commit internal state change; the APU controller can no longer flush the instruction.</p> <p>In normal cases, this signal is asserted for one FCM clock cycle. In some cases when a non-blocking multi-cycle operation is followed by an autonomous or blocking multi-cycle operation while using a large clock ratio, the signal may be asserted for two back-to-back FCM clock cycles.</p>
APUFCMLOADDATA[0:31]	Data word loaded from storage to the APU register file.
APUFCMLOADDVALID	When asserted the data word on the APUFCMLOADDATA[0:31] data bus is valid.
APUFCMLOADBYTEEN[0:3]	Specifies the valid bytes for the word on the load data bus APUFCMLOADDATA[0:31].
APUFCMENDIAN	When asserted, the load/store instruction being presented to the FCM has true little-endian storage attribute.
APUFCMXERCA	Reflects the XerCA bit used for extended arithmetic.
APUFCMDECODED	Asserted when the APU controller decoded the instruction being sent to the FCM.
APUFCMDECUDI[0:2]	Specifies which UDI the APU controller decoded (binary encoded).
APUFCMDECUDIVALID	Valid signals for APUFCMDECUDI.

## APU Controller Attributes

The following input signals are used as reset values for the APU controller configuration registers. The reset values can be over-written using DCR. For details see the “[APU Controller Configuration](#)” section in this chapter.

Table 4-9: APU Controller Attributes

Attribute Signal	Function
TIEAPUUDI1[0:23]	Reset value for UDI register 1.
TIEAPUUDI2[0:23]	Reset value for UDI register 2.
TIEAPUUDI3[0:23]	Reset value for UDI register 3.
TIEAPUUDI4[0:23]	Reset value for UDI register 4.
TIEAPUUDI5[0:23]	Reset value for UDI register 5.
TIEAPUUDI6[0:23]	Reset value for UDI register 6.
TIEAPUUDI7[0:23]	Reset value for UDI register 7.
TIEAPUUDI8[0:23]	Reset value for UDI register 8.
TIEAPUCONTROL[0:15]	Reset values for the APU control register.

Table 4-10: Bit Map Between TIEAPUUDIn and UDI Configuration Registers

UDI Configuration Field	TIEAPUUDI Bits
PriOpCodeSel	0
ExtOpCode	(1:11)
PrivOp	12
RaEn	13
RbEn	14
GPRWrite	15
XerOVEn	16
XerCAEn	17
CRFieldEn	(18:20)
Type	(21:22)
UDIEn	23

Table 4-11: Bit Map Between TIEAPUCONTROL and APU Configuration Register

APU Controller Configuration Field	TIEAPUCONTROL Bits
LdStDecDis	0
UDIDecDis	1
ForceUDINonB	2
FPUDecDis	3
FPUCArithDis	4
FPUConvIDis	5
FPUEstimIDis	6
ForceFPUNonB	7
StoreWBOK	8
LdStPrivOp	9
ForceAlign	10
LETrap	11
BETrap	12
BESSteer	13
APUDiv	14
FCMEn	15

## FCM Instruction Execution

There are two distinct classes of instruction execution, non-storage and storage instructions. Each type has different execution options and certain execution restrictions.

### FCM Non-storage Instructions

There are a few options that non-storage instructions can take advantage of. The CPU can send RA and RB source data (from the CPU's GPR) along with a carry-in value for extended arithmetic. The FCM can also return RT result to the CPU's GPR along with an overflow bit, a carry bit, and condition record values. The only restriction to these instructions is that any instruction returning a result or status to the CPU must be non-autonomous.

### FCM Storage Instructions

FCM storage instructions can be loads or stores of byte, half word, word, double word, or quad word. Additionally loads and stores can be executed with Update (meaning the RA, base address, will be updated with the new effective address). The CPU actually handles the majority of the FCM loads and stores. The CPU calculates the address, retrieves data from the cache or other memory, and then sends the data to the FCM in the case of a load.

For stores, the CPU receives the data from the FCM and puts it in cache or other memory at the address the CPU calculates. The CPU will also take care of any Update on the RA register. Because the CPU is so involved in the execution of FCM storage instructions, the user must make sure the FCM adheres to formats and regulations of these instructions.

A load or store without Update will automatically align to the transfer's natural address boundary. In other words, a word transfer is aligned on a word boundary, a double word on a double word boundary, and so on. Even if the address given is not aligned, the resulting load or store will be aligned on the natural boundary. In the case of multi-word loads, the CPU sends them in address increasing order. For example, if a quad word load starts at address 0, the FCM will receive the word at address 0 first followed by the next three words. The same ordering is expected for multi-word stores. For byte or half word loads, the aligned word is sent to the FCM along with the corresponding byte enables based on the address given. For stores, a half word is expected to be positioned at FCMAPURESULT[16:31] and a byte is expected to be positioned at FCMAPURESULT[24:31]. The CPU uses the address given to store the data in the correct place in memory.

A load or store with Update is required on a byte boundary for byte transfers, half word boundary for half word transfers, and word boundary for any other transfer. If the address for the load/store is not aligned properly it will cause an Alignment Exception in the CPU (see “CPU Exceptions” for more details). To prevent this exception the instruction can use the ForceAlign option. This will force alignment on the transfer's natural boundary (similar to load/store without Update).

## FCM Interface Timing Specification

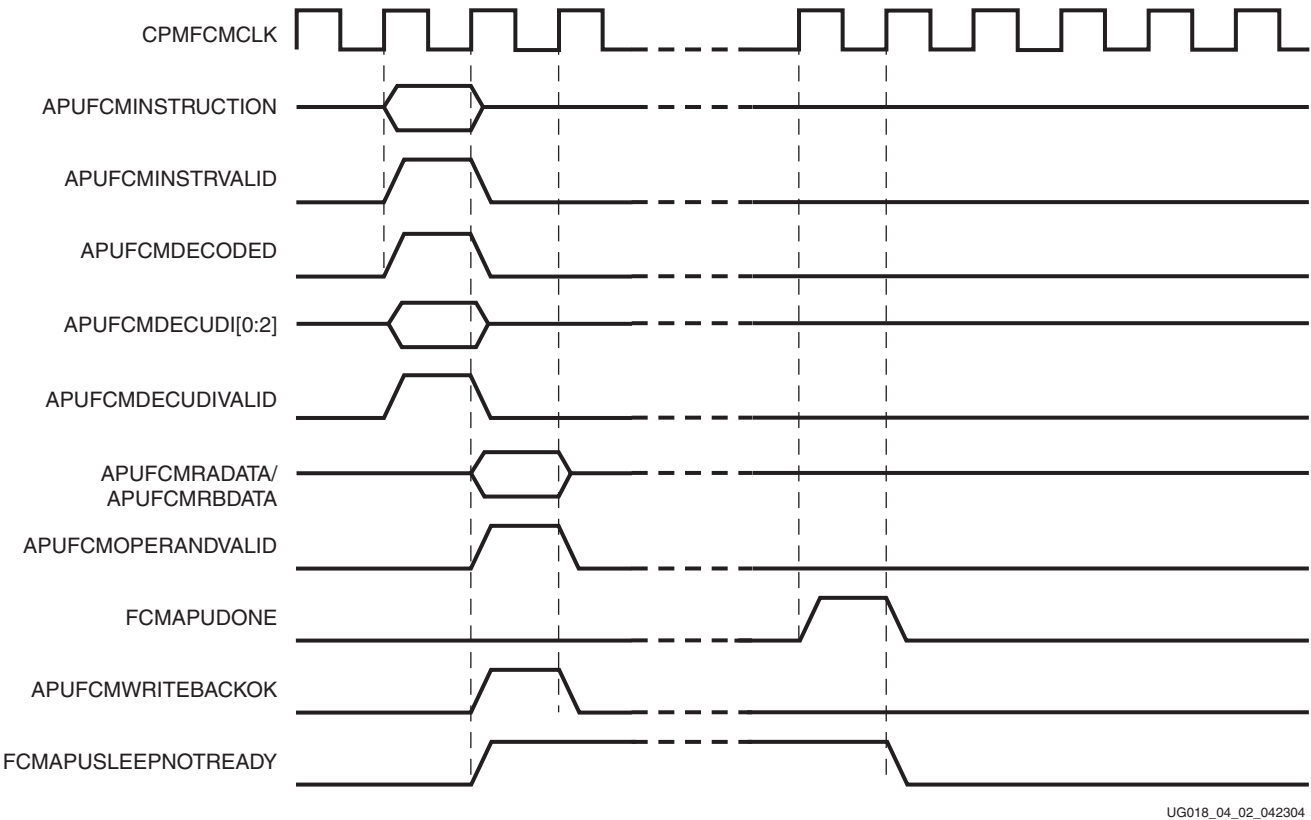
### Autonomous Transactions

For an APU controller decoded UDI instruction, the following signals will always arrive at the same cycle: APUFCMINSTRUCTION[0:31], APUFCMINSTRVALID, APUFCMDECODED, APUFCMDECUDI[0:2], and APUFCMDECUDIVALID.

In the example shown in Figure 4-3, APUFCMRADATA/APUFCMRBDATA, APUFCMOPERANDVALID, and APUFCMWRITEBACKOK are seen one cycle after the instruction is presented. These signals can arrive at the same time as the instruction signals or anytime afterward. The timing of these signals depend both on the current situation in the CPU pipeline and the clock ratio between the FCM and CPU. However, in the case of autonomous instructions APUFCMRADATA/APUFCMRBDATA, APUFCMOPERANDVALID, and APUFCMWRITEBACKOK, they will all arrive during the same cycle. Not shown in this example, APUFCMXERCA will be valid the same cycle as APUFCMOPERANDVALID. This signal indicates the carry in for any extended arithmetic.

After the FCM has received APUFCMWRITEBACKOK it can respond with FCMAPUDONE in the same cycle or anytime afterward. The Figure 4-3 timing diagram assumes the signals from the FCM are coming off of flip-flops, so FCMAPUDONE arrives the cycle after APUFCMWRITEBACKOK.

Also in this example, FCMAPUSLEEPNOTREADY is set High the cycle after the instruction signals are received. This is the latest cycle the signal should go High and it should remain High until the instruction is completely finished in the FCM (even if FCMAPUDONE has been sent). FCMAPUSLEEPNOTREADY will not prevent a new instruction from being sent to the FCM.



UG018\_04\_02\_042304

Figure 4-3: APU Controller Decoded Autonomous Transaction Example

**Note:** Actual timing results may vary from those shown in Figure 4-3. For example, the instruction and operands can be valid on the same FCM clock cycle, or they can be many cycles apart.

For an FCM decoded UDI instruction, the following signals will always arrive at the same cycle: APUFCMINSTRUCTION[0:31] and APUFCMINSTRVALID.

In the example shown in Figure 4-4, the FCM responds in the next cycle by setting FCMAPUINSTRACK High and sending any of the FCMAPUOPTIONS that are used by this instruction. To respond in a later cycle, please see Figure 4-12 and Figure 4-13. Table 4-12 shows the FCM signals included in FCMAPUOPTIONAS,

Table 4-12: FCM Signals Included in FCMAPUOPTIONS

FCMAPUDCDGPRWRITE	FCMAPUDCDRAEN	FCMAPUDCDRBEN
FCMAPUDCDPRIVOP	FCMAPUDCDFORCEALIGN	FCMAPUDCDXERCAEN
FCMAPUDCDXEROVEN	FCMAPUDCDLOAD	FCMAPUDCDCREN
FCMAPUEXECRFIELD[0:2]	FCMAPUDCDLDSTBYTE	FCMAPUDCDSTORE
FCMAPUDCDUPDATE	FCMAPUDCDLDSTDW	FCMAPUDCDLDSTHW
FCMAPUDCDLDSTDW	FCMAPUDCDTRAPBE	FCMAPUDCDLDSTQW
FCMAPUDCDTRAPLE	FCMAPUEXEBLOCKINGMCO	FCMAPUDCDFORCEBESTEERING
FCMAPUFPUOP	FCMAPUEXENONBLOCKINGMCO	

In the example shown in Figure 4-4, APUFCMRADATA / APUFCMRBDATA, APUFCMOPERANDVALID, and APUFCMWRITEBACKOK are seen two cycles after the FCM acknowledges the instruction. This is the earliest cycle these signals can be sent to the FCM, but they can also arrive anytime afterward. For autonomous instructions, the signals APUFCMRADATA / APUFCMRBDATA, APUFCMOPERANDVALID, and APUFCMWRITEBACKOK will all arrive during the same cycle. Not shown in this example, APUFCMXERCA will be valid the same cycle as APUFCMOPERANDVALID. This signal indicates the carry in for any extended arithmetic.

After the FCM has received APUFCMWRITEBACKOK, it can respond with FCMAPUDONE in the same cycle or anytime afterward. This timing diagram assumes the signals from the FCM are coming off of flip-flops, so FCMAPUDONE arrives the cycle after APUFCMWRITEBACKOK.

In the example FCMAPUSLEEPNOTREADY is set High the same cycle the instruction acknowledge is sent. This is the latest cycle the signal should go High and it should remain High until the instruction is completely finished in the FCM (even if FCMAPUDONE has been sent). FCMAPUSLEEPNOTREADY will not prevent a new instruction from being sent to the FCM.

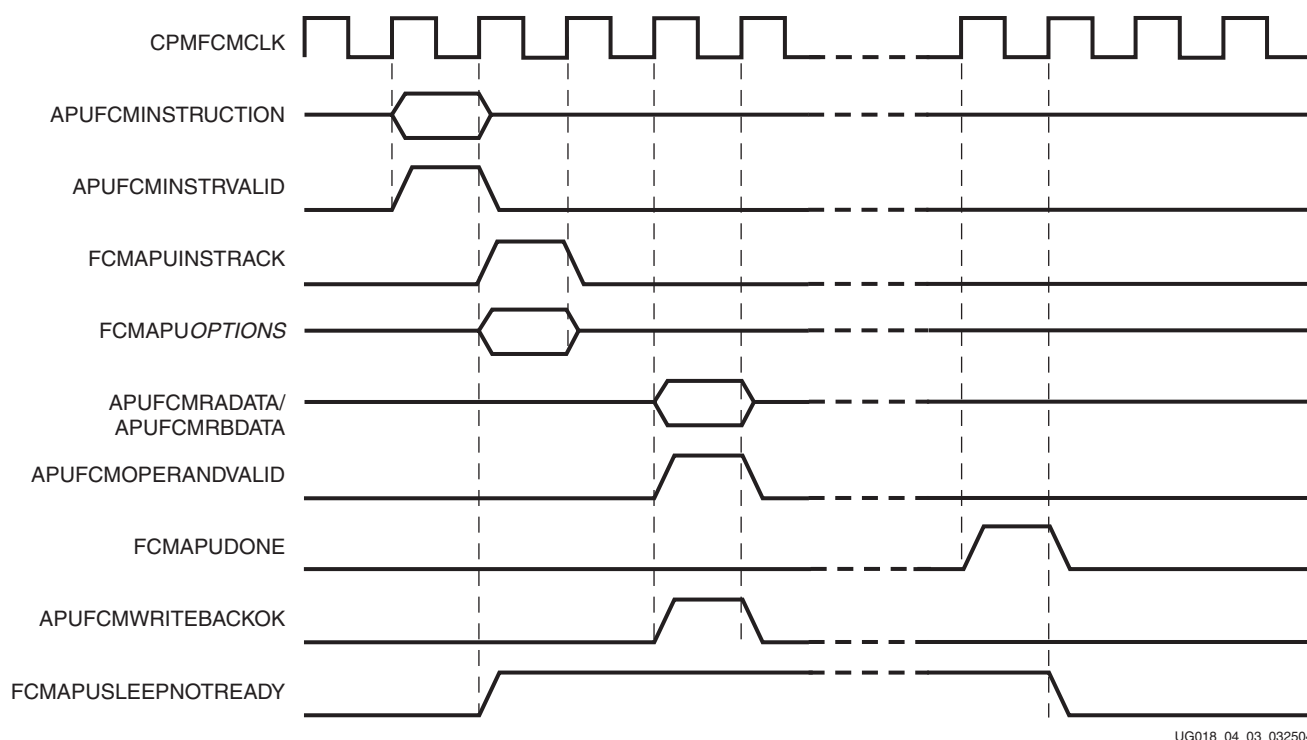


Figure 4-4: FCM Decoded Autonomous Transaction Example

**Note:** Actual timing results may vary from those shown in Figure 4-4. For example, the operands could come later than shown.

## Blocking Transactions

For an FCM decoded UDI instruction the following signals will always arrive at the same cycle: APUFCMINSTRUCTION[0:31], APUFCMINSTRVALID.

In the example shown in Figure 4-5, the FCM responds in the next cycle by setting FCMAPUINSTRACK High and sending any of the FCMAPUOPTIONS that are used by this instruction. To respond in a later cycle please see Figure 4-12 and Figure 4-13.

In this example APUFCMRADATA/APUFCMRBDATA, APUFCMOPERANDVALID, and APUFCMWRITEBACKOK are seen two cycles after the FCM acknowledges the instruction. This is the earliest cycle they can be sent to the FCM but can also arrive anytime afterward. In the case of blocking instructions APUFCMRADATA/APUFCMRBDATA, APUFCMOPERANDVALID, and APUFCMWRITEBACKOK will all arrive during the same cycle. It is not shown in this example, but APUFCMXERCA will be valid the same cycle as APUFCMOPERANDVALID. This signal indicates the carry in for any extended arithmetic.

After the FCM has received APUFCMWRITEBACKOK it can respond with FCMAPURESULT[0:31], FCMAPURESULTVALID, and FCMAPUDONE in the same cycle or anytime after that. The timing diagram in Figure 4-5 assumes the signals from the FCM are coming off of flip-flops, so FCMAPURESULTVALID, etc. arrive the cycle after APUFCMWRITEBACKOK. FCMAPURESULT[0:31] and FCMAPURESULTVALID must arrive the same cycle. FCMAPUDONE can arrive during the same cycle as these or any

cycle later. FCMAPUXEROV, FCMAPUXERCA, and FCMAPUCR[0:3] are not shown in this example, but should be sent at the same cycle as FCMAPURESULT[0:31] and FCMAPURESULTVALID.

In the example shown in Figure 4-5, FCMAPUSLEEPNOTREADY is set High the same cycle the instruction acknowledge is sent. This is the latest cycle the signal should go High and it should remain High until the instruction is completely finished in the FCM (even if FCMAPUDONE has been sent). FCMAPUSLEEPNOTREADY will not prevent a new instruction from being sent to the FCM.

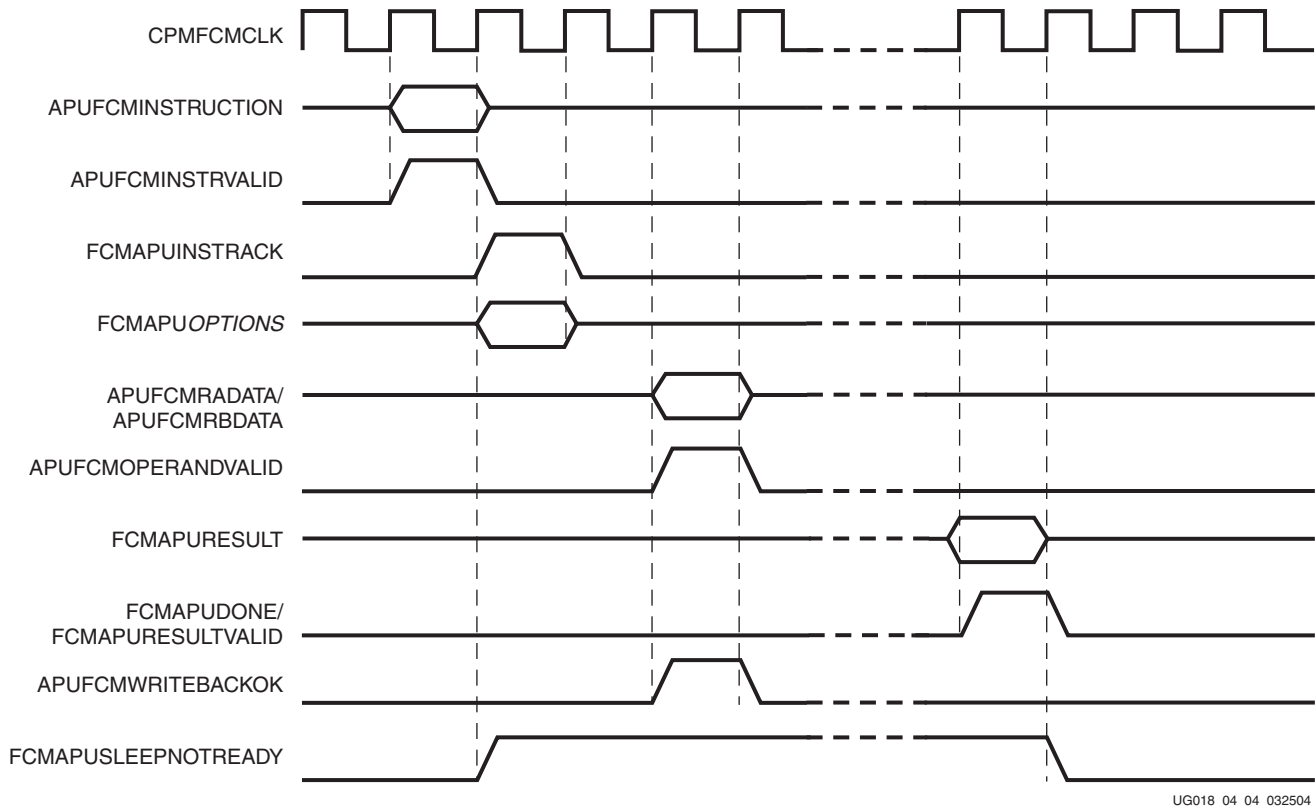


Figure 4-5: FCM Decoded Blocking Transaction Example

**Note:** Actual timing results may vary from those shown in Figure 4-5. For example, the operands could come later than shown.

## Non-Blocking Transactions

For an APU controller decoded UDI instruction the following signals will always arrive at the same cycle: APUFCMINSTRUCTION[0:31], APUFCMINSTRVALID, and APUFCMDECODED (APUFCMDECODEI[0:2] and APUFCMDECODEIVALID are not shown in Figure 4-6, but they will also arrive the same cycle).

In the example shown in Figure 4-6, APUFCMRADATA/APUFCMRBDATA and APUFCMOPERANDVALID are seen one cycle after the instruction is presented. These signals can arrive at the same time as the instruction signals or anytime afterward. The timing of these signals depend both on the current situation in the CPU pipeline and the clock ratio between the FCM and CPU. However, in the case of non-blocking instructions APUFCMRADATA/APUFCMRBDATA and APUFCMOPERANDVALID will all arrive during the same cycle. It is not shown in this example, but APUFCMXERCA will be valid



the same cycle as APUFCMOPERANDVALID. This signal indicates the carry in for any extended arithmetic.

After the FCM has received APUFCMOPERANDVALID along with the operands it can respond with FCMAPURESULT[0:31], FCMAPURESULTVALID, and FCMAPUDONE in the same cycle or anytime afterward. FCMAPURESULT[0:31] and FCMAPURESULTVALID must arrive the same cycle. FCMAPUDONE can arrive during the same cycle or any cycle later. FCMAPUXEROV, FCMAPUXERCA, and FCMAPUCR[0:3] are not shown in this example, but should be sent at the same cycle as FCMAPURESULT[0:31] and FCMAPURESULTVALID.

APUFCMWRITEBACKOK is shown as arriving two cycles after FCMAPUDONE. This is the earliest cycle it can arrive.

In the example shown in [Figure 4-6](#), FCMAPUSLEEPNOTREADY is set High the cycle after the instruction signals are received. This is the latest cycle the signal should go High and it should remain High until the instruction is completely finished in the FCM. The signal should not be lowered until APUFCMWRITEBACKOK has arrived but could remain High afterward.

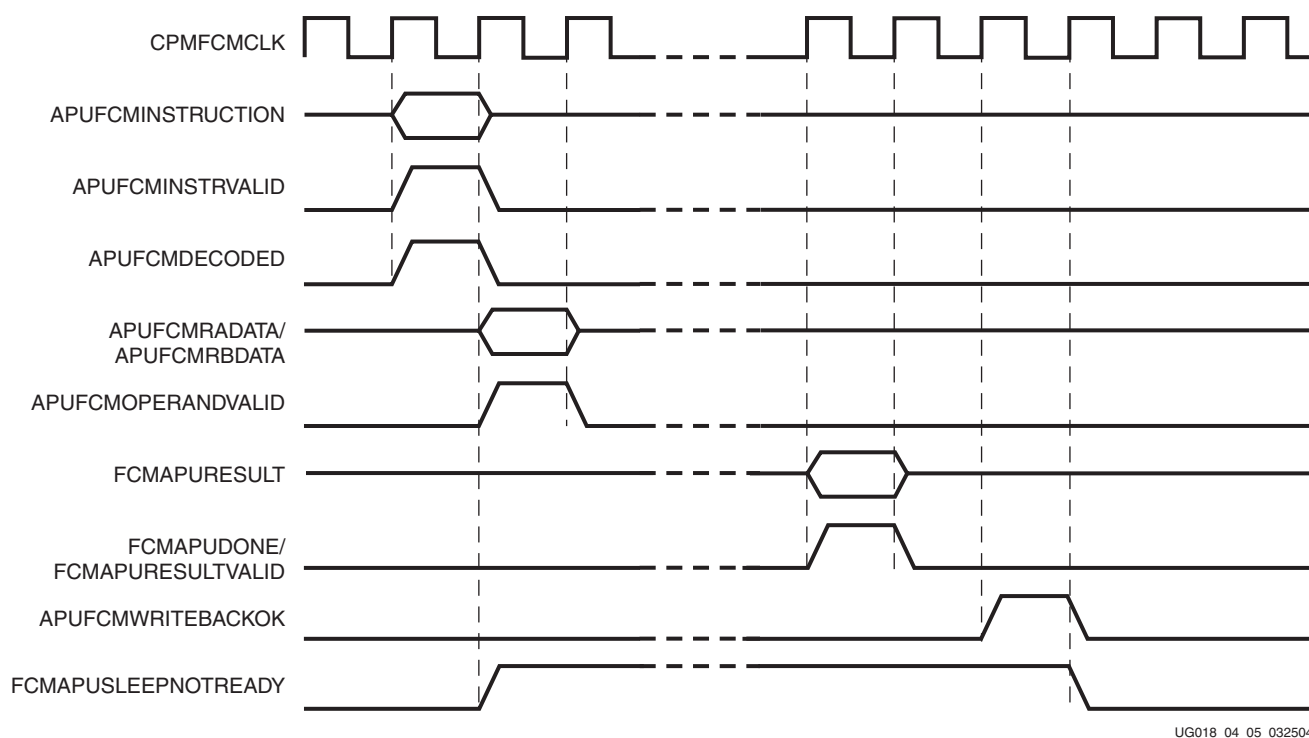


Figure 4-6: APU Controller Decoded Non-Blocking Transaction Example

**Note:** Actual timing results may vary from those shown in [Figure 4-6](#). For example, the operands could come later than shown.

## FCM Load Instruction

For an APU controller decoded load instruction the following signals will always arrive at the same cycle: APUFCMINSTRUCTION[0:31], APUFCMINSTRVALID, and APUFCMDECODED.

In the example shown in Figure 4-7 (load word), APUFCMLOADDATA[0:31], APUFCMLOADVALID, and APUFCMWRITEBACKOK are seen one cycle after the instruction is presented. These signals can arrive at the same time as the instruction signals or anytime afterward. The timing of these signals depend both on the current situation in the CPU pipeline and the clock ratio between the FCM and CPU. However, in the case of load byte, load half word, and load word instructions APUFCMAPUFCMLOADDATA[0:31], APUFCMLOADVALID, and APUFCMWRITEBACKOK will all arrive during the same cycle. For multi-word loads APUFCMWRITEBACKOK will arrive no later than the final word. APUFCMLOADBYTEEN[0:3] is not shown in this example. It will be valid the same cycle as APUFCMLOADVALID. The byte enables map to the load data bus in the following manner:

APUFCMLOADBYTEEN[0] = APUFCMAPUFCMLOADDATA[0:7]  
 APUFCMLOADBYTEEN[1] = APUFCMAPUFCMLOADDATA[8:15]  
 APUFCMLOADBYTEEN[2] = APUFCMAPUFCMLOADDATA[16:23]  
 APUFCMLOADBYTEEN[3] = APUFCMAPUFCMLOADDATA[24:31]

It is not shown in this example, but APUFCMENDIAN will also be valid the same cycle as APUFCMLOADVALID.

After the FCM has received APUFCMWRITEBACKOK along with the final word, the FCM can respond with FCMAPUDONE in the same cycle or anytime afterward.

In the example shown in Figure 4-7, FCMAPUSLEEPNOTREADY is set High the cycle after the instruction signals are received. This is the latest cycle the signal should go High and it should remain High until the instruction is completely finished in the FCM (even if FCMAPUDONE has been sent). FCMAPUSLEEPNOTREADY will not prevent a new instruction from being sent to the FCM.

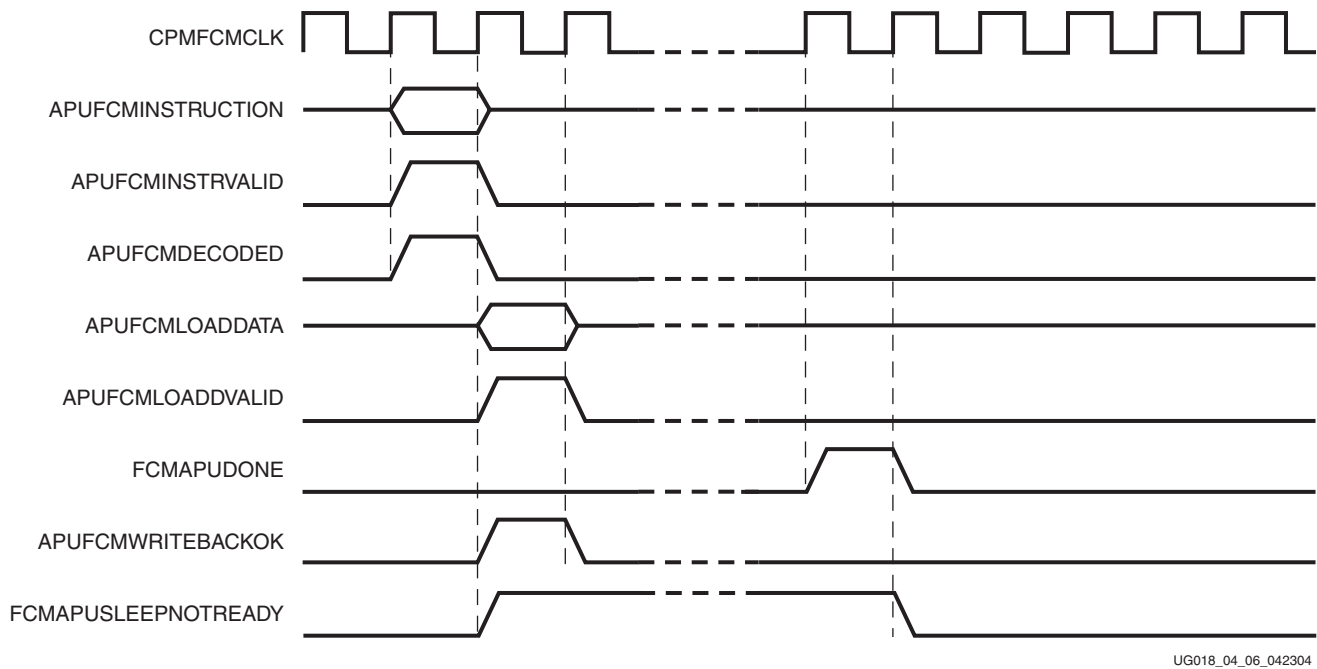


Figure 4-7: APU Controller Decoded Load Instruction Example

UG018\_04\_06\_042304

**Note:** Load data can arrive at the same time as the instruction or at a later clock cycle than shown in Figure 4-7.

The example shown in Figure 4-8 is similar to Figure 4-7 with two exceptions. This example shows two words being transferred and the FCM is using FCMAPULOADWAIT.

In Figure 4-7, the FCM does not have enough room to accept the second load word and therefore uses FCMAPULOADWAIT. If FCMAPULOADWAIT is set High APUFCMLOADDATA[0:31] and APUFCMLOADVALID will remain asserted until FCMAPULOADWAIT is deasserted. It should be noted that the second word may arrive later than shown. Multi-words are not necessarily sent back-to-back.

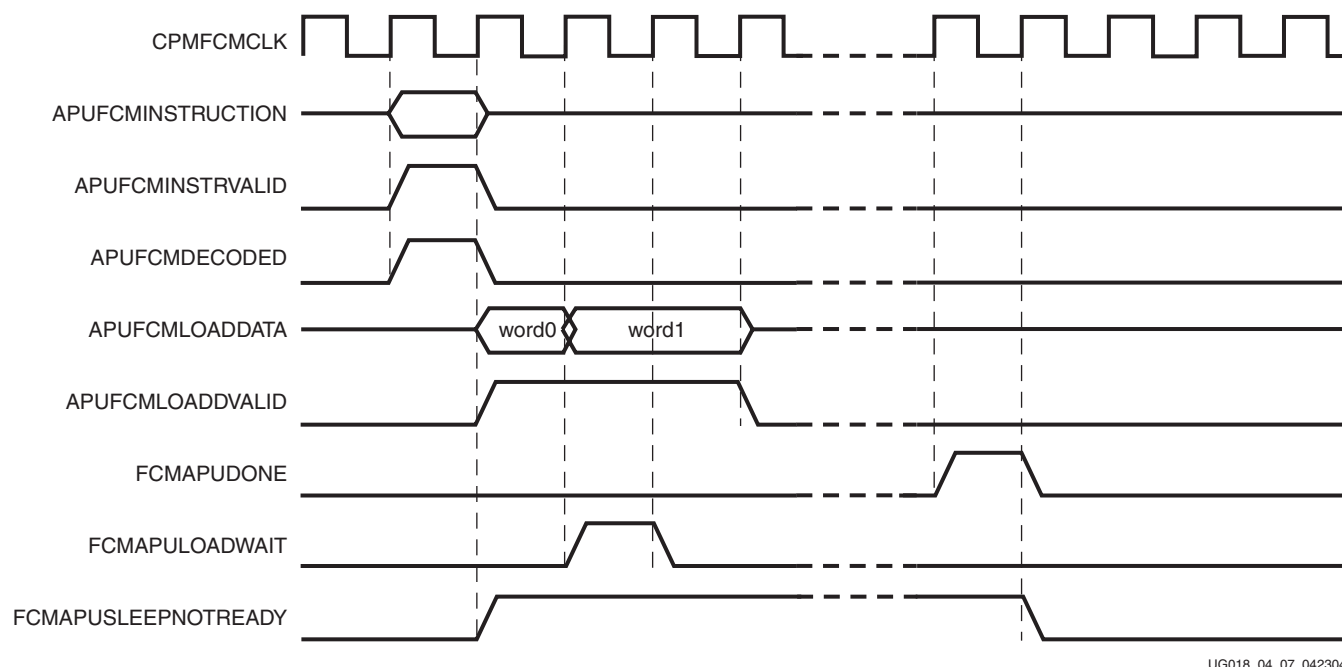


Figure 4-8: APU Controller Decoded a Double Word Load Instruction with LoadWait Example

**Note:** Load data can arrive at the same time as the instruction or at a later clock cycle than shown in Figure 4-8. Also, load data might not be sent back-to-back. Users should look at the valid signal.

## FCM Store Instruction

For an APU controller decoded store instruction the following signals will always arrive at the same cycle: APUFCMINSTRUCTION[0:31] and APUFCMINSTRVALID. APUFCMDECODED is shown in Figure 4-9, but the FCM should not expect this signal to arrive with every APU controller decoded store.

After the FCM has received APUFCMINSTRVALID along with the instruction, it can respond with FCMAPURESULT[0:31], FCMAPURESULTVALID, and FCMAPUDONE in the next cycle or anytime afterward. FCMAPURESULT[0:31] and FCMAPURESULTVALID must arrive the same cycle for each word. The words do not have to arrive back-to-back. FCMAPUDONE can arrive during the same cycle as the last word or any cycle later.

APUFCMWRITEBACKOK is not sent in this example. Please see Figure 4-10 for the case when StoreWBOK has been enabled.

In the example shown in Figure 4-9, FCMAPUSLEEPNOTREADY is set High the cycle after the instruction signals are received. This is the latest cycle the signal should go High and it should remain High until the instruction is completely finished in the FCM (even if FCMAPUDONE has been sent). FCMAPUSLEEPNOTREADY will not prevent a new instruction from being sent to the FCM.

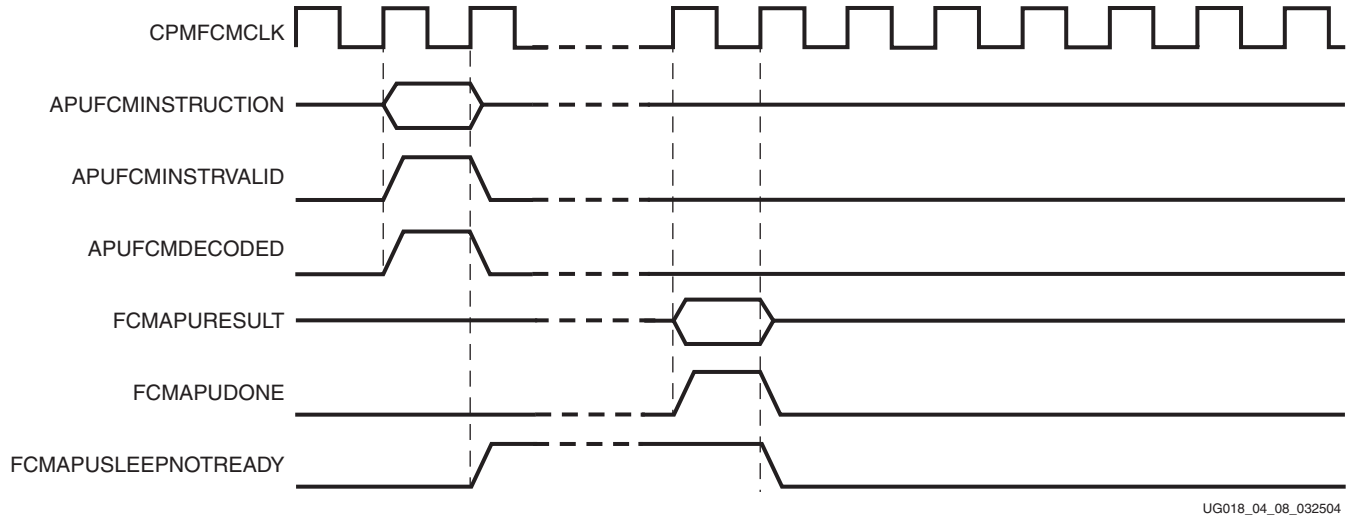
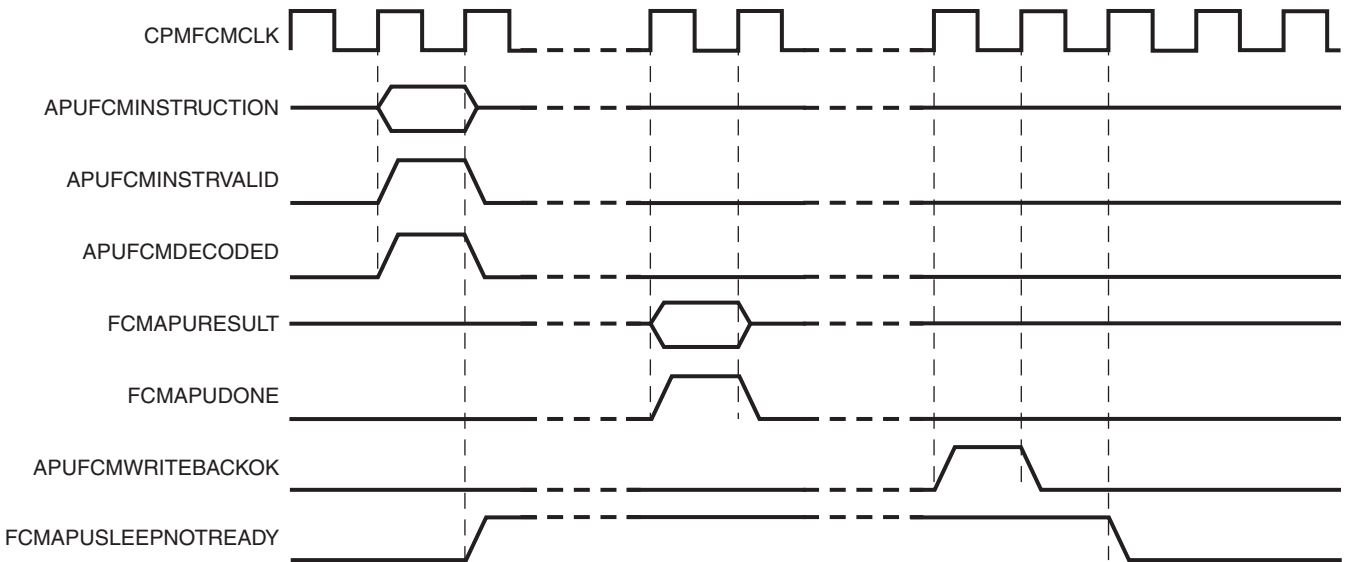


Figure 4-9: APU Controller Decoded Store Instruction

The example in Figure 4-10 is very similar to Figure 4-9 except the APU controller has the StoreWBOK bit set to 1. This means the FCM will expect to receive APUFCMWRITEBACKOK after the store instruction is completed. As shown APUFCMWRITEBACKOK will arrive some time after FCMAPUDONE. The timing of these signals depend both on the current situation in the CPU pipeline and the clock ratio between the FCM and CPU.

In the example in Figure 4-10, FCMAPUSLEEPNOTREADY is set High the cycle after the instruction signals are received. This is the latest cycle the signal should go High and it should remain High until the instruction is completely finished in the FCM. The signal should not be lowered until APUFCMWRITEBACKOK has arrived but could remain High afterward.



UG018\_04\_09\_032504

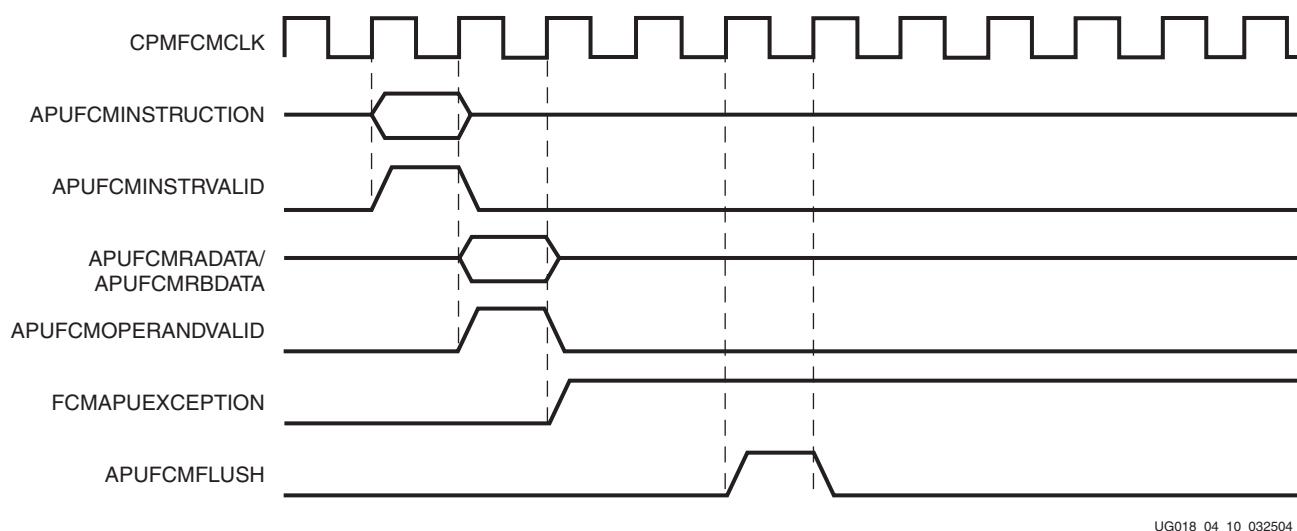
Figure 4-10: APU Controller Decoded Store Instruction with StoreWBOK=1

FCM Exception

In the example in [Figure 4-11](#), an FCM instruction has caused an exception in the FCM. As stated in the “[FCM Exceptions](#)” section, only blocking and non-blocking instructions will produce precise exceptions.

In this example FCMAPUEXCEPTION is set High one cycle after APUFCMOPERANDVALID. The signal can also be set High the cycle after the instruction is received or any later time (but obviously before FCMAPUDONE is sent). This signal must remain High until it is reset by software.

APUFCMFLUSH will be sent to the FCM after the CPU has seen the exception. This indicates the FCM instruction has been flushed from the CPU pipeline.



UG018\_04\_10\_032504

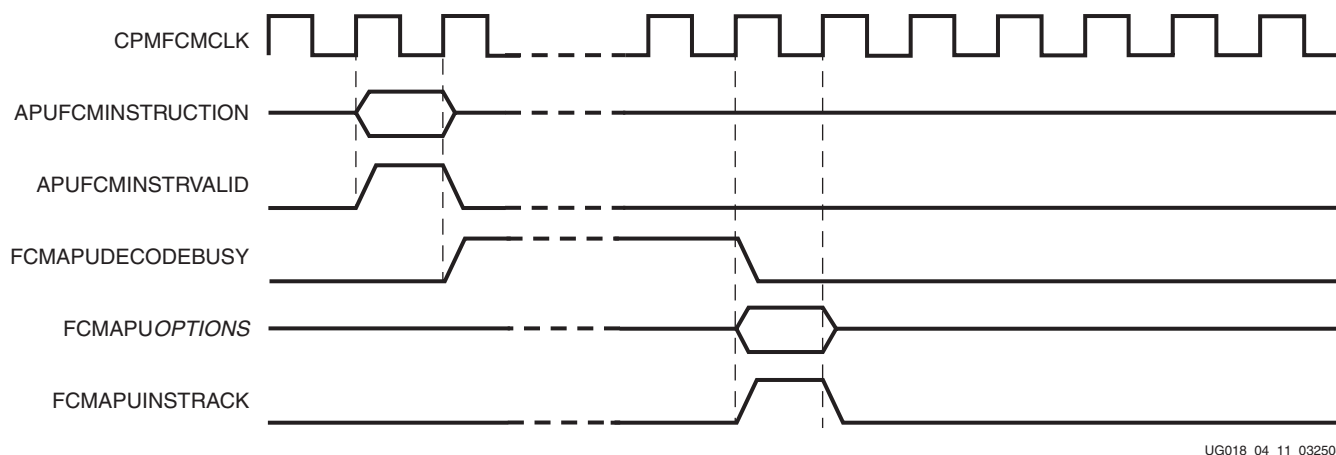
Figure 4-11: FCM Exception

**Note:** FCMAPUEXCEPTION may be sent at any time during the execution of a non-autonomous instruction.

## FCM Decoding Using Decode Busy Signal

The example in Figure 4-12 only shows the decode part of an FCM decoded instruction using DecodeBusy. In this example, the FCM asserts FCMAPUDECODEBUSY the cycle after it receives the instruction signals. APUFCMINSTRUCTION[0:31] and APUFCMINSTRVALID will not be valid after their one cycle in this case.

The same cycle FCMAPUDECODEBUSY is removed the FCM must send FCMAPUINSTRACK along with any FCMAPUOPTIONS.



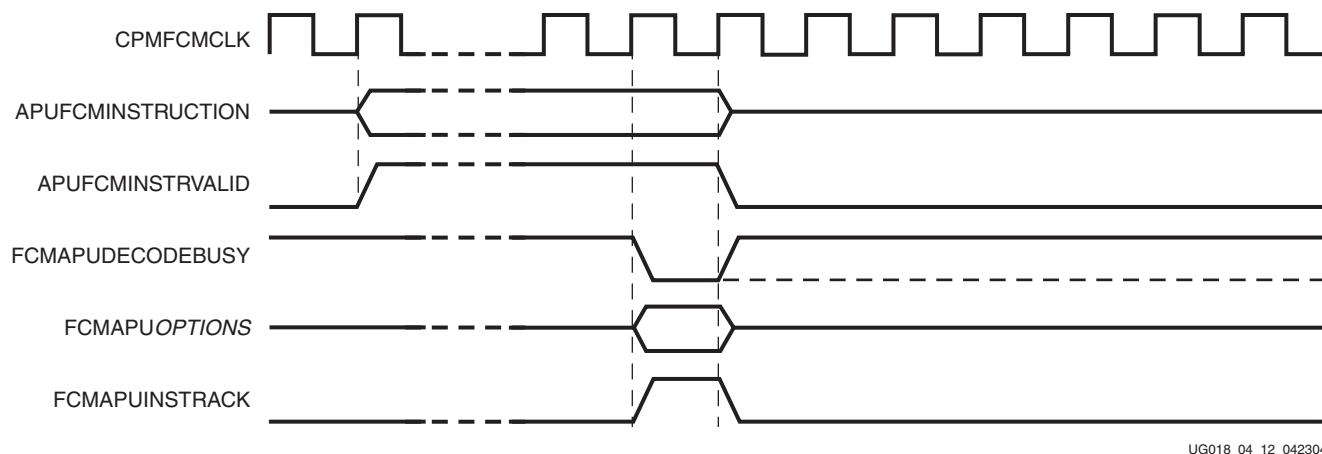
UG018\_04\_11\_032504

Figure 4-12: FCM Decode Asserting DecodeBusy

The example in Figure 4-13 only shows the decode part of an FCM decoded instruction using DecodeBusy. In this example, FCMAPUDECODEBUSY is asserted before it receives

the instruction signals. APUFCMINSTRUCTION[0:31] and APUFCMINSTRVALID will remain valid until FCMAPUDECODEBUSY is deasserted in this case.

The cycle FCMAPUDECODEBUSY is pulsed Low the FCM must send FCMAPUINSTRACK along with any FCMAPUOPTIONS.



UG018\_04\_12\_042304

Figure 4-13: FCM Deasserting DecodeBusy

## Flushed FCM Instructions

The example in Figure 4-14 is very similar to Figure 4-3. The major difference is that this FCM instruction is flushed.

For an APU controller decoded UDI instruction the following signals will always arrive at the same cycle: APUFCMINSTRUCTION[0:31], APUFCMINSTRVALID, APUFCMDECODED, APUFCMDECUDI[0:2], APUFCMDECUDIVALID.

In the example in Figure 4-14, APUFCMRADATA/APUFCMRBDATA, APUFCMOPERANDVALID, and APUFCMXERCA do not arrive until several cycles later. These signals can arrive at the same time as the instruction signals or anytime afterward. The timing of these signals depend both on the current situation in the CPU pipeline and the clock ratio between the FCM and CPU.

APUFCMFLUSH also arrives the same cycle as the operand signals. In the case of an autonomous or blocking instruction, this is the last cycle APUFCMFLUSH can be sent. It could also be sent at the same cycle as APUFCMINSTRVALID. APUFCMWRITEBACKOK will remain Low. Once APUFCMFLUSH has been sent to the FCM, the FCM cannot update its internal registers and must be able to re-execute the same instruction at a later time.

In the example in Figure 4-14, FCMAPUSLEEPNOTREADY is set High the cycle after the instruction signals are received. This is the latest cycle the signal should go High and it should remain High until the instruction is completely finished in the FCM (even if

APUFCMFLUSH has been sent). FCMAPUSLEEPNOTREADY will not prevent a new instruction from being sent to the FCM.

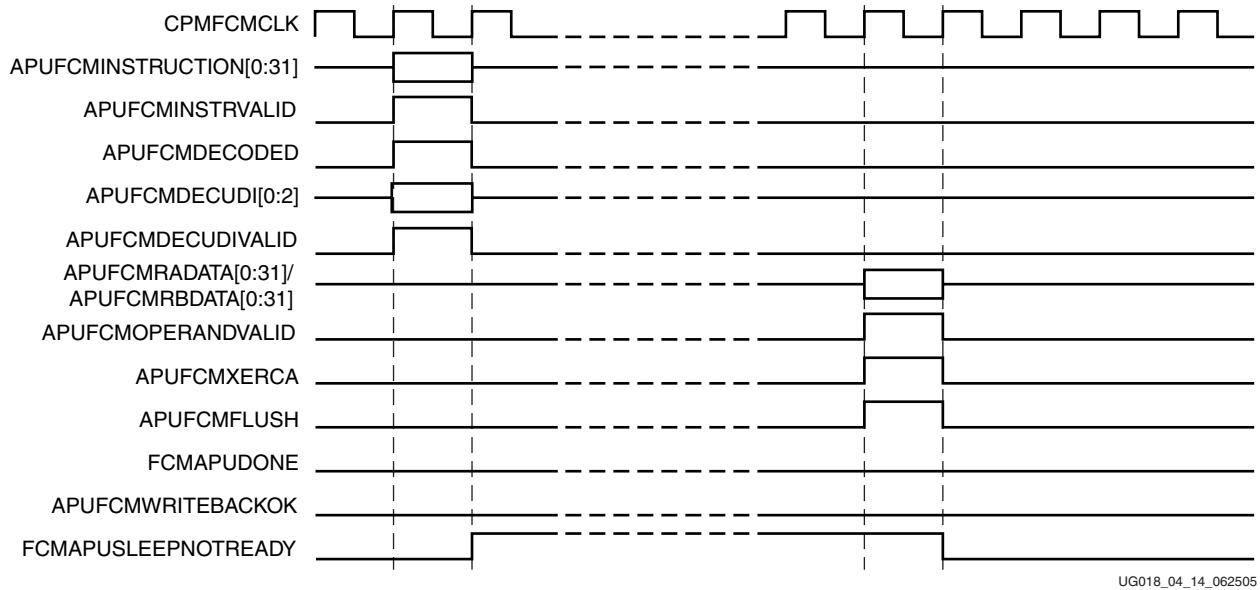


Figure 4-14: APU Controller Decoded Autonomous Transaction with Flush

The example in Figure 4-15 is very similar to Figure 4-6. The major difference is that this FCM instruction is flushed.

For an APU controller decoded UDI instruction the following signals will always arrive at the same cycle: APUFCMINSTRUCTION[0:31], APUFCMINSTRVALID, APUFCMDECODED, APUFCMDECUDI[0:2], APUFCMDECUDIVALID.

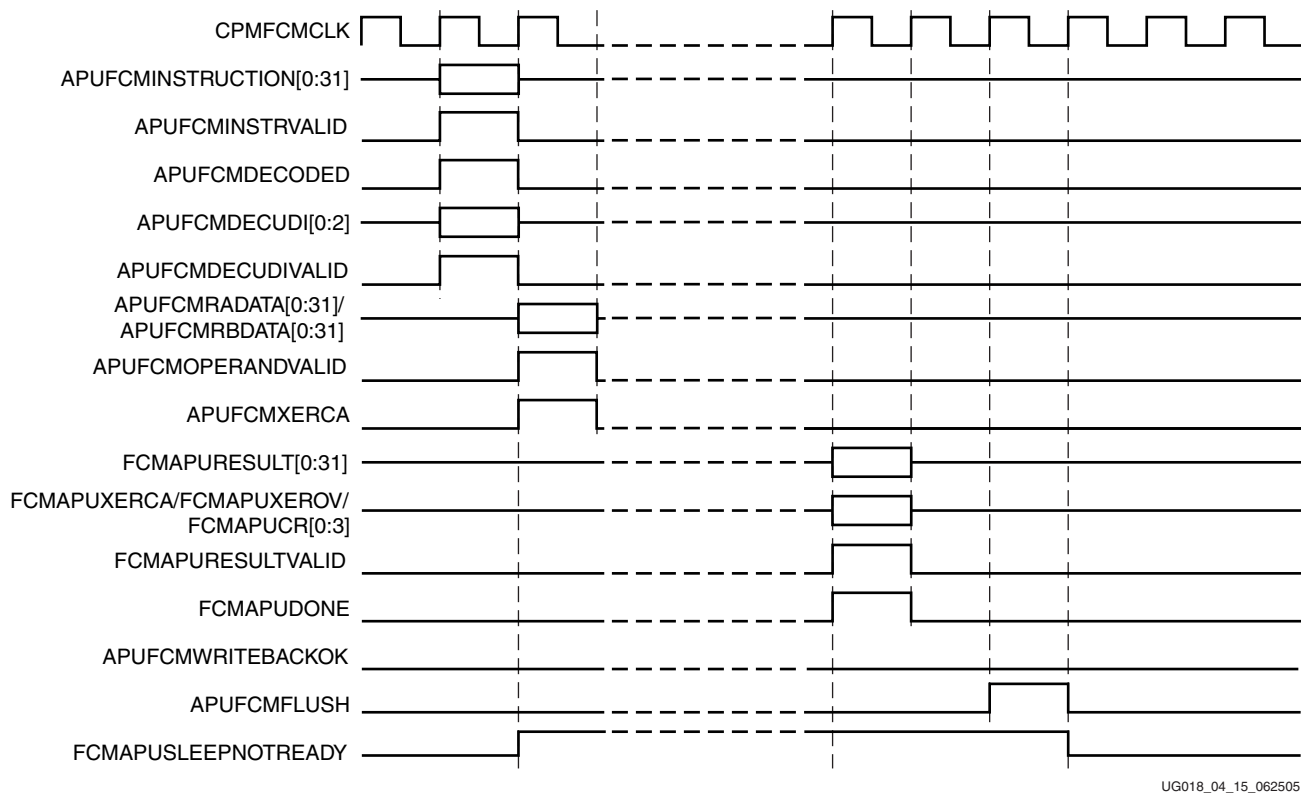
In the example in Figure 4-15, APUFCMRADATA/APUFCMRBDATA, APUFCMOPERANDVALID, and APUFCMXERCA are seen one cycle after the instruction is presented. These signals can arrive at the same time as the instruction signals or anytime afterward.

After the FCM has received APUFCMOPERANDVALID along with the operands it can respond with FCMAPURESULT[0:31], FCMAPURESULTVALID, FCMAPUXEROV, FCMAPUXERCA, FCMAPUCR[0:3], and FCMAPUDONE in the next cycle or anytime afterward.

At this point in the example, APUFCMFLUSH is sent to the FCM. This signal can arrive anytime from the same cycle as the instruction until after FCMAPUDONE has been sent. You will notice that APUFCMWRITEBACKOK remains Low. Once the FCM sees that APUFCMFLUSH has been asserted it cannot update internal registers and must be able to re-execute the same instruction at a later time.

In the example in Figure 4-15, FCMAPUSLEEPNOTREADY is set High the cycle after the instruction signals are received. This is the latest cycle the signal should go High and it should remain High until the instruction is completely finished in the FCM. The signal should not be lowered until APUFCMWRITEBACKOK or APUFCMFLUSH has arrived but could remain High afterward.





**Figure 4-15: APU Controller Decoded Non-blocking Transaction with Flush**

For an APU controller decoded load instruction the following signals will always arrive at the same cycle: APUFCMINSTRUCTION[0:31], APUFCMINSTRVALID, and APUFCMDECODED.

In the example in [Figure 4-16](#) (load quad word), APUFCMLOADDATA[0:31], APUFCMLOADDVALID, and APUFCMLOADBYTEEN[0:3] are seen two cycles after the instruction is presented. These signals can arrive at the same time as the instruction signals or anytime afterward. The timing of these signals depend both on the current situation in the CPU pipeline and the clock ratio between the FCM and CPU.

In the example in [Figure 4-16](#), APUFCMFLUSH is asserted when the second of the four words is sent to the FCM. At this point, the FCM will not receive more words. The FCM cannot update its internal registers and must be able to re-execute the quad word load at a later time. APUFCMWRITEBACKOK remains Low.

FCMAPUSLEEPNOTREADY is set High the cycle after the instruction signals are received. This is the latest cycle the signal should go High and it should remain High until the instruction is completely finished in the FCM (even if APUFCMFLUSH has been sent). FCMAPUSLEEPNOTREADY will not prevent a new instruction from being sent to the FCM.

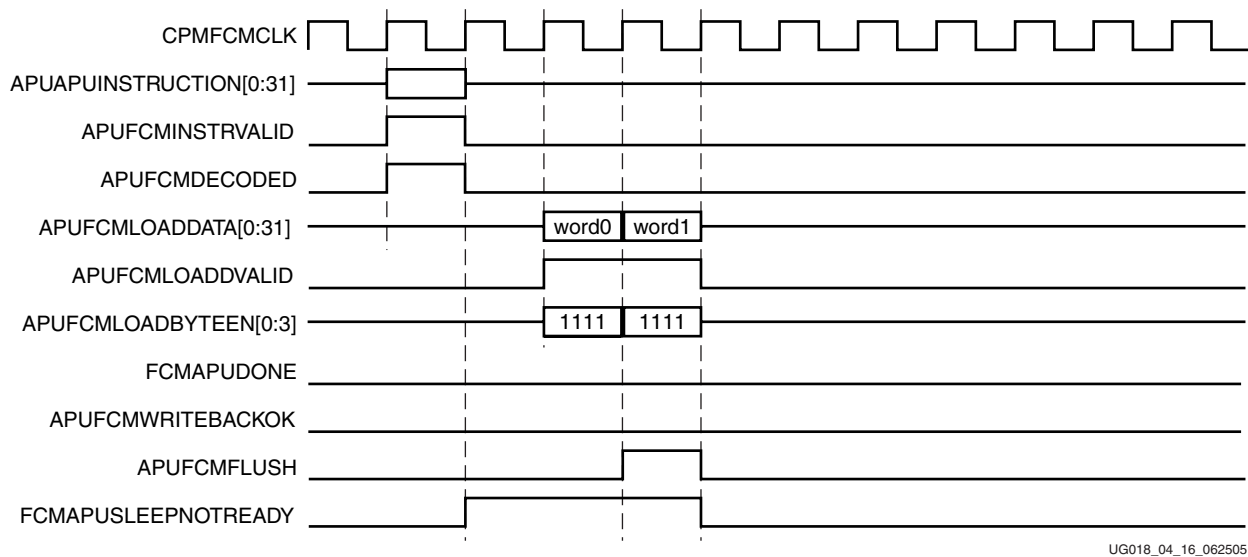


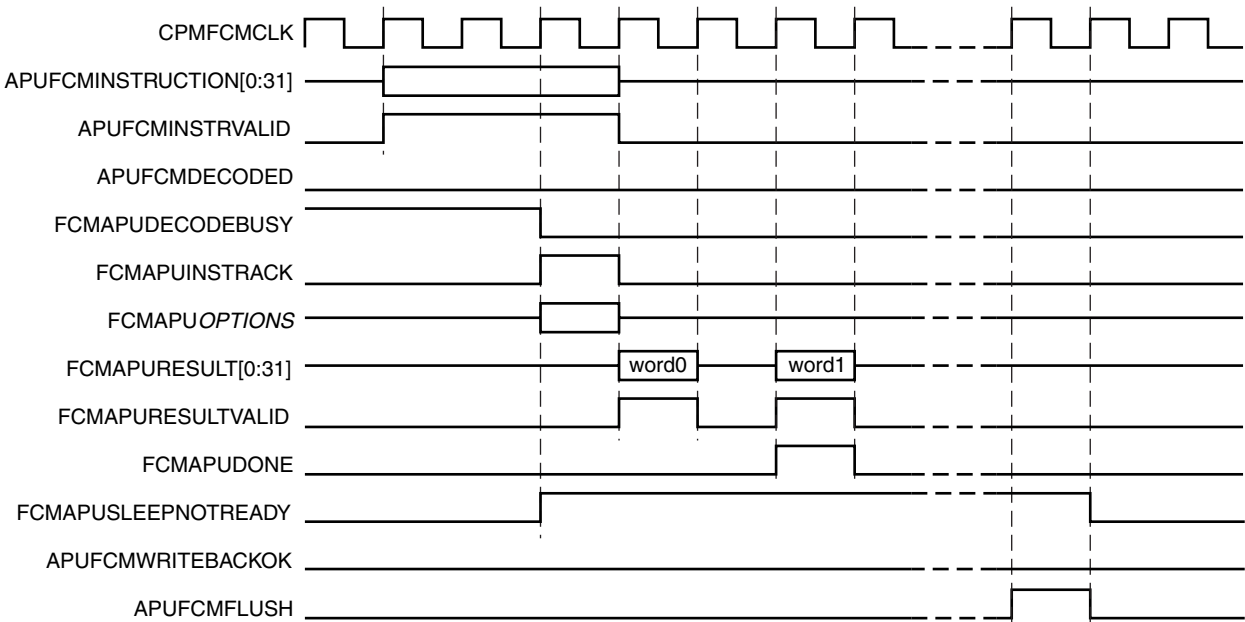
Figure 4-16: APU Controller Decoded Quad Word Load Transaction with Flush

For an FCM decoded store instruction the following signals will always arrive at the same cycle: APUFCMINSTRUCTION[0:31] and APUFCMINSTRVALID. In the example in Figure 4-17, the FCM uses FCMAPUDECODEBUSY before responding with FCMAPUINSTRACK and the execution options.

After the FCM has sent FCMAPUINSTRACK it can send FCMAPURESULT[0:31] and FCMAPURESULTVALID for the first word in the next cycle or anytime after that. FCMAPURESULT[0:31] and FCMAPURESULTVALID must arrive the same cycle for each word. The words do not have to arrive back-to-back as shown in this example. FCMAPUDONE can arrive during the same cycle as the last word or any cycle later.

Because the APU controller has the StoreWBOK bit set to 1, the FCM must wait for APUFCMWRITEBACKOK. However, in this case the store was flushed from the CPU pipeline and APUFCMFLUSH is sent. The FCM cannot update its internal registers and must be able to re-execute the store at a later time.

In the example in Figure 4-17, FCMAPUSLEEPNOTREADY is set High the same cycle FCMAPUINSTRACK is sent. This is the latest cycle the signal should go High and it should remain High until the instruction is completely finished in the FCM. The signal should not be lowered until APUFCMWRITEBACKOK or APUFCMFLUSH has arrived but could remain High afterward.



UG018\_04\_17\_062505

Figure 4-17: FCM Decoded Double Word Store Transaction with Flush and StoreWBOK = 1

