

Pico Processor **Specification**

Kernel Processor

Revision 0.2

31 July 2025

English ver.

Copyright 2011 ArchiTek All Rights Reserved

Confidential and Proprietary

1. Overview
 - 1.1. Introduction
 - 1.2. Main Parameters
 - 1.3. Implementation Parameters
2. Signal Lines
 - 2.1. Control Bus Interface
 - 2.2. PSS Interface
 - 2.3. Aux Bus Interface
 - 2.4. Memory Interface (Data Use)
 - 2.5. Memory Interface (Parameter Read Use)
 - 2.6. Register File Interface (Write Port X)
 - 2.7. Register File Interface (Write Port Y)
 - 2.8. Register File Interface (Read Port)
 - 2.9. Scalar Register Interface
 - 2.10. Interrupt for psss
 - 2.11. Utility
3. Architecture and Operation Description
 - 3.1. Notation
 - 3.2. Structural Overview
 - 3.3. Driving Interface (Initiator)
 - 3.4. Notes on Fragmentation
 - 3.5. Start and End of Instructions
 - 3.6. Constant Setting
 - 3.7. Integer Operations
 - 3.8. Floating-Point Operations
 - 3.9. Floating-Point Adjustment
 - 3.10. Operands
 - 3.11. Constants
 - 3.12. Type Conversion
 - 3.13. Scalarization
 - 3.14. Branch Control
 - 3.15. Register Update Control
 - 3.16. Condition Codes (CC) and User Flags (Flag)
 - 3.17. Memory Access
 - 3.18. Flow Control
4. Instruction Description

- 4.1. Overview
- 4.2. Control Instructions (Cntl)
- 4.3. Integer Arithmetic Instructions (Int)
- 4.4. Memory Instructions (Mem)
- 4.5. Floating-Point Instructions (Mad)
- 4.6. Hyperfunction Instructions (Hyp)
- 5. Control Register Description
 - 5.1. Overview
 - 5.2. Definition
 - 5.3. Details
 - 5.3.1. Reset Register
 - 5.3.2. Clock Register
 - 5.3.3. Info Register
 - 5.3.4. IntEn Register
 - 5.3.5. Cntl Register
 - 5.3.6. IRVal Register
 - 5.3.7. SeedX Register
 - 5.3.8. MonitorXY Register
 - 5.3.9. MonitorZW Register
 - 5.3.10. MonitorPC Register
 - 5.3.11. BreakXY Register
 - 5.3.12. BreakZW Register
 - 5.3.13. BreakPC Register
- 6. Application Notes
 - 6.1. Processing Volume
 - 6.2. Regarding Hyperbolic Instructions
 - 6.3. Image Processing Examl
 - 6.3.1. Mandelbrot Rendering

1. Overview

1.1. Introduction

- The Pico Processor (Kernel Processor version, hereafter referred to as “kp”) is a compact, high-performance processor. Designed with a SIMD (Single Instruction Multiple Data) architecture, it delivers excellent performance for applications involving repetitive operations such as image processing.
- It supports easy addition of arithmetic pipelines, increased computation precision, and finer granularity of pipeline stages, offering excellent scalability in both functionality and performance.
- The instruction set includes control instructions, integer arithmetic, memory access, floating-point arithmetic, and transcendental functions (e.g., trigonometric functions). All instructions execute in a single cycle, although memory access may incur system-level penalties.
- At implementation, arithmetic pipelines can be parallelized in powers of two. Similarly, SRAM capacity can be scaled in powers of two to support an increased number of logical processors.
- Users do not need to be aware of the degree of parallelism in the physical processors when programming. Performance efficiency does not degrade due to increased parallelism.
- The processing units support 16-bit and 32-bit integers, as well as half-precision and single-precision floating-point formats. It handles registers and constants in units of 16 entries \times the number of banks.
- It supports conditional branching and conditional register updates.
- The assignment of command fields varies depending on the Ver register. In addition, a simplified 64-bit instruction format is provided alongside the standard 128-bit instructions.

1.2. Key Parameters

- **Memory Bus**
Memory Read/Write: 32-bit \times 2PNR
Instruction Read: 64-bit
- **Physical Processors**
 \times 2PNR (PNR is determined at implementation)
- **Logical Processors**

- × 2LNR (LNR is determined at implementation; proportional to SRAM capacity: 64 Bytes
× 2LNR)
- **Throughput**
Up to 2PNR instructions per cycle
- **Clock**
Undefined (depends on the implementation process)

1.3. Implementation Parameters

Parameter Name	Description	Default Value
LNR	• Radix of logical processor number	12
PNR	• Radix of physical processor number	3
BLR	• Radix of Burst Length for Command List Loading • Configure Burst Unit for 64-bit Memory Access	1 (4 以下)
BKR	• Radix of register bank number	3
CLR	• Radix of pss channel number	6
BSR	• Data R/W の Radix of burst length • Configure Burst Unit for 64-bit Memory Access	2 (4 以下)
SLR	• Radix of scalar number	11

2. Signal

2.1. Control Bus Interface

Signal Name	IO	Pol	Source	Description
cntlReq	I	+	clk	• Request signal • Evaluate cntlGnt
cntlGnt	O	+	clk	• Grant signal

cntlRw	I	+	clk	<ul style="list-style-type: none"> R/W signal Evaluate cntlReq & cntlGnt 0: Write 1: Read
cntlAddr[31:0]	I	+	clk	<ul style="list-style-type: none"> Address signal Evaluate cntlReq & cntlGnt
cntlWrStrb	I	+	clk	<ul style="list-style-type: none"> Write strobe signal Evaluate cntlWrAck
cntlWrAck	O	+	clk	<ul style="list-style-type: none"> Write acknowledge signal
cntlWrData[31:0]	I	+	clk	<ul style="list-style-type: none"> Write data signal Evaluate cntlWrAck
cntlRdStrb	I	+	clk	<ul style="list-style-type: none"> Read strobe signal Evaluate cntlRdAck
cntlRdAck	O	+	clk	<ul style="list-style-type: none"> Read acknowledge signal
cntlRdData[31:0]	O	+	clk	<ul style="list-style-type: none"> Read data signal Sync cntlRdAck
cntlIrq	O	+	clk	<ul style="list-style-type: none"> Interrupt signal Level hold type

2.2. PSS Interface

Signal Name	IO	Pol	Source	Description
iVld	I	+	clk	<ul style="list-style-type: none"> Pipeline start valid signal
iStall	O	+	clk	<ul style="list-style-type: none"> Pipeline start stall signal
iCID[CNR-1:0]	I	+	clk	<ul style="list-style-type: none"> Logical channel number
iEnd[3:0]	I	+	clk	<ul style="list-style-type: none"> Information of end of indexes

iAddr[31:0]	I	+	clk	<ul style="list-style-type: none"> • Address to fetch context data • Evaluate iVld & !iStall • iAddr[31:BKR+4] and iAddr[1:0] indicates program start address • iAddr[BKR+3:4] indicates offset of logical register bank when Cntl.BankAdd=1 otherwise indicates start address[BKR+3:4] • iAddr[3] indicates offset of program counter when Cntl.BaseAdd=1 otherwise indicates start address[3] • iAddr[2] indicates select of volume whether iDelta or TR register
iDelta[15:0]	I	+	clk	<ul style="list-style-type: none"> • Transfer volume • Evaluate iVld & !iStall
iIndex[64:0]	I	+	clk	<ul style="list-style-type: none"> • Five coordinates to specify the processing • Evaluate iVld & !iStall
oVld	O	+	clk	<ul style="list-style-type: none"> • Pipeline end valid signal
oStall	I	+	clk	<ul style="list-style-type: none"> • Pipeline end stall signal

2.3. Aux Bus Interface

Signal Name	IO	Pol	Source	Description
auxReq	O	+	clk	<ul style="list-style-type: none"> • Request signal • Evaluate cntlGnt
auxGnt	I	+	clk	<ul style="list-style-type: none"> • Grant signal
auxRxw	O	+	clk	<ul style="list-style-type: none"> • R/W signal • Evaluate cntlReq & cntlGnt 0: Write 1: Read
auxAddr[31:0]	O	+	clk	<ul style="list-style-type: none"> • Address signal • Evaluate cntlReq & cntlGnt
auxWrStrb	O	+	clk	<ul style="list-style-type: none"> • Writ strobe signal
auxWrAck	I	+	clk	<ul style="list-style-type: none"> • Write ack
auxWrData[31:0]	O	+	clk	<ul style="list-style-type: none"> • Write data signal • Evaluate auxAck
auxWrMask[3:0]	O	+	clk	<ul style="list-style-type: none"> • Write mask signal

auxRdStrb	O	+	clk	• Read strobe signal
auxRdAck	I	+	clk	• Read acknowledge signal
auxRdData[31:0]	I	+	clk	• Read data signal

2.4. Memory Interface (Data Use)

Signal Name	IO	Pol	Source	Description
miReq[2 ^{PNR} -1:0]	O	+	clk	• Request signal
miGnt[2 ^{PNR} -1:0]	I	+	clk	• Grant signal
miRxxw[2 ^{PNR} -1:0]	O	+	clk	• R/W signal
miAddr[35*2 ^{PNR} -1:0]	O	+	clk	• Address signal
miWrStrb[2 ^{PNR} -1:0]	O	+	clk	• Write strobe signal
miWrAck[2 ^{PNR} -1:0]	I	+	clk	• Write acknowledge signal
miWrData[2 ^{PNR} *3-2:1:0]	O	+	clk	• Write data signal
miWrMask[2 ^{PNR} *3-1:0]	O	+	clk	• Write mask signal
miRdStrb[2 ^{PNR} -1:0]	O	+	clk	• Read strobe signal
miRdAck[2 ^{PNR} -1:0]	I	+	clk	• Read acknowledge signal
miRdData[2 ^{PNR} *3-2:1:0]	I	+	clk	• Read data signal

2.5. Memory Interface (Parameter Read Use)

Signal Name	IO	Pol	Source	Description
meReq	O	+	clk	• Request signal
meGnt	I	+	clk	• Grant signal
meAddr[31:0]	O	+	clk	• Address signal
meStrb	O	+	clk	• Read strobe signal
meAck	I	+	clk	• Read acknowledge signal
meFlush	O	+	clk	• Read flush signal
meData[63:0]	I	+	clk	• Read data signal

2.6. Register File Interface (Write Port X)

Signal Name	IO	Pol	Source	Description
rxWE[2 ^{PNR} -1:0]	O	+	clk	• Write enable signal
rxWB[BKR-1:0]	O	+	clk	• Bank signal
rxWA[15:PNR]	O	+	clk	• Address signal
rxWD[2 ^{PNR} *32-1:0]	O	+	clk	• Write data
rxWM[3:0]	O	+	clk	• Write mask
rxWS[3:0]	O	+	clk	• Register number (0-15)

2.7. Register File Interface (Write Port Y)

Signal Name	IO	Pol	Source	Description
ryWE[2 ^{PNR} -1:0]	O	+	clk	• Write enable signal
ryWB[BKR-1:0]	O	+	clk	• Bank signal
ryWA[15:PNR]	O	+	clk	• Address signal
ryWD[2 ^{PNR} *32-1:0]	O	+	clk	• Write data
ryWM[3:0]	O	+	clk	• Write mask
ryWS[3:0]	O	+	clk	• Register number (0-15)

2.8. Register File Interface (Read Port)

Signal Name	IO	Pol	Source	Description
rgRE[15:0]	O	+	clk	• Read enable signal each of register number
rgRB[16*BKR-1:0]	O	+	clk	• Bank signal each of register number
rgRA[(16*(16-PNR)-1:0]	O	+	clk	• Address signal each of register number
rgRD[2 ^{PNR} *32-1:0]	I	+	clk	• Write data each of register number

2.9. Scalar Register Interface

Signal Name	IO	Pol	Source	Description
scWE[2 ^{PNR} -1:0]	O	+	clk	• Write enable signal

scWA[SLR-1:0]	O	+	clk	• Address signal
scWD[31:0]	O	+	clk	• Write data
scRE[2 ^{PNR} -1:0]	O	+	clk	• Read enable signal
scRA[(2 ^{PNR} *SLR-1:0]	O	+	clk	• Address signal
scRD[2 ^{PNR} *32-1:0]	I	+	clk	• Read data
scRS	I	+	clk	• Read strobe from another core access
scRSD	I	+	clk	• Read strobe from another core access (ahead)

2.10. Interrupt for pss

Signal Name	IO	Pol	Source	Description
iqWE	O	+	clk	• Interrupt pulse
iqWA[CLR-1:0]	O	+	clk	• Interrupt vectorl
iqWD	O	+	clk	• Interrupt value

2.11. Utility

Signal Name	IO	Pol	Source	Description
rstReq	O	+	clk	• Internal reset signal to reset the external system
rstAck	I	+	clk	• Acknowledge of rstReq
reg_swap	O	+	clk	• Indicates 64bit word swap
fReq	I	+	clk	• 1 clock early request against the miReq signal • Use to generate gate signal (for memory controller)
pReq	O	+	clk	• 1 clock early request against the meReq signal • Use to generate gate signal (for memory controller)
gate[4:0]	O	+	clk	• Gated clock control signal signifying condition of each internal block(expansion case is gate) • Equal to {cor, mad, mem, int, all}
gclk[4:0]	I	+	clk	• Gated clock(expansion case is gclk)
mclk	I	+	clk	• Gated clock(for internal sram)
clk	I	+	clk	• Clock
reset	I	+	clk	• Synchronous reset signal

reset_n		-	clk	• Asynchronous reset
---------	--	---	-----	----------------------

3. Architecture and Operation Description

3.1. Notation

- This document uses the following notations and abbreviations for explanation.

Symbol Legend:

Symbol	Description
.	A delimiter or register concatenation indicating hierarchy X.Y means X is the upper (higher) part, and Y is the lower part.
:	A delimiter indicating the hierarchy of instructions Int:tfr represents a transfer (tfr) instruction within the Int instruction category.
[:]	Indicates a bit range A[X:Y] represents data A with MSB at position X and LSB at position Y.
R _b [n]	Represents the n-th 32-bit register in bank b of the Vector Register (b = 0 to 2BKR - 1, n = 0 to 15) (The b designation may be omitted in some cases)
R _b [n]@shift	Represents the n-th 32-bit register in bank b of the Vector Register belonging to a logical processor offset by shift (The b designation may be omitted in some cases)
SR[n]	Represents the n-th 32-bit register of the Scalar Register (n = 0 to 255)

CC _{sub}	<ul style="list-style-type: none"> • A 4-bit condition code (N: Negative, Z: Zero, V: Overflow, C: Carry) The subscript sub indicates the associated instruction. (It may be omitted if the context is clear) • The condition code from an Int instruction result is CCInt • The condition code from a Mad instruction result is CCMadHyp 命令結果の CC は CC_{Hyp}
Flag	User flags, generated by condition codes (CC), by the user directly, or via parameters
CCR	Represents the 8-bit Condition Code Register A packed register containing both CC and Flags
TR	Represents the 32-bit Context Register
PC	Program Counter (in multiples of 16)
C _n	Represents a 32-bit constant in the Constant Register (n = 0 to 7) C0 is directly specified in normal instructions, while C1 to C7 are preloaded using set instructions.
mem _n [address]	Access of n bytes to a byte address n = 1, 2, 4 (defaults to 4-byte word access when omitted)
mem _n (X,Y)	2D access of n bytes, assigning R[n] to both X and Y n = 1, 2, 4 (defaults to 4-byte word access when omitted)

A_{sub} B_{sub} X_{sub}	<ul style="list-style-type: none"> Input operands <p>The subscript sub indicates the associated instruction. (It may be omitted if the context is clear)</p> <p>Int Instructions:</p> <ul style="list-style-type: none"> • AInt, BInt, XInt <p>Mem Instructions:</p> <ul style="list-style-type: none"> • AMem, BMem <p>Mad Instructions:</p> <ul style="list-style-type: none"> • AMad, BMad, XMad <p>Hyp Instructions:</p> <ul style="list-style-type: none"> • AHyp, BHyp •
X_{sub} Y_{sub}	<ul style="list-style-type: none"> Output operands; part of X is also used as an input <p>The subscript sub indicates the associated instruction. (It may be omitted if the context is clear)</p> <p>Int Instructions:</p> <ul style="list-style-type: none"> • XInt, YInt <p>Mem Instructions:</p> <ul style="list-style-type: none"> • YMem <p>Mad Instructions:</p> <ul style="list-style-type: none"> • XMad, YMad <p>Hyp Instructions:</p> <ul style="list-style-type: none"> • XHyp, YHyp
Mod_{sub}	<p>Operand modifiers</p> <p>The subscript sub indicates the associated operand (A, B, or X) (It may be omitted if the context is clear)</p>
$\$_{\text{sub}}$	<p>Operand modified by Modsub</p> <p>The subscript sub indicates the associated operand (A, B, X, or Y)</p>
Set_{sub}	<p>Controls register updates for output operands</p> <p>The subscript sub indicates the associated operand (X or Y)</p>

CX, CY	XY coordinates assigned to the logical processor
@(CX,CY)	Indicates memory access using XY coordinates $R[n] = @(CX, CY)$ means transferring the data at XY coordinates to register $R[n]$
Signed	Signed integer, represented in two's complement format
Unsigned	Unsigned integer
Command	Instruction, 16 bytes
Cntl	Control instructions within a Command (e.g., branches and term)
Instr	Arithmetic instructions within a Command
Infinity	Infinity in IEEE 754 floating-point representation
NaN	Not a Number (NaN) in IEEE 754 floating-point representation

Color coding for Registers and Commands:

	Register
	Cntl Instruction (Control Instruction)
	Int Instruction (Integer Instruction)
	Mem Instruction (Integer Instruction)
	Mad Instruction (Integer Instruction)
	Hyp Instruction (Integer Instruction)
	Constant

3.2. Architecture Overview

- As shown in Figure 1, the Kp is a processor in which N processor elements sequentially process M sets of register files. Although there are N physical processors, it appears logically as if there are M processors.

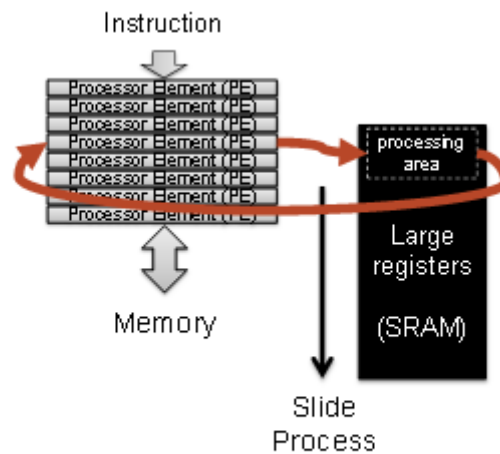


Figure 1 Architecture of Kp

- Unlike conventional architectures, Kp executes a large number of operations from a single instruction before moving on to the next. By expanding the cycle count between instructions from 1 to several tens, it efficiently avoids penalties that would normally occur—such as having to wait for the result of the previous instruction before continuing computation.

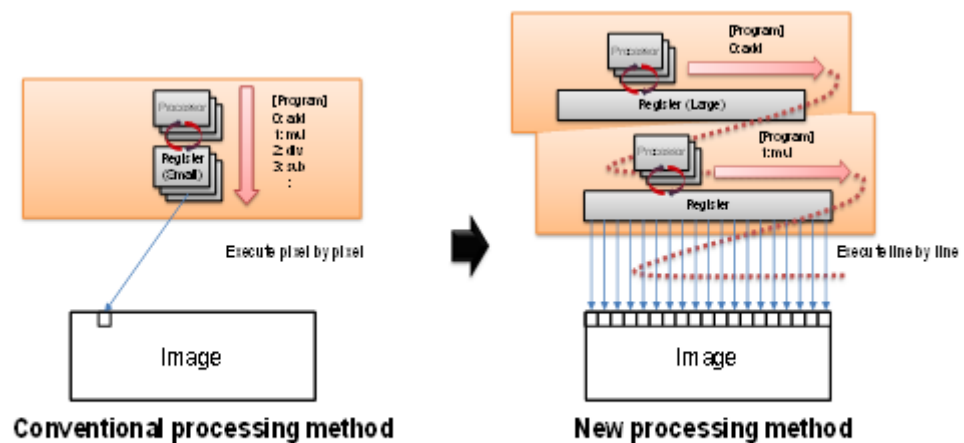


Figure 2 Processing of Kp

- The processor consists of four pipelines, each of which can read data from a register file and update the results.

A single logical processor can access $R[n]$, CCR, and the shared registers SR, TR, and Cn.

- Vector Register $Rb[n]$: 16 32-bit registers per bank, with n banks (b depends on the implementation)
- Condition Code Register CCR: A single 8-bit register indicating the result of an operation
- Scalar Register SR: 2SLR 32-bit registers
- Context Register TR: A single 32-bit register (internally managed per instance using the iCID signal)
- Constant Register C: 8 32-bit registers (C0 is directly specified in the Command)

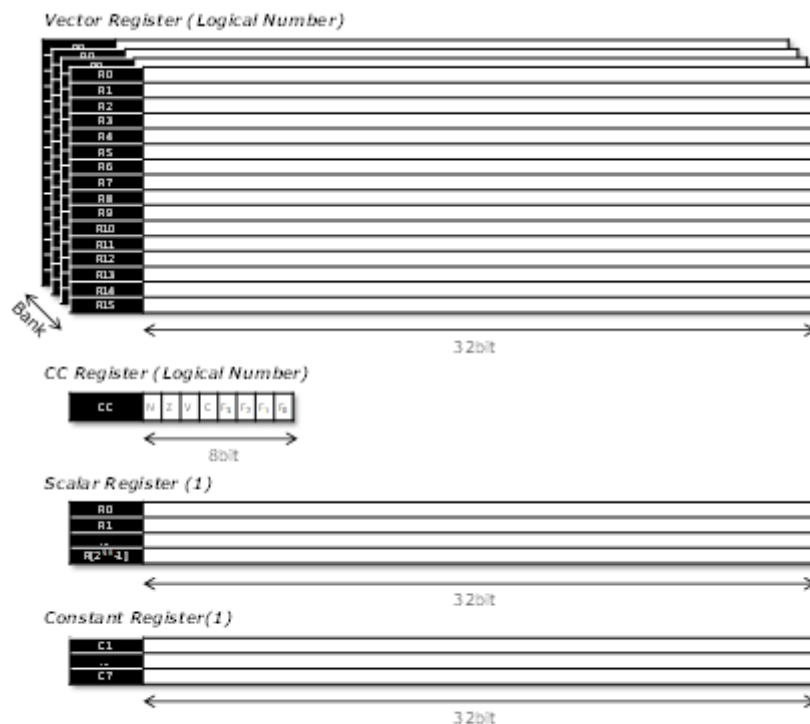


Figure 4 Register File

- The Int, Mem, Mad, and Hyp instructions can each specify arbitrary registers. However, the registers to be updated must generally be exclusive. If not exclusive, the results are not guaranteed.
- All operands can access $R[n]$ of different logical processors with an offset that is a multiple of 64. By applying a fixed offset to all register accesses, the number of usable registers can be expanded through partitioning.
- Operands can also access $R[n]$ of different logical processors with an offset ranging from -32 to +31, or in multiples of 64. This is used for overlapping calculations such as those involving neighboring pixels.
- Scalar processing allows integration of computations across logical processors. For example, scalar values such as the sum or accumulation of designated registers from multiple logical

processors can be computed. This requires a set of multiple instructions.

- The condition code (CC) attached to computation results consists of the following flags: N for negative, Z for zero, V for overflow, and C for carry. Flags are generated by combining these CC values. Flags can be accumulated up to 4 bits and are used in branches and conditional register updates.
- Branching is performed collectively after the specified processing unit (iDelta) of the instruction has completed. All logical processors' user flags are collected and evaluated, and if the condition is true, a jump to the specified instruction occurs. A delayed jump method is used, meaning the next instruction (delay slot) is executed before jumping.
- Conditional register updates evaluate the flag for each logical processor individually to determine whether to perform an update. Unlike branching, this allows handling different conditions for each logical processor.

3.3. Drive Interface (Initiator)

- The PSS sends XYZW indices to the Kp's Initiator (PX, PY, PZ, PW). The configuration for PSS (e.g., processing units) is pre-arranged in memory. The PSS manages multiple configurations (numbered N , depending on implementation) via time division and drives the Kp after scheduling.

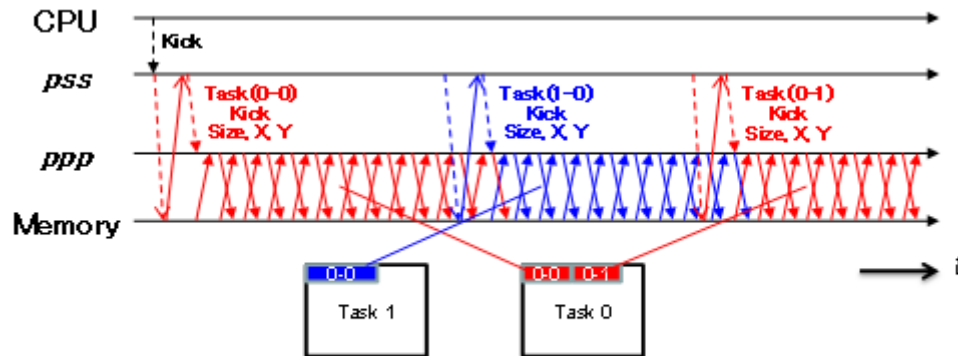


Figure 5 Resource Sharing using *pss*

- The Initiator reads from memory based on the starting address of the instructions sent by the PSS and sets up the pipelines. It continues processing either for the duration indicated by iDelta from PSS, or by the amount specified in TR, until a termination instruction appears. The amount of processing is determined by iAddr[2]: if '0', iDelta is referenced; if '1', TR is used.
- Among the XY indices, the X index is assigned to each logical processor's identifier. Therefore, increasing the range of the X index is a key point for maximizing performance.

When needed, a scan conversion function can be used to generate new XY coordinates.

- The bank of the logical processor's Vector Register can be adjusted by setting the Cntl.BankAdd register to '1', which causes iAddr[BKR+3:4] to be added.
- For the Program Counter, if the Cntl.BaseAdd register is '1' and iAddr[3] is also '1', iAddr[31:4] is added (or iAddr[31:BKR+4] if Cntl.BankAdd is also '1').

3.4. Fragmentation Considerations

- In fragmented processing, alternating between different instruction sets generally does not cause inconsistencies. However, there are some exceptions:

Case	Description
Referencing past register results	<p>If another task is inserted, the register values may be overwritten. Consider the following:</p> <p>Use exclusive register access by utilizing register banks and offsets</p> <p>Save necessary data to memory after each processing step</p>
Processing very short fragments (iDelta signal)	<p>Performance per cycle will degrade.</p> <p>The following considerations are necessary:◦</p> <p>Making fragment length as long as possible</p>

- An instruction set is indivisible; other tasks cannot interrupt it. An instruction set refers to the complete sequence of operations from the first to the last instruction—typically corresponding to one image line (iDelta).
- When transitioning from the last instruction to the next line, an interrupt from another process may occur. If you need to retain previous results (e.g., data in the register file), configure the PSS to avoid interruption until all processing for one frame is complete, or save register file contents to memory beforehand.
- To interrupt tasks at finer time intervals within a line, the instruction set must be broken into smaller segments. This requires inserting a termination instruction periodically. Since termination instructions occupy exclusive fields, they don't significantly alter the instruction set. However, if interrupted, register values may be corrupted—avoid reuse or

manage banks exclusively.

- Kp includes registers that count flags based on true/false conditions. There are 2CLR such registers managed by the iCID[CLR-1:0] signal. If CLR is minimized to reduce hardware size, even different iCID values from PSS may be treated as identical by Kp, potentially causing incorrect behavior. As with registers, prevent interruption until all processing is complete or avoid generating duplicate iCID[CLR-1:0] values.
- When referencing TR[iCID[CLR-1:0]] for workload, writes to this register must be tracked as part of workload management.
- If the workload is less than 30 cycles \times number of physical processors N (i.e., 2PNR), NOPs are automatically inserted to preserve data dependency consistency. For example, in 4-way parallel Kp image processing, widths below 120 degrade performance. Two methods can be used to avoid this, and they can be combined.

Method	Description
The data dependency check interval is specified by Command.Cntl (default is 0)	Adjust Cntl:lat for each instruction If the write register number is not used between the current PC and the instruction lat cycles before, no NOP is inserted
Increase the fragment size For 2D or higher-dimensional processing, expand the processing range along the X index and reduce it along the Y index	Use the scan conversion function For example, a 64 \times 64 operation can be processed as 256 \times 16

3.5. Instruction Start and Termination

- Kp sequentially executes the program starting from the specified instruction address. Once a termination code appears in the program, it completes the current instruction and then enters standby mode.
- Until a termination instruction is encountered, the program counter increments continuously while executing instructions.

3.6. Constant Setup

- The Constant Registers C1 to C7 and special parameters are preloaded using set

instructions. If they are not used, setting them is unnecessary.

- Since set instructions are not vector operations, they consume only the minimum number of cycles.
- Part of C1 along with C2 to C4 is set using the Set0 instruction, and the remaining part of C1 along with C5 to C7 is set using the Set1 instruction.

Note that C0 is specified directly in regular instructions.

3.7. Integer Arithmetic

- Integer arithmetic supports **signed types** for addition and subtraction, and **unsigned types** are also supported for other operations. All operations have a throughput of one cycle.
- Only signed operations are supported for addition and subtraction. Carry can also be included in calculations:

$Y = A + B$	$X = A++$
$Y = A + B + \text{Carry}$	$X = A++$
$Y = B - A$	$X = A++$
$Y = B - A - \text{Carry}$	$X = A++$

For multiplication (mul), the lower 32 bits of the result are used. Both signed and unsigned types are supported:

$Y = A * B$	$X = A$
-------------	---------

Division (div) computes both quotient and remainder simultaneously. It also supports 32-bit left-shifted dividends. Both signed and unsigned types are supported:

$Y = A / B$	$X = A \% B$
$Y = (A \ll 32) / B$	$X = (A \ll 32) \% B$

Ternary operations are supported. The condition is specified by any bit of the CCR:

$$Y = \text{CCR}[\text{X}[2:0]] ? B : A$$

Scalar registers can be indexed using any register value:

$$Y = \text{SR}[B]$$

Arbitrary shift operations (in two's complement format) are possible, with shift amount specified by A. A positive shift shifts left, and a negative shift shifts right. The empty bits resulting from a shift can be filled using one of the following modes, specified by operand X (0–3):

$Y = \text{rot}(B)$ $\#X[1:0] = 0$ Circular (rotate)
 $Y = \text{rot}(B)$ w/ carry $\#X[1:0] = 1$ Fill with carry
 $Y = \text{rot}(B)$ w/ zero $\#X[1:0] = 2$ Fill with 0
 $Y = \text{rot}(B)$ w/ lsb, msb $\#X[1:0] = 3$ Fill with LSB (if positive) or MSB (if negative)

The number of logical 1s (i.e., the Hamming weight) in operand B returns a value from 0 to 32:

$$Y = \sum B[i] \quad i=0-31$$

Boolean algebra operations are supported via bitwise binary functions. The function type is specified using operand X (0–15):

$$Y[i] = f(A[i], B[i]) \quad \text{Type} = \#X[3:0] \quad i=0-31$$

Type	$Y[i]$
0	0
1	$\sim A[i] \ \& \ \sim B[i]$
2	$A[i] \ \& \ \sim B[i]$
3	$\sim B[i]$
4	$\sim A[i] \ \& \ B[i]$
5	$\sim A[i]$
6	$A[i] \wedge B[i]$
7	$\sim A[i] \mid \sim B[i]$
8	$A[i] \ \& \ B[i]$
9	$A[i] == B[i]$
10	$A[i]$
11	$A[i] \mid \sim B[i]$
12	$B[i]$
13	$\sim A[i] \mid B[i]$
14	$A[i] \mid B[i]$
15	1

Type conversion is supported. Conversions are available between float (single precision) and integer types, as well as between float (single precision) and float (half precision).

Values that exceed the range of 32-bit integers are clamped.

Both signed and unsigned types are supported.

$Y = f(B)$

$X = A_{\epsilon}$

	f(B)
hf2f	From floating-point (half precision) to floating-point (single precision)*
f2hf	From floating-point (single precision) to floating-point (half precision)*
bf2f	From floating-point (8-bit) to floating-point (single precision)*
f2bf	From floating-point (single precision) to floating-point (8-bit)*
si2f	From signed integer to floating-point (single precision) C2[30:23] - 0x7F is added to the exponent part of the result
us2f	From unsigned integer to floating-point (single precision) C2[30:23] - 0x7F is added to the exponent part of the result
f2si	From floating-point (single precision) to signed integer C2[30:23] - 0x7F is added to the exponent part for reference Values greater than or equal to 0x7FFFFFFF are clamped to 0x7FFFFFFF Values less than or equal to 0x80000000 are clamped to 0x80000000 NaN is converted to 0
f2ui	From floating-point (single precision) to signed integer C2[30:23] - 0x7F is added to the exponent part for reference Values greater than or equal to 0x7FFFFFFF are clamped to 0x7FFFFFFF Values less than or equal to 0x80000000 are clamped to 0x80000000 NaN is converted to 0

*Lane designation is specified in the Mod field.

Special type conversions are supported, including exponent extraction from floating-point (single precision) (f2log), conversion to integer and fractional parts (f2sif, f2usif), and normalization (f2ef).

$Y = f(B)$

$X = g(B)$

	f(B)
f2ef	<p>r when decomposing floating-point value B into $r * m$</p> <p>r is a floating-point value representing a scaling factor that is a power of $2^{<sup>n</sup>}$</p> <p>The condition is $0.5 \leq m < 2.0$</p>
f2flog	<p>Convert the exponent part of a floating-point number to a floating-point value</p> <p>The operation is performed on the value obtained by subtracting 0x7F from the exponent in IEEE representation.</p> <p>For example:</p> <p>If the result is 0x00, then the output is 0x00000000 (0.0)</p> <p>If the result is 0x01, then the output is 0x3F800000 (1.0)</p> <p>If the result is 0xFF, then the output is 0xBF800000 (– 1.0)</p>
f2ilog	Corresponding integer (0–255)
f2sif	<p>Extract the integer part of a floating-point value and convert it to a signed integer (up to 8-bit precision)</p> <p>The exponent is adjusted by adding C[30:23] – 0x7F</p>
f2uif	The above, for unsigned integer

	g(B)
f2ef	<p>m when decomposing a floating-point value B into $r * m$</p> <p>m is a normalized floating-point value satisfying $0.5 \leq m < 2.0$</p> <p>The condition is that r must be a power of $2^{<sup>n</sup>}$</p>
f2flog f2ilog	<p>Convert to a floating-point number whose exponent part becomes 0x7F</p> <p>However, if the exponent is 0 or the value is NaN, the input value is retained.</p>
f2sif f2uif	<p>Extract the fractional part of a floating-point number and convert it to a floating-point value</p> <p>The exponent is adjusted by adding C[30:23] – 0x7F</p>

3.8. Floating-Point Operations

- Floating-point operations support IEEE 754 single-precision format.
- The following operations are available, some of which can be executed simultaneously.
- All operations have a throughput of one cycle.

$$Y = A * B$$

$$Y = A + B$$

$$Y = B / A$$

$$Y = \min(A, B)$$

$$Y = \max(A, B)$$

$$Y = A * B + X$$

$$Y = (A < B) ? A : A - B$$

$$Y = A * \sin(B)$$

$$X = A * \cos(B)$$

$$Y = A * \sinh(B)$$

$$X = A * \cosh(B)$$

$$Y = \operatorname{atan}(B, A)$$

$$X = \sqrt{A^2 + B^2}$$

$$Y = \operatorname{atanh}(B, A)$$

$$X = \sqrt{A^2 - B^2}$$

Depending on the operation result, NaN or Infinity may be generated.

No exception interrupts are triggered.

However, the overflow flag may change, and an interrupt can be asserted to the system if necessary.

$$A + B / A - B$$

B \ A	0	A	Infinity	NaN
0	0	A	Infinity	NaN
B	B, -B	A + B, A - B	Infinity	NaN
Infinity	Infinity	Infinity	Infinity, NaN	NaN
NaN	NaN	NaN	NaN	NaN

$$A * B$$

B \ A	0	A	Infinity	NaN
0	0	0	NaN	NaN
B	0	A * B	Infinity	NaN
Infinity	NaN	Infinity	Infinity	NaN
NaN	NaN	NaN	NaN	NaN

$$B / A$$

B \ A	0	A	Infinity	NaN
0	NaN	0	0	NaN
B	Infinity	B / A	0	NaN

Infinity	Infinity	Infinity	NaN	NaN
NaN	NaN	NaN	NaN	NaN

$A \cdot \sin(B)$, $A \cdot \cos(B)$

$\begin{array}{c} B \\ \backslash \\ A \end{array}$	0	A	Infinity	NaN
0	0	0, A	NaN, Infinity	NaN
B	0	$A \circ B$	Infinity	NaN
Infinity	0	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN

*lefthand is $\sin()$, righthand is $\cos()$

$A \cdot \sinh(B)$, $A \cdot \cosh(B)$

$\begin{array}{c} B \\ \backslash \\ A \end{array}$	0	A	Infinity	NaN
0	0	0, A	NaN, Infinity	NaN
B	0	$A \circ B$	Infinity	NaN
Infinity	NaN	Infinity	Infinity	NaN
NaN	NaN	NaN	NaN	NaN

*lefthand is $\sinh()$, righthand is $\cosh()$

$\text{atan}(B/A)$, $\text{atanh}(B/A)$

$\begin{array}{c} B \\ \backslash \\ A \end{array}$	0	A	Infinity	NaN
0	0	A, 0	Infinity, 0(0.5)	NaN
B	B, 0.25	$A \circ B$	Infinity, 0(0.5)	NaN
Infinity	Infinity, ± 0.25	Infinity, ± 0.25	NaN	NaN
NaN	NaN	NaN	NaN	NaN

*lefthand is square root(), righthand is angle

*hyperbolic function: $|B| > |A| \rightarrow \text{NaN}$, $|B| = |A| \rightarrow \text{Infinity}$

*() is selected according to sign flag

- Type conversion between integer and floating-point is performed using the Command.Int:tfr instruction.

Additionally, the Int:mem instruction includes functionality for coordinate conversion (floating-point \rightarrow integer).

- In Hyp instructions, angles used for trigonometric functions are expressed such that 1.0 corresponds to 2π radians.
The result of atan ranges from -0.5 to 0.5.
- For hyperbolic functions processed by Hyp instructions, accurate results for atanh cannot be obtained unless the condition $|A| \geq |B|$ is met.
However, it is possible to determine whether the condition is met based on the result of $\sqrt{(A^2 - B^2)}$:
If the result overflows, then $|A| < |B|$; otherwise, $|A| \geq |B|$.
- For hyperbolic functions such as sinh and cosh used in Hyp instructions, due to algorithm precision limitations, the absolute value of B should be less than 1.12.
- In Mad instructions, conversion from fixed-point (when Exp is '0') to floating-point is performed automatically.
However, if the operand is specified as 1.0 (in the case of multiplication) or 0.0 (in the case of addition), the other operand's value is directly passed through as the result.
Since the input and output remain unchanged, this feature can be used via the Mad instruction to access integer values from another logical processor.
- Using the Hyp instruction, exponential functions can be computed as shown below.
However, for $\log(x)$, since the atanh condition $|A| \geq |B|$ limits the input range, adjustments are required.
You need to manipulate Command.Exp to either zero out or extract the exponent in order to bring the value within the allowable range.

$$\begin{aligned}\exp(x) &= \sinh(x) + \cosh(x) \\ \log(x) &= 2 \operatorname{atanh}(x-1, x+1) \\ \operatorname{sqr}(x) &= \operatorname{sqr}((x+0.25)^2 - (x-0.25)^2)\end{aligned}$$

3.9. Floating-Point Adjustment

- The Mad instruction allows up to four adjustment options to be applied simultaneously with the result computation. However, only one of these options can be applied at a time.
- **Exponent Bit Shift (Left or Right)**
Refers to C2[30:23].
The offset is computed by subtracting 0x7F from the IEEE-format exponent.
If the result is positive (in two's complement format), the exponent is shifted left; if negative, it is shifted right.
After shifting, 0x7F is added back to produce the new exponent.

• Exponent Offset

Refers to C2[30:23].

The offset is calculated by subtracting 0x7F from this value.

If C2[30:23] is 0, it is treated as a special case and the offset becomes zero.

• Rounding

Refers to C2[30:23].

Indicates the rounding unit.

For example:

- 0x7F indicates standard rounding
- 0x80 rounds to the nearest multiple of 2
- 0x7E rounds to the nearest multiple of 0.5

If C2[30:23] is 0, it is treated as a special case and interpreted as 0x7F.

方式	Description
Trunc	Rounds toward zero
Round	Rounds to the nearest value if exactly halfway, rounds to make the LSB of the mantissa 0)
Floor	Rounds toward negative infinity
Ceil	Rounds toward positive infinity

• Pass-Through Assignment

Normally, custom-defined denormalized values (i.e., fixed-point representations) are normalized through computation.

Denormalized values may be generated when using random numbers or setting constants.

To avoid normalization, use an addition-type Mad instruction and set one operand to 0.0.

3.10. Operands

- The operands for Int, Mad, and Hyp instructions can each be selected independently.

The number of operands required depends on the instruction.

If there are two outputs writing to the same R[n], the result will be the logical OR of both

$$\underline{R[0]} = \sin(\underline{R[1]})^{\text{c}}$$

$$\underline{R[0]} = \cos(\underline{R[1]})^{\text{c}}$$

$$\rightarrow \underline{R[0]} = \sin(\underline{R[1]}) \mid \sin(\underline{R[1]})^{\text{c}}$$

Type	Instruction	Input	Output
Int	scalar, ham	1	1
	tern	2	1
	add, sub, mul, div, rot, bool, mem	2	2
Mad	mul, add, min, max, diff	2	1
	mad	3	1
	mul, div, sin, cos, sinh, cosh, atan, atanh	2	2

- Operands A and B can refer to $R[n]$ of another logical processor using *operand shift*, as described below.

The total shift amount is $\text{Offset0} + \text{Offset1}$:

- Offset0 = k** ($k = -32$ to 31), or **Offset0 = 64k** ($k = 0$ to 63)
 - Offset1 = 64** ($n = 0$ to 63 , $m = \text{selection index: 1, Y, Z, W}$)
- Operand shift has restrictions: specifying different shift amounts for the same $R[n]$ in a single instruction is not supported (behavior is undefined):
 - $R[x] = R[0] * R[0]@4$ # Invalid: $R[0]$ is not unique
 - $R[x] = R[2]@-2 * R[2]@-2$ # Valid: same register, same shift amount
- When using high-granularity operand shifts ($k = -32$ to 31), writing to the same reference register in the immediately following instruction is not allowed (result of the previous instruction is not guaranteed). Insert NOPs if necessary.
- When using register references with relative logical processor numbers that exceed the defined endpoints, edge handling becomes active.

Edge	Description
0	<p>Default Mode</p> <p>Values that exceed the endpoints are replaced with a fixed value.</p> <p>if ($X < 0 \parallel X \geq \text{width}$) then default value(C2)</p> <p>width represents the coordinate of the maximum endpoint + 1.</p> <p>If width is 0, it is treated as infinite (same applies below).</p>
1	<p>Copy Mode</p> <p>Values that exceed the endpoints are replaced with the register value of the nearest endpoint.</p> <p>if ($X < 0$) then value[0]</p> <p>if ($X \geq \text{width}$) then value[width-1]</p>

2	Ring Mode Values that exceed the endpoints wrap around to the opposite endpoint's register value. if $(X < 0)$ then $\text{value}[(X + \text{width})\% \text{width}]$ if $(X \geq \text{width})$ then $\text{value}[(X - \text{width})\% \text{width}]$
3	Reserved

- Operands can be modified using the modify field, which allows specification of register modifications, constants, CCR, and relative positions of logical processors. These modifications are selectable for each operand individually.

Note: The default setting (0) disables both operand reading and writing.

Available operand types include:

- Standard register access
- Negation and absolute value modifiers
- Constants C0–C7
- Operand constants from -127 to 127
- Random numbers
- CCR (Condition Code Register)
- SCR (register values selected by relative processor position)
- TR (Context Register)
- PC (Program Counter)
- Random numbers use the xorshift algorithm.

Each logical processor generates a different sequence.

The initial seed is set via the Seed Register, not by the Command itself.

Initialization is controlled through a Command instruction.

- There are restrictions when specifying operand $Rb[n]$ from bank b.

You cannot access different banks using the same register number.

For example, the combination below is invalid, as it attempts to access register number 1 from both Bank 3 and Bank 4, resulting in undefined behavior:

Int Instruction: $R[0] = R4[1] + R3[1]$

3.11. Constants

- Cn represents utility constants defined using the Set instruction.

The usage varies depending on the instruction type.

Typically, they are configured as 8 freely usable constants, but for branch instructions or

memory access, specific parts of the constants function as control codes.

Cn	Case	Description
C1[31:0]	Scan Conversion	Converts the PSS indices X, Y, Z, W into unique X, Y coordinates for each logical processor
C2[31:0]	Edge Handling Fixed Value Exponent Correction	Fixed value used outside the valid region and exponent specification used in Mad instruction rounding (only C2[31:24] is referenced)
C3[15:0]	Branch Target (Lower)	Branch Target (Lower)
C3[31:16]	BranchTarget (Upper) / Count	Branch Target (Upper) Or the maximum number of loop iterations
C4[31:16]	Flag Generation	Sets the bit that evaluates to true based on one of the 16 possible combinations of CC or Flag after computation
C5,C6,C7	MemoryAccess (mem)	16-bit configuration for each of the X and Y window sizes in 2D access (used to perform edge handling)
		Configure format-related settings
		Configure the update amount for 2D access
		Configure the starting address and access type

There are special parameters that cannot be referenced via registers.

These can only be defined using the Set instruction.

Name	Case	Description
Step	Logical Processor Shift	Select the index using Step[7:6] Multiply Step[5:0] by 64 to shift the register number of the logical processor to be used
Shrink	Limiter for Scalarization	Configure the limit for 1/N scalarization Specified as a power of 2; 0 means no limit

Util	utility	Util[2:0] is added from the most significant bits of the address during memory access Util[3] skips dummy operations during loops (for performance improvement; it does not affect the result)
------	---------	--

- Operand constants can be used as immediate constants and specified when modifying registers.
- For Int and Mem instructions, the 8-bit register selection field is sign-extended to 32 bits:

$R[x] = \{24\{x[7]\}, x[7:0]\}$ // $\{24\{x[7]\}$ replicates bit $x[7]$ 24 times

- For Mad and Hyp instructions, the 8-bit register selection field is converted into a single-precision floating-point value.

In this case, the MSB represents the sign, and the remaining bits are interpreted as an integer.

$R[x] = \underline{x[7]} ? \text{itof}(\underline{x[6:0]}) | \underline{0x80000000} : \text{itof}(\underline{x[6:0]})$

3.12. Type Conversion

- When referencing operand constants or parameter constants, automatic type conversion is performed depending on the instruction used.

For example, the operand constant 1 becomes 0x00000001 for an Int instruction and 0x3F800000 for a Mad instruction.

Note that constants like C_n are not subject to automatic type conversion.

- When selecting the upper or lower 16 bits from a 32-bit word register for use as an operand, automatic type conversion and bit extension are also applied depending on the instruction.

For example, the register value 0x0000BC00 when referencing the lower 16 bits becomes:

- 0xFFFFBC00 for an Int instruction (MSB is sign-extended)
- 0xBF800000 (−1.0) for a Mad instruction

- When operand $R[n]$ types or word lengths differ, the following conversions are applied

automatically.

If this results in unintended conversions, bit manipulation may be required before or after the operation.

Source	Destination	Description
R[n]	Operands for Int and Mem instructions	R[n]
-R[n]		$\sim R[n] + 1$ (two's complement))
R[n]		R[n][31] ? $\sim R[n] + 1$: R[n]
#n		n[7] ? {0xfffff, n[7:0]} : n[7:0]
C(n)		C(n)
Rnd		Rnd (32-bit random number)
CCR		CCR, \sim CCR
LocalX,Y,Z,W		LocalX,Y,Z,W
iCID signal		<i>Logical channel number from PSS</i>
iAddr signal		<i>Context address from PSS</i>
iDelta signal		<i>Workload amount from PSS</i>
ScanX,Y		ScanX,Y
R[n][15:0]		R[n][15] ? {0xffff, R[n][15:0]} : R[n][15:0]
R[n][31:16]		R[n][31] ? {0xffff, R[n][31:16]} : R[n][31:16]
R[n][7:0]		R[n][7] ? {0xfffff, R[n][7:0]} : R[n][7:0]
R[n][15:8]		R[n][15] ? {0xfffff, R[n][15:8]} : R[n][15:8]
R[n][23:16]		R[n][23] ? {0xfffff, R[n][23:16]} : R[n][23:16]
R[n][31:24]		R[n][31] ? {0xfffff, R[n][31:24]} : R[n][31:24]
TR		TR
SrialCount		SerialCount[CID] (assigns internal counter)
TickCount		TickCount (assigns internal clock counter)
PC		PC
R[n]	Operands for Mad and Hyp instructions	R[n]
-R[n]		{ $\sim R[n][31]$, R[n][30:0]} (単精度符号反転)
R[n]		R[n][31] ? { $\sim R[n][31]$, R[n][30:0]} : R[n]
#n		n[7] ? \sim itof(n[6:0]) : itof(n[6:0])
C(n)		C(n)
Rnd		Rnd[22:0] (fractional part only of single-precision → range: 0 to 1.0, normalized before computation)

CCR		0
iCID signal		0
iAddr signal		0
iDelta signal		0
LocalX,Y,Z,W		itof(LocalX,Y,Z,W) (Unsigned)
ScanX,Y		itof(ScanX,Y) (Unsigned)
R[n][15:0]		ftof(R[n][15:0]) (Half precision → Single precision)
R[n][31:16]		ftof(R[n][31:16]) (Half precision → Single precision)
R[n][7:0]		R[n][7:0]<<15(Single-precision fractional part left-aligned → range: 0 to 1.0)
R[n][15:8]		R[n][15:8]<<15(Single-precision fractional part left-aligned → range: 0 to 1.0)
R[n][23:16]		R[n][23:16]<<15(Single-precision fractional part left-aligned → range: 0 to 1.0)
R[n][31:24]		R[n][31:24]<<15(Single-precision fractional part left-aligned → range: 0 to 1.0)
TR		0
SrialCount		0
TickCount		
PC		0
Results of Int and Mem instructions	R[n]	Result
	R[n][15:0]	Result[15:0] (Upper bits are masked)
	R[n][31:16]	Result[15:0] (lower bits are masked)
	SCR	Result
	TR	Result
	Int	Result(Assigned to interrupt vector)
	PC	Result (indicates that a Branch instruction is attached)
Results of Mad, Hyp	R[n]	Result
	R[n][15:0]	ftof(Result) (Upper bits are masked, single precision → half precision)
	R[n][31:16]	ftof(Result) (lower bits are masked, single precision → half precision)

	SCR	Result
	TR	Result (Not recommended designation)
	Int	Result (Not recommended designation)
	PC	Result (Not recommended designation)

3.13. Scalarization

By utilizing Command.Opt, it is possible to perform scalarization of each logical processor's registers. Scalarization requires the specification of one initialization instruction followed by a sequence of three scalarization instructions. The positions of the initialization and scalarization programs can be arbitrary, provided that their order is maintained.

Type	Opt	その他設定	備考
initialization	One	Example: $R[y] = 0$	Initialize only one logical processor (e.g., set to 0).
Scalarization	1/N	Set arbitrary operations using Int or Mad instructions (e.g., use add for summation). Configure both the read and write register numbers to be the same. In binary operations, apply an operand shift (+1) to one operand. Example: $R[x] = R[x] + R[x]@+1$	The contents of the relevant register will be temporarily overwritten. A limit can be set using the Shrink parameter.
	Unite	Write the result to a Scalar Register. One of the register read and write numbers uses the initialized register number No operand shift is used $y \neq x$ is the condition 例: $R[y] = R[y] + R[x]$	Result goes into initialized register Not necessary if the number of logical processors does not exceed N

	Unite	Write the result to a Scalar Register. Example: $S[z] = R[y]$	Specification of writing to the Scalar Register cannot be omitted.
--	-------	--	--

After scalarization, the instruction should generally be terminated. Subsequent processing should read the result stored in the Scalar Register using a different program.

Command.Edge must always be set to Default mode. Operand values accessed beyond boundaries will be the constant C2. For example, if obtaining a sum using the Mad instruction, set C2 to 0.0; for a product, set it to 1.0.[TEL – Thèses en ligne](#)

3.14. Branch Control

- Branches can be made to any Program Counter (PC) location. However, all logical processors branch synchronously; they cannot have differing PC values.
- Branching employs a delayed jump mechanism: the branch instruction and the subsequent instruction (the delay slot) are executed before the branch occurs. Do not place two branch instructions consecutively; if a branch instruction occupies the delay slot, it will be ignored.
- The jump destination is specified either by setting C3 at the time of issuing the branch instruction or by writing directly to the PC. Note that writing to the PC must be executed as a branch instruction to be effective. Specify the PC in the modify field of the write operand.
- When a branch instruction is present, each logical processor generates a single judgment result (Judge) based on its 4-bit Flag and the 16-bit C4[15:0]. C4[15:0] encompasses all combinations of the Flag. For example, if the Flag is '0101' in binary, the fifth bit of C4[15:0] is used as the result. All combination results can be described in C4[15:0]. This judgment result is also utilized in the update control described later.

Temporary = C4[15:0]

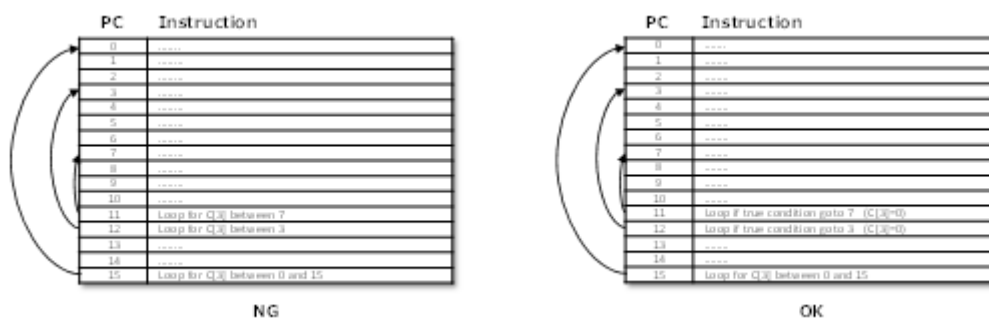
Judge = Temporary[Flag]

The judgment result of each logical processor is determined by the evaluation expression defined in Command's Cntl.Branch.

Branch[1:0]	Description
0	NOP
1	Always branching
2	Reserved
3	Reserved
4	Branching (AND) with all logic processor decision results true
5	Branch (OR) when one of the logic processor decision results is true
6	Branch (NAND) when one of the logic processor decision results is false
7	Branching (NOR) with all logic processor decision results false
8	Reserved
9	Reserved
10	Reserved
11	Reserved
12	Branching (AND) with all logic processor decision results true
13	Branch (OR) when one of the logic processor decision results is true
14	Branch (NAND) when one of the logic processor decision results is false
15	Branching (NOR) with all logic processor decision results false

If branching is based on a representative scalar value (i.e., the result of one logical processor's computation) or always branches, it's recommended to set `Command.Opt = 2 (One)` to reduce processing time.

When using the loop function, specify the maximum number of loops in `C3[31:16]`. If the branch condition is always true, it results in a simple loop count. Setting it to 0 creates an infinite loop; however, after 65,536 iterations, a limiter prevents further branching. Setting the control register `Cntl.LoopFree` to 1 disables this limiter. Loop count specification cannot be used in nesting unless all `C3[31:16]` within the nest are 0. Alternatively, create a local loop variable in the logical processor and nest based on that condition.



Even if the instruction in the delay slot is an end instruction (Instr:period), processing continues if branching occurs. Otherwise, the instruction in the delay slot is executed, and processing ends.

Set Util[3] to skip operations when the branch condition is false. This setting does not change the result but improves performance by skipping unnecessary logical processor blocks during slide execution on the hardware.

Cannot be used for processing involving more than 2LNR logical processors.

Subroutine Jump Example

```

n      Branch(1, n+100 → PC      // Jump, Opt=2 (One)
n+1    PC+1 → R[y]                // Delay Slot, save PC+1 to R[y], Opt=2
n+2    (main routine)             // Return from main routine

n+100  (subroutine)               // Subroutine start
n+101  (subroutine)
n+102  Branch(1, R[y] → PC        // Return by assigning saved address to PC, Opt=2
n+103  (subroutine)               // Delay Slot, subroutine end

```

3.15. Register Update Control

- The Set field in the Command determines whether to update registers and memory. By utilizing this field, it is possible to change only the CCR without updating the register, or vice versa.
- In addition to direct specification via the Command, control can also be achieved using the Judge flag generated from the Flag and C4. In this case, control is performed using the Deselect field in the Command.

Deselect Field	Description
[0]	Flag 制御
[1]	R[n], CC 制御

Deselect	Description
0	R[n] update always
1	R[n] update if Judge=0 (Judge = C4[Flag])

The two controls, Set and Deselect, can be combined. If conditions for both not updating and updating overlap, the result is not updating.

For Scalar Register access, it must be set to 0. Also, it cannot be used in scalarization processing.

3.16. Special Control

- Special operations are performed by setting the Cntl.Act field in the command.
- When Cntl.Act[0] = 0, this configures edge-handling for logical-processor numbers in operand shifts. It defines what kind of value is returned if the register number computed by the shift falls outside the valid range:

Act[2:1]	Description
0	If outside the range, return the default value C2.
1	If outside the range, return the value of the nearest effective register number.
2	If outside the range, return the register number linked to the other edge.
3	Reserved

When Cntl.Act[0] = 1, this defines various control actions:

Act[2:1]	Description
0	Reset the random-number seed based on the current register value. The random numbers generated by this instruction use the post-reset seed.
1	Reset the CCR to 0. The CCR value used by this instruction is the pre-reset value

2	Gather information beforehand to skip unnecessary logical-processor block processing, improving performance when a mask is later applied ($Act[2:1] = 3$). Not needed for other branch types. 3 (Ver. 2 or later)
3	Mask execution of any logical processor whose specified Flag bit is 1. The bit to test is chosen in advance via a register setting, not via a command-line parameter.

3.17. CC and Flags

- **CC and Flags** are packed into the Condition Code Register (CCR) and updated after each instruction completes.
- Each of the **Int**, **Mem**, **Mad**, and **Hyp** pipelines produces two results, **X** and **Y**, and each result generates a 4-bit condition code (**CC_X** and **CC_Y**), for a total of 8 bits. The FlagCC field in the Command selects either CC_X or CC_Y and stores it in the upper 4 bits of the CCR:

FlagCC	Description
0	CC _X
1	CC _Y

- The Flags occupy the lower 4 bits of the CCR. They are newly generated from the current 4-bit CC (from the pipeline), the existing 4-bit Flag, and the 16-bit constant C4[31:16]. The new update bit FNew is selected by indexing into C4[31:16] with the CC value

Temporary = C4[31:16]

FNew = Temporary[CC]

- The new flag bit FNew is then combined with the old Flag bits according to the FlagOp and FlagComb fields in the Command, producing the final 4-bit Flag that is written back into the CCR. FlagComb specifies how to merge FNew (Fi) and the old Flag (Fo):

FlagComb	Description
0	0
1	$\sim Fi \ \& \ \sim Fo$

2	$F_i \& \sim F_o$
3	$\sim F_o$
4	$\sim F_i \& F_o$
5	$\sim F_i$
6	$F_i \wedge F_o$
7	$\sim F_i \mid \sim F_o$
8	$F_i \& F_o$
9	$F_i = F_o$
10	F_i
11	$F_i \mid \sim F_o$
12	F_o
13	$\sim F_i \mid F_o$
14	$F_i \mid F_o$
15	1

FlagOp=1: $F_i = F_{\text{New}}$, $F_o = \text{FLAG}[0]$

FlagOp=3: $F_i = \text{FLAG}[2]$, $F_o = \text{FLAG}[3]$

- The CC bits themselves are not suppressed by the Set field, but they are affected by the Mod field of a write operand. If the Mod field is disabled, CC will not update (though Flags always will).
- To preserve the existing Flags (i.e., prevent any update), set:

FlagOp = 0
 FlagComb = 12
- Per-instruction update sequence:
 1. Generate new CC from the pipeline result.
 2. Generate FNew from CC, FlagOp, and FlagComb.
 3. Generate CCnew from old Flag, FlagRefer, and Deselect.
 4. Write CCnew → CC field; write FNew → Flag field.
- Specific bits of the Flag can be used to mask execution on individual logical processors. The bit to test is pre-configured via a register setting, and whether to apply the mask is controlled by Cntl.Act[0]=1 (Version 2 and later).
- By configuring Cntl.Deselect, you can control per-instruction whether CC and Flag are updated or preserved.
- By configuring Cntl.Act, you can clear the CC and Flag on a per-instruction basis.
- When the CCR is the target via a write's Mod field, the lower 8 bits of the computed result $R[y]$ are written into the CCR.

- If the Set field for R[y] is not enabled, the existing CCR value is retained.
- If Cntl.Seed = 1, the clear operation takes precedence, but updates or preservation governed by Deselect remain in effect.
- For flag-referencing operations, setting Cntl.IR = 1 allows the lower 4 bits of a specified register to be used as an immediate value. The particular register is chosen via the register-configuration settings.

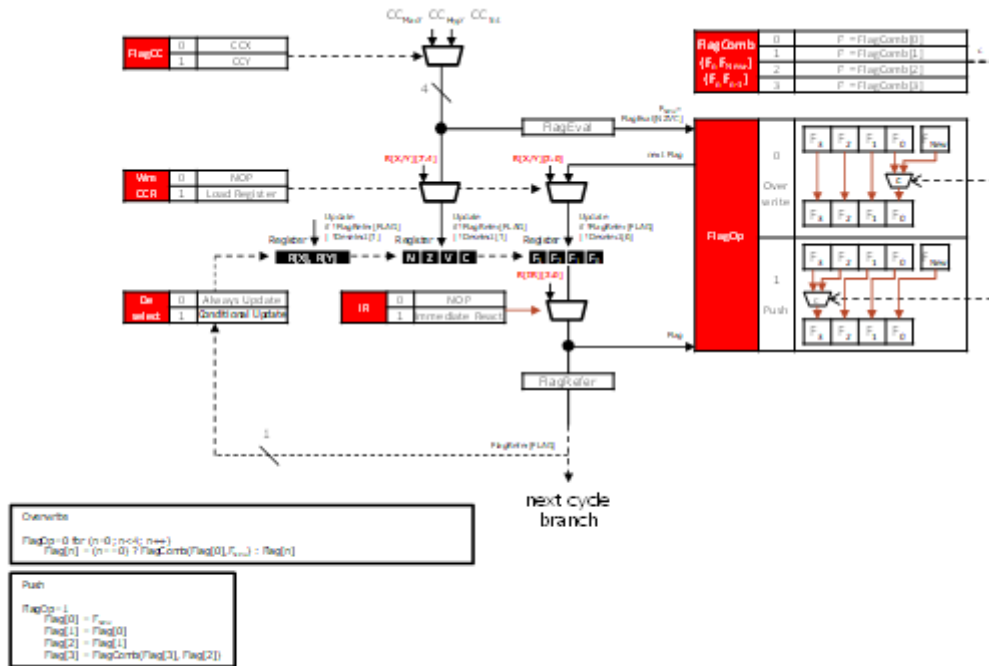


Figure 7 Combine the Flag (Ver.0-1)

Buf	C6[5]	Assigned to bit 1 of the base address
Bound	C6[7:6]	Access instruction outside the area set by WidthX,Y.
Swap	C6[15:8]	Byte-swap
Stride	C6[31:16]	Address increment in X direction
Ser	C7[3]	Serial-addressing enable
Base	C7[31:4]	Base address
Type	C7[1:0]	Conversion from floating-point to integer

- 2-dimensional coordinates are provided directly in R[n]. The register may hold either an integer or a floating-point value; the type of each half (Y in the high half, X in the low half) is explicitly indicated by the Type field. See the next section for coordinate-modification details.

$\text{mem}(A, B) = X \quad \approx \quad \text{mem}_{\text{Format}}[(A' + B' * (\text{Stride} + 1)) \ll \text{Format}' + \text{Base}] = \text{swap}(X)$
 $X = \text{mem}(A, B) \quad \approx \quad X = \text{swap}(\text{mem}_{\text{Format}}[(A' + B' * (\text{Stride} + 1)) \ll \text{Format}' + \text{Base}])$

↵

$A' = \text{Type}[0] ? \text{ftoi}(A) : A$

$B' = \text{Type}[0] ? \text{ftoi}(B) : B$

$\text{Format}' = (\text{Format} == 3) ? 2 : \text{Format}$

Type	Description
0	Treat as integer coordinates
1	Treat as floating-point coordinates. (Convert to integer type during address calculation)

- Stride is the increment applied to the address on each access, minus one. This stride is measured not in bytes but in units of the chosen word size (8-, 16-, or 32-bit).
- Edge Handling for out-of-range coordinates is selected by the Bound field (C[7:6]). For 2D accesses, the valid coordinate rectangle runs from (0, 0) up to (WidthX-1, WidthY-1). The values of WidthX and WidthY are specified as (width - 1) and (height - 1), respectively, in units of the chosen word size. Versions A and B do not support negative indices.



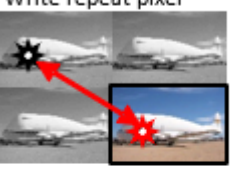

Bound	0	<p>Default Read 'default(C2)' value Write masking</p> 
	1	<p>Copy Read nearest pixel Write nearest pixel</p> 
	2	<p>Ring Read repeat pixel Write repeat pixel</p> 
	3	<p>Mirror Read mirror position Write mirror position</p> 

Figure 1 Bound Operation

- If both WidthX and WidthY are 0, processing is performed without judgment of crossing endpoints.
- The following are the parameters used in the method where the address is given directly (Instr.Instr[3]=1).

Parameter	Source	Description
Value	Value'0'	No bounds handling (always within range)
WidthX	Value'0'	No bounds handling
WidthY	Value'0'	No bounds handling
Format	Instr.Instr[1:0]	Memory format Select of 8bit, 16bit, 32bit

Extension	Instr.Wm[3:0] < 6 ? Instr.Wm[1:0] : Value'0'	Sign-extension of read data (extension applied when the field value equals 3).
Opt	Instr.Op[0]	Assigned to the 0th bit of the 0 base address
Buf	Instr.Op[1]	Assigned to the 0th bit of the 1 base address
Bound	Value'0'	No bounds handling
Swap	Value'0'	Fix 0
Stride	Value'0'	No bounds handling
Ser	Value'0'	Fix 0
Base	Value'0'	Fix 0
Type	Value'0'	Fix 0

- **Direct Addressing Mode:**

In this mode, the memory address is calculated as $R[n] + R[m]$. If the resulting address is not aligned to the word size, it is automatically aligned. The access unit (word size) varies depending on the instruction. Note that $R[n]$ must be of integer type.

$$\text{memformat}[A+B] = X \leftarrow$$

$$Y = \text{sign}_{\text{extend or not}}(\text{memformat}[A+B]) \leftarrow$$

-

This operation rearranges the byte lanes within a 32-bit word arbitrarily. Byte swapping is often used to handle differences in endianness between systems.

-

Memory is accessed in 64-bit units using Big Endian format. Each 64-bit unit automatically packs data from adjacent physical processors. For example, during an 8-bit access, data from eight physical processors are packed sequentially from the left.

-

Beyond simple addressing based on the position of logical processors, there exists a mechanism for serial addressing based on conditions. This applies only to write operations. Setting the Ser field to '1' enables serial addressing starting from 0, instead of using XY coordinates. In this mode, accesses masked by Deselect[1] are skipped, allowing only valid data to be written consecutively to memory.

-

When serial addressing is performed, a serial count value is recorded internally for each iCID. This serial count remains valid until the next serial addressing operation. Subsequent instructions can transfer this serial count value to another register for use.

3.19. External Bus (Aux) Access

- Access to the system bus is possible by setting `Command.Cntl:Option = 1 (Aux)`. Other parameters follow standard instruction formats.
- For write operations, assign the value to `R[x]` and the address to `R[y]`. The aux signal facilitates access to the external bus.
- For read operations, assign the address to `R[y]`. The aux signal facilitates access to the external bus, and the read data is written to the TR register.
- Similar to `Command.Cntl:Option = 2 (One)`, only logical processor 0 executes the operation.
- Be aware that operations like resetting the processor via the system bus may lead to undefined behavior and are not guaranteed to function correctly.

3.20. Flow Control

- The apparent latency can be calculated using the formula below. If the number of operations (`VolumeNum`) is sufficiently large, the apparent latency will be less than or equal to 1. For example, if the actual number of pipeline stages (`PipeStageNum`) is 32, the degree of parallelism (`ParallelNum`) is 4, the number of operations (`VolumeNum`) is 256, and the flow control number (`FlowControlNum`) is 1, then the latency will be 1/2.

$$\text{Latency} = \frac{\text{PipeStageNum} \times \text{ParallelNum}}{\text{VolumeNum} \times \text{FlowControlNum}}$$

- If the apparent latency is less than or equal to 1, no performance penalty occurs, and throughput of 1 can be achieved. It is assumed that the number of operations (`VolumeNum`) is sufficiently large when using `Kp`. If this number is small, a penalty will occur, and the expected performance will not be achieved. By default, `R[n]` can be used continuously as shown below:

命令 0 `xxx→R[y]`
命令 1 `R[y]→xxx`

- The following methods can be used to suppress penalties by adjusting the flow control number (`FlowControlNum`):
 - Set the `Lat` field in the Control register to 3 (release up to 3).
 - Set the `Lat` field in `Command.Cntl` to `FlowControlNum - 1` (static setting).
 - Use `R[n]` considering the latency (do not use the register `R[n]` between the writing

instruction and the current instruction based on the Lat setting).

- If this usage is incorrect, the result is not guaranteed.
- For example, when FlowControlNum = 3, a register written in instruction n can be referenced from instruction n + 3 onwards. Unlike in typical processors, registers cannot be used at the exact timing corresponding to the latency, but only after the latency has passed. In other words, it is not guaranteed when the register will be updated before the latency period ends after instruction issuance. In the example below, R[y] written by instruction 0 can be used from instruction 3 onward, but it might be overwritten by instruction 1. Therefore, R[y] cannot be used in instructions 1–2.

```
命令 0   xxx→R[y];  
命令 1   xxx→R[y];  
命令 2   ---←  
命令 3   R[y]→xxx;  
命令 4   R[y]→xxx;
```

4. Instruction Description

4.1. Overview

- A single instruction, referred to as a **Command**, consists of 16 bytes in total when it is a normal instruction. This includes a control instruction (**Cntl**), an operation instruction, and a constant (**C0**). In the case of a set instruction, it consists of **Cn** (n = 1–7). The meaning of each field varies depending on the **Ver** register setting. The default is **Ver2**.

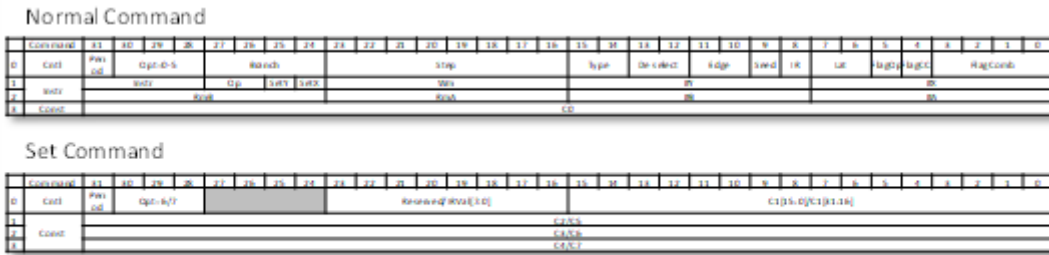


Figure 9 Command Field (Ver1)

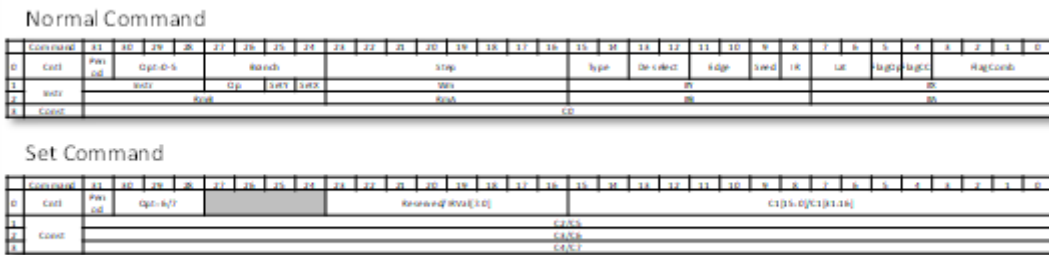


Figure 9 Command Field (Ver2)

- When a single instruction is executed, the Program Counter (PC) advances by 16. The PC starts from 0.
- The same instruction is issued to all logical processors. The only difference lies in the operand constants, which are indexed to indicate the position of each logical processor.
- Operand specification is done through the A, B, X, and Y fields, which point to Rb[n]. The lower 4 bits represent the register number n, while the upper bits represent the bank number b. The number of banks supported depends on the implementation and can be 1, 2, 4, 8, or 16. Use bank numbers within this supported range. Please refer to the FPGA/LSI implementation parameters for the maximum number of banks.
- Banks serve as double buffers used for data input/output from sources such as DMA, operating in the background. In such cases, programs are written to use only a single bank. If the program has exclusive access to the bank, the bank number b can be freely specified just like the register number n.
- Operand inputs can also be modified using the Mod field. To avoid unnecessary circuit activity, be aware that setting the Mod field to '0' means the operand will not be processed.
- The Mod field for read operations (Rm) is specified separately for each operand being read.

Rm[7:6]	Description
0	NOP (interpreted when 0 is read)
1	Normal
2	Logical processor number offset (-32 to 31, valid for registers A

	and B only)
3	Logical processor number offset (in multiples of 64)

Rm[5:4]	Description
0-3	Reserved

Rm[3:0]	Description
0	R[n]
1	-R[n] (Negative number)
2	R[n] (Absolute value)
3	Operand constant (register number converted to two's complement)
4	C0-C7 (Constant number is specified by the register number)
5	Parameter (parameter number is specified by the register number)
6	Extends the 16-bit value from R[n][15:0]; for Int/Mem instructions, sign extension is applied; for Mad/Hyp instructions, extension is from half-precision to single-precision.
7	Extends the 16-bit value from R[n][31:16]; for Int/Mem instructions, sign extension is applied; for Mad/Hyp instructions, extension is from half-precision to single-precision.
8	Extends the 8-bit value from R[n][7:0]; for Int/Mem instructions, sign extension is applied; for Mad/Hyp instructions, extension is from half-precision to single-precision.
9	Extends the 8-bit value from R[n]15:8; for Int/Mem instructions, sign extension is applied; for Mad/Hyp instructions, extension is from half-precision to single-precision.
10	Extends the 8-bit value from R[n][23:16]; for Int/Mem instructions, sign extension is applied; for Mad/Hyp instructions, extension is from half-precision to single-precision.
11	Extends the 8-bit value from R[n][31:24]; for Int/Mem instructions, sign extension is applied; for Mad/Hyp instructions, extension is from half-precision to single-precision.
12	TR (per CID)
13	SerialCount (per CID)

14	TickCount
15	PC

Rm[3:0] is invalid unless Rm[7:6] = '1'.

The specified number is automatically modulo-adjusted to stay within the valid range.

For Rm[3:0] = 8–11, half-precision representation uses a custom format where the sign is 0, exponent is 0, and 0xFF represents 255/256.

- The parameters for Rm[3:0] = 5 are as follows.

レジスタ#	Description
0	Random number For Int/Mem instructions: 32-bit For Mad/Hyp instructions: 24-bit (fixed-point representation with Exp = 0)
1	Reserved
2	CCR(Lower 8 bits are valid)
3	Reserved
4	Index X (from PSS), equal to the logical processor number
5	Index Y(from PSS)
6	Index Z(from PSS)
7	Index W(from PSS)
8	Reserved
9	
10	
11	
12	Coordinate X obtained by scan-converting index XYZW (from PSS)
13	Coordinate Y obtained by scan-converting index XYZW (from PSS)
14	Reserved
15	

- The Mod field for write operations (Wm) is set commonly for all write operands within the instruction.

Wm[7:6]	Description
0	NOP (no write is performed)
1	Normal
2	Reserved
3	Logical processor number offset (multiple of 64)

Wm[3:0]	Description
0	R[n] In memory access instructions where the address is directly specified, Wm[1:0] determines whether sign extension is applied during read operations (sign extension is applied if the value is 3).
1	
2	
3	
4	
5	
6	R[n] (Lower 16 bits)
7	R[n] (Upper 16 bits)
8	SCR(0–255 is specified by the logical OR of register numbers X and Y)
9	Reserved
10	SCR(Specified by logical processor coordinate CX plus register number X)
11	SCR(Specified by logical processor coordinate CX plus register number Y)
12	TR Since the output of an arbitrary logical processor is selected, it is desirable that the output value be the same regardless of the logical processor number. (No issue when using serial count selection)
13	CCR (the lower 8 bits of Y are written); Cntl.Seed takes precedence (if set to 1, it clears the register); control via Cntl.Deselect is also valid.
14	Interrupt Generation In addition to the ilrq signal, an arbitrary interrupt vector can be asserted to the system (how it is used depends on the external system).

15	PC (can only be assigned in conjunction with a Branch instruction)
----	--

Wm[3:0] is invalid unless Wm[7:6] = '1'.

- Operand X functions as either a read or write operand depending on the instruction. Since operand X shares the Mod field (Wm) with operand Y, conflicting settings are not guaranteed to operate correctly.

4.2. Control Instructions (Cntl)

- Control instructions consist of termination, branching, and flag operations.

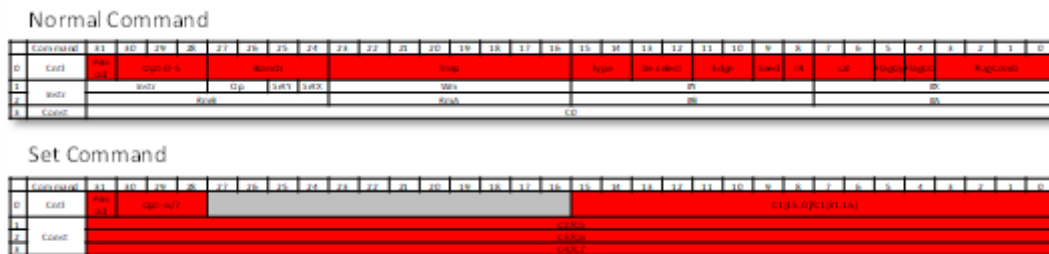


Figure 10 Cntl Instruction

- The end of the instruction is indicated by the Period field.

Period	Description
0	Execute the instruction and continue
1	Execute the instruction and terminate

- The Opt field is used to select the system configuration.

Opt	Description
0	Normal order
1	External Bus Access Instruction
2	Executes only the first instruction regardless of the number of operations (e.g., for SCR initialization)

3	Executes only the first instruction for each logical processor Used when the processing volume exceeds the number of logical processors.
4	Performs computation using adjacent logical processors, reducing the number of operations by half. Any remainder is added to the total number of operations.
5	Performs scalarization processing Calculations are carried out in a tournament-style manner using adjacent logical processors, ultimately producing one result per logical processor group.
6	Set Instruction 0
7	Set Instruction 1

- **Branch defines the branching behavior. The branch is taken based on the condition set by the combination of C4 Flags and Branch[2:0].**

A delayed jump mechanism is used, meaning the instruction immediately following the branch instruction is executed before the actual branch occurs.

Branch	Description
0	NOP
1	Branch (constantly); if not writing to PC, refer to C3
2	Reserved
3	
4	Branch (AND condition); if not writing to PC, refer to C3.
5	Branch (OR condition); if not writing to PC, refer to C3."
6	Branch (NAND condition); if not writing to PC, refer to C3.
7	Branch (NOR condition); if not writing to PC, refer to C3.
8	Reserved
9	
10	
11	
12	Branch (AND condition); C3[15:0] is the jump destination, and C3[31:16] is the loop count
13	Branch (OR condition); C3[15:0] is the jump destination, and C3[31:16] is the loop count

14	Branch (NAND condition); C3[15:0] is the jump destination, and C3[31:16] is the loop count
15	Branch (NOR condition); C3[15:0] is the jump destination, and C3[31:16] is the loop count

- Step sets the offset value for the logical processor number. The offset value is calculated as $\text{Step}[5:0] \times 64 \times \text{Index}$. The Index is selected from Step[7:6]. If the total offset value exceeds the actual entity size, the modulo of that value is used as the effective offset.

Step[7:6]	Description
0	1
1	Index Y (iIndex[31:16]) input from the PSS
2	Index Z (iIndex[47:32]) input from the PSS"
3	Index W (iIndex[63:48]) input from the PSS"

- Select the operation via the Type field.

Opt	Description
0	Int instruction"
1	Mem instruction"
2	Mad instruction"
3	Hyp instruction"

When Op is 3 in the Int instruction, it is equivalent to the mem instruction.

- Deselect controls updates to R[n] and memory. For memory writes performed by the Int instruction, it functions as a mask control.

Deselect[0]	Description
0	Flag update always
1	Flag update if C4[Flag]=0

Deselect[1]	Description
0	R[n] and CC update always
1	R[n] and CC update update if C4[Flag]=0

- The Edge field specifies how to handle R[n] values that exceed the valid range.

Edge	Description
0	Default mode : Values outside the valid range return C2
1	Copy mode : Returns the R[n] value at the nearest endpoint.
2	Ring mode : Returns the R[n] at the modulo position of Width + 1.
3	Reserved

- When the Seed field is set to '1', random number generation and CCR initialization are performed. The random seeds specified in the control registers Seed0 to Seed3 are used. Note that only the random number generator is initialized after a reset.
- When the IR (Immediate React) field is set to '1', operations related to Flag (Deselect and Branch) can use the lower 4 bits of the computation result R[#IRVal] as an immediate value. #IRVal is specified either via the Set instruction 1 or through register settings. If the IRValEn in the Cntl register is '1', the value in the IRVal register is used; if IRValEn is '0', the value set by the Set instruction is used.
- The Lat field specifies the tolerance for NOP insertion in coherence control. It applies to instructions from the current one back by Lat steps. The default value '0' means that instructions before the immediately preceding one are subject to the check.
- FlagOp, FlagCC, and FlagComb instruct the generation of a new Flag. F_{New} is newly generated and integrated into a 4-bit Flag. To retain the existing Flag, set FlagOp = 0 and FlagComb = 12.

FlagOp	new Flag[3]	new Flag[2]	new Flag[1]	new Flag[0]
0	-	-	-	FlagComb(Flag[0], F _{New})
1	FlagComb(Flag[3], Flag[2])	Flag[1]	Flag[0]	F _{New}

- F_{New} is generated based on the following set of expressions.

$F_{New} = \text{Temporary}[CC], \text{Temporary} = C4[31:16]$

FlagComb	Description
0	0

1	$\sim F_i \& \sim F_o$
2	$F_i \& \sim F_o$
3	$\sim F_o$
4	$\sim F_i \& F_o$
5	$\sim F_i$
6	$F_i \wedge F_o$
7	$\sim F_i \mid \sim F_o$
8	$F_i \& F_o$
9	$F_i = F_o$
10	F_i
11	$F_i \mid \sim F_o$
12	F_o
13	$\sim F_i \mid F_o$
14	$F_i \mid F_o$
15	1

- In the Set instruction, only the fields for setting the Period and the Constant Register are valid.
- C1 to C7, in addition to being used as regular constants, are also defined as parameters for special purposes.

1	Region	Region	Area	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City	City
---	--------	--------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

4		X
5		Y
6		Z
7		W

C1* [3]	C0* [3]	X'	Y'
0	0	$U0 \% 2^{16-\text{MaskX}} + U1 * 2^{\text{BoxX}}$	$V0 \% 2^{16-\text{MaskY}} + V1 * 2^{\text{BoxY}}$
0	1	$U0 / 2^{\text{MaskX}} + U1 * 2^{\text{BoxX}}$	$V0 / 2^{\text{MaskY}} + V1 * 2^{\text{BoxY}}$
1	0	$U0 \% 2^{16-\text{MaskX}} + U1 / 2^{\text{BoxX}}$	$V0 \% 2^{16-\text{MaskY}} + V1 / 2^{\text{BoxY}}$
1	1	$U0 \% 2^{\text{BoxX}} + U1 / 2^{\text{BoxY}} * 2^{\text{BoxX}}$	$V0 / 2^{\text{BoxX}} \% 2^{16-\text{MaskX}} + V1 / 2^{\text{BoxY}} \% 2^{16-\text{MaskY}} * 2^{\text{MaskX}}$

- **C2: Default Value**

Sets the value to be assigned when an operand shift or out-of-range memory access occurs. Additionally, C2[30:23] is used to configure the following parameters for the Mad instruction: right or left shift of the exponent, offset to the exponent, and rounding.

- **C3: Branch Parameters**

The lower 16 bits specify the branch destination PC (absolute value), and the upper 16 bits specify the loop count.

- **C4: Flag Selection**

The lower 16 bits define the selection value used by control operations referencing Flags (Deselect and Branch), while the upper 16 bits define the selection value used by CC for Flag updates.

Control = FlagRefer[Flag]

FNew = FlagEval[CC]"

- **C5, C6, C7: Memory Access Parameters (see section 3.11)**

4.3. Integer Arithmetic Instructions (Int)

- The integer arithmetic instruction (Cntl:Type = 0) performs integer operations. It uses two operands, A and B. The register selection value X is used as a configuration parameter.

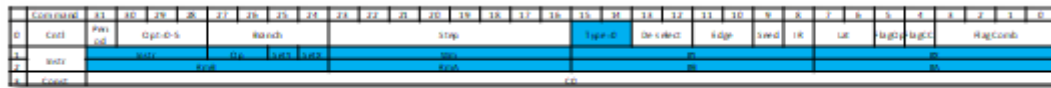


Figure 11 Int Instruction

Arithmetic operations (Op = 0):

Intr (Op=0)	Y	X	Y Flag				X Flag			
			N	Z	V	C	N	Z	V	C
0	add	A+B	Y[31]	Y=0	new Carry ! = Y[31]	new Carry	X[31]	X=0	0	A=-1
1	addc	A+B+Carry								
2	sub	B-A								
3	addc	B-A-Carry								
4	padd	B								
5	paddc	B+Carry								
6	psub	B								
7	psubc	B-Carry								
8	mul	A*B	Y[31]	Y=0	0	0	X[31]	X=0	0	0
9	mul*	A*B								
10	abs	B								
11	mov	B								
12	div	B/A								
13	divu*	B%A								
14	ediv	(B<<32)/A								
15	edivu*	(B<<32)%A								

Bit/Byte operations (Op = 1):

Intr (Op=1)		Y	X	Y Flag				X Flag			
				N	Z	V	C	N	Z	V	C
10	tern	CCR[#X] ? B : A	-	Y[31]	Y=0	0	0	-	-	-	-
11	lut	SR[B]									
***	rot0	rot _A (B)									
	rot1	shift _A (B) w/carry					new carry				
	rot2	shift _A (B) w/zero*									
	rot3	shift _A (B) w/LSB,MSB **									
13	ham	Σ B[i]					0				

		(i=0-31)									
	bool0	0									
	bool1	$\sim A[i] \& \sim B[i]$									
	bool2	$A[i] \& \sim B[i]$									
	bool3	$\sim B[i]$									
	bool4	$\sim A[i] \& B[i]$									
	bool5	$\sim A[i]$									
14	bool6	$A[i] \wedge B[i]$									
***	bool7	$\sim A[i] \mid \sim B[i]$									
*	bool8	$A[i] \& B[i]$									
	bool9	$A[i] \sim \wedge B[i]$									
	bool10	$A[i]$									
	bool11	$A[i] \mid \sim B[i]$									
	bool12	$B[i]$									
	bool13	$\sim A[i] \mid B[i]$									
	bool14	$A[i] \mid B[i]$									
	bool15	1									

*: zero stuff

** : if ($A \geq 0$) B[0] else B[31] stuff

***: rot number is assigned by #X[1:0]

****: bool number is assigned by #X[3:0]

Transfer operations (Op = 2):

Intr (Op=2)	Y	X	Y Flag				X Flag			
			N	Z	V	C	N	Z	V	C
0	hf2f	half float to float(B)								
1	f2hf	float to half float(B)								
2	bf2f	8bit float to float(B)	A	Y[31]	Y=0	0	0	X[31]	X=0	0
3	f2bf	float to 8bit float(B)								
4	si2f	signed int to float(B)								

perform memory access. There are two access modes: 2D access mode and direct access mode. In the latter, most parameters can be specified through Command.Instr.

- **2D Access Mode**

- **Write2D** accesses the address using operands A and B as coordinates and writes the value of register X. If Deselect is set, masking is applied based on conditions.

Intr (Op=Don't care)		mem(A,B)	Y Flag				X Flag			
			N	Z	V	C	N	Z	V	C
0	write2D (A,B)	swap(X & 0xff)@Byte								
		swap(X & 0xffff)@Half	-	-	-	-	-	-	-	-
		swap(X)@Word								

- **Read2D** access uses operands A and B as coordinates to access the address and writes the data to register Y.

Intr (Op=3)		Y	X	Y Flag				X Flag			
				N	Z	V	C	N	Z	V	C
4	read2D (A,B)	swap(mem[B,A] & 0xff)	0	Y[31]	Y=0	0	0	0	1	0	0
		swap(mem[B,A] & 0xffff)									
		swap(mem[B,A])									

Value	Swap[7:6]	Swap[5:4]	Swap[3:2]	Swap[1:0]
	swap0[31:24]	swap0[23:16]	swap0[15:8]	swap0[7:0]
0	mem[31:24]	mem[23:16]	mem[15:8]	mem[7:0]
1	mem[23:16]	mem[15:8]	mem[7:0]	mem[31:24]
2	mem[15:8]	mem[7:0]	mem[31:24]	mem[23:16]
3	mem[7:0]	mem[31:24]	mem[23:16]	mem[15:8]

- Figure 13 shows the data type conversion table for Read2D and Write2D. "Destination" refers to the transfer target, and "Source" refers to the transfer origin.

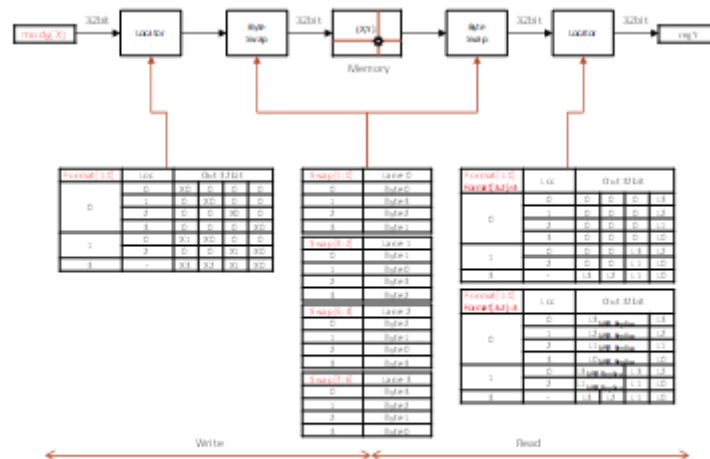


Figure 13 Read Data Transfer

- By setting Ser (C7[3]) to '1', serial addressing is enabled. This is valid only for Write operations, and its behavior is not guaranteed for Read. Operands related to coordinates are ignored, and if Deselect[1] & ~FlagRefer[Flag] is true, the Write is skipped and the address is incremented.
- The Type field (C7[1:0]) specifies the data type of the coordinates used for address calculation. Type[0] corresponds to the X coordinate, and Type[1] to the Y coordinate. Set '0' for integer type and '1' for floating-point type.
- When performing a Read, setting the Format[3:2] field to '3' enables the following sign extension for integer types.

Format[1:0]	Description
0 8bit Read	The MSB of the 8-bit data retrieved from memory is copied to extend it to 32 bits
1 16bit Read	The MSB of the 16-bit data retrieved from memory is copied to extend it to 32 bits.
2 24bit Read	Reserved
3 32bit Read	The 32-bit data retrieved from memory is used as-is.

- During memory access, the Base field specifies the memory address at coordinate (0,0), and the Stride field specifies the increment amount when the Y coordinate is updated (in units specified by Format[1:0]) minus 1.

- **Direct Access Mode**

- **WriteN** writes the value of register X to the memory address obtained by adding operands A and B. If Deselect is set, masking is applied based on conditions. N represents the data unit size.

Intr		mem[A+B+Base]	Y Flag				X Flag			
			N	Z	V	C	N	Z	V	C
8	write8 (A,B)	swap(X & 0xff)@Byte	-	-	-	-	-	-	-	-
9	write16 (A,B)	swap(X & 0xffff)@Half	-	-	-	-	-	-	-	-
10										
11	write (A,B)	swap(X)@Word	-	-	-	-	-	-	-	-

Value	Swap[7:6]	Swap[5:4]	Swap[3:2]	Swap[1:0]
	swap0[31:24]	swap0[23:16]	swap0[15:8]	swap0[7:0]
0	X[31:24]	X[23:16]	X[15:8]	X[7:0]
1	X[7:0]	X[31:24]	X[23:16]	X[15:8]
2	X[15:8]	X[7:0]	X[31:24]	X[23:16]
3	X[23:16]	X[15:8]	X[7:0]	X[31:24]

- **ReadN** writes the data from the memory address obtained by adding operands A and B into register Y. N represents the data unit size. When Command.Op = 3, sign extension is performed according to the word length.

Intr (Op=3)		Y	X	Y Flag				X Flag			
				N	Z	V	C	N	Z	V	C
12	read8 (A,B)	swap(signExtend8 (mem[B,A] & 0xff))	0	Y[31]	Y=0	0	0	0	1	0	0
13	read16 (A,B)	swap(signExtend16 (mem[B,A] & 0xffff))									

14											
15	read (A,B)	swap(mem[B,A])									

Value	Swap[7:6]	Swap[5:4]	Swap[3:2]	Swap[1:0]
	swap0[31:24]	swap0[23:16]	swap0[15:8]	swap0[7:0]
0	X[31:24]	X[23:16]	X[15:8]	X[7:0]
1	X[7:0]	X[31:24]	X[23:16]	X[15:8]
2	X[15:8]	X[7:0]	X[31:24]	X[23:16]
3	X[23:16]	X[15:8]	X[7:0]	X[31:24]

- The Buf and Opt fields are passed to the memory subsystem. The behavior depends on the memory subsystem specifications, but for example, a memory subsystem with the following functionality is available.

Buf	Opt	Description
0	0	Cache access
0	1	Cache access However, it transitions to the Valid state but not to the Modified state (on write), and no write-back to external memory occurs
1	0	Buffer access (bypassing the cache))
1	1	Reserved

4.5. Floating-Point Instruction (Mad)

- The floating-point instruction performs floating-point arithmetic operations. It uses three operands: A, B, and X.

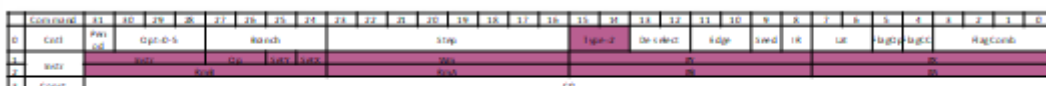


Figure 14 Mad Instruction

Intr		Y	X	Y Flag				X Flag			
				N	Z	V	C	N	Z	V	C
0	mul	A*B	0	Y[31]	Y=0	Y=NaN	Y[22:0]	Y[31]	1	0	0

1	add	A+B				Y=Inf	≠ 0			
2	min	min(A,B)				*	**	A<B		
3	max	max(A,B)						A≥B		
4	mad***	A*B+X						Y[31]		
5	Reserved									
6	Reserved									
7	diff	(A<B) ? A : A-B						A<B		

*: NaN/Inf is exp[7:0]=0xff

**: frac[22:0] is not zero (NaN C=1, Inf C=0)

***: Result X is not affected by round/magnification operation

- The result of the operation can be subject to rounding, exponent shifting, and addition processing. For the mad operation, these effects apply only to register Y and do not affect register X.
- If Intr[3] = 1, rounding is performed according to the table below, using the decimal point position indicated by Exp (C2[31:24]) as the reference.

Op	Description
0	Trunc: mode Discards the fractional part below the specified decimal point position (e.g., 1.5 becomes 1).
1	Round mode: Performs rounding to the nearest even number based on the specified decimal point position. In rounding to even, values exactly at 0.5 are rounded to the nearest even number (e.g., 2.5 → 2, 3.5 → 4). (For example, 0.5 becomes 0, and 1.5 becomes 2.)
2	Floor mode: Discards toward the negative direction based on the specified decimal point position. (For example, 1.5 becomes 1, and -1.5 becomes -2)
3	Ceil mode: Rounds up toward the positive direction based on the specified decimal point position

	(e.g., 1.5 becomes 2, -1.5 becomes -1)
--	--

- When Intr[3] = 0 and Op = 1, if the multiplicand is 1.0, one of the input values is directly output; likewise, if the addend is 0.0, one of the input values is directly output.
- When Intr[3] = 0 and Op = 2, the exponent part is shifted to perform exponentiation by n. Here, n = Exp (C2[31:24]), with Exp represented in two's complement form.
- When Intr[3] = 0 and Op = 3, the exponent part is modified through addition to perform multiplication by 2ⁿ. Here, n = Exp (C2[31:24]) – 127. If EXP = 0, then n = 0.
- The result of the operation may produce NaN or Infinity. Since these values propagate, care must be taken. They can be distinguished using the Overflow flag and Carry flag.

4.6. Hyperfunction Instruction (Hyp)

- The hyperfunction instruction performs special floating-point arithmetic operations. It uses two operands: A and B.

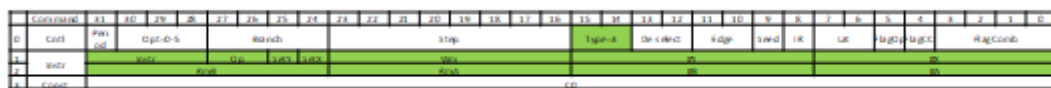


Figure 15 Hyp Instruction

Intr	Y	X	Y Flag				X Flag			
			N	Z	V	C	N	Z	V	C
0	tri	A*sin(B)	Y[31]	Y=0	Y=NaN Y=Inf *	Y[22:0] ≠ 0 **	X[31]	X=0	X=NaN X=Inf *	X[22:0] ≠ 0 **
1	hyp	A*sinh(B)								
2	mul	A*B								
3	thru	B								
4	atan	tan ⁻¹ (B/A)								
5	atanh***	tanh ⁻¹ (B/A)								
6	div	B/A								
7	thru	B								

*: NaN/Inf is exp[7:0]=0xff

**: frac[22:0] is not zero (NaN C=1, Inf C=0)

***: if A is negative X is negative

- Unlike the Mad instruction, rounding, exponent shifting, or addition processing cannot be applied to the result of the operation.
- The result of the operation may produce NaN or Infinity. Since these values propagate, care must be taken. They can be distinguished using the Overflow flag and Carry flag.
- For trigonometric functions, the unit of angle is not in radians but is normalized such that 2π equals 1.0. Since the CORDIC algorithm is used, errors near the least significant bits may occur depending on the operand values.

5. Control Register Description

5.1. Overview

- Control registers are accessed via the control bus. Unlike R[n], they are common settings shared across processors. They include scalar registers.
- In the detailed register descriptions, the following symbols are used to indicate access types:
 - R – Read Only (writes have no effect)
 - R/W – Read / Write
 - R/WC – Read / Write, Clear on Write
- Do not access reserved registers. Also, when writing to reserved fields, set them to '0'.
- Any 'x' in address or data fields indicates a “don't care” value.

5.2. Definition

Address	Register Name	Description
0000_0000	Reset	Reset Control
0000_0004	Clock	Clock Control
0000_0018	Info	Interrupt information
0000_001c	IntEn	Interrupt enable
0000_0010	Cntl	Master control
0000_0014	IRVal	IR value
0000_0018	–	Reserved
0000_001c	–	Reserved
0000_0020	Seed0	Random Seed 0
0000_0024	Seed1	Random Seed 1
0000_0028	Seed2	Random Seed 2
0000_002c	Seed3	Random Seed 3
0000_0030	–	Reserved
0000_0034	–	Reserved
0000_0038	–	Reserved
0000_003c	–	Reserved
0000_0040	MonitorXY	Active Index XY

0000_0044	MonitorZW	Active Index ZW
0000_0048	MonitorPC	Active Index PC
0000_004c	–	Reserved
0000_0050	–	Reserved
0000_0054	–	Reserved
0000_0058	–	Reserved
0000_005c	–	Reserved
0000_0060	BreakXY	Break Index XY
0000_0064	BreakZW	Break Index ZW
0000_0068	BreakPC	Break Target PC
0000_006c	–	Reserved

5.3. Details

5.3.1.1. Reset Register

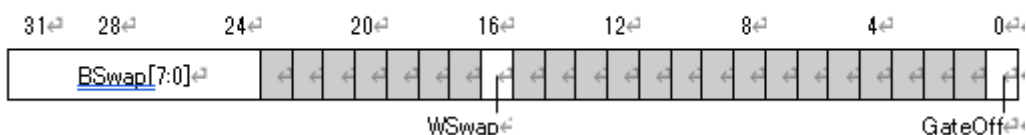
[Address: 0x0000_0000]



Name	Type	Default	Description
Reset	R/W	0	Synchronous Reset After setting to '1', the system enters an internal reset state and then automatically clears it. Unlike the reset_n signal, the contents of other registers are retained.

5.3.1.2. Clock Register

[Address: 0x0000_0004]



Name	Type	Default	Description
BSwap	R/W	0	Configures byte swapping for input/output data in memory instructions.

For input, byte-level mapping is performed from input data In[31:0] to internal data Pipe[31:0].

Be careful, as configurations other than one-to-one mapping may result in unknown values or overlapping.

Value	BSwap[7:6]	BSwap[5:4]	BSwap[3:2]	BSwap[1:0]
	Pipe[31:24]	Pipe[23:16]	Pipe[15:8]	Pipe[7:0]
0	In[31:24]	In[23:16]	In[15:8]	In[7:0]
1	In[7:0]	In[31:24]	In[23:16]	In[15:8]
2	In[15:8]	In[7:0]	In[31:24]	In[23:16]
3	In[23:16]	In[15:8]	In[7:0]	In[31:24]

For output, byte-level mapping is performed from internal data Pipe[31:0] to output data Out[31:0].

Configurations other than one-to-one mapping may result in unknown values or overlapping.

Value	BSwap[7:6]	BSwap[5:4]	BSwap[3:2]	BSwap[1:0]
	Out[31:24]	Out[23:16]	Out[15:8]	Out[7:0]
0	Pipe[31:24]	Pipe[23:16]	Pipe[15:8]	Pipe[7:0]
1	Pipe[23:16]	Pipe[15:8]	Pipe[7:0]	Pipe[31:24]
2	Pipe[15:8]	Pipe[7:0]	Pipe[31:24]	Pipe[23:16]
3	Pipe[7:0]	Pipe[31:24]	Pipe[23:16]	Pipe[15:8]

WSwap

R/W

0

Configures final word swapping with memory.

The specification is the same as the swap used in instructions (see the 2D access method of memory instructions).

For Write operations, this swap is applied *after* the swap specified in the instruction.

For Read operations, this swap is applied *before* the swap specified in the instruction.

GateOff

R/W

0

Gated Clock Off Mode.

When set to '1', all bits of the gate signal are fixed to '1'.

5.3.1.3. Info Register

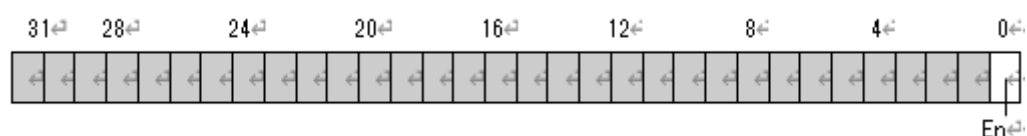
[Address: 0x0000_0008]



Name	Type	Default	Description
Int	R/WC	0	Indicates that an interrupt has occurred. Cleared by writing '1'.

5.3.1.4. IntEn Register

[Address: 0x0000_000c]



Name	Type	Default	Description
En	R/W	0	Configures permission for interrupts to the system.

5.3.1.5. Cntl Register

[Address: 0x0000_0010]



Name	Type	Default	Description
IRValEn	R/W	0	When set to '1', the value of the IRVal register is used as the immediate value for register specification.
LoopFree	R/W	0	When set to '1', the maximum number of loop iterations is unlimited; when set to '0', the maximum number of loop iterations is limited to 65,536.
Lat	R/W	0	Sets the maximum latency minus one. Values less than or equal to the configured value will

not be blocked.

5.3.1.6. IRVal Register

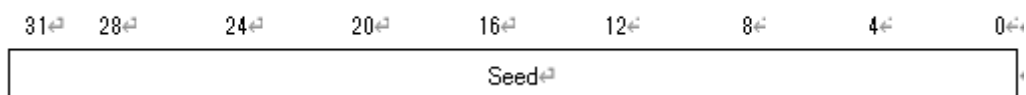
[Address: 0x0000_0014]



Name	Type	Default	Description
IRVal	R/W	0	Used for immediate value support. If Cntl register's IRValEn is set to '1', this takes precedence over the IRVal specified in the Set instruction.

5.3.1.7. SeedX Register

[Address: 0x0000_0020 - 0x0000_002c]

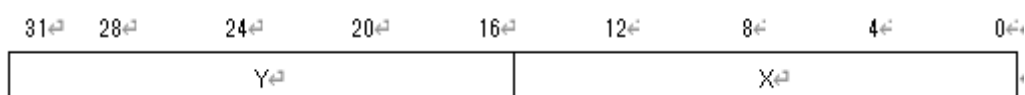


Name	Type	Default	Description
Seed	R/W	Table Reference	Sets the seed for random number generation. Four values together constitute a single random seed.

Address	Register Name	Default (Decimal)
0000_0020	Seed0	123456789
0000_0024	Seed1	362436069
0000_0028	Seed2	521288629
0000_002c	Seed3	88675123

5.3.1.8. MonitorXY Register

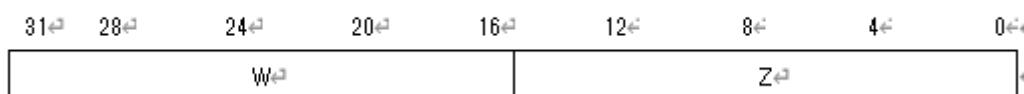
[Address: 0x0000_0040]



Name	Type	Default	Description
X	R	0	Indicates the currently active index X.
Y	R	0	Indicates the currently active index Y

5.3.1.9. MonitorZW Register

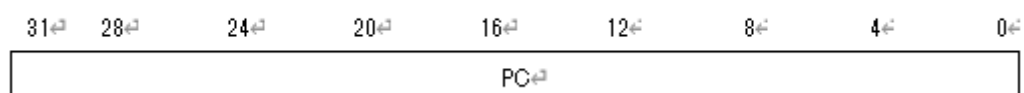
[Address: 0x0000_0044]



Name	Type	Default	Description
Z	R	0	Indicates the currently active index Z
W	R	0	Indicates the currently active index W

5.3.1.10. MonitorPC Register

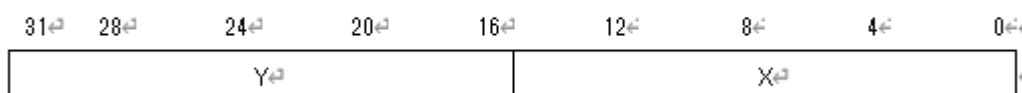
[Address: 0x0000_0048]



Name	Type	Default	Description
PC	R	0	Indicates the currently active PC (Program Counter).

5.3.1.11. BreakXY Register

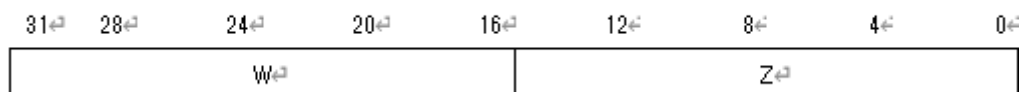
[Address: 0x0000_0050]



Name	Type	Default	Description
X	R/W	0	Specifies the break index X. Execution will halt just before processing this index. Setting it to '0' disables the break.
Y	R/W	0	Specifies the break index Y. It behaves the same as index X.

5.3.1.12. BreakZW Register

[Address: 0x0000_0054]

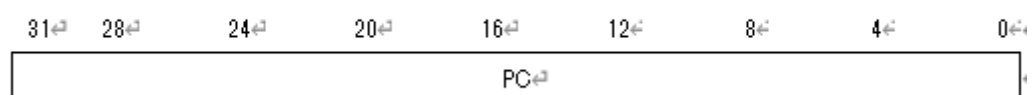


Name	Type	Default	Description
Z	R/W	0	Specifies the break index Z. It behaves the same as index X.

W	R/W	0	Specifies the break index W. It behaves the same as index X.
---	-----	---	---

5.3.1.13. BreakPC Register

[Address: 0x0000_0058]



Name	Type	Default	Description
PC	R/W	0	Specifies the break PC. It behaves the same as index X.

6. Application Notes

6.1. Processing Volume

- There are two ways to specify the processing volume: either through PSS or by setting a value in the TR register.

Both methods perform **horizontal processing**, where n logical processors operate in parallel.

- For **vertical processing**, in which loops are controlled by program branching, the processing volume is defined either as a register value or a constant.

6.2. Regarding Hyperbolic Instructions

- The built-in hyperbolic instructions use the CORDIC algorithm, which imposes limitations on the input value range.

For forward functions such as sinh or cosh, the absolute value must be less than or equal to 1.12.

For inverse functions such as atanh, the ratio B/A must have an absolute value less than or equal to 0.807.

Since this limitation affects all operations using hyperbolic instructions, care must be taken with any composed functions as well.

- The function $\text{sqrt}(x)$ can be expressed as $\text{sqrt}((x + 0.25)^2 - (x - 0.25)^2)$, which can be calculated using $A = x + 0.25$ and $B = x - 0.25$.

To satisfy the $B/A \leq 0.807$ constraint above, x must be less than or equal to approximately 2.3.

By using the f2ef instruction, one can obtain r and m such that $B = r * m$ with $r = 2^{2^n}$ and $0.5 \leq |m| < 2.0$.

First use this instruction, then calculate $\text{sqrt}(m) * \text{sqrt}(r)$ using the MAD instruction to construct the $\text{sqrt}(x)$ function across the full range.

Note that $\text{sqrt}(r) = 2^n$, so the exponent shift option in the MAD instruction ($\text{Intr}[3] = 0$, $\text{Op} = 2$) can be used.

- The function $\exp(x) = \sinh(x) + \cosh(x)$ is valid under the condition $|x| < 1.12$.

For values outside this range, decompose x into integer and fractional parts. Use a lookup table for the integer part and compute only the fractional part using the function.

- When computing atanh , the result of the simultaneous operation $\pm\sqrt{A^2 - B^2}$ reflects the sign of A.

This is because A belongs to the negative quadrant of the hyperbola.

6.3. Image Processing Example

- **Assumed Implementation Parameters**
 - Number of physical processors: 212
 - Number of logical processors: 22
 -

6.3.1. Mandelbrot Rendering

- **Conditions**
 - Image size: 64×64 ($mX = 64$, $mY = 64$)
 - Calculation coordinates: $X = -2.0$ to 2.0 , $Y = -2.0$ to 2.0
($dX = 2.0$, $dY = dX \times mY / mX$)
 - Maximum loop count: 64
 - Using PSS's $\Delta X = 1024$ and $\Delta Y = 4$, the image is expanded in 2D with $\Delta X = 32$ and $\Delta Y = 32$, then divided into 4 sections to render a 64×64 image
- **PSS Settings**
 - 1D Processing (65):
 - Configure the scalar register with a color palette read and (maximum loop count + 1) entries
 - 2D Processing (64×64):
 - Rendering
- **Scan Settings**
 - $x' = \{y[0], x[4:0]\} = ((y \& 1) \ll 5) + (x \& 0x1F)$
 - $y' = \{y[1], x[9:5]\} = (((y \gg 1) \& 1) \ll 5) + ((x \gg 5) \& 0x1F)$
 - $C1 = 0x1EDC_5BD8$
- **Instructions**
 - Composed of two sets: one with 2 instructions and the other with 14 instructions

PC	Cntl	Instruction		Comment
0		Set1	C5, C6, C7	Memory Parameters
16	Period	Int	SR[X]=@(X,Y)	Color Palette

PC	Cntl	Instruction		Comment
----	------	-------------	--	---------

0		Set0	C1(Lower), C2, C3, C4	exp = 2, expand index X to 32×32 and use with Scan()
16		Set1	C1(Upper), C5, C6, C7	Memory Parameters
32	Hold, Seed	Hyp	$R[0]=R[1]=0$	Clearing is performed using the two-operand output of Hyp.
64	Hold	Int	$R[3]=-1$	Count value (starts from -1)
96	Hold	Mad	$R[4]=\exp(\text{Scan}(X)*C0-1)$	Normalize coordinate X
128	Hold	Mad	$R[5]=\text{Scan}(Y)*C0-1$	Normalize coordinate Y
160	Hold,Des	Mad	$R[2]=R[0]*R[0]+R4$	Real part (initial) — Start of loop
192	Hold,Des	Mad	$R[2]=-R[1]*R[1]+R[2]$	Real part (final)
224	$F \sim V\&\sim Z, \text{Des}$	Hyp	$=\text{atanh}(R[2], 2)$	Overwrite the flag if the real part > 2
256	Hold,Des	Mad	$R[1]=\exp(R[0]R[1]+R[5])$	Imaginary Part
288	$F \sim V\&\sim Z, \text{Des}$ $F \neq 1$ Loop 6	Hyp	$=\text{atanh}(R[1], 2)$	Overwrite the flag if the real part > 2
320	Hold, Des	Int	$R[0]=R[2], R[3]++$	Delay Slot — End of Loop
352		Int	$R[3]=\text{SR}[R[3]]$	Color Palette Lookup
384	Period	Mem	$\text{Mem}[\text{Scan}(X), \text{Scan}(Y)]=R[3]$	Write to Memory

Hold: Holds the flag and prevents it from changing

Seed: Clears the CCR

Des: Sets Deselect and saves $R[n]$ and CCR