PLX 1.1 ISA Reference

February 2002

PLX is a small, general-purpose, subword-parallel instruction set architecture (ISA) designed at Princeton University, Department of Electrical Engineering. PLX was designed to be a simple yet high-performance ISA for multimedia information processing.

PLX History

- The design goals and architecture for PLX were specified by Prof. Ruby B. Lee of Princeton University.
- PLX version 0.1 was encoded, documented and implemented as a class project for the ELE-572 Class during Spring 2001, by Princeton graduates R. Adler '01 and G. Reis '01.
- This version of PLX was then completely re-done, with numerous additions and deletions of instructions and features, and re-encoded by R.B. Lee and A.M. Fiskiran.
- PLX 1.0 was released in September 2001.
- PLX 1.1 was released in February 2002. The changes include addition and deletion of some instructions and an improved predication scheme.

PLX Architectural Highlights

PLX is a RISC architecture designed for high-performance multimedia information processing.

PLX specifies 32 general-integer registers, numbered R0 through R31. For a given PLX implementation, the register size can be 32, 64 or 128 bits. The default size is 64 bits. No ISA changes are required to scale the datapath down to 32 bits or up to 128 bits. However, for word size and datapaths of 128 bits, some features are incomplete. Of the 32 general-integer registers, R0 is hardwired to 0, therefore it always returns 0 when read. Writing a value to R0 has no effect. GR31 is the implied link register for `jmp.reg` (jump register) and `jmp.reg.link` (jump register and link) instructions.

PLX is a fully subword-parallel ISA. Subword parallelism has been shown to be critical for achieving high-performance in multimedia applications. Subword sizes in PLX can be 1,2,4 or 8-bytes. The size of the largest subword for a given PLX implementation is limited by the datapath width of that implementation.

PLX uses 32-bit instructions, which are classified under 5 major instruction formats.

All PLX instructions are predicated. There are eight 1-bit predicate registers, numbered P0 through P7, forming a predicate register set of 1 byte long. There are 16 of these 1-byte predicate register sets, however only one of them is active at a given time. The active predicate register set is changed in software. In the active predicate register set, P0 is hardwired to 1, therefore, the instructions predicated on P0 always execute. This definition of predication is novel to PLX.

Currently, PLX does not have floating-point instructions, but this is viewed as a necessary future addition.

Program Flow Exceptions

Memory Alignment and Unaligned Address Trap

PLX enforces aligned memory accesses. This requires the following:

- Because PLX instructions are 4-bytes long, they need to be aligned at 4-byte boundaries. The least-significant byte of any instruction needs to be at an address whose two least significant bits are zero (xxx…xxx00). The remaining three bytes of the instruction will be at the next three addresses (xxx…xxx11, xxx…xxx10 and xxx…xxx01).

- Load and store instructions that read or write 4-byte data can only access data aligned at 4-byte boundaries. In a load instruction, the least significant byte of the data will be loaded from an address whose two least significant bits are zero (xxx…xxx00). The remaining three bytes of the data will be loaded from the following three addresses (xxx…xxx11, xxx…xxx10 and xxx…xxx01). In a store instruction, the least significant byte of the data will be stored to an address whose two least significant bits are zero (xxx…xxx00). The remaining three bytes of the data will be stored to the following three addresses (xxx…xxx11, xxx…xxx10 and xxx…xxx01).

- Load and store instructions that read or write 8-byte data can only access data aligned at 8-byte boundaries. In a load instruction, the least significant byte of the data will be loaded from an address whose three least significant bits are zero (xxx…xxx000). The remaining seven bytes of the data will be loaded from the following seven addresses (xxx…xxx111 through xxx…xxx001). In a store instruction, the least significant byte of the data will be stored to an address whose three least significant bits are zero (xxx…xxx000). The remaining seven bytes of the data will be stored to the following seven addresses (xxx…xxx111 through xxx…001).

If an unaligned memory access attempt is made, this raises an Unaligned Address Trap and the program control is returned to the operating system.

Illegal Instruction Trap

The PLX instructions are 32-bits in length but not all of the 2^{32} possibilities will correspond to a legal instruction. Some of the opcodes are reserved for future instructions, and therefore cannot be used. Even when an opcode may be valid, the subop or the immediate fields may have some restrictions on the values that can go in them. As an example, consider the `loadi.z.pos` instruction in a 32-bit PLX implementation. The 17th bit in the instruction can never be a one in this case, since this would correspond to `pos` field values of two and three, which are not possible in a 32-bit datapath. Therefore, any `loadi.z.pos` instruction with the 17th bit encoded as a one would be an illegal instruction. Whenever an the encoding of an instruction is illegal (either because the opcode is unused or reserved, or because some other field of the instruction has a value that is disallowed), this raises an Illegal Instruction Trap and the program control is returned to the operating system.

PLX Instructions (grouped by functionality)

Program Flow Control Instructions

jmp	Jump
jmp.link	Jump and Link
jmp.reg	Jump Register
jmp.reg.link	Jump Register and Link
trap	Trap

Compare Instructions and Predication

changepr	Change Predicate Register Set
changepr.ld	Change Predicate Register Set and Load
cmp.rel	Compare
cmp.rel.pw0	Compare Parallel Write Zero
cmp.rel.pw1	Compare Parallel Write One
cmpi.rel	Compare Immediate
testbit	Test Bit

Memory Access Instructions

loadi.z.pos	Load Immediate
load.sw	Load
load.sw.update	Load Update
loadx.sw	Load Indexed
loadx.sw.update	Load Indexed Update
store.sw	Store
store.sw.update	Store Update

ALU Instructions (Immediate)

addi	Add Immediate
andi	And Immediate
ori	Or Immediate
subi	Subtract Immediate
xori	Xor Immediate

Shift and Bit Field Instructions (Immediate)

slli	Shift Left Logical Immediate
srai	Shift Right Arithmetic Immediate
srli	Shift Right Logical Immediate
shrp	Shift Right Pair
extract	Extract
deposit	Deposit

ALU Instructions

padd.sw	Parallel Add
padd.sw.u	Parallel Add Unsigned Saturation
padd.sw.s	Parallel Add Signed Saturation
paddincr	Parallel Add Increment
psub.sw	Parallel Subtract
psub.sw.u	Parallel Subtract Unsigned Saturation
psub.sw.s	Parallel Subtract Signed Saturation
psubdecr.sw	Parallel Subtract Decrement
pavg.sw	Parallel Average
pavg.sw.raz	Parallel Average Round Away From Zero
psubavg.sw	Parallel Subtract Average
and	And
andcm	And Complement
or	Or
xor	Xor
not	Not
pcmp.sw.eq	Parallel Compare Equal To
pcmp.sw.gt	Parallel Compare Greater Than
pmax.sw	Parallel Maximum
pmin.sw	Parallel Minimum
pshiftadd.sw.l	Parallel Shift Left and Add
pshiftadd.sw.r	Parallel Shift Right and Add

Multiply Instructions

pmul.odd	Parallel Multiply Odd
pmul.odd.u	Parallel Multiply Odd Unsigned
pmul.even	Parallel Multiply Even
pmul.even.u	Parallel Multiply Even Unsigned
pmulshr.sa	Parallel Multiply and Shift Right Logical
pmulshr.sa.a	Parallel Multiply and Shift Right Arithmetic

Shift Instructions

pshift.sw.l	Parallel Shift Left Logical
pshift.sw.r	Parallel Shift Right Logical
pshift.sw.ra	Parallel Shift Right Arithmetic

Shift Instructions (Immediate)

pshifti.sw.l	Parallel Shift Immediate Left Logical
pshifti.sw.r	Parallel Shift Immediate Right Logical
pshifti.sw.ra	Parallel Shift Immediate Right Arithmetic

Subword Permutation Instructions

mix.sw.l	Mix Left
mix.sw.r	Mix Right
mux.sw.rev	Mux Reverse
mux.sw.mix	Mux Mix
mux.sw.shuf	Mux Shuffle
mux.sw.alt	Mux Alternate
mux.sw.brcst	Mux Broadcast
perm	Permute

PLX Instructions (in alphabetical order of major mnemonics)

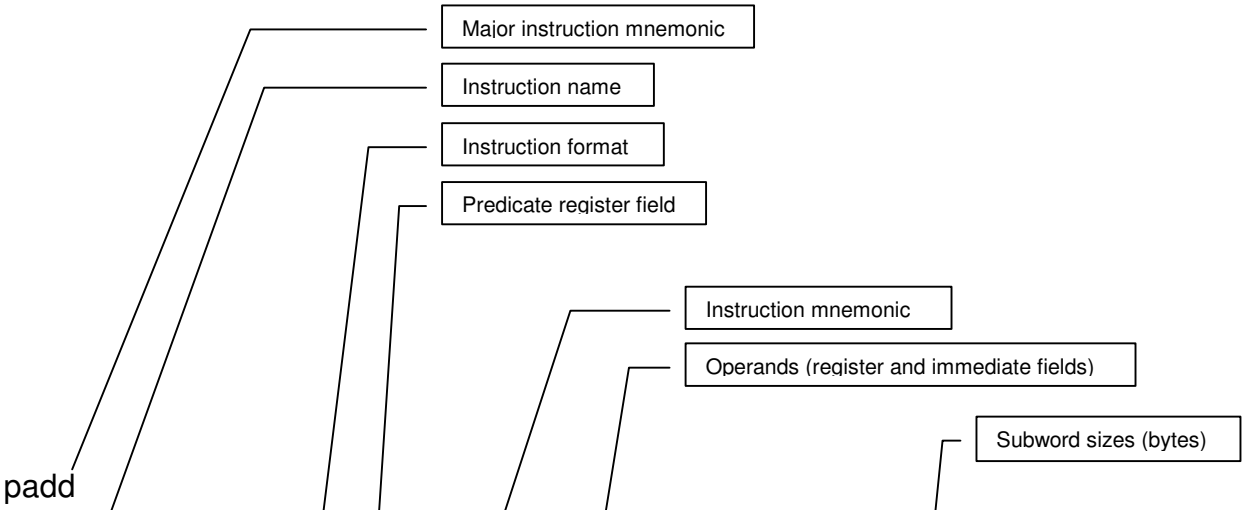
addi	Add Immediate
and	And
andcm	And Complement
andi	And Immediate
changepr	Change Predicate Register Set
cmp	Compare
cmpi	Compare Immediate
deposit	Deposit
extract	Extract
jmp	Jump
load	Load
loadi	Load Immediate
loadx	Load Indexed
mix	Mix
mux	Mux
not	Not
or	Or
ori	Or Immediate
padd	Parallel Add
paddincr	Parallel Add Increment
pavg	Parallel Average
pcmp	Parallel Compare
perm	Permute
pmax	Parallel Maximum
pmin	Parallel Minimum
pmul	Parallel Multiplication
pmulshr	Parallel Multiply Shift Right
pshift	Parallel Shift
pshiftadd	Parallel Shift Add
pshifti	Parallel Shift Immediate
psub	Parallel Subtract
psubavg	Parallel Subtract Average
psubdecr	Parallel Subtract Decrement
shrp	Shift Right Pair
slli	Shift Left Logical Immediate
srai	Shift Right Arithmetic Immediate
srli	Shift Right Logical Immediate
store	Store
subi	Subtract Immediate
testbit	Test Bit
trap	Trap
xor	Xor
xori	Xor Immediate

Last revised on 12/9/2002 by A. Murat Fiskiran

PLX Instruction Reference

This document is available from the PLX website: http://palms.ee.princeton.edu/plx

Formatting Used in Instruction Descriptions



Parallel Add

Format:

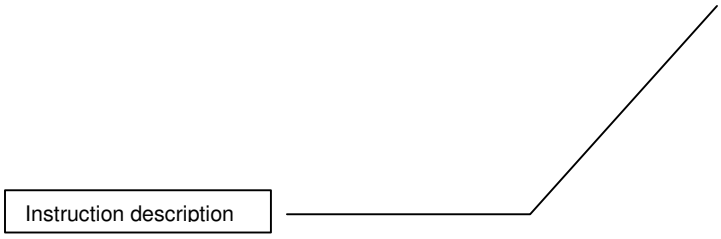
4a	(P)	padd.sw Rd,Rs1,Rs2	1,2,4,8
4a	(P)	padd.sw.u Rd,Rs1,Rs2	1,2,4,8
4a	(P)	padd.sw.s Rd,Rs1,Rs2	1,2,4,8

Description:

Rs1 and Rs2 are added, and the result is written to Rd.

Subword size is specified in the sw field, and can be 1,2,4 or 8 bytes.

Padd.sw uses modular arithmetic, padd.sw.u uses unsigned saturation, and padd.sw.s uses signed saturation during the add.



addi

Add Immediate

Format: 2 (P) addi Rd,Rs1,imm13

Description: Imm13 is sign extended and added to Rs1. The result is written to Rd.

and

And

Format: 4a (P) and Rd,Rs1,Rs2

Description: Rs1 and Rs2 are anded. The result is written to Rd.

andcm

And Complement

Format: 4a (P) andcm Rd,Rs1,Rs2

Description: Rs1 and the complement of Rs2 are anded. The result is written to Rd.

andi

And Immediate

Format: 2 (P) andi Rd,Rs1,imm13

Description: Imm13 is zero extended and anded with Rs1. The result is written to Rd.

changepr

Change Predicate Register Set

Format: 5b (P) changepr imm4
5b (P) changepr.ld imm4,imm8

Description: In changepr, the active predicate register set is changed to the predicate register set specified by imm4.

In changepr.ld, the active predicate register set is changed to the predicate register set specified by imm4, and imm8 is written to this predicate register set.

cmp

Compare

Format: 5a (P) cmp.rel Rs1,Rs2,Pd1,Pd2

Description: Rs1 and Rs2 are compared with each other according to the relation specified in the rel field (see table below for a listing of possible rel values).

If the relation is true, the value 1 is written to Pd1, and its complement, 0, is written to Pd2. If the relation is false, the value 0 is written to Pd1, and its complement, 1, is written to Pd2.

rel	Relation	Sign of a and b
eq	a == b	N/A
ne	a != b	N/A
lt	a < b	Signed
le	a <= b	Signed
gt	a > b	Signed
ge	a >= b	Signed
ltu	a < b	Unsigned
leu	a <= b	Unsigned
gtu	a > b	Unsigned
geu	a >= b	Unsigned

cmp

Compare Parallel Write

Format:

5a (P) cmp.rel.pw0 Rs1,Rs2,Pd1,Pd2
5a (P) cmp.rel.pw1 Rs1,Rs2,Pd1,Pd2

Description:

Rs1 and Rs2 are compared with each other according to the relation specified in the rel field. (See the table in the description of the cmp instruction.)

For cmp.rel.pw0, the instruction works as follows:
If the relation is true, the value 0 is written to Pd1, and its complement, 1, is written to Pd2. If the relation is false, nothing is written to Pd1 or Pd2. Multiple cmp.rel.w0 instructions can be executed in the same cycle, targeting the same predicate registers, since only 0 can be written (concurrently) to Pd1, and only 1 can be written (concurrently) to Pd2.

For cmp.rel.pw1, the instruction works as follows:
If the relation is true, the value 1 is written to Pd1, and its complement, 0, is written to Pd2. If the relation is false, nothing is written to Pd1 or Pd2. Multiple cmp.rel.w1 instructions can be executed in the same cycle, targeting the same predicate registers, since only 1 can be written (concurrently) to Pd1, and only 0 can be written (concurrently) to Pd2.

cmpi

Compare Immediate

Format: 5b (P) cmpi.rel Rs1,imm8,Pd1,Pd2

Description: Rs1 is compared to sign-extended imm8 according to the relation specified in the rel field. (See the table in the description of the cmp instruction.)

If the relation is true, the value 1 is written to Pd1 and its complement, 0, is written to Pd2. If the relation is false, the value 0 is written to Pd1 and its complement, 1, is written to Pd2.

deposit

Deposit

Format: 3 (P) deposit Rd,Rs1,imm7,imm6

Description: Right-aligned bit field of length imm6 from Rs1, is written to Rd, starting at location specified by imm7. Remaining bits of Rd are unchanged.

extract

Extract

Format: 3 (P) extract Rd,Rs1,imm7,imm6

Description: Bit field of length imm6 of Rs1, starting at a location specified by imm7, is written right-aligned to Rd. High order bits of Rd are cleared.

jmp

Jump

Format:
0 (P) jmp imm23
0 (P) jmp.link imm23
1 (P) jmp.reg Rd
1 (P) jmp.reg.link Rd

Description: In jmp, imm23 is added to the current PC. The result becomes the new PC.

In jmp.link, PC + 4 is written to GR[31]. The previous value of GR[31] is destroyed. Then, imm23 is added to the PC. The result becomes the new PC.

In jmp.reg, Rd is added to the current PC. The result becomes the new PC.

In jmp.reg.link, PC + 4 is written to GR[31]. The previous value of GR[31] is destroyed. Then, Rd is added to the PC. The result becomes the new PC.

load

Load

Format:

2	(P) load.sw Rd,Rs1,imm13	4,8
2	(P) load.sw.update Rd,Rs1,imm13	4,8

Description:

The data in a memory location is loaded into register Rd. The size of the loaded data is indicated in the sw field, and can be 4 or 8 bytes. The addressing mode is displacement mode, where the effective address is calculated as Rs1+imm13. The base address register, Rs1, can be updated with the new address, using post-modify, i.e., for load.sw.update, the memory address used to fetch the data is given by Rs1, then Rs1 is replaced with (Rs1+imm13). Memory addresses must be aligned, otherwise an Unaligned Address Trap occurs.

The instruction works as follows:
For load.4, if (Rs1+ imm13) is not equal to 0 in mod 4, then Unaligned Address Trap occurs. Otherwise Rd is loaded with the 4 bytes from M[Rs1+imm13].

For load.8, if (Rs1+imm13) is not equal to 0 in mod 8, then Unaligned Address Trap occurs. Otherwise Rd is loaded with the 8 bytes from M[Rs1+ imm13].

For load.4.update, if Rs1 is not equal to 0 in mod 4, then Unaligned Address Trap occurs. Otherwise Rd is loaded with the 4 bytes from M[Rs1], and Rs1 is replaced with (Rs1+imm13).

For load.8.update, if Rs1 is not equal to 0 in mod 8, then Unaligned Address Trap occurs. Otherwise Rd is loaded with the 8 bytes from M[Rs1], and Rs1 is replaced with (Rs1+imm13).

loadi

Load Immediate

Format:

1 (P) loadi.z.pos Rd,imm16
1 (P) loadi.k.pos Rd,imm16

Description:

The loadi.z.pos instruction writes imm16 into one of four different positions in the lower 64-bits of register Rd, clearing the rest of Rd to zeros. (For 32-bit PLX implementations, pos=2 and pos=3 result in an Illegal Instruction Trap.) The pos field is specified by bits 16 and 17 in the instruction, giving 4 possible 16-bit field positions in Rd:

- For Loadi.z.0, bits 0-15 of Rd is replaced with imm16, the other bits of Rd are cleared to zeros.
- For Loadi.z.1, bits 16-31 of Rd is replaced with imm16, the other bits of Rd are cleared to zeros.
- For Loadi.z.2, bits 32-47 of Rd is replaced with imm16, the other bits of Rd are cleared to zeros.
- For Loadi.z.3, bits 48-64 of Rd is replaced with imm16, the other bits of Rd are cleared to zeros."

The loadi.k.pos instruction writes imm16 into one of four different positions in the lower 64-bits of register Rd, keeping the rest of Rd unchanged. (For 32-bit PLX implementations, pos=2 and pos=3 result in an Illegal Instruction Trap.) The pos field is specified by bits 16 and 17 in the instruction, giving 4 possible 16-bit field positions in Rd:

- For Loadi.k.0, bits 0-15 of Rd is replaced with imm16, the other bits of Rd are left unchanged.
- For Loadi.k.1, bits 16-31 of Rd is replaced with imm16, the other bits of Rd are left unchanged.
- For Loadi.k.2, bits 32-47 of Rd is replaced with imm16, the other bits of Rd are left unchanged.
- For Loadi.k.3, bits 48-64 of Rd is replaced with imm16, the other bits of Rd are left unchanged.

loadx

Load Indexed

Format:

4a	(P) loadx.sw Rd,Rs1,Rs2	4,8
4a	(P) loadx.sw.update Rd,Rs1,Rs2	4,8

Description:

The data in a memory location is loaded into register Rd. The size of the loaded data is indicated in the sw field, and can be 4 or 8 bytes. The addressing mode is indexed mode, where the effective address is calculated as Rs1+ Rs2. The base address register, Rs1, can be updated with the new address, using post-modify, i.e., the memory address used to fetch the data is given by Rs1, then Rs1 is replaced with (Rs1+Rs2). Memory addresses must be aligned, otherwise an Unaligned Address Trap occurs.

The instruction works as follows:
For loadx.4, if (Rs1+Rs2) is not equal to 0 in mod 4, then Unaligned Address Trap occurs. Otherwise, Rd is loaded with the 4 bytes from M[Rs1+Rs2].

For loadx.8, if (Rs1+Rs2) is not equal to 0 in mod 8, then Unaligned Address Trap occurs. Otherwise, Rd is loaded with the 8 bytes from M[Rs1+Rs2].

For loadx.4.update, if Rs1 is not equal to 0 in mod 4, then Unaligned Address Trap occurs. Otherwise Rd is loaded with the 4 bytes from M[Rs1], and Rs1 is replaced with (Rs1+Rs2).

For loadx.8.update, if Rs1 is not equal to 0 in mod 8, then Unaligned Address Trap occurs. Otherwise Rd is loaded with the 8 bytes from M[Rs1], and Rs1 is replaced with (Rs1+Rs2).

mix

Mix

Format:

3	(P) mix.sw.l Rd,Rs1,Rs2	1,2,4
3	(P) mix.sw.r Rd,Rs1,Rs2	1,2,4

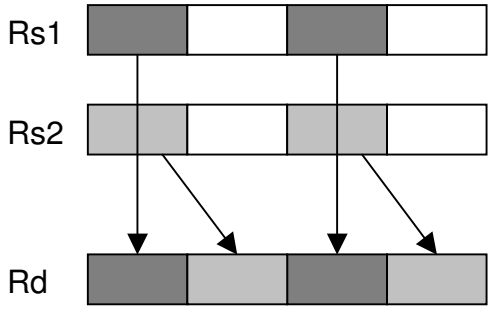
Description:

Even or odd-indexed subwords are selected alternately from Rs1 and Rs2, and written to Rd.

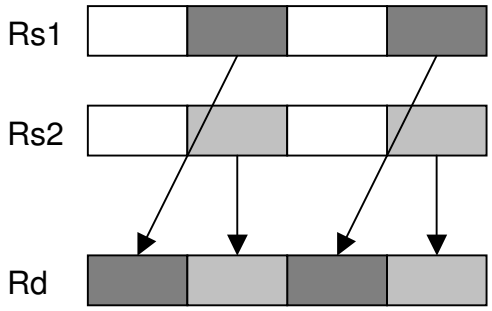
Subword size is indicated in the sw field, and can be 1,2 or 4 bytes.

In mix.sw.l, odd-indexed subwords are selected alternately from Rs1 and Rs2, and written to Rd. The first subword of Rd is the first subword of Rs1.

In mix.sw.r, even-indexed subwords are selected alternately from Rs1 and Rs2 are written to Rd. The first subword of Rd is the second subword of Rs1.



mix.2.l Rd,Rs1,Rs2



mix.2.r Rd,Rs1,Rs2

mux

Mux

Format:

4b	(P) mux.sw.rev Rd,Rs1	1
4b	(P) mux.sw.mix Rd,Rs1	1
4b	(P) mux.sw.shuf Rd,Rs1	1
4b	(P) mux.sw.alt Rd,Rs1	1
4b	(P) mux.sw.brcst Rd,Rs1	1,2

Description:

A permutation is performed on the subwords of Rs1 and the result is written to Rd.

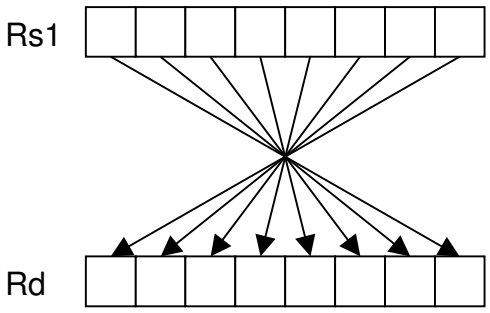
Mux.sw.rev reverses the order of the subwords of Rs1.

Mux.sw.mix divides Rs1 into left and right halves, then a mix operation is performed on these two halves of Rs1.

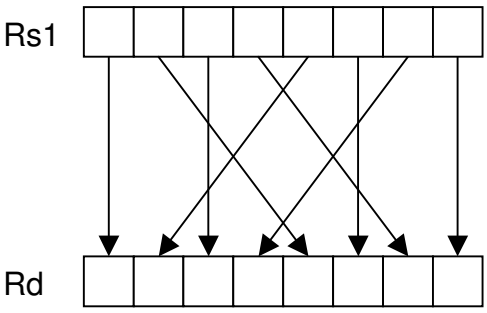
Mux.sw.shuf divides Rs1 into left and right halves, then a shuffle operation is performed on these two halves of Rs1.

Mux.sw.alt divides Rs1 into left and right halves, then an alternate operation is performed on these two halves of Rs1.

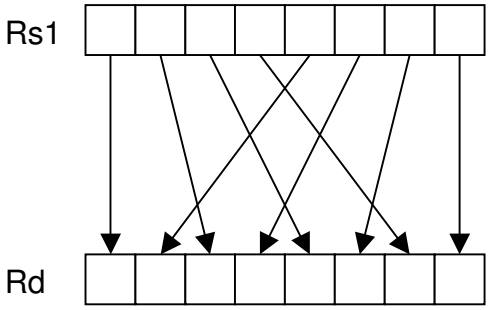
Mux.sw.brcst writes the least-significant subword of Rs1 to all subwords of Rd.



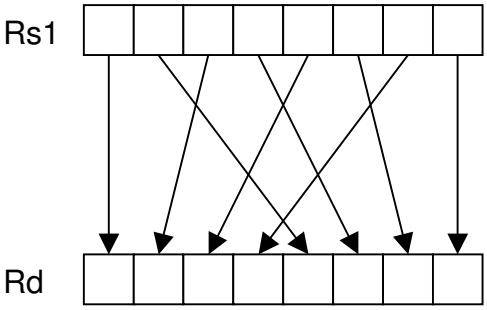
mux.1.rev Rd,Rs1



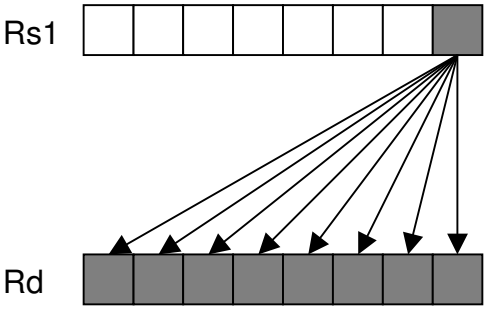
mux.1.mix Rd,Rs1



mux.1.shuf Rd,Rs1



mux.1.alt Rd,Rs1



mux.1.brcst Rd,Rs1

not

Not

Format: 4a (P) not Rd,Rs1

Description: Rs1 is complemented. The result is written to Rd.

or

Or

Format: 4a (P) or Rd,Rs1,Rs2

Description: Rs1 and Rs2 are ored. The result is written to Rd.

ori

Or Immediate

Format: 2 (P) ori Rd,Rs1,imm13

Description: Imm13 is zero extended and ored with Rs1. The result is written to Rd.

padd

Parallel Add

Format:

4a	(P) padd.sw Rd,Rs1,Rs2	1,2,4,8
4a	(P) padd.sw.u Rd,Rs1,Rs2	1,2,4,8
4a	(P) padd.sw.s Rd,Rs1,Rs2	1,2,4,8

Description: Rs1 and Rs2 are added, and the result is written to Rd.

Subword size is specified in the sw field, and can be 1,2,4 or 8 bytes.

Padd.sw uses modular arithmetic, padd.sw.u uses unsigned saturation, and padd.sw.s uses signed saturation during the add.

paddincr

Parallel Add Increment

Format: 4a (P) paddincr.sw Rd,Rs1,Rs2 1,2,4,8

Description: Rs1 and Rs2 are added, and their sum is incremented by one. The result is written to Rs2. Modular arithmetic is used.

Subword size is specified in the sw field, and can be 1,2,4 or 8 bytes.

pavg

Parallel Average

Format:

4a	(P) pavg.sw Rd,Rs1,Rs2	1,2
4a	(P) pavg.sw.raz Rd,Rs1,Rs2	1,2

Description:

Averages of the subwords from Rs1 and Rs2 are written to Rd.

Subword size is specified in the sw field, and can be 1 or 2 bytes.

In pavg.sw, unsigned subwords from Rs1 and Rs2 are added, and the sums are shifted right by one bit. The highest order bit becomes the carryout of the add operation. The shifted results are written to Rs2. The least-significant bit of each result subword is the or of the two least-significant bits of the shifted sums.

In pavg.sw.raz (raz stands for round away from zero), unsigned subwords from Rs1 and Rs2 are added, and the sums are incremented by one. The incremented sums are then shifted right by one bit. The highest order bit becomes the carryout of the add operation. The shifted results are written to Rs2. The least-significant bit of each result subword is the least-significant bit of the shifted sums.

pcmp

Parallel Compare

Format:

4a	(P) pcmp.sw.eq Rd,Rs1,Rs2	1,2,4,8
4a	(P) pcmp.sw.gt Rd,Rs1,Rs2	1,2,4,8

Description:

In pcmp.eq, subwords from Rs1 and Rs2 are tested for equality.

In pcmp.ge, signed subwords from Rs1 are tested for being greater-than the signed subwords of Rs2.

Subword size is specified in the sw field, and can be 1,2,4 or 8 bytes.

If the comparison condition is true, then corresponding subword of Rd is set to all ones, otherwise it is set to all zeros.

perm

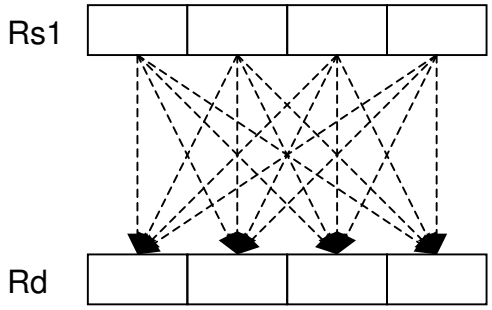
Permute

Format: 4a (P) perm Rd,Rs1,Rs2 2

Description: A permutation is performed on the 2-byte subwords of Rs1 and the result is written to Rd.

All possible permutations can be performed, with or without repetitions of subwords. The permutation is specified by the bits read from Rs2.

(If Rs1 has n subwords, this requires nlogn bits to specify a permutation. These bits are read from the low-order nlogn bits of Rs2.)



perm Rd,Rs1,Rs2

pmax

Parallel Maximum

Format: 4a (P) pmax Rd,Rs1,Rs2 1,2

Description: The greater of the subwords from Rs1 and Rs2 is written to Rd.

Subwords are treated as signed values.

Subword size is specified in the sw field, and can be 1 or 2 bytes.

pmin

Parallel Minimum

Format: 4a (P) pmin Rd,Rs1,Rs2 1,2

Description: The lesser of the subwords from Rs1 and Rs2 is written to Rd.

Subwords are treated as signed values.

Subword size is specified in the sw field, and can be 1 or 2 bytes.

pmul

Parallel Multiply

Format:

4a	(P) pmul.odd Rd,Rs1,Rs2	2
4a	(P) pmul.odd.u Rd,Rs1,Rs2	2
4a	(P) pmul.even Rd,Rs1,Rs2	2
4a	(P) pmul.even.u Rd,Rs1,Rs2	2

Description:

In pmul.odd, odd indexed signed 16-bit subwords from Rs1 and Rs2 are multiplied, and the 32-bit products are written to Rd.

In pmul.odd.u, odd indexed unsigned 16-bit subwords from Rs1 and Rs2 are multiplied, and the 32-bit products are written to Rd.

In pmul.even, even indexed signed 16-bit subwords from Rs1 and Rs2 are multiplied, and the 32-bit products are written to Rd.

In pmul.even.u, even indexed unsigned 16-bit subwords from Rs1 and Rs2 are multiplied, and the 32-bit products are written to Rd.

pmulshr

Parallel Multiply Shift Right

Format:

4a	(P) pmulshr.sa Rd,Rs1,Rs2	2
4a	(P) pmulshr.sa.a Rd,Rs1,Rs2	2

Description:

In pmulshr.sa, unsigned 16-bit subwords from Rs1 and Rs2 are multiplied. Each product is then logically shifted to the right by sa bits, where sa can be 0,8,15, or 16. The lower halves of the shifted products are written to Rd.

In pmulshr.sa.a, signed 16-bit subwords from Rs1 and Rs2 are multiplied. Each product is then arithmetically shifted to the right by sa bits, where sa can be 0,8,15, or 16. The lower halves of the shifted products are written to Rd.

pshift

Parallel Shift

Format:

4a	(P) pshift.sw.l Rd,Rs1,Rs2	2,4,8
4a	(P) pshift.sw.r Rd,Rs1,Rs2	2,4,8
4a	(P) pshift.sw.ra Rd,Rs1,Rs2	2,4,8

Description:

Subwords of Rs1 are shifted and the result is written to Rd.

Subword size is specified in the sw field, and can be 2,4 or 8 bytes.

In pshift.sw.l, subwords of Rs1 are logically shifted to the left by Rs2 bits.

In p.shift.sw.r, subwords of Rs1 are logically shifted to the right by Rs2 bits.

In, pshift.sw.ra, subwords of Rs1 are arithmetically shifted to the right by Rs2 bits.

pshiftadd

Parallel Shift and Add

Format:

4a	(P) pshiftadd.sa.l Rd,Rs1,Rs2	2
4a	(P) pshiftadd.sa.r Rd,Rs1,Rs2	2

Description:

In pshiftadd.sa.l, signed 2-byte subwords of Rs1 are shifted left by sa bits, where sa can be 1, 2, or 3. The result is added to Rs2 using signed saturation arithmetic. The result is written to Rd.

In pshiftadd.sa.r, signed 2-byte subwords of Rs1 are shifted right by sa bits, where sa can be 1, 2, or 3. The result is added to Rs2 using signed saturation arithmetic. The result is written to Rd.

pshifti

Parallel Shift Immediate

Format:

4b	(P) pshifti.sw.l Rd,Rs1,imm5	2,4,8
4b	(P) pshifti.sw.r Rd,Rs1,imm5	2,4,8
4b	(P) pshifti.sw.ra Rd,Rs1,imm5	2,4,8

Description:

In pshifti.sw.l, subwords of Rs1 are logically shifted to the left by imm5 bits.

Subword size is specified in the sw field, and can be 2,4 or 8 bytes.

In pshifti.sw.r, subwords of Rs1 are logically shifted to the right by imm5 bits.

In pshifti.sw.ra, subwords of Rs1 are arithmetically shifted to the right by imm5 bits.

psub

Parallel Subtract

Format:

4a	(P) psub.sw Rd,Rs1,Rs2	1,2,4,8
4a	(P) psub.sw.u Rd,Rs1,Rs2	1,2,4,8
4a	(P) psub.sw.s Rd,Rs1,Rs2	1,2,4,8

Description:

Rs2 is subtracted from Rs1. The result is written to Rd.

Subword size is specified in the sw field, and can be 1,2,4 or 8 bytes.

Psub uses modular arithmetic, psub.u uses unsigned saturation, and psub.s uses signed saturation during the subtract.

psubavg

Parallel Subtract Average

Format:

4a	(P) psubavg.sw Rd,Rs1,Rs2	1,2

Description:

Unsigned Rs2 is subtracted from unsigned Rs1. The differences are shifted right by one bit. The highest order bit becomes the carryout of the subtract operation. The shifted results are written to Rsd. The least-significant bit of each result subword is the or of the two least-significant bits of the shifted differences.

Subword size is specified in the sw field, and can be 1 or 2 bytes.

psubdecr

Parallel Subtract Decrement

Format: 4a (P) psubdecr.sw Rd,Rs1,Rs2 1,2,4,8

Description: Rs2 is subtracted from Rs1, and the difference is decremented by one. The result is written to Rd. Modular arithmetic is used.

Subword size is specified in the sw field, and can be 1,2,4 or 8 bytes.

shrp

Shift Right Pair

Format: 4c (P) shrp Rd,Rs1,Rs2,imm8

Description: Rs1 and Rs2 are concatenated and logically shifted to the right by imm8 bits. The lower-order half of the shifted result is written to Rd. (Most-significant bit of imm8 is ignored for 64-bit processors; most-significant two bits of imm8 are ignored for 32-bit processors.)

slli

Shift Left Logical Immediate

Format: 2 (P) slli Rd,Rs1,imm13

Description: Rs1 is shifted to the left by imm13 bits. If imm13 is greater than the word size, then only the low-order bits of imm13 are used as the shift amount. The vacated bits are filled with zeroes. The result is written to Rd.

srai

Shift Right Arithmetic Immediate

Format: 2 (P) srai Rd,Rs1,imm13

Description: Rs1 is shifted to the right by imm13 bits. If imm13 is greater than the word size, then only the low-order bits of imm13 are used as the shift amount. The vacated bits are filled with the sign bit. The result is written to Rd.

srli

Shift Right Logical Immediate

Format: 2 (P) srli Rd,Rs1,imm13

Description: Rs1 is shifted to the right by imm13 bits. If imm13 is greater than the word size, then only the low-order bits of imm13 are used as the shift amount. The vacated bits are filled with zeroes. The result is written to Rd.

store

Store

Format:

2	(P) store.sw Rd,Rs1,imm13	1,2,4,8
2	(P) store.sw.update Rd,Rs1,imm13	1,2,4,8

Description:

The data in register Rd is stored to a memory location. The size of the stored data is indicated in the sw field, and can be 1,2,4 or 8 bytes. The addressing mode is displacement mode, where the effective address is calculated as Rs1+imm13. The base address register, Rs1, can be updated with the new address, using post-modify, i.e., for store.sw.update, the memory address used to store the data is given by Rs1, then Rs1 is replaced with (Rs1+imm13). Memory addresses must be aligned, otherwise an Unaligned Address Trap occurs.

The instruction works as follows:
For store.sw, if (Rs1+ imm13) is not equal to 0 in mod sw, then Unaligned Address Trap occurs. Otherwise sw least significant bytes of Rd are stored to M[Rs1+imm13].

For store.sw.update, if Rs1 is not equal to 0 in mod sw, then Unaligned Address Trap occurs. Otherwise sw least significant bytes of Rd are stored to M[Rs1], and Rs1 is replaced with (Rs1+imm13).

subi

Subtract Immediate

Format: 2 (P) subi Rd,Rs1,imm13

Description: Imm13 is sign extended and subtracted from Rs1. The result is written to Rd.

testbit

Test Bit

Format: 5b (P) testbit Rs1,imm8,Pd1,Pd2

Description: The bit specified by imm8 is selected from Rs1. If this bit is 1, the value 1 is written to Pd1, and its complement, 0, is written to Pd2. If the bit is 0, then the value 0 is written to Pd1, and its complement, 1, is written to Pd2. If imm8 is larger than the number of bits in Rs1, an Illegal Instruction Trap occurs.

trap

Trap

Format: 0 (P) trap

Description: The processor halts execution unconditionally.

xor

Xor

Format: 4a (P) xor Rd,Rs1,Rs2

Description: Rs1 and Rs2 are xored. The result is written to Rd.

xori

Xor Immediate

Format: 2 (P) xori Rd,Rs1,imm13

Description: Imm13 is zero extended and xored with Rs1. The result is written to Rd.

PLX 1.1 ISA Encoding

Refer to Tables 1-7 in the most recent revision of the document titled “PLX 1.1 ISA Encoding”.