P-Code Reference Manual

Last updated September 5, 2019

COPY	INT_ADD	BOOL_OR
LOAD	INT_SUB	FLOAT_EQUAL
STORE	INT_CARRY	FLOAT_NOTEQUAL
BRANCH	INT_SCARRY	FLOAT_LESS
CBRANCH	INT_SBORROW	FLOAT_LESSEQUAL
BRANCHIND	INT_2COMP	FLOAT_ADD
CALL	INT_NEGATE	FLOAT_SUB
CALLIND	INT_XOR	FLOAT_MULT
USERDEFINED	INT_AND	FLOAT_DIV
RETURN	INT_OR	FLOAT_NEG
PIECE	INT_LEFT	FLOAT_ABS
SUBPIECE	INT_RIGHT	FLOAT_SQRT
POPCOUNT	INT_SRIGHT	FLOAT_CEIL
INT_EQUAL	INT_MULT	FLOAT_FLOOR
INT_NOTEQUAL	INT_DIV	FLOAT_ROUND
INT_LESS	INT_REM	FLOAT_NAN
INT_SLESS	INT_SDIV	INT2FLOAT
INT_LESSEQUAL	INT_SREM	FLOAT2FLOAT
INT_SLESSEQUAL	BOOL_NEGATE	TRUNC
INT_ZEXT	BOOL_XOR	CPOOLREF
INT_SEXT	BOOL_AND	NEW

A Brief Introduction to P-Code

P-code is a *register transfer language* designed for reverse engineering applications. The language is general enough to model the behavior of many different processors. By modeling in this way, the analysis of different processors is put into a common framework, facilitating the development of retargetable analysis algorithms and applications.

Fundamentally, p-code works by translating individual processor instructions into a sequence of **p-code operations** that take parts of the processor state as input and output variables (**varnodes**). The set of unique p-code operations (distinguished by **opcode**) comprise a fairly tight set of the arithmetic and logical actions performed by general purpose processors. The direct translation of instructions into these operations is referred to as **raw p-code**. Raw p-code can be used to directly emulate instruction execution and generally follows the same control-flow, although it may add some of its own internal control-flow. The subset of opcodes that can occur in raw p-code is described in the section called “P-Code Operation Reference” and in the section called “Pseudo P-CODE Operations”, making up the bulk of this document.

P-code is designed specifically to facilitate the construction of *data-flow* graphs for follow-on analysis of disassembled instructions. Varnodes and p-code operators can be thought of explicitly as nodes in these graphs. Generation of raw p-code is a necessary first step in graph construction, but additional steps are required, which introduces some new opcodes. Two of these, **MULTIEQUAL** and **INDIRECT**, are specific to the graph construction process, but other opcodes can be introduced during subsequent analysis and transformation of a graph and help hold recovered data-type relationships. All of the new opcodes are described in the section called “Additional P-CODE Operations”, none of which can occur in the original raw p-code translation. Finally, a few of the p-code operators, **CALL**, **CALLIND**, and **RETURN**, may have their input and output varnodes changed during analysis so that they no longer match their *raw p-code* form.

The core concepts of p-code are:

Address Space

The **address space** for p-code is a generalization of RAM. It is defined simply as an indexed sequence of bytes that can be read and written by the p-code operations. For a specific byte, the unique index that labels it is the byte's **address**. An address space has a name to identify it, a size that indicates the number of distinct indices into the space, and an **endianess** associated with it that indicates how integers and other multi-byte values are encoded into the space. A typical processor will have a **ram** space, to model memory accessible via its main data bus, and a **register** space for modeling the processor's general purpose registers. Any data that a processor manipulates must be in some address space. The specification for a processor is free to define as many address spaces as it needs. There is always a special address space, called a **constant** address space, which is used to encode any constant values needed for p-code operations. Systems generating p-code also generally use a dedicated **temporary** space, which can be viewed as a bottomless source of temporary registers. These are used to hold intermediate values when modeling instruction behavior.

P-code specifications allow the addressable unit of an address space to be bigger than just a byte. Each address space has a **wordsize** attribute that can be set to indicate the number of bytes in a unit. A wordsize which is bigger than one makes little difference to the representation of p-code. All the offsets into an address space are still represented internally as a byte offset. The only exceptions are the **LOAD** and **STORE** p-code operations. These operations read a pointer offset that must be scaled properly to get the right byte offset when dereferencing the pointer. The wordsize attribute has no effect on any of the other p-code operations.

Varnode

A **varnode** is a generalization of either a register or a memory location. It is represented by the formal triple: an address space, an offset into the space, and a size. Intuitively, a varnode is a contiguous sequence of bytes in some address space that can be treated as a single value. All manipulation of data by p-code operations occurs on varnodes.

Varnodes by themselves are just a contiguous chunk of bytes, identified by their address and size, and they have no type. The p-code operations however can force one of three *type* interpretations on the varnodes: integer, boolean, and floating-point.

- Operations that manipulate integers always interpret a varnode as a twos-complement encoding using the endianess associated with the address space containing the varnode.
- A varnode being used as a boolean value is assumed to be a single byte that can only take the value 0, for *false*, and 1, for *true*.
- Floating-point operations use the encoding expected by the processor being modeled, which varies depending on the size of the varnode. For most processors, these encodings are described by the IEEE 754 standard, but other encodings are possible in principle.

If a varnode is specified as an offset into the **constant** address space, that offset is interpreted as a constant, or immediate value, in any p-code operation that uses that varnode. The size of the varnode, in this case, can be treated as the size or precision available for the encoding of the constant. As with other varnodes, constants only have a type forced on them by the p-code operations that use them.

P-code Operation

A **p-code operation** is the analog of a machine instruction. All p-code operations have the same basic format internally. They all take one or more varnodes as input and optionally produce a single output varnode. The action of the operation is determined by its **opcode**. For almost all p-code operations, only the output varnode can have its value modified; there are no indirect effects of the operation. The only possible exceptions are *pseudo* operations, see the section called "Pseudo P-CODE Operations", which are sometimes necessary when there is incomplete knowledge of an instruction's behavior.

All p-code operations are associated with the address of the original processor instruction they were translated from. For a single instruction, a 1-up counter, starting at zero, is used to enumerate the multiple p-code operations involved in its translation. The address and counter as a pair are referred to as the p-code op's unique **sequence number**. Control-flow of p-code operations generally follows sequence number order. When execution of all p-code for one instruction is completed, if the instruction has *fall-through* semantics, p-code control-flow picks up with the first p-code operation in sequence corresponding to the instruction at the fall-through address. Similarly, if a p-code operation results in a control-flow branch, the first p-code operation in sequence executes at the destination address.

The list of possible opcodes are similar to many RISC based instruction sets. The effect of each opcode is described in detail in the following sections, and a reference table is given in the section called "Syntax Reference". In general, the size or precision of a particular p-code operation is determined by the size of the varnode inputs or output, not by the opcode.

P-Code Operation Reference

For each possible p-code operation, we give a brief description and provide a table that lists the inputs that must be present and their meaning. We also list the basic syntax for denoting the operation when describing semantics in a processor specification file.

COPY

Parameters	Description
input0	Source varnode.
output	Destination varnode.

Semantic statement

```
output = input0;
```

Copy a sequence of contiguous bytes from anywhere to anywhere. Size of input0 and output must be the same.

LOAD

Parameters	Description
input0 (**special**)	Constant ID of space to load from.
input1	Varnode containing pointer offset to data.
output	Destination varnode.

Semantic statement

```
output = *input1;
output = *[input0]input1;
```

This instruction loads data from a dynamic location into the output variable by dereferencing a pointer. The “pointer” comes in two pieces. One piece, input1, is a normal variable containing the offset of the object being pointed at. The other piece, input1, is a constant indicating the space into which the offset applies. The data in input1 is interpreted as an unsigned offset and should have the same size as the space referred to by the ID, i.e. a 4-byte address space requires a 4-byte offset. The space ID is not manually entered by a user but is automatically generated by the p-code compiler. The amount of data loaded by this instruction is determined by the size of the output variable. It is easy to confuse the address space of the output and input1 variables and the Address Space represented by the ID, which could all be different. Unlike many programming models, there are multiple spaces that a “pointer” can refer to, and so an extra ID is required.

It is possible for the addressable unit of an address space to be bigger than a single byte. If the **wordsize** attribute of the space given by the ID is bigger than one, the offset into the space obtained from input1 must be multiplied by this value in order to obtain the correct byte offset into the space.

STORE

Parameters	Description
input0 (**special**)	Constant ID of space to store into.
input1	Varnode containing pointer offset of destination.
input2	Varnode containing data to be stored.

Semantic statement

```
*input1 = input2;
*[input0]input1 = input2;
```

This instruction is the complement of **LOAD**. The data in the variable input2 is stored at a dynamic location by dereferencing a pointer. As with **LOAD**, the “pointer” comes in two pieces: a space ID part, and an offset variable. The size of input1 must match the address space specified by the ID, and the amount of data stored is determined by the size of input2.

Its possible for the addressable unit of an address space to be bigger than a single byte. If the **wordsize** attribute of the space given by the ID is bigger than one, the offset into the space obtained from input1 must be multiplied by this value in order to obtain the correct byte offset into the space.

BRANCH

Parameters	Description
input0 (**special**)	Location of next instruction to execute.

Semantic statement

```
goto input0;
```

This is an absolute jump instruction. The varnode parameter input0 encodes the destination address (address space and offset) of the jump. The varnode is not treated as a variable for this instruction and does not store the destination. Its address space and offset *are* the destination. The size of input0 is irrelevant.

Confusion about the meaning of this instruction can result because of the translation from machine instructions to p-code. The destination of the jump is a *machine* address and refers to the *machine* instruction at that address. When attempting to determine which p-code instruction is executed next, the rule is: execute the first p-code instruction resulting from the translation of the machine instruction(s) at that address. The resulting p-code instruction may not be attached directly to the indicated address due to NOP instructions and delay slots.

If input0 is constant, i.e. its address space is the **constant** address space, then it encodes a *p-code relative branch*. In this case, the offset of input0 is considered a relative offset into the indexed list of p-code operations corresponding to the translation of the current machine instruction. This allows branching within the operations forming a single instruction. For example, if the **BRANCH** occurs as the pcode operation with index 5 for the instruction, it can branch to operation with index 8 by specifying a constant destination "address" of 3. Negative constants can be used for backward branches.

CBRANCH

Parameters	Description
input0 (**special**)	Location of next instruction to execute.
input1	Boolean varnode indicating whether branch is taken.

Semantic statement

```
if (input1) goto input0;
```

This is a conditional branch instruction where the dynamic condition for taking the branch is determined by the 1 byte variable input1. If this variable is non-zero, the condition is considered *true* and the branch is taken. As in the **BRANCH** instruction the parameter input0 is not treated as a variable but as an address and is interpreted in the same way. Furthermore, a constant space address is also interpreted as a relative address so that a **CBRANCH** can do *p-code relative branching*. See the discussion for the **BRANCH** operation.

BRANCHIND

Parameters	Description
input0	Varnode containing offset of next instruction.

Semantic statement

```
goto [input0];
```

This is an indirect branching instruction. The address to branch to is determined dynamically (at run-time) by examining the contents of the variable input0. As this instruction is currently defined, the variable input0 only contains the *offset* of the destination, and the address space is taken from the address associated with the branching instruction itself. So *execution can only branch within the same address space* via this instruction. The size of the variable input0 must match the size of offsets for the current address space. P-code relative branching is not possible with **BRANCHIND**.

CALL

Parameters	Description
input0 (**special**)	Location of next instruction to execute.
[input1]	First parameter to call (never present in raw p-code)
...	Additional parameters to call (never present in raw p-code)

Semantic statement

```
call [input0];
```

This instruction is semantically equivalent to the **BRANCH** instruction. **Beware:** This instruction does not behave like a typical function call. In particular, there is no internal stack in p-code for saving the return address. Use of this instruction instead of **BRANCH** is intended to provide a hint to algorithms that try to follow code flow. It indicates that the original machine instruction, of which this p-code instruction is only a part, is intended to be a function call. The p-code instruction does not implement the full semantics of the call itself; it only implements the final branch.

In the raw p-code translation process, this operation can only take input0, but in follow-on analysis, it can take arbitrary additional inputs. These represent (possibly partial) recovery of the parameters being passed to the logical *call* represented by this operation. These additional parameters have no effect on the original semantics of the raw p-code but naturally hold the varnode values flowing into the call.

CALLIND

Parameters	Description
input0	Varnode containing offset of next instruction.
[input1]	First parameter to call (never present in raw p-code)
...	Additional parameters to call (never present in raw p-code)

Semantic statement

```
call [input0];
```

This instruction is semantically equivalent to the **BRANCHIND** instruction. It does not perform a function call in the usual sense of the term. It merely indicates that the original machine instruction is intended to be an indirect call. See the discussion for the **CALL** instruction.

As with the **CALL** instruction, this operation may take additional inputs when not in raw form, representing the parameters being passed to the logical call.

RETURN

Parameters	Description
input0	Varnode containing offset of next instruction.
[input1]	Value returned from call (never present in raw p-code)

Semantic statement

```
return [input0];
```

This instruction is semantically equivalent to the **BRANCHIND** instruction. It does not perform a return from subroutine in the usual sense of the term. It merely indicates that the original machine instruction is intended to be a return from subroutine. See the discussion for the **CALL** instruction.

Similarly to **CALL** and **CALLIND**, this operation may take an additional input when not in raw form. If input1 is present it represents the value being *returned* by this operation. This is used by analysis algorithms to hold the value logically flowing back to the parent subroutine.

PIECE

Parameters	Description
input0	Varnode containing most significant data to merge.
input1	Varnode containing least significant data to merge.
output	Varnode to contain resulting concatenation.

Semantic statement

Cannot (currently) be explicitly coded

This is a concatenation operator that understands the endianess of the data. The size of input0 and input1 must add up to the size of output. The data from the inputs is concatenated in such a way that, if the inputs and output are considered integers, the first input makes up the most significant part of the output.

SUBPIECE

Parameters	Description
input0	Varnode containing source data to truncate.
input1 (**constant**)	Constant indicating how many bytes to truncate.
output	Varnode to contain result of truncation.

Semantic statement

```
output = input0(input1);
```

This is a truncation operator that understands the endianess of the data. Input1 indicates the number of least significant bytes of input0 to be thrown away. Output is then filled with any remaining bytes of input0 *up to the size of output*. If the size of output is smaller than the size of input0 plus the constant input1, then the additional most significant bytes of input0 will also be truncated.

POPCOUNT

Parameters	Description
input0	Input varnode to count.
output	Resulting integer varnode containing count.

Semantic statement

```
output = popcount(input0);
```

This is a bit count (population count) operator. Within the binary representation of the value contained in the input varnode, the number of 1 bits are counted and then returned in the output varnode. A value of 0 returns 0, a 4-byte varnode containing the value 2^{32}-1 (all bits set) returns 32, for instance. The input and output varnodes can have any size. The resulting count is zero extended into the output varnode.

INT_EQUAL

Parameters	Description
input0	First varnode to compare.
input1	Second varnode to compare.
output	Boolean varnode containing result of comparison.

Semantic statement

```
output = input0 == input1;
```

This is the integer equality operator. Output is assigned *true*, if input0 equals input1. It works for signed, unsigned, or any contiguous data where the match must be down to the bit. Both inputs must be the same size, and the output must have a size of 1.

INT_NOTEQUAL

Parameters	Description
input0	First varnode to compare.
input1	Second varnode to compare.
output	Boolean varnode containing result of comparison.

Semantic statement

```
output = input0 != input1;
```

This is the integer inequality operator. Output is assigned *true*, if input0 does not equal input1. It works for signed, unsigned, or any contiguous data where the match must be down to the bit. Both inputs must be the same size, and the output must have a size of 1.

INT_LESS

Parameters	Description
input0	First unsigned varnode to compare.
input1	Second unsigned varnode to compare.
output	Boolean varnode containing result of comparison.

Semantic statement

```
output = input0 < input1;
```

This is an unsigned integer comparison operator. If the unsigned integer input0 is strictly less than the unsigned integer input1, output is set to *true*. Both inputs must be the same size, and the output must have a size of 1.

INT_SLESS

Parameters	Description
input0	First signed varnode to compare.
input1	Second signed varnode to compare.
output	Boolean varnode containing result of comparison.

Semantic statement

```
output = input0 s< input1;
```

This is a signed integer comparison operator. If the signed integer input0 is strictly less than the signed integer input1, output is set to *true*. Both inputs must be the same size, and the output must have a size of 1.

INT_LESSEQUAL

Parameters	Description
input0	First unsigned varnode to compare.
input1	Second unsigned varnode to compare.
output	Boolean varnode containing result of comparison.

Semantic statement

```
output = input0 <= input1;
```

This is an unsigned integer comparison operator. If the unsigned integer input0 is less than or equal to the unsigned integer input1, output is set to *true*. Both inputs must be the same size, and the output must have a size of 1.

INT_SLESSEQUAL

Parameters	Description
input0	First signed varnode to compare.
input1	Second signed varnode to compare.
output	Boolean varnode containing result of comparison.

Semantic statement

```
output = input0 s<= input1;
```

This is a signed integer comparison operator. If the signed integer input0 is less than or equal to the signed integer input1, output is set to *true*. Both inputs must be the same size, and the output must have a size of 1.

INT_ZEXT

Parameters	Description
input0	Varnode to zero-extend.
output	Varnode containing zero-extended result.

Semantic statement

```
output = zext(input0);
```

Zero-extend the data in input0 and store the result in output. Copy all the data from input0 into the least significant positions of output. Fill out any remaining space in the most significant bytes of output with zero. The size of output must be strictly bigger than the size of input.

INT_SEXT

Parameters	Description
input0	Varnode to sign-extend.
output	Varnode containing sign-extended result.

Semantic statement

```
output = sext(input0);
```

Sign-extend the data in input0 and store the result in output. Copy all the data from input0 into the least significant positions of output. Fill out any remaining space in the most significant bytes of output with either zero or all ones (0xff) depending on the most significant bit of input0. The size of output must be strictly bigger than the size of input0.

INT_ADD

Parameters	Description
input0	First varnode to add.

Semantic statement

```
output = input0 + input1;
```

Parameters	Description
input1	Second varnode to add.
output	Varnode containing result of integer addition.

Semantic statement

```
output = input0 + input1;
```

This is standard integer addition. It works for either unsigned or signed interpretations of the integer encoding (twos complement). Size of both inputs and output must be the same. The addition is of course performed modulo this size. Overflow and carry conditions are calculated by other operations. See **INT_CARRY** and **INT_SCARRY**.

INT_SUB

Parameters	Description
input0	First varnode input.
input1	Varnode to subtract from first.
output	Varnode containing result of integer subtraction.

Semantic statement

```
output = input0 - input1;
```

This is standard integer subtraction. It works for either unsigned or signed interpretations of the integer encoding (twos complement). Size of both inputs and output must be the same. The subtraction is of course performed modulo this size. Overflow and borrow conditions are calculated by other operations. See **INT_SBORROW** and **INT_LESS**.

INT_CARRY

Parameters	Description
input0	First varnode to add.
input1	Second varnode to add.
output	Boolean result containing carry condition.

Semantic statement

```
output = carry(input0,input1);
```

This operation checks for unsigned addition overflow or carry conditions. If the result of adding input0 and input1 as unsigned integers overflows the size of the varnodes, output is assigned *true*. Both inputs must be the same size, and output must be size 1.

INT_SCARRY

Parameters	Description
input0	First varnode to add.

Semantic statement

```
output = scarry(input0,input1);
```

Parameters	Description
input1	Second varnode to add.
output	Boolean result containing signed overflow condition.

Semantic statement

```
output = scarry(input0,input1);
```

This operation checks for signed addition overflow or carry conditions. If the result of adding input0 and input1 as signed integers overflows the size of the varnodes, output is assigned *true*. Both inputs must be the same size, and output must be size 1.

INT_SBORROW

Parameters	Description
input0	First varnode input.
input1	Varnode to subtract from first.
output	Boolean result containing signed overflow condition.

Semantic statement

```
output = sborrow(input0,input1);
```

This operation checks for signed subtraction overflow or borrow conditions. If the result of subtracting input1 from input0 as signed integers overflows the size of the varnodes, output is assigned *true*. Both inputs must be the same size, and output must be size 1. Note that the equivalent unsigned subtraction overflow condition is **INT_LESS**.

INT_2COMP

Parameters	Description
input0	First to negate.
output	Varnode result containing twos complement.

Semantic statement

```
output = -input0;
```

This is the twos complement or arithmetic negation operation. Treating input0 as a signed integer, the result is the same integer value but with the opposite sign. This is equivalent to doing a bitwise negation of input0 and then adding one. Both input0 and output must be the same size.

INT_NEGATE

Parameters	Description
input0	Varnode to negate.

Semantic statement

```
output = ~input0;
```

Parameters	Description
output	Varnode result containing bitwise negation.

Semantic statement

```
output = ~input0;
```

This is the bitwise negation operation. Output is the result of taking every bit of input0 and flipping it. Both input0 and output must be the same size.

INT_XOR

Parameters	Description
input0	First input to exclusive-or.
input1	Second input to exclusive-or.
output	Varnode result containing exclusive-or of inputs.

Semantic statement

```
output = input0 ^ input1;
```

This operation performs a logical Exclusive-Or on the bits of input0 and input1. Both inputs and output must be the same size.

INT_AND

Parameters	Description
input0	First input to logical-and.
input1	Second input to logical-and.
output	Varnode result containing logical-and of inputs.

Semantic statement

```
output = input0 & input1;
```

This operation performs a Logical-And on the bits of input0 and input1. Both inputs and output must be the same size.

INT_OR

Parameters	Description
input0	First input to logical-or.
input1	Second input to logical-or.
output	Varnode result containing logical-or of inputs.

Semantic statement

```
output = input0 | input1;
```

This operation performs a Logical-Or on the bits of input0 and input1. Both inputs and output must be the same size.

INT_LEFT

Parameters	Description
input0	Varnode input being shifted.
input1	Varnode indicating number of bits to shift.
output	Varnode containing shifted result.

Semantic statement

```
output = input0 << input1;
```

This operation performs a left shift on input0. The value given by input1, interpreted as an unsigned integer, indicates the number of bits to shift. The vacated (least significant) bits are filled with zero. If input1 is zero, no shift is performed and input0 is copied into output. If input1 is larger than the number of bits in output, the result is zero. Both input0 and output must be the same size. Input1 can be any size.

INT_RIGHT

Parameters	Description
input0	Varnode input being shifted.
input1	Varnode indicating number of bits to shift.
output	Varnode containing shifted result.

Semantic statement

```
output = input0 >> input1;
```

This operation performs an unsigned (logical) right shift on input0. The value given by input1, interpreted as an unsigned integer, indicates the number of bits to shift. The vacated (most significant) bits are filled with zero. If input1 is zero, no shift is performed and input0 is copied into output. If input1 is larger than the number of bits in output, the result is zero. Both input0 and output must be the same size. Input1 can be any size.

INT_SRIGHT

Parameters	Description
input0	Varnode input being shifted.
input1	Varnode indicating number of bits to shift.
output	Varnode containing shifted result.

Semantic statement

```
output = input0 s>> input1;
```

This operation performs a signed (arithmetic) right shift on input0. The value given by input1, interpreted as an unsigned integer, indicates the number of bits to shift. The vacated bits are filled with the original value of the most significant (sign) bit of input0. If input1 is zero, no shift is performed and input0 is copied into output. If input1 is larger than the number of bits in output, the result is zero or all 1-bits (-1), depending on the original sign of input0. Both input0 and output must be the same size. Input1 can be any size.

INT_MULT

Parameters	Description
input0	First varnode to be multiplied.
input1	Second varnode to be multiplied.
output	Varnode containing result of multiplication.

Semantic statement

```
output = input0 * input1;
```

This is an integer multiplication operation. The result of multiplying input0 and input1, viewed as integers, is stored in output. Both inputs and output must be the same size. The multiplication is performed modulo the size, and the result is true for either a signed or unsigned interpretation of the inputs and output. To get extended precision results, the inputs must first by zero-extended or sign-extended to the desired size.

INT_DIV

Parameters	Description
input0	First varnode input.
input1	Second varnode input (divisor).
output	Varnode containing result of division.

Semantic statement

```
output = input0 / input1;
```

This is an unsigned integer division operation. Divide input0 by input1, truncating the result to the nearest integer, and store the result in output. Both inputs and output must be the same size. There is no handling of division by zero. To simulate a processor’s handling of a division-by-zero trap, other operations must be used before the **INT_DIV**.

INT_REM

Parameters	Description
input0	First varnode input.
input1	Second varnode input (divisor).
output	Varnode containing remainder of division.

Semantic statement

```
output = input0 % input1;
```

This is an unsigned integer remainder operation. The remainder of performing the unsigned integer division of input0 and input1 is put in output. Both inputs and output must be the same size. If q = input0/input1, using the **INT_DIV** operation defined above, then output satisfies the equation q*input1 + output = input0, using the **INT_MULT** and **INT_ADD** operations.

INT_SDIV

Parameters	Description
input0	First varnode input.
input1	Second varnode input (divisor).
output	Varnode containing result of signed division.

Semantic statement

```
output = input0 s/ input1;
```

This is a signed integer division operation. The resulting integer is the one closest to the rational value input0/input1 but which is still smaller in absolute value. Both inputs and output must be the same size. There is no handling of division by zero. To simulate a processor’s handling of a division-by-zero trap, other operations must be used before the **INT_SDIV**.

INT_SREM

Parameters	Description
input0	First varnode input.
input1	Second varnode input (divisor).
output	Varnode containing remainder of signed division.

Semantic statement

```
output = input0 s% input1;
```

This is a signed integer remainder operation. The remainder of performing the signed integer division of input0 and input1 is put in output. Both inputs and output must be the same size. If q = input0 s/ input1, using the **INT_SDIV** operation defined above, then output satisfies the equation q*input1 + output = input0, using the **INT_MULT** and **INT_ADD** operations.

BOOL_NEGATE

Parameters	Description
input0	Boolean varnode to negate.
output	Boolean varnode containing result of negation.

Semantic statement

```
output = !input0;
```

This is a logical negate operator, where we assume input0 and output are boolean values. It puts the logical complement of input0, treated as a single bit, into output. Both input0 and output are size 1. Boolean values are implemented with a full byte, but are still considered to only support a value of *true* or *false*.

BOOL_XOR

Parameters	Description
input0	First boolean input to exclusive-or.
input1	Second boolean input to exclusive-or.
output	Boolean varnode containing result of exclusive-or.

Semantic statement

```
output = input0 ^^ input1;
```

This is an Exclusive-Or operator, where we assume the inputs and output are boolean values. It puts the exclusive-or of input0 and input1, treated as single bits, into output. Both inputs and output are size 1. Boolean values are implemented with a full byte, but are still considered to only support a value of *true* or *false*.

BOOL_AND

Parameters	Description
input0	First boolean input to logical-and.
input1	Second boolean input to logical-and.
output	Boolean varnode containing result of logical-and.

Semantic statement

```
output = input0 && input1;
```

This is a Logical-And operator, where we assume the inputs and output are boolean values. It puts the logical-and of input0 and input1, treated as single bits, into output. Both inputs and output are size 1. Boolean values are implemented with a full byte, but are still considered to only support a value of *true* or *false*.

BOOL_OR

Parameters	Description
input0	First boolean input to logical-or.
input1	Second boolean input to logical-or.
output	Boolean varnode containing result of logical-or.

Semantic statement

```
output = input0 || input1;
```

This is a Logical-Or operator, where we assume the inputs and output are boolean values. It puts the logical-or of input0 and input1, treated as single bits, into output. Both inputs and output are size 1. Boolean values are implemented with a full byte, but are still considered to only support a value of *true* or *false*.

FLOAT_EQUAL

Parameters	Description
input0	First floating-point input to compare.
input1	Second floating-point input to compare.
output	Boolean varnode containing result of comparison.

Semantic statement

```
output = input0 f== input1;
```

This is a floating-point equality operator. Output is assigned *true*, if input0 and input1 are considered equal as floating-point values. Both inputs must be the same size, and output is size 1. If either input is **NaN**, output is set to *false*.

FLOAT_NOTEQUAL

Parameters	Description
input0	First floating-point input to compare.
input1	Second floating-point input to compare.
output	Boolean varnode containing result of comparison.

Semantic statement

```
output = input0 f!= input1;
```

This is a floating-point inequality operator. Output is assigned *true*, if input0 and input1 are not considered equal as floating-point values. Both inputs must be the same size, and output is size 1. If either input is **NaN**, output is set to *false*.

FLOAT_LESS

Parameters	Description
input0	First floating-point input to compare.
input1	Second floating-point input to compare.
output	Boolean varnode containing result of comparison.

Semantic statement

```
output = input0 f< input1;
```

This is a comparison operator, where both inputs are considered floating-point values. Output is assigned *true*, if input0 is less than input1. Both inputs must be the same size, and output is size 1. If either input is **NaN**, output is set to *false*.

FLOAT_LESSEQUAL

Parameters	Description
input0	First floating-point input to compare.
input1	Second floating-point input to compare.
output	Boolean varnode containing result of comparison.

Semantic statement

```
output = input0 f<= input1;
```

This is a comparison operator, where both inputs are considered floating-point values. Output is assigned *true*, if input0 is less than or equal to input1. Both inputs must be the same size, and output is size 1. If either input is **NaN**, output is set to *false*.

FLOAT_ADD

Parameters	Description
input0	First floating-point input to add.
input1	Second floating-point input to add.
output	Varnode containing result of addition.

Semantic statement

```
output = input0 f+ input1;
```

This is a floating-point addition operator. The result of adding input0 and input1 as floating-point values is stored in output. Both inputs and output must be the same size. If either input is **NaN**, output is set to **NaN**. If any overflow condition occurs, output is set to **NaN**.

FLOAT_SUB

Parameters	Description
input0	First floating-point input.
input1	Floating-point varnode to subtract from first.
output	Varnode containing result of subtraction.

Semantic statement

```
output = input0 f- input1;
```

This is a floating-point subtraction operator. The result of subtracting input1 from input0 as floating-point values is stored in output. Both inputs and output must be the same size. If either input is **NaN**, output is set to **NaN**. If any overflow condition occurs, output is set to **NaN**.

FLOAT_MULT

Parameters	Description
input0	First floating-point input to multiply.
input1	Second floating-point input to multiply.
output	Varnode containing result of multiplication.

Semantic statement

```
output = input0 f* input1;
```

This is a floating-point multiplication operator. The result of multiplying input0 to input1 as floating-point values is stored in output. Both inputs and output must be the same size. If either input is **NaN**, output is set to **NaN**. If any overflow condition occurs, output is set to **NaN**.

FLOAT_DIV

Parameters	Description
input0	First floating-point input.
input1	Second floating-point input (divisor).
output	Varnode containing result of division.

Semantic statement

```
output = input0 f/ input1;
```

This is a floating-point division operator. The result of dividing input1 into input0 as floating-point values is stored in output. Both inputs and output must be the same size. If either input is **NaN**, output is set to **NaN**. If any overflow condition occurs, output is set to **NaN**.

FLOAT_NEG

Parameters	Description
input0	Floating-point varnode to negate.
output	Varnode containing result of negation.

Semantic statement

```
output = f- input0;
```

This is a floating-point negation operator. The floating-point value in input0 is stored in output with the opposite sign. Both input and output must be the same size. If input is **NaN**, output is set to **NaN**.

FLOAT_ABS

Parameters	Description
input0	Floating-point input.

Semantic statement

```
output = abs(input0);
```

Parameters	Description
output	Varnode result containing absolute-value.

Semantic statement

```
output = abs(input0);
```

This is a floating-point absolute-value operator. The absolute value of input0 is stored in output. Both input0 and output must be the same size. If input0 is **NaN**, output is set to **NaN**.

FLOAT_SQRT

Parameters	Description
input0	Floating-point input.
output	Varnode result containing square root.

Semantic statement

```
output = sqrt(input0);
```

This is a floating-point square-root operator. The square root of input0 is stored in output. Both input0 and output must be the same size. If input0 is **NaN**, output is set to **NaN**.

FLOAT_CEIL

Parameters	Description
input0	Floating-point input.
output	Varnode result containing result of truncation.

Semantic statement

```
output = ceil(input0);
```

This operation rounds input0, as a signed floating-point value, towards *positive infinity*. For instance, the value 1.2 rounds to 2.0; -1.2 rounds to -1.0. The integral value obtained by rounding input0 up is stored in output, as a floating-point value. Both input0 and output must be the same size. If input0 is **NaN**, output is set to **NaN**.

FLOAT_FLOOR

Parameters	Description
input0	Floating-point input.
output	Varnode result containing result of truncation.

Semantic statement

```
output = floor(input0);
```

This operation rounds input0, as a floating-point value, towards *negative infinity*. For instance, the value 1.2 rounds to 1.0 and -1.2 rounds to -2.0. The integral value obtained by rounding input0 down is stored in output, as a floating-point value. **FLOAT_FLOOR** does *not* produce a twos complement integer output (See the **TRUNC** operator). Both input0 and output must be the same size. If input0 is **NaN**, output is set to **NaN**.

FLOAT_ROUND

Parameters	Description
input0	Floating-point input.
output	Varnode result containing result of truncation.

Semantic statement

```
output = round(input0);
```

This is a floating-point rounding operator. The integral value closest to the floating-point value in input0 is stored in output, as a floating-point value. For example, 1.2 rounds to 1.0 and 1.9 rounds to 2.0. **FLOAT_ROUND** does *not* produce a twos complement integer output (See the **TRUNC** operator). Both input0 and output must be the same size. If input0 is **NaN**, output is set to **NaN**.

FLOAT_NAN

Parameters	Description
input0	Floating-point input.
output	Boolean varnode containing result of NaN test.

Semantic statement

```
output = nan(input0);
```

This operator returns *true* in output if input0 is interpreted as **NaN**. Output must be size 1, and input0 can be any size.

INT2FLOAT

Parameters	Description
input0	Signed integer input.
output	Result containing floating-point conversion.

Semantic statement

```
output = int2float(input0);
```

This is an integer to floating-point conversion operator. Input0 viewed as a signed integer is converted to floating-point format and stored in output. Input0 and output do not need to be the same size. The conversion to floating-point may involve a loss of precision.

FLOAT2FLOAT

Parameters	Description
input0	Floating-point input varnode.
output	Result varnode containing conversion.

Semantic statement

```
output = float2float(input0);
```

This is a floating-point precision conversion operator. The floating-point value in input0 is converted to a floating-point value of a different size and stored in output. If output is smaller than input0, then the operation will lose precision. Input0 and output should be different sizes. If input0 is **NaN**, then output is set to **NaN**.

TRUNC

Parameters	Description
input0	Floating-point input varnode.
output	Resulting integer varnode containing conversion.

Semantic statement

```
output = trunc(input0);
```

This is a floating-point to integer conversion operator. The floating-point value in input0 is converted to a signed integer and stored in output using the default twos complement encoding. The fractional part of input0 is dropped in the conversion by rounding *towards zero*. Input0 and output can be different sizes.

Pseudo P-CODE Operations

Practical analysis systems need to be able to describe operations, whose exact effect on a machine's memory state is not fully modeled. P-code allows for this by defining a small set of *pseudo* operators. Such an operator is generally treated as a placeholder for some, possibly large, sequence of changes to the machine state. In terms of analysis, either the operator is just carried through as a black-box or it serves as a plug-in point for operator substitution or other specially tailored transformation. Pseudo operators may violate the requirement placed on other p-code operations that all effects must be explicit.

USERDEFINED

Parameters	Description
input0 (**special**)	Constant ID of user-defined op to perform.
input1	First parameter of user-defined op.
...	Additional parameters of user-defined op.
[output]	Optional output of user-defined op.

Semantic statement

```
userop(input1, ... );
output = userop(input1,...);
```

This is a placeholder for (a family of) user-definable p-code instructions. It allows p-code instructions to be defined with semantic actions that are not fully specified. Machine instructions that are too complicated or too esoteric to fully implement can use one or more **USERDEFINED** instructions as placeholders for their semantics.

The first input parameter input0 is a constant ID assigned by the specification to a particular semantic action. Depending on how the specification defines the action associated with the ID, the **USERDEFINED** instruction can take an arbitrary number of input parameters and optionally have an output parameter. Exact details are processor and specification dependent. Ideally, the output parameter is determined by the input parameters, and no variable is affected except the output parameter. But this is no longer a strict requirement, side-effects are possible. Analysis should generally treat these instructions as a “black-box” which still have normal data-flow and can be manipulated symbolically.

CPOOLREF

Parameters	Description
input0	Varnode containing pointer offset to object.
input1 (**special**)	Constant ID indicating type of value to return.
...	Additional parameters describing value to return.
output	Varnode to contain requested size, offset, or address.

Semantic statement

```
output = cpool(input0,intput1);
```

This operator returns specific run-time dependent values from the *constant pool*. This is a concept for object-oriented instruction sets and other managed code environments, where some details about how instructions behave can be deferred until run-time and are not directly encoded in the instruction. The **CPOOLREF** operator acts a query to the system to recover this type of information. The first parameter is a pointer to a specific object, and subsequent parameters are IDs or other special constants describing exactly what value is requested, relative to the object. The canonical example is requesting a method address given just an ID describing the method and a specific object, but **CPOOLREF** can be used as a placeholder for recovering any important value the system knows about. Details about this instruction, in terms of emulation and analysis, are necessarily architecture dependent.

NEW

Parameters	Description
input0	Varnode containing class reference
[input1]	If present, varnode containing count of objects to allocate.
output	Varnode to contain pointer to allocated memory.

Semantic statement

```
output = new(input0);
```

This operator allocates memory for an object described by the first parameter and returns a pointer to that memory. This is used to model object-oriented instruction sets where object allocation is an atomic operation. Exact details about how memory is affected by a **NEW** operation is generally not modeled in these cases, so the operator serves as a placeholder to allow analysis to proceed.

Additional P-CODE Operations

The following opcodes are not generated as part of the raw translation of a machine instruction into p-code operations, so none of them can be used in a processor specification. But, they may be introduced at a later stage by various analysis algorithms.

MULTIEQUAL

Parameters	Description
input0	Varnode to merge from first basic block.
input1	Varnode to merge from second basic block.
[...]	Varnodes to merge from additional basic blocks.
output	Merged varnode for basic block containing op.

Semantic statement

Cannot be explicitly coded.

This operation represents a copy from one or more possible locations. From the compiler theory concept of Static Single Assignment form, this is a **phi-node**. Each input corresponds to a control-flow path flowing into the basic block containing the **MULTIEQUAL**. The operator copies a particular input into the output varnode depending on what path was last executed. All inputs and outputs must be the same size.

INDIRECT

Parameters	Description
input0	Varnode on which output may depend.
input1 (**special**)	Code iop of instruction causing effect.
output	Varnode containing result of effect.

Semantic statement

Cannot be explicitly coded.

An **INDIRECT** operator copies input0 into output, but the value may be altered in an indirect way by the operation referred to by input1. The varnode input1 is not part of the machine state but is really an internal reference to a specific p-code operator that may be affecting the value of the output varnode. A special address space indicates input1's use as an internal reference encoding. An **INDIRECT** op is a placeholder for possible indirect effects (such as pointer aliasing or missing code) when data-flow algorithms do not have enough information to follow the data-flow directly. Like the **MULTIEQUAL**, this op is used for generating Static Single Assignment form.

A constant varnode (zero) for input0 is used by analysis to indicate that the output of the **INDIRECT** is produced solely by the p-code operation producing the indirect effect, and there is no possibility that the value existing prior to the operation was used or preserved.

PTRADD

Parameters	Description
input0	Varnode containing pointer to an array.

Semantic statement

Cannot be explicitly coded.

Parameters	Description
input1	Varnode containing integer index.
input2 (**constant**)	Constant varnode indicating element size.
output	Varnode result containing pointer to indexed array entry.

Semantic statement

Cannot be explicitly coded.

This operator serves as a more compact representation of the pointer calculation, input0 + input1 * input2, but also indicates explicitly that input0 is a reference to an array data-type. Input0 is a pointer to the beginning of the array, input1 is an index into the array, and input2 is a constant indicating the size of an element in the array. As an operation, **PTRADD** produces the pointer value of the element at the indicated index in the array and stores it in output.

PTRSUB

Parameters	Description
input0	Varnode containing pointer to structure.
input1	Varnode containing integer offset to a subcomponent.
output	Varnode result containing pointer to the subcomponent.

Semantic statement

Cannot be explicitly coded.

A **PTRSUB** performs the simple pointer calculation, input0 + input1, but also indicates explicitly that input0 is a reference to a structured data-type and one of its subcomponents is being accessed. Input0 is a pointer to the beginning of the structure, and input1 is a byte offset to the subcomponent. As an operation, **PTRSUB** produces a pointer to the subcomponent and stores it in output.

CAST

Parameters	Description
input0	Varnode containing value to be copied.
output	Varnode result of copy.

Semantic statement

Cannot be explicitly coded.

A **CAST** performs identically to the **COPY** operator but also indicates that there is a forced change in the data-types associated with the varnodes at this point in the code. The value input0 is strictly copied into output; it is not a conversion cast. This operator is intended specifically for when the value doesn't change but its interpretation as a data-type changes at this point.

INSERT

Parameters	Description
input0	Varnode where the value will be inserted.
input1	Integer varnode containing the value to insert.
position (**constant**)	Constant indicating the bit position to insert at.
size (**constant**)	Constant indicating the number of bits to insert.
output	Varnode result containing input0 with input1 inserted.

Semantic statement

Cannot be explicitly coded.

The values *position* and *size* must be constants. The least significant *size* bits from input1 are inserted into input0, overwriting a range of bits of the same size, but leaving any other bits in input0 unchanged. The least significant bit of the overwritten range is given by *position*, where bits in index0 are labeled from least significant to most significant, starting at 0. The value obtained after this overwriting is returned as output. Varnodes input0 and output must be the same size and are intended to be the same varnode. The value *size* must be not be bigger than the varnode input1, and *size* + *position* must not be bigger than the varnode input0.

This operation is never generated as raw p-code, even though it is equivalent to SLEIGH **bitrange** syntax such as input0[10,1] = input1.

EXTRACT

Parameters	Description
input0	Varnode to extract a value from.
position (**constant**)	Constant indicating the bit position to extract from.
size (**constant**)	Constant indicating the number of bits to extract.
output	Varnode result containing the extracted value.

Semantic statement

Cannot be explicitly coded.

The values *position* and *size* must be constants. The operation extracts *size* bits from input0 and returns it in output. The *position* indicates the least significant bit in the range being extracted, with the bits in input0 labeled from least to most significant, starting at 0. The varnodes input0 and output can be different sizes, and the extracted value is zero extended into output. The value *size* must not be bigger than the varnode output, and *size* + *position* must not be bigger than the varnode input0.

This operation is never generated as raw p-code, even though it is equivalent to SLEIGH **bitrange** syntax such as output = input0[10,1].

Syntax Reference

Name	Syntax	Description
COPY	`v0 = v1;`	Copy v1 into v0.

LOAD	`* v1`
	`*[spc]v1`
	`*:2 v1`
	`*[spc]:2 v1`
		Dereference v1 as pointer into default space. Optionally specify a space to load from and size of data in bytes.

STORE	`*v0 = v1;`
	`*[spc]v0 = v1;`
	`*:4 v0 = v1;`
	`*[spc]:4 v0 = v1;`
		Store in v1 in default space using v0 as pointer. Optionally specify space to store in and size of data in bytes.

BRANCH	`goto v0;`	Branch execution to address of v0.
CBRANCH	`if (v0) goto v1;`	Branch execution to address of v1 if v0 equals 1 (true).
BRANCHIND	`goto [v0];`	Branch execution to value in v0 viewed as an offset into the current space.
CALL	`call v0;`	Branch execution to address of v0. Hint that the branch is a subroutine call.
CALLIND	`call [v0];`	Branch execution to value in v0 viewed as an offset into the current space. Hint that the branch is a subroutine call.
RETURN	`return [v0];`	Branch execution to value in v0 viewed as an offset into the current space. Hint that the branch is a subroutine return.
PIECE	`<na>`	Concatenate two varnodes into a single varnode.
SUBPIECE	`v0:2`	The least signficant n bytes of v0.
SUBPIECE	`v0(2)`	All but the least significant n bytes of v0.
POPCOUNT	`popcount(v0)`	Count 1 bits in v0.
INT_EQUAL	`v0 == v1`	True if v0 equals v1.
INT_NOTEQUAL	`v0 != v1`	True if v0 does not equal v1.

INT_LESS	`v0 < v1`
	`v1 > v0`
		True if v0 is less than v1 as an unsigned integer.

INT_SLESS	`v0 s< v1`
	`v1 s> v0`
		True if v0 is less than v1 as a signed integer.

INT_LESSEQUAL	`v0 <= v1`	True if v0 is less than or equal to v1 as an unsigned integer.

Name	**Syntax**	**Description**
	`v1 >= v0`	
INT_SLESSEQUAL	`v0 s<= v1` `v1 s>= v0`	True if v0 is less than or equal to v1 as a signed integer.
INT_ZEXT	`zext(v0)`	Zero extension of v0.
INT_SEXT	`sext(v0)`	Sign extension of v0.
INT_ADD	`v0 + v1`	Addition of v0 and v1 as integers.
INT_SUB	`v0 - v1`	Subtraction of v1 from v0 as integers.
INT_CARRY	`carry(v0,v1)`	True if adding v0 and v1 would produce an unsigned carry.
INT_SCARRY	`scarry(v0,v1)`	True if adding v0 and v1 would produce an signed carry.
INT_SBORROW	`sborrow(v0,v1)`	True if subtracting v1 from v0 would produce a signed borrow.
INT_2COMP	`-v0`	Twos complement of v0.
INT_NEGATE	`~v0`	Bitwise negation of v0.
INT_XOR	`v0 ^ v1`	Bitwise Exclusive Or of v0 with v1.
INT_AND	`v0 & v1`	Bitwise Logical And of v0 with v1.
INT_OR	`v0 | v1`	Bitwise Logical Or of v0 with v1.
INT_LEFT	`v0 << v1`	Left shift of v0 by v1 bits.
INT_RIGHT	`v0 >> v1`	Unsigned (logical) right shift of v0 by v1 bits.
INT_SRIGHT	`v0 s>> v1`	Signed (arithmetic) right shift of v0 by v1 bits.
INT_MULT	`v0 * v1`	Integer multiplication of v0 and v1.
INT_DIV	`v0 / v1`	Unsigned division of v0 by v1.
INT_REM	`v0 % v1`	Unsigned remainder of v0 modulo v1.
INT_SDIV	`v0 s/ v1`	Signed division of v0 by v1.
INT_SREM	`v0 s% v1`	Signed remainder of v0 modulo v1.
BOOL_NEGATE	`!v0`	Negation of boolean value v0.
BOOL_XOR	`v0 ^^ v1`	Exclusive-Or of booleans v0 and v1.
BOOL_AND	`v0 && v1`	Logical-And of booleans v0 and v1.
BOOL_OR	`v0 || v1`	Logical-Or of booleans v0 and v1.
FLOAT_EQUAL	`v0 f== v1`	True if v0 equals v1 viewed as floating-point numbers.
FLOAT_NOTE-QUAL	`v0 f!= v1`	True if v0 does not equal v1 viewed as floating-point numbers.
FLOAT_LESS	`v0 f< v1` `v1 f> v0`	True if v0 is less than v1 viewed as floating-point numbers.

Name	**Syntax**	**Description**

FLOAT_LESSE-QUAL	`v0 f<= v1`
	`v1 f>= v0`
		True if v0 is less than or equal to v1 viewed as floating-point numbers.

FLOAT_ADD	`v0 f+ v1`	Addition of v0 and v1 as floating-point numbers.
FLOAT_SUB	`v0 f- v1`	Subtraction of v1 from v0 as floating-point numbers.
FLOAT_MULT	`v0 f* v1`	Multiplication of v0 and v1 as floating-point numbers.
FLOAT_DIV	`v0 f/ v1`	Division of v0 by v1 as floating-point numbers.
FLOAT_NEG	`f- v0`	Additive inverse of v0 as a floating-point number.
FLOAT_ABS	`abs(v0)`	Absolute value of v0 as a floating-point number.
FLOAT_SQRT	`sqrt(v0)`	Square root of v0 as a floating-point number.
FLOAT_CEIL	`ceil(v0)`	Nearest integral floating-point value greater than v0, viewed as a floating-point number.
FLOAT_FLOOR	`floor(v0)`	Nearest integral floating-point value less than v0, viewed as a floating-point number.
FLOAT_ROUND	`round(v0)`	Nearest integral floating-point to v0, viewed as a floating-point number.
FLOAT_NAN	`nan(v0)`	True if v0 is not a valid floating-point number (NaN).
INT2FLOAT	`int2float(v0)`	Floating-point representation of v0 viewed as an integer.
FLOAT2FLOAT	`float2float(v0)`	Copy of floating-point number v0 with more or less precision.
TRUNC	`trunc(v0)`	Signed integer obtained by truncating v0 viewed as a floating-point number.
CPOOLREF	`cpool(v0,...)`	Obtain constant pool value.

NEW	`newobject(v0)`
	`newobject(v0,v1)`
		Allocate an object or an array of objects.

MULTIEQUAL	`<na>`	Compiler phi-node: values merging from multiple control-flow paths.
INDIRECT	`<na>`	Indirect effect from input varnode to output varnode.
CAST	`<na>`	Copy from input to output. A hint that the underlying datatype has changed.
PTRADD	`<na>`	Construct a pointer to an element from a pointer to the start of an array and an index.

Name	**Syntax**	**Description**
PTRSUB	`<na>`	Construct a pointer to a field from a pointer to a structure and an offset.
INSERT	`<na>`	Insert a value as a bit-range into a varnode
EXTRACT	`<na>`	Extract a bit-range from a varnode