10 - COPROCESSOR EXTENSION REFERENCE GUIDE

10.1 - Introduction

10.1.1 - Chapter Overview

The purpose of this chapter is to serve as a reference guide for the ST122 Coprocessor Extension architecture and to enable the development of coprocessor extensions. This chapter addresses the following topics:

– Rationale

– Coprocessor Extension Features

– Coprocessor Extension Instruction Set Architecture

– Micro-architectural mechanisms and connectivity to the ST122 core.

– Non-uniform instruction latency in a coprocessor.

– Multiple Coprocessor Extensions

– Identifying the presence of a Coprocessor extension.

– Managing interrupts.

– Saving/Restoring the Coprocessor state.

In this document the terms coprocessor, coprocessor cluster, coprocessor extension, and core extension have the same meaning.

10.1.2 - Rationale

Instruction set requirements change with application domains. Current methods for accommodating the instruction set requirements of new application domains primarily follow the approach of extending existing instruction sets. This trend of extending instruction sets contributes to increase the complexity of instruction sets and leads to complex designs.

An alternative to this is to build Coprocessor Extensions specifically targeted to the application. The Extension can be built to support special instructions to accelerate the kernel functions of the application and thereby provide significant improvement in performance, power savings and time to market.

The following application kernels of ADSL were considered as potential Extension candidates while developing the Coprocessor Extension architecture:

– Mapper/De-mapper functions

– CRC function

– Floating-point support for the equalizer functions

– Galois Multiplier/Horner accumulator for Reed-Solomon function

10.2 - ST122 Coprocessor Extension Features

The Coprocessor Extension architecture supports the following features:

– Transaction oriented architecture.

– Tightly coupled to the ST122 Data Unit (DU) appearing like another Data Unit.

– Sharing of the Load/Store sub-system of the ST122.

– Special Coprocessor Extension instructions reserved in the ISA (GP32 and SLIW modes only).

– Extension specific fields in the instruction allowing customization.

– Coprocessor Extensions addressed using a 2-bit ID, which implies the support of up to 4 Coprocessor Extensions.

– Single guarded Coprocessor Extension Instruction issued per cycle and in parallel with another ST122 core instruction in slot-1.

– Two 32-bit source operands from the ST122 DRF (Data Register File), available to the Coprocessor Extension.

– One 40-bit result retrievable per cycle from the Coprocessor Extension, to be written into the ST122 DRF (Data Register File).

– Architecture supporting software pipelining and static scheduling.

– Coprocessor Extension instruction scheduling fully left to the optimizer/compiler.

– Support of multi-cycle Coprocessor Instruction latencies.

– Coprocessor Extension architecture that fully exploits the decoupling techniques of the ST122.

– Support of a trap mechanism to address internal exceptions.

– Use of a FIFO (First-In-First-Out) to hold results which did not reach the ST122 core due to interrupts/ stall conditions local to the ST122 core.

10.2.1 - ST122 Coprocessor Extension Principles

The Coprocessor Extension, in the most simplistic form, performs an application specific function that has a latency of L cycles and a throughput of T cycles when used in a pipelined fashion. Coprocessor Extension instructions follow a split-transaction model where a minimum of two Coprocessor Extension instructions are required to execute an operation on the Coprocessor and retrieve the result of the operation from the Coprocessor.

In the case the result is stored inside the Coprocessor (either as an internal state or in its own Register File) only a single coprocessor instruction is needed to initiate the operation and to provide its internal destination. This is a single-transaction model of the coprocessor.

10.2.2 - Split-transaction model and store /load coprocessor instructions

In case of the split-transaction model, the first instruction provides the operands to the Coprocessor and the second instruction retrieves the result generated by the previous instruction. This split-transaction model allows user-defined latencies of Coprocessor Extension functions.

This split-transaction model therefore implies that one would have to wait for a minimum of L cycles before retrieving the result. In pipelined operation mode (where one coprocessor operation is executed during each cycle), this latency is paid only once, at the beginning of the operation.

10.2.2.1 - Store /Load Coprocessor Extension Instructions

The first Coprocessor Extension instruction **XSD** (Store Data to eXtension) that sends operands has the following mnemonic:

```
XSD  Rn, Rp, #xspc11    // XSD (Store Data to eXtension)
                        // Data to store: registers Rn, Rp of DRF.
                        // #xspc11 specifies 11 user-defined bits.
```

The second Coprocessor Extension instruction **XLD** (Load Data from eXtension) that retrieves result has the following mnemonic:

```
XLD     Rm, #xspc15     // XLD (Load Data from eXtension)
                        // Retrieved data is written into Rm.
                        // #xspc15 specifies 15 user-defined bits.
```

The diagram below illustrates the split-transaction for L = 3 in non-pipelined operation mode. The diagram illustrates the case with optimal scheduling of XLD (Load Data) that is 3 cycles behind XSD (Store Data). The corresponding assembly is shown below.

Table 162: Split-transaction Programming Model (Non-pipelined Operation Mode)

PC [31:0](GP32 Mode)	DU Instruction	
	Slot 1	Slot 0
0x100	DU-OP	XSDRn, Rp, #xspc11
0x108	DU-OP	DU-OP
0x110	DU-OP	DU-OP
0x118	DU-OP	XLD Rm, #xspc15
0x120	DU-OP	XSDRn, Rp, #xspc11
0x128	DU-OP	DU-OP
0x130	DU-OP	DU-OP
0x138	DU-OP	XLDRm, #xspc15

Table 163: Split-transaction Example (Non-pipelined Operation Mode)

Cycle / Internal pipeline Stages	1	2	3	4	5	6	7	
DU-OPF (Operand Fetch) (DOF)	XSD	3 cycles →		XLD				SLOT0
								SLOT1
DU-EX1 (Execute Stage 1)		XSD			XLD			SLOT0
								SLOT1
DU-EX2 (Execute Stage 2)			XSD			XLD ▲		SLOT0
								SLOT1
DU-WB (Write-back Stage) (DWB)				XSD			XLD	SLOT0
								SLOT1
Inst Dispatch to Extension	XSD							
Operands to Extension		XSD						
EX1 Stage of Extension			XSD					
EX2 Stage of Extension				XSD				
Write to ResultQ (FIFO)					XSD			
Result on Extension BUS						XSD ●	Result available to core	

Note: Shaded pipeline stages correspond to the internal stages of the coprocessor.

The diagram illustrates that optimal scheduling of XLD (Load Data) is 3 cycles behind XSD (Store Data) and is equal to the latency of the Coprocessor Extension.

10.2.2.2 - Store data and load data coprocessor extension instruction XSLD

Obviously if the Coprocessor Extension is operated in a pipelined manner one result from the Coprocessor can be produced every cycle. The first definition of the Coprocessor Extension architecture states that only one coprocessor instruction per cycle can be issued.

Given the above constraint the Coprocessor cannot be operated in a pipelined manner with just the XSD and XLD instructions. An instruction of the form XSDLD that implies " store data to extension and load data from extension " is needed. This instruction has the following mnemonic:

```
XSDLD  Rm, Rn, Rp #xspc7   // XSDLD (Store Data to coprocessor,
                           // Load Result from Coprocessor)
                           // Data to store: registers Rn, Rp.
                           // Retrieved data is written into Rm.
                           // #xspc7 specifies 7 user-defined bits.
```

The diagram shown below illustrates the split-transaction for L = 3 operated in a pipelined mode.

Table 164: Split-transaction Programming Model (Pipelined Operation Mode)

PC [31:0] *(GP32 Mode)*	**DU Instruction**		
	Slot 1	**Slot 0**	
0x100	DU-OP	XSD	Rn, Rp, #xspc11 (1)
0x108	DU-OP	XSD	Rn, Rp, #xspc11 (2)
0x110	DU-OP	XSD	Rn, Rp, #xspc11 (3)
0x118	DU-OP	XSDLD	Rm, Rn, Rp #xspc7 (4)(1)
0x120	DU-OP	XSDLD	Rm, Rn, Rp #xspc7 (5)(2)
0x128	DU-OP	XLD	Rm, #xspc15 (3)
0x130	DU-OP	XLD	Rm, #xspc15 (4)
0x138	DU-OP	XLD	Rm, #xspc15 (5)

Note: Shaded pipeline stages correspond to the internal stages of the coprocessor.

The coprocessor extension instructions are tagged with an identifier in parentheses to indicate the XLD matching with the XSD. For XSDLD a pair of parentheses is available, the first one is the identifier for the XSD part and the 2nd holds the identifier for the XLD part.

The "store data to extension" part of an XSDLD instruction is also referred as an "XSD component", while the "load data from extension" part of an XSDLD instruction is also referred to as an "XLD component".

Table 165: Split-transaction Example (Pipelined operation Mode)

Cycle / Internal Pipeline	1	2	3	4	5	6	7	8	
DU-OPF (DOF)	XSD (1)	XSD (2)	XSD (3)	XSDLD (4)(1)	XSDLD (5)(2)	XLD (3)	XLD (4)	XLD (5)	SLOT0
									SLOT1
DU-EX1		XSD (1)	XSD (2)	XSD (3)	XSDLD (4)(1)	XSDLD (5)(2)	XLD (3)	XLD (4)	SLOT0
									SLOT1
DU-EX2			XSD (1)	XSD (2)	XSD (3)	XSDLD (4)(1)	XSDLD (5)(2)	XLD (3)	SLOT0
									SLOT1
DU-WB (DWB)				XSD (1)	XSD (2)	XSD (3)	XSDLD (4)(1)	XSDLD (5)(2)	SLOT0
									SLOT1
Inst Dispatch to Extension	XSD (1)	XSD (2)	XSD (3)	XSDLD (4)(1)	XSDLD (5)(2)				
Operands to Extension		XSD (1)	XSD (2)	XSD (3)	XSDLD (4)(1)	XSDLD (5)(2)			
EX1 Stage of Extension			XSD (1)	XSD (2)	XSD (3)	XSDLD (4)(1)	XSDLD (5)(2)		
EX2 Stage of Extension				XSD (1)	XSD (2)	XSD (3)	XSDLD (4)(1)	XSDLD (5)(2)	
Write to ResultQ					XSD (1)	XSD (2)	XSD (3)	XSDLD (4)(1)	XSDLD (5)(2)
Result to Extension Bus						XSD (1)	XSD (2)	XSD (3)	XSDLD (4)(1)

Note: Shaded pipeline stages correspond to the internal stages of the coprocessor.

10.2.3 - FIFO or Result Queue

10.2.3.1 - Rationale

The previous example shows how one can pipeline the operation of Coprocessor Extension. The example assumes that the result is retrieved using the XLD or the XLD component of XSDLD instruction, if done optimally. However if the ST122 core stalls or switches threads due to an interrupt and is therefore unable to pick up the result in an optimal manner, there is a clear possibility to loose coprocessor results when there is no temporary storage.

To solve this problem, a notion of a Result Queue or a FIFO structure is needed to hold the spill over. This enables the coprocessor extension pipeline to run independently of the ST122 core and not to be influenced by the core stall conditions. The Coprocessor Extension does not stall when the ST122 core stalls.

This implies that XLD would always retrieve the result from the bottom of the FIFO.

The FIFO has a depth D, which is a function of both the coprocessor latency L and the coprocessor throughput 1/T (1 result every T cycles) as given by the formula:

D = Ceiling (L/T) (Note: Ceiling (A) is equal to the smallest integer greater than or equal to A).

Extra entries in the FIFO can ease instruction scheduling. Note that the terms FIFO and ResultQ are used interchangeably.

Up to 4 tightly coupled coprocessors can be connected to the ST122 and each of them must have its own FIFO. Latency and throughput are two parameters of a coprocessor extension. The latencies between different coprocessors can be different; this implies that their respective FIFO depths are also different.

10.2.3.2 - Guards

The coprocessor result will be written into the FIFO and the FIFO valid counter will be updated always independently of the guard value. However the internal state of the coprocessor will be updated only if the guard (the value of which is asserted in stage DU-EX2) is true.

XLD or XLD components will always read from the FIFO and update the FIFO status independently of the guard value. This choice removes the timing constraints seen due to the late arrival of the guard value in DU-EX2 stage.

10.2.4 - Coprocessor Extension without a FIFO

Let us consider coprocessor extension without a FIFO. It includes at least a *result register* that is connected to the extension result bus. This leads to a very simple implementation of a coprocessor. The assembly code and the corresponding diagram illustrating this example is shown below.

In this example it is assumed that CR0, CR1 represent either an internal state or register identifiers for a coprocessor specific register file. In addition xfn indicates a coprocessor function that is specified using the #xspc field of the instruction. Rn, Rp indicate source registers in the ST122 DRF and Rm refers to the ST122 DRF destination.

PC [31:0] *(GP32 Mode)*	**DU Instruction**		
	Slot 1	**Slot 0**	**Comments**
0x100	DU-OP	XSD CR0, Rn, Rp, xfn (1)	CR0 = xfn (Rn, Rp)
0x108	DU-OP	XSD CR1, Rn, Rp, xfn (2)	CR1 = xfn (Rn, Rp)
0x110	DU-OP	XLD Rm, CR0, xfn_nop (3)	Rm = CR0
0x118	DU-OP	XLD Rm, CR1, xfn_nop (4)	Rm = CR1

Table 166: Split Transaction example: Coprocessor Extension without a FIFO

Cycle / Internal Pipeline	1	2	3	4	5	6	7	
DU-OPF (DOF) (Operand Fetch)	XSD (1)	XSD (2)	XLD (3)	XLD (4)				SLOT0
								SLOT1
DU-EX1 (Execute Stage 1)		XSD (1)	XSD (2)	XLD (3)	XLD (4)			SLOT0
								SLOT1
DU-EX2 (Execute Stage 2)			XSD (1)	XSD (2)	XLD (3)	XLD (4)		SLOT0
								SLOT1
DU-WB (DWB) (Write-back Stage)				XSD (1)	XSD (2)	XLD (3)	XLD (4)	SLOT0
								SLOT1
Inst Dispatch to Extension	XSD (1)	XSD (2)	XLD (3)	XLD (4)				
Operands to Extension		XSD (1)	XSD (2)					
Execute & Write Internal State of Extension			XSD (1)	XSD (2)				
Read Internal State & Write Result_Register of Extension				XLD (3)	XLD (4)			
Result to Extension Bus					XLD (3)	XLD (4)		

Note: Shaded pipeline stages correspond to the internal stages of the coprocessor.

The critical pipeline stage is "Read Internal State and Write Result Register" in terms of pipeline timing. The ST122 core stalls do not impact this coprocessor model. In the case the internal state is maintained in coprocessor register file, this model creates register pressure.

This model is well suited for short latency coprocessor solutions. Careful consideration must be paid to the fact that the guard values are available only in DU-EX2 stage.

– Coprocessor extensions with FIFO are efficient when the compiler employs loop pipelining with the added complexity of saving and restoring the FIFO.

– Some coprocessor solutions might not want the complexity of a FIFO. However these solutions would have to deal with register pressure or internal state constraints.

10.3 - ST122 Coprocessor Extension ISA

The Coprocessor Extension instructions XSD, XLD and XSDLD have been already described in previous sections. Since they are guarded, as all coprocessor instructions, their full mnemonics are the following:

– Gx? XSD Rn, Rp, #xspc11

– Gx? XLD Rm, #xspc15

– Gx? XSDLD Rm, Rn, Rp #xspc7

In addition we have one more operation mode where no operands are sent to the coprocessor and no result is retrieved from the coprocessor. This mode enables the coprocessor extension to perform an internal operation and update its internal state. This instruction mnemonic is the following:

– Gx? XOP #xspc19 (#xspc19 specifying 19 user-defined bits.)

The ST122 Coprocessor ISA provides 4 versions of XSD (XSD1, XSD2, XSD3, XSD4) that are identical in the encoding and differ only in the primary op-code description. In addition we have 4 versions of XSDLD (XSDLD1, XSDLD2, XSDLD3, XSDLD4), 2 versions of XLD (XLD1, XLD2) and 2 versions of XOP (XOP1, XOP2). These instructions versions have been created to give more encoding space to co-processors instructions, in addition to the space already provided by the extension specific field.

The Extension specific field is indicated by "xspc". All coprocessor instructions have an extension specific field. The width of "xspc"is described in the Coprocessor Extension set-encoding table shown below:

XSD encoding

[31:28]	[27:24]	[23:22]	[21]	[20:17]	[16:11]	[10:7]	[6:0]
Gx	*xspc [10:7]*	Ext#[1:0]	*xspc [6]*	Rn *Source*	*xspc [5:0]*	Rp *Source*	XSD *(4 Opcodes)*

XSDLD encoding

[31:28]	[27:24]	[23:22]	[21]	[20:17]	[16:11]	[10:7]	[6:0]
Gx	Rm *Dest.*	Ext#[1:0]	*xspc [6]*	Rn *Source*	xspc [5:0]	Rp *Source*	XSDLD *(4 Opcodes)*

XLD encoding

[31:28]	[27:24]	[23:22]	[21:7]	[6:0]
Gx	Rm *Dest.*	Ext#[1:0]	*xspc [14:0]*	XLD *(2 Opcodes)*

XOP encoding

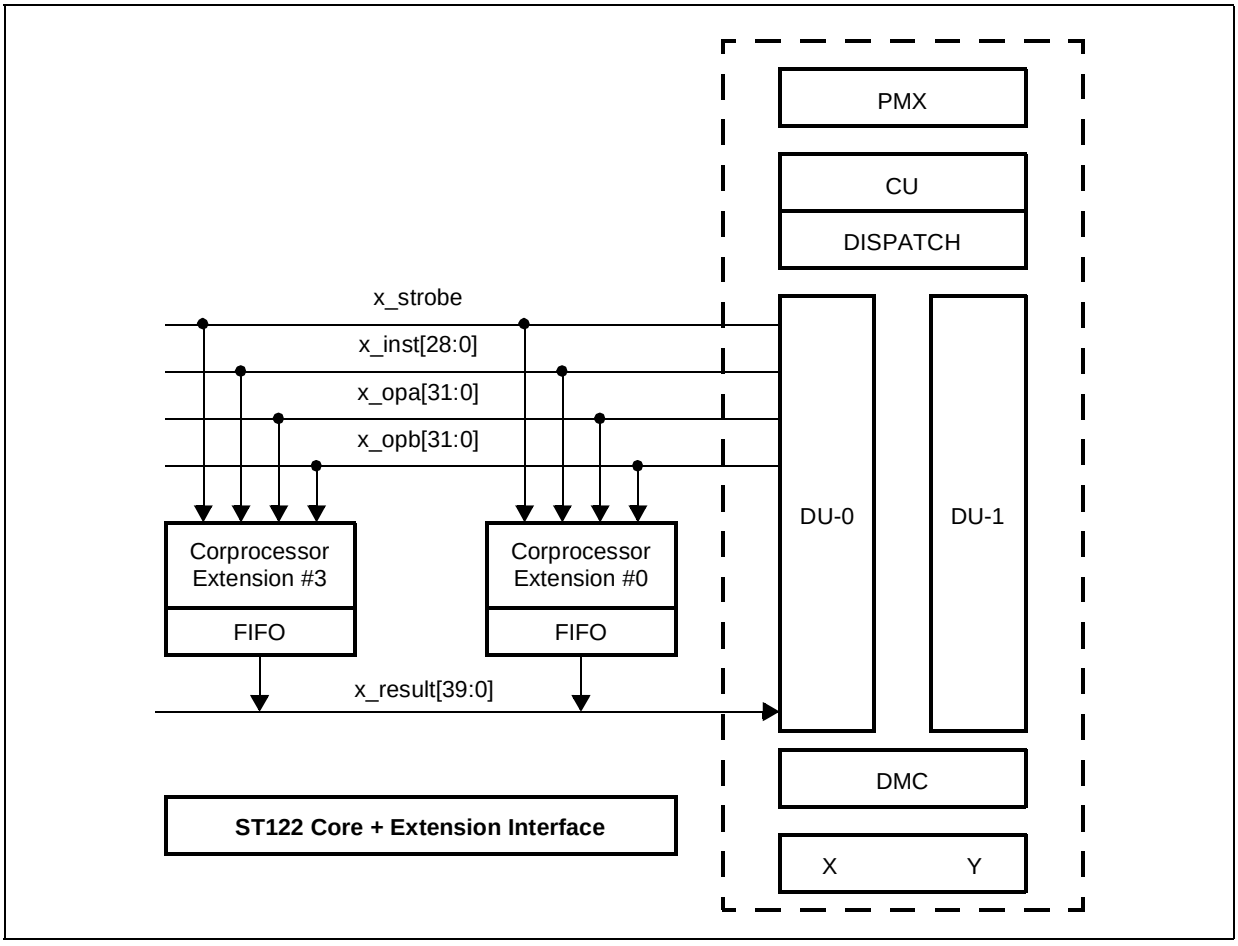
[31:28]	[27:24]	[23:22]	[21:7]	[6:0]
Gx	*xspc [18:15]*	Ext#[1:0]	*xspc [14:0]*	XOP *(2 Opcodes)*

The instruction set encoding holds a 2-bit Ext#[1:0] field that allows addressing of up to 4 Coprocessor Extension units.

The GP32 semantics and syntax is shown in the table below where "xfn" refers to Coprocessor Extension function, "Xa" refers to the result FIFO and "a" refers to the 2-bit coprocessor extension number. When "Xa" is on the Left Hand Side of the equation it refers to writing onto the FIFO and when it is on the Right Hand Side it refers to reading out of the FIFO. Also <i> indicates the forms of XSD such as XSD1, XSD2, XSD3, XSD4.

Mnemonic Syntax	GP32 Semantics	xfn Field Width
Gx? XSD<i> a, Rn, Rp, #xspc11	Gx? [Xa =] xfn (Rn, Rp)	xspc [10:0]
Gx? XSDLD<i> a, Rm, Rn, Rp, #xspc7	Gx? Rm = Xa || xfn (Rn, Rp)	xspc [6:0]
Gx? XLD<i> a, Rm, #xspc15	Gx? Rm = Xa, xfn	xspc [14:0]
Gx? XOP<i> a, #xspc19	Gx? [Xa=] xfn	xspc [18:0]

Figure 107: ST122 Coprocessor Extension Interface



The electrical interface to the Coprocessor Extension includes:

– x_strobe signals that give the following information arriving from different pipeline stages of the ST122 core:
  - x_valid: Coprocessor Extension instruction is valid.
  - x_schedule: Coprocessor Extension instruction that was valid is scheduled and is in EX1 pipe-stage.
  - x_guard: Coprocessor Extension Instruction guard value.
  - x_wb: Coprocessor Extension Instruction is expected to produce a result.

– x_inst: The Coprocessor Extension instruction itself is available from the core.

– x_opa: First Coprocessor Extension operand bus.

– x_opb: Second Coprocessor Extension operand bus.

– x_result: Coprocessor Extension result bus.

Table 167: Coprocessor Extension interface signals

Signal Name	Width	Type for ST122	ST122 Pipe Stage	Description
x_valid [1]	1	Output	DOF	Coprocessor instruction is valid.
x_inst	28:0	Output	DOF	Coprocessor Extension Instruction [31:7 | 3:0]
x_schedule [1]	1	Output	EX1	Coprocessor instruction that was valid in stage DOF has entered DU Pipe EX1 stage.
x_opa	31:0	Output	EX1	First coprocessor operand.
x_opb	31:0	Output	EX1	Second coprocessor operand.
x_wb [1]	1	Output	EX1	Coprocessor instruction with XLD component is in EX1 stage.
x_guard [1]	2	Output	EX2	Guard Gx, (Gx+16) value.
x_result	39:0	Input	EX2	Coprocessor result bus.

Note 1. These signals belong to what is characterized as x_strobe in the diagram above. (DOF: Data Operand Fetch, EX1: Execute Stage 1, EX2: Execute Stage 2).

The ST122 core after issuing a coprocessor instruction in DOF (Data Operand Fetch) stage can potentially stall. In case it stalls the "x_schedule" signal would be inactive in the following cycle. Hence until "x_schedule" becomes active the coprocessor extension should continue to decode the "x_inst" issued in the prior cycle.

10.4 - Use of Guards in Coprocessor Extension Instructions

Each coprocessor instruction is guarded. Coprocessor extensions can be one of two types:

– Type A: Coprocessor without any internal state or register file; a pure execution unit with a result queue. (RQ)

– Type B: Coprocessor with internal state and/or register file, execution unit(s) and a result queue.

The semantics of the coprocessor instruction with regards to the usage of the guards for both types of coprocessors is shown in the table below. "RQ" is used to refer to the result queue or the result FIFO, "Cn" refers to the coprocessor internal state (e.g. internal register), "x_result" refers to the coprocessor result bus connecting to the core.

Note the notation [RQ=] in the table below indicates that the write to RQ is optional based on the coprocessor solution.

Instruction	Coprocessor Semantics	Comment
Gx? XSD Rn, Rp, #xspc11	[RQ =] xfn (Rn, Rp)	Write to RQ is unconditional.
	Gx? [Cn =] xfn (Rn, Rp) [1]	In case of internal register, Cn is written to only if Gx is true.
Gx? XLD Rm, #xspc15	x_result = RQ	RQ is read from unconditionally.
	Gx? Rm = x_result	Rm written to only if Gx is true.
	Gx? xfn ()	Guarded internal operation.
Gx? XSDLD Rm, Rn, Rp, #xspc7	[RQ =] xfn (Rn, Rp)	Write to RQ is unconditional .
	Gx? [Cn =] xfn (Rn, Rp) [1]	In case of internal register, Cn is written to only if Gx is true.
	x_result = RQ	RQ is read from unconditionally.
	Gx? Rm = x_result	Rm written to if Gx is true.
Gx? XOP #xspc19	[RQ=] xfn ()	Write to RQ is unconditional.
	Gx? [Cn =] xfn () [1]	In case of internal register, Cn is written to only if Gx is true.

Note 1. Gx? Cn <= xfn (Rn, Rp) exists only when the coprocessor has an internal state (Type B).

In case of Type A coprocessor extensions (without any internal state), the guard usage is similar to the one used in the ST122; the XSD component or XOP instruction are executed speculatively; only the part of the XLD component writing to the ST122 register file is guarded. It is important to note that writes to result queue (RQ) and reads from result queue are not guarded. Type A coprocessor extensions therefore do not require the guard value from the ST122 core.

In case of Type B coprocessor extensions (with internal state), the internal state update and pure internal coprocessor operations are guarded. XSD and XLD components can be merged into one XSDLD only when both components use the same guard. As examples, typical cases are "not guarded" code (Gx = G15 for both components) or a code block guarded by Gx (a complete set of instructions "if converted" using Gx).

When XSD and XLD components cannot have the same guard, the XSDLD instruction is broken into two instructions XSD and XLD with their respective specific guards. The example in the table below illustrates this case.

In the example below the use of guards is illustrated with the instructions at PC=0x100, 0x110 and 0x120 being guarded by Gx for a Type B coprocessor. The instructions at PC=0x108, 0x118 are guarded by Gy.

PC *(GP32)*	Slot -1	Slot - 0	Instruction Components	
			XSD	XLD
0x100	DU-OP	Gx? XSDRn, Rp, #xspc11	*Gx? XSD*	
0x108	DU-OP	Gy? XSDRn, Rp, #xspc11	*Gy? XSD*	
0x110	DU-OP	Gx? XSDLDRm, Rn, Rp, #xspc7	*Gx? XSD*	*Gx? XLD*
0x118	DU-OP	Gy? XLDRm, #xspc15		*Gy? XLD*
0x120	DU-OP	Gx? XLDRm, #xspc15		*Gx? XLD*

Rule for Type A & Type B Coprocessor Extensions:

1. Writing and updating the FIFO is irrespective of the guard value of the instruction.
2. Reading the FIFO and updating the FIFO counters on a XLD component is irrespective of the guard value of the instruction.

Rule for Type B Coprocessor Extension:

The coprocessor instruction supplying the operands and extension specific opcode followed at a later time by an instruction retrieving the result out of the result FIFO should share the same guard.

An exception to this rule is made when an interrupt handler saves the state of the FIFO using G15 (always true) as guard. It should be noted that the restore routine would use G15 to restore the FIFO as well when exiting the interrupt handler.

10.5 - Managing non-uniform Instruction Latencies in a Coprocessor

It is conceivable that a coprocessor could have several types of instructions with each of the types having different execution latency. The question then arises as to how one can use this extension optimally and what are the rules the optimizer/compiler should enforce.

In answering these questions it is important to remember the rule that **results of coprocessor instructions are retrieved in-order**. Assume that we have two consecutive instructions that are to be sent to the coprocessor, not necessarily in consecutive cycles, denoted by i1, i2 with latencies L1 and L2 respectively. It is also assumed that the execution units with latencies L1, L2 have their results sent to the FIFO via a single port. Given these assumptions the following cases have to be analyzed:

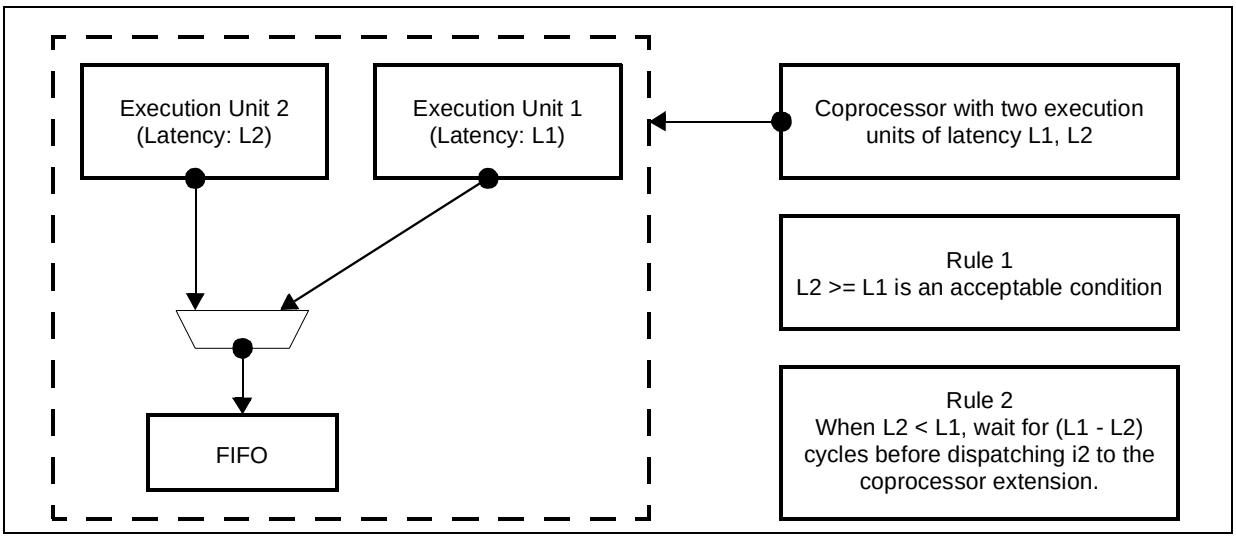
a) L2 >= L1

This implies that the Extension result FIFO would not be updated for (L2 - L1) + 1 Cycles (Note the "+1" is due to the fact that i1 & i2 in the best base are issued in consecutive cycles). This is an acceptable case since they will not be competing for the FIFO and the FIFO is updated in order.

b) L2 < L1

This case can cause a potential conflict when both the execution units could end up competing for the FIFO and i2 might attempt to update the FIFO before i1. Specifically if the ST122 core does not stall the issue of i2, then i2 updates result FIFO before i1. On the contrary if i2 stalls, then it is likely that the result FIFO is updated in order. It is an indeterminate case. Without special scheduling logic to detect this case, it could be catastrophic.

Figure 108: Non-Uniform Instruction Latencies Block Diagram



The discussion above is for two instructions with latencies L1, L2. The scheduler to optimally use the execution engines in the coprocessor should apply these rules.

10.6 - Multiple Coprocessor Extensions

The current architecture definition as noted earlier allows up to 4 different extensions. It is conceivable that these extensions have different instruction latencies.

Each extension by definition has its own FIFO. With this requirement, the scheduling rules of each individual extension need to be respected.

The instructions to the coprocessor can be obviously split in two categories:

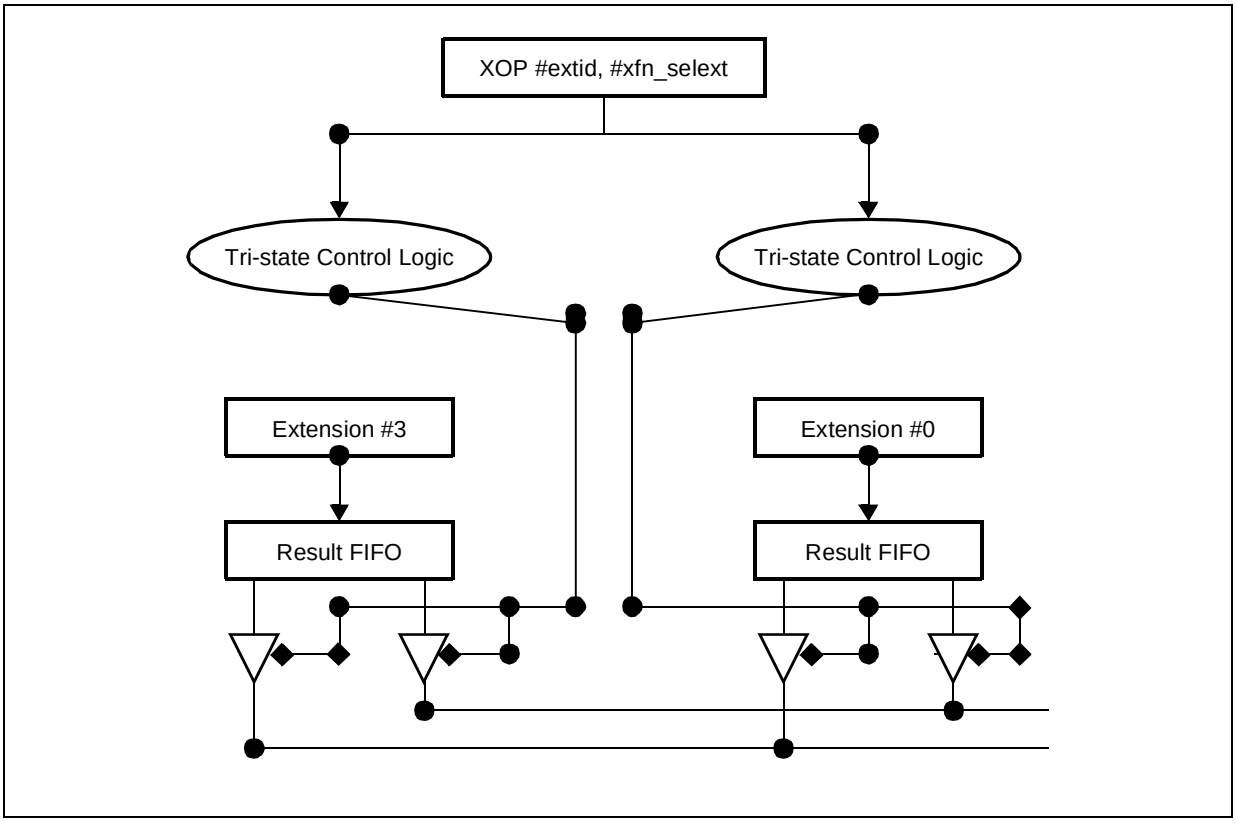
a) One that issues operands optionally (XSD or XOP) and an extension specific operation.

b) One that retrieves the result from the FIFO (XLD or XSDLD)

To get optimum performance a user can issue instructions belonging to group (a) to different extensions by merely selecting the appropriate extension in the extension field.

However instructions belonging to group (b) can be issued only after an Extension is selected as an active extension by issuing an Extension Select instruction. This is discussed in the next section.

10.6.1 - Connecting result buses of multiple coprocessor extensions



Multiple coprocessor extension result buses can be connected in at least one of two ways:

a) OR Logic

It is assumed that each coprocessor has a row of OR logic that is driven by its result FIFO and it immediate neighbor's result FIFO connected in a daisy chain fashion. It is also assumed that except for the selected coprocessor extension, all the others drive a logic zero as output of its result FIFO. With these assumptions we would always be presenting the result FIFO of the selected extension to the core.

This may impose a critical timing constraint. For most coprocessor implementations OR Logic type result bus connectivity is acceptable. However the architecture does allow Tri-state logic type result bus connectivity as discussed in the next paragraph.

b) Tri-State Logic

In this approach all the extensions share the same bus and therefore the problem of routing congestion of case (a) is eliminated. However we have introduced a problem of switching an Extension on and off the tri-state bus.

The following points have to be considered when a tri-state bus is used:

– The Extension scheduled to drive the bus must be identified early enough.

– Settling time must be allowed as dictated by the electrical interface when switching from one Extension to another Extension. This could be in the order of a cycle, and during this time, results on the bus would be ignored.

– It must be also guaranteed that there is always one Extension owning the bus at all times to prevent the bus from entering a tri-state condition.

– The result bus driven to the ST122 must never see tri-state value on the positive edge of the clock.

The notion of an Extension Selection instruction is introduced. All coprocessors are expected to decode this instruction. In response to this "Extension Selection instruction":

– Coprocessor extensions with their two-bit ID different from the ID specified in the "Extension Selection" instruction would have to turn off their tri-state buses in the first half of the cycle.

– The coprocessor extension matching two-bit ID will turn on its tri-state bus in the second half of the cycle.

In order to identify the ID of the selected extension, a special extension specific opcode, labeled "X_chkbsy" instruction, must be supported by the co-processor (see table of reserved extension opcodes). This instruction returns the ID together with a bit that indicates whether the coprocessor extension is busy or not.

After selecting the Extension a minimum of one cycle delay should be allowed before retrieving results from the selected extension.

It is assumed in this discussion that a half-cycle is sufficient to turn-off and turn-on tri-state buses. In the event the analysis indicates that half a cycle is insufficient then it is assumed that logic would be in place to respect the turn-on and turn-off delays. Appropriate delays have to be introduced after the "Extension Selection" instruction.

A typical sequence to work with multiple extensions is indicated below in a pseudo-assembly form where the macro SELEXT is shown as below:

```
macroSELEXTextid
    xop extid, #xfn_selext
    endm
```

PC [31:0]	DU Instruction	
	Slot 1	Slot 0
0x100	DU-OP	XSD Rn, Rp, #xspc11, Ext#0
0x108	DU-OP	SELEXT 1 // Select Ext#1
0x110	DU-OP	XLD Rm, #xspc15, #Ext1
0x118	DU-OP	SELEXT 0 // Select Ext#0
0x120	DU-OP	XLD Rm, #xspc15, #Ext0

Since it takes one cycle to switch the ownership of the tri-state bus, it is suggested that the user uses coarse grain parallelism techniques while operating the coprocessor extensions.

10.7 - Identifying the presence of a Coprocessor Extension

It is recommended that all coprocessor extensions have a unique signature as defined below. If you define your own coprocessor, please get in touch with mail box st100_marketing@st.com to have the exact coprocessor signature defined and registered.

The signature includes:

– A vendor field [31:24].

– An application specific field [23:16].

– A FIFO depth field [15:11].

– A user-defined field [10:8].

– Coprocessor Result Bus Type: 1: Tri-State, 0: OR-Logic.

– Two-bit identifier of the currently selected extension.

– Power Active Status: set to indicate active, reset to indicate inactive.

– Extension present indicators to indicate the presence of Extensions with identifier 0, 1, 2 and 3.

Signature Format for Extension

Vendor Part	Application Specific Field	FIFO Depth	User Field	Copro-Bus	Selected Ext.ID	Power Status	Extensions Present			
[31:24]	[23:16]	[15:11]	[10:8]	7	[6:5]	4	3	2	1	0

The macro for extracting the signature of an extension is shown below:

PC [31:0]	DU Instruction	
	Slot 1	Slot 0
0x100	DU-OP	XSD Rn, Rp, #xspc11, Ext#0
0x108	DU-OP	SELEXT 1 // Select Ext#1
0x110	DU-OP	XLD Rm, #xspc15, #Ext1
0x118	DU-OP	SELEXT 0 // Select Ext#0
0x120	DU-OP	XLD Rm, #xspc15, #Ext0

```
MacroSignatureextID, R0
   XOP extid, xfn_signature
   NOP

   XLD extid, R0, xfn_nop
   NOP
   Endm
```

In the above macro extid refers to Ext# [1:0] field in the GP32 format description of the instruction. Note that we have reserved xfn_signature to read the signature (see table of reserved extension opcodes). The signature is written onto the FIFO. A XLD is used to retrieve the signature, from the currently selected extension.

The identification software first examines the signature of coprocessor with id 0. Examining the "Copro-Bus" bit of the signature, the connectivity of the coprocessor buses can be inferred; 0: indicates OR-Logic type connectivity, 1: indicates tri-state bus type connectivity.

10.7.1 - Coprocessor Result Buses with OR-Logic Connectivity

In this case, the signature instruction can be applied to extensions 0 to 3 and by examining the appropriate "Extensions Present" bit in the signature; one can infer the presence of the coprocessor extension. Also in this case, it is therefore not required to always have a coprocessor id 0. As an example, one can have coprocessor id 3 as the only coprocessor and still be able to identify its presence with this type of result bus connectivity.

10.7.2 - Coprocessor Result Buses with Tri-state Logic Connectivity

In this case if we have only one Extension in the system it should be wired to extension identifier zero. In the absence of any coprocessor extension, the result bus should be tied to ground so that it is clear to the program that none are present.

It is assumed that a four-bit wide bus indicating the presence of extensions 0,1,2 and 3 are wired to all the extensions. This makes it possible for all the extensions to report the presence of the extensions 0 thru' 3. Also the "Copro-Bus" bit of the signature should be returned set to indicate tri-state type connectivity.

10.7.3 - Pseudo-C for Discovering Coprocessors

```
//-------------------------------------------------------------------------------
// Generic C Code to discover extensions
//-------------------------------------------------------------------------------
// Macros used:
//-------------------------------------------------------------------------------
// Signature (int i): Signature of extension "i" using xfn_signature instruction.
// Select (int i); Select extension "i" as the active extension using xfn_selext
//-------------------------------------------------------------------------------
// Function bool bit (x, y) : Returns the bit value of bit "y" in variable "x'.
//-------------------------------------------------------------------------------
bool extension_present[4];        // Extensions present
int extension_signature[4];       // Signature of extensions present.

void discover ()
{
  Select (0);                     // Select coprocessor extension#0
  for (int i=0; i<4; i++) {
    extension_present[i] = false;
  }
  extension_signature[0] = Signature (0);
                                  // Read Signature of Extension 0
                                  // This reports the presence of all extensions
                                  // in case of tri-state connectivity.
  extension_present[0] = bit (extension_signature[0], 0);

  if ( bit (extension_signature[0], 7) { // Result bus type in bit 7
    for ( int i=1; i<4; i++) {          // For Tri-state bus, find the signature
      extension_present[i] = bit (extension_signature[0], i);
      if ( bit (extension_signature[0], i) {// Is the extension present as
                                          // reported by id 0
        Select (i);               // Select the active extension
        extension_signature[i] = Signature(i);
      }
    }
    Select (0);                   // Select Extension 0 again
  } else {                        // Result bus type is OR-Logic
    for ( int i=1; i<4; i++) {
      extension_signature[i] = Signature(i);
      extension_present[i] = bit (extension_signature[i], i);
    }
  }
}
```

10.8 - Managing Interrupts

It is assumed that multiple threads of execution would use semaphores or other synchronization techniques to share a coprocessor extension resource.

In response to an interrupt, the ST122 can elect to save the state of the FIFO and other resources of the coprocessor extension. Assuming that the only resource that needs to be saved is the FIFO, here are some points worth considering.

a. First we need to determine if the Coprocessor has completed execution of all pending operations, if the FIFO has been updated and if the coprocessor instruction pipeline is empty. This is similar to a "coprocessor barrier" instruction.

b. We need to know the state of the FIFO:

- Number of valid entries in the FIFO
- Content of the FIFO.

To determine the condition specified in (a) we can opt for one of two ways:

– Assuming we have only coprocessors with a single uniform latency attached to the core, we would need to issue coprocessor NOP instructions (see table of reserved extension opcodes) equal to the value of the single uniform latency before saving its state.

– For a more complex situation where we have coprocessor(s) with non-uniformity latency, a special extension specific opcode "X_chkbsy" is defined (see table of reserved extension opcodes) to indicate busy status. In response to this instruction, the extension indicates that it is busy by returning a one. In addition it returns the two-bit identifier of the currently selected extension.

The assembly sequence to determine (a) is given below

```
XLD extid, Rn, xfn_chkbsy
Nop
```

Response to X_chkbsy instruction:

Don't-Care Bits	Selected Extension ID	Busy Indicator
[31:3]	[2:1]	0

In the above sequence the extid refers to the currently selected extension. This instruction is unique since in a single transaction we have initiated an operation and retrieved the results of the operation. This is possible since the results bypass the FIFO i.e. the FIFO counter is not updated in response to this instruction.

In theory the extid field is not relevant in this specific case, since XLD always refers to the currently selected extension and FIFO counters are not updated for this X_chkbsy instruction. A program to discover the identifier of the currently active extension without disturbing the FIFO state of the extensions can therefore use this instruction. It is worth noting that this is the only case where we would bypass the FIFO to read a result. Assuming that the extension is not busy, then the save and restore mechanism can proceed in a fashion outlined in section on ST122 Coprocessor FIFO Save/Restore.

10.8.1 - Single Stepping Coprocessor Extension

When the ST122 core operates in single step mode, the coprocessor instruction issuance runs in single step mode as well. However once the instruction is issued to the coprocessor, the coprocessor will continue execution as long as its own clock is not stopped.

In the event the coprocessor chooses to generate an interrupt, then the ST122 core will acknowledge the interrupt provided it obeys the priority rules. A coprocessor needing to indicate an internal condition to the core can use this mechanism (see next section).

10.9 - Managing exceptions in coprocessor extensions.

A signal x_except that can be driven by a coprocessor to indicate that an exception has occurred. The currently active coprocessor should drive this signal in the event that there is more than one coprocessor extension in the system.

This signal can be connected to the ST122 ITC (Interrupt Controller) inputs in order to generate an interrupt (see chapter on exception handling and interrupt mechanism). The ST122 core's interrupt handler in response can examine the exception status recorded in the ST122 core.

This an imprecise exception and the interrupt handler (when x_except is connected to the ST122 ITC) should use additional internal coprocessor specific states to identify the cause of the exception.

10.10 - ST122 Coprocessor Extension Compliance List

10.10.1 - Reserved Extension Op-codes

Some extension opcodes have been predefined in order to allow, for instance a generic OS function, to save/restore the state of a coprocessor without having to know anything about its intimate working.

Function Name	Purpose	Inst.Target	Guard	Instruction Encoding	
Signature	– Identify the presence of a coprocessor. – Identify the number of extensions present in the configuration. – Identify the active extension t owning the tri-state result bus	Currently Selected Extension	G15	**Extension ISA**	**Specific Part**
				XOP1	xfn_signature
					xspc [18:0] = 0x00
Select Extension	– Select the extension to own the tri-state result bus. – Not required for an OR-Logic configuration of coprocessor result buses.	Broadcast	G15	XOP1	xfn_selext
					xspc [18:0] = 0x10
Extension NOP	– To aid instruction scheduling.	ID field of the instruction	Gx	XOP1	xfn_nop
					xspc [18:0] = 0x20
Power-Up	– Power up the selected extension.	ID field of the instruction	G15	XOP1	xfn_pwrup
					xspc [18:0] = 0x30
Power-down	– Power down the selected extension.	ID field of the instruction	G15	XOP1	xfn_pwrdn
					xspc [18:0] = 0x40
Reset FIFO pointer	– First instruction issued as a part of the restore sequence.	ID field of the instruction	G15	XOP1	xfn_rstfifo
					xspc [18:0] = 0x50
Software Reset	– Used to perform a software reset of the Coprocessor; this includes reset of the FIFO counter.	ID field of the instruction	G15	XOP1	xfn_srst
					xspc [18:0] = 0x60
Copy FIFO count	– FIFO state: copy count of valid entries onto FIFO and read FIFO. (Copy ResultQ count to ResultQ)	Currently selected Extension	G15	XLD1	xfn_rdrqcnt
					xspc[14:0] = 0x0
Extension busy status (X_chkbsy instructions)	– To determine if there are any pending operations in the Extension pipeline	Currently Selected Extension	G15	XLD1	xfn_chkbsy
	– To determine the identifier of the currently selected extension.				xspc[14:0] = 0x10
Read FIFO contents	– To read the result FIFO with a coprocessor specific field indicating a NOP; used for saving FIFO contents during context switch.	Currently Selected Extension	Gx	XLD1	xfn_rdrq
					xspc[14:0] = 0x20
Write FIFO count	– FIFO state: write count of valid entries (Write operand A as the ResultQ count)	Currently Selected Extension	G15	XSD1	xfn_wrrqcnt
					xspc[10:0] = 0x0
Write FIFO	– Write to the top of FIFO, used by restore routine (Write operand A to the ResultQ)	ID field of the instruction	G15	XSD1	xfn_wrrq
					xspc[10:0] = 0x10

10.11 - Examples

The following examples are meant to illustrate the operation of this extension interface and the programming model for supporting it.

10.11.1 - Galois Multiplier Extension

The following information is assumed about this extension:

– This extension is referred to as GMX in the Ext# field.

– It is assumed that it is a fully pipelined, two-cycle execution extension engine followed by a single cycle to write to ResultQ.

– The latency seen by the program on the core is 3-cycles.

– It takes two 32-bit operands and produces a 32-bit result performing Galois Multiply on four 8-bit fields.

10.11.1.1 - Programming Model

With software pipelining it is possible to produce one result per cycle after the initial latency as illustrated in the sequence below:

```
    . sliwmd
// Software pipelining: prologue

    10 G15? ld R0, @(P0 !+ 4)
    11 G15? ld R1, @(P1 !+ 4)
    12 G15? xsd gmx, R0, R1, gmpy  // gmx: Ext#
    13 Nop                         // gmpy: Extension Specific

    14 G15? ld R0, @(P0 !+ 4)
    15 G15? ld R1, @(P1 !+ 4)
    16 G15? xsd gmx, R0, R1, gmpy
    17 nop

    18 G15? ld R0, @(P0 !+ 4)
    19 G15? ld R1, @(P1 !+ 4)
    20 G15? xsd gmx, R0, R1, gmpy
    21 nop
// Loop Starts
    22 G15? ld R0, @(P0 !+ 4)
    23 G15? ld R1, @(P1 !+ 4)
    24 G15? xsdld gmx, R2, R0, R1, gmpy
    25 nop

    26 G15? sdw @(P2 !+ 4), R2
    27 nop
    28 nop
    29 nop
// Loop Ends
// Epilogue

    30 G15? xld gmx, R2
    31 nop
    32 G15? sdw @(P2 !+ 4), R2
    33 nop

    34 G15? xld gmx, R2
    35 nop
    36 G15? sdw @(P2 !+ 4), R2
    37 nop

    38 G15? xld gmx, R2
    39 nop
    40 G15? sdw @(P2 !+ 4), R2
    41 nop
```

The basic model is to read the Galois fields from memory and write two 32-bit DRF operands to the Extension using XSD to prime the pipeline. Once the pipeline is set-up, we use XSDLD to retrieve the result and supply the next two 32-bit DRF operands.

Note that in instruction 24 we are able to supply two operands and retrieve the result using a single instruction.

10.11.1.2 - Internal Pipeline Operation

	Cycles											
Core Stages	1	2	3	4	5	6	7	8	9	10	11	Extension Stages
DU-OPF (DOF)	12	16	20	24		24		24				Dispatch to Extension
DU-EX1		12	16	20	24		24		24			Operands to Extension
DU-EX2			12	16	20	24		24		24		Ext-EX1
DU-WB (DWB)				12	16	20	24		24		24	Ext-EX2
					12	16	20	24		24		Write to ResultQ
						12	16	20	24		24	Extension Result Bus

Note : The numbers inside the diagram refer to the instruction numbers in the instruction sequence shown before.

Extension Pipeline Stages	Description
Dispatch to Extension	Coprocessor Extension instruction is dispatched to the coprocessor.
Operands to Extension	Operands from the ST122 are stored into the Coprocessor Extension.
Ext-EX1	Extension Execute pipeline stage 1
Ext-EX2	Extension Execute pipeline stage 2
Write to ResultQ	Extension result is written to ResultQ or result FIFO
Extension Result Bus	Extension result is available on the extension result bus.

This example illustrates the use of the Extension when we have to retrieve operands from memory and store the result of the Extension instruction in memory. In this example we are able to retrieve a result every other cycle, but one can readily observe that the interface does have the capability to produce a result every cycle.

Another interesting observation is that we need a result FIFO that is three deep to hold the results before returning them to the ST122 core. This becomes very important when we take an interrupt inside this loop. The interrupt handler should be able to retrieve the results from the three-entry deep FIFO and store them away in case another thread requires the use of this resource. Once we retrieve the contents of the three-entry deep FIFO we need a mechanism to restore the contents prior to return from an interrupt.

A suggested method to save and restore the FIFO is part of the Appendix.

10.12 - Saving and Restoring Coprocessor State

10.12.1 - Introduction

When the coprocessor is intensively used (pipelined operation mode), the coprocessor pipeline is full:

– At each cycle a new calculation is started.

– At each cycle a new result is written to the DU register file.

As a result, when an interrupt occurs, results produced by the coprocessor cannot be moved into DU during the draining of the pipeline. Therefore these results would be lost without any temporary storage. A FIFO is thus placed between the last pipe stage of the coprocessor and DU to drain these results produced by the coprocessor.

The programmer must make sure that at any cycle the number of pending results is less or equal than the FIFO depth. This way the FIFO cannot overflow if an interrupt occurs otherwise data items produced by the coprocessor will be lost.

10.12.2 - FIFO Behavior

To describe the behavior of this FIFO, we assume that the number of valid entries in the FIFO is a contained into a 32-bit register called FIFO Counter. FIFO reads and writes obey the following rules:

– When **read**, an empty FIFO returns back the last valid result it has already returned. FIFO Counter remains unchanged to 0.

– When **read**, a non-empty FIFO returns back its oldest entry and drops it. **FIFO Counter** is decremented.

– When **read** and simultaneously written, an empty FIFO returns back the last valid result it has returned. A new entry is created with the written data. **FIFO Counter** is incremented.

– When **read** and simultaneously **written**, a non-empty FIFO returns back its oldest result and drops it. A new entry is created with the written data. **FIFO Counter** remains unchanged.

– When **read** and simultaneously **written**, a full FIFO must work as explained in the previous case. This behaviour can be guaranteed by the FIFO micro-architecture or by implementing one more entry than required by the FIFO theoretical size.

– When **written**, an empty FIFO creates a new entry with the written data. **FIFO Counter** is incremented (it takes the value 1).

– When **written**, a non-empty FIFO creates a new entry with the written data. **FIFO Counter** is incremented.

– When **written**, a full FIFO has an UNDEFINED behaviour.

Note: the physical implementation might differ from this concept but same behavior must be kept.

10.12.3 - FIFO Save/Restore Instructions

To be able to save and restore the FIFO, some coprocessor instructions must be dedicated to this task, as shown by the table below. Note that the FIFO width can be either 32 or 40 bits.

Coprocessor Instruction	Description
XLD Rm, xfn_rdrqcnt	– Move the FIFO into the DU register Rm and copy the FIFO counter value into the FIFO. – If the FIFO Counter is 0, it is incremented; else it remains unchanged.
XLD Rm, xfn_rdrq	– Move the FIFO into the DU register Rm. – If FIFO counter is 0, it remains unchanged; else it is decremented.
XOP xfn_rstfifo	– Reset FIFO Counter.
XSD Rn, Rp, xfn_wrrq	Dedicated to 32-bit FIFO: – Move Rn[31:0] into the 32-bit FIFO (Rp[31:0] is lost). – FIFO Counter is incremented by 1. Dedicated to 40-bit FIFO: – Move the concatenation of Rp[23:16] and Rn[31:0] into the 40-bit FIFO (Rp part in 8 MSB and Rn part in 32 LSB). – FIFO Counter is incremented by 1.
XSD Rn, xfn_wrrqcnt	– Move Rn[31:0] into the 32-bit FIFO counter.

10.12.4 - FIFO Save/Restore Routines

– Assuming a 32-bit or a 40-bit FIFO with 4 entries, the FIFO saving routine is shown below:

```
G15? XLD R0, xfn_rdrqcnt;  // state 1 -> state 2
G15? XLD R1, xfn_rdrq;     // state 2 -> state 3
G15? XLD R2, xfn_rdrq;     // state 3 -> state 4
G15? XLD R3, xfn_rdrq;     // state 4 -> state 5
G15? XLD R4, xfn_rdrq;     // state 5 -> state 6
...                        // code saving R0 to R4 somewhere
```

The figure "FIFO Saving Examples" shown below depicts the effects of this saving routine.

– Assuming a 32-bit FIFO with 4 entries the FIFO restoring routine is shown below:

```
...                        // code which restores R0 to R4 from the
...                        // location used by the save routine.
G15? XOP xfn_rstfifo;      // state 7-> state 8
G15? XSD R0, R0 xfn_wrrq;  // state 8-> state 9 (2nd R0 is dummy)
G15? XSD R1, R1 xfn_wrrq;  // state 9-> state 10 (2nd R1 is dummy)
G15? XSD R2, R2 xfn_wrrq;  // state 10-> state 11 (2nd R2 is dummy)
G15? XSD R3, R3 xfn_wrrq;  // state 11-> state 12 (2nd R3 is dummy)
G15? XSD R4, xfn_wrrqcnt;  // state 12   -> state 13
```

– Assuming a 40-bit FIFO with 4 entries the FIFO restoring routine is shown below:

```
...                        // code which restores R0-R3, R10-R13 and R4
                           // from the location used by the save routine.
G15? XOP xfn_rstfifo;      // state 7-> state 8
G15? XSD R0, R10 xfn_wrrq; // state 8-> state 9
G15? XSD R1, R11 xfn_wrrq; // state 9-> state 10
G15? XSD R2, R12 xfn_wrrq; // state 10-> state 11
G15? XSD R3, R13 xfn_wrrq; // state 11-> state 12
G15? XSD R4, xfn_wrrqcnt;  // state 12-> state 13
```

Note: clearing the FIFO counter is not necessary if the FIFO is empty before the execution of the restore routine. This should be the normal case, since a strongly recommended software practice is to consume all the results generated by the instructions issued to the coprocessor.

The figure "FIFO Restoring Examples" depicts the (same) effects of these restoring routines

FIFO Saving Examples

FIFO full before the save routine execution	FIFO half-full before the save routine execution	FIFO empty before the save routineexecution
State 1 FIFO: data 3, data 2, data 1, data 0 (in / out) FIFO counter: 4	2 data 1 data 0	0 data 0
State 2 4 data 3 data 2 data1 4	2 2 data 1	1 0
State 3 4 data 3 data 2 3	1 2	0 0
State 4 4 data 3 2	0 2	0 0
State 5 4 1	0 2	0 0
State 6 4 0	0 2	0 0

FIFO Restoring Examples

FIFO full before the save routine execution	FIFO half-full before the save routine execution	FIFO empty before the save routineexecution
State 7 FIFO (in / out) FIFO counter ? | ? ? ? ? ? | ? ? ? ?	**State 7** ? | ? ? ? ? ? | ? ? ? ?	**State 7** ? | ? ? ? ? ? | ? ? ? ?
State 8 0 | ? ? ? ? 0 | ? ? ? ?	**State 8** 0 | ? ? ? ? 0 | ? ? ? ?	**State 8** 0 | ? ? ? ? 0 | ? ? ? ?
State 9 1 | 4 data 3 data 2 1 | data 0	**State 9** 1 | data 0 1 | data 0	**State 9** 1 | data 0 1 | data 0
State 10 2 | data 0 2 | data 0	**State 10** 2 | data 1 data 0 2 | data 1 data 0	**State 10** 2 | 0 data 0 2 | 0 data 0
State 11 3 | data 2 data 1 data 0 3 | data 0	**State 11** 3 | 2 data 1 data 0 3 | data 0	**State 11** 3 | 0 0 data 0 3 | 0 0 data 0
State 12 4 | data 3 data 2 data 1 data 0 4 | data 0	**State 12** 4 | 2 2 data 1 data 0 4 | data 0	**State 12** 4 | 0 0 0 data 0 4 | 0 0 0 data 0
State 13 4 | data 3 data 2 data 1 data 0 4 | data 0	**State 13** 2 | data 1 2 | data 0	**State 13** 0 | data 0 0 | data 0