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. 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. 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 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 a special address space, called the **constant** address space, which is used to encode constant values in p-code operations.

P-code specifications now 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 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, integers, floating-point, and boolean. Operations that manipulate integers always interpret a varnode as a twos complement encoding using the endianess associated with the processor being modeled. Similarly, floating-point operations assume the endianess and encoding expected by the processor being modeled. 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*.

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 a size or precision of that constant. For example, the varnode (constant, 0x100, 4) is treated as the constant value 256 within an operation.

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. Only the output varnode can have its value modified by the operation, and there are never any indirect effects of the operation. The action of the operation is determined by its **opcode**. The list of possible opcodes are similar to many RISC based instruction sets. The effect of each opcode is described in detail in the next chapter, and a reference table is given at the end of this document. 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 in general.

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`
Corresponding 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`
Corresponding semantic statement	
`output = *input1;   OR   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 a typical C implementation, there are multiple spaces that a "pointer" can refer to, and so an extra ID is required.

Recent additions to p-code allow 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`
Corresponding semantic statement	
`*input1 = input2;    OR    *[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.

Recent additions to p-code allow 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`
Corresponding semantic statement	
	`goto input0;`

This is an absolute jump instruction; the destination of the jump is encoded in the instruction. The parameter input0 is not really treated as a variable for this particular instruction. It is treated as an address (address space and offset) where the instruction to be executed is located. 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 address is a machine address and, as a destination, 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.

There is now a different form of branch destination called a *p-code relative branch*. If the address space of the destination is the **constant** address space, the offset is then 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 p-code 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`
Corresponding 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`
Corresponding semantic statement	
	`goto [input0];`

This is an indirect branching instruction. The address to branch to is determined dynamically (at runtime) 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.

CALL

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

CALLIND

Parameters	**Description**
`input0`	`Varnode containing offset of next instruction`
Corresponding 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.

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`

Corresponding semantic statement

```
userop(input1,…);    OR    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 user to a particular semantic action. Depending on how the user wants to define the action associated with the ID, the **USERDEFINED** instruction can take an arbitrary number of input parameters and optionally have an output parameter. Algorithms should still treat these instructions as having normal data-flow. The output parameter is determined by the input parameters and no variable is affected except the output parameter. The instruction should be treated as a “black-box” which can still be manipulated symbolically.

RETURN

Parameters	Description
`input0`	`Varnode containing offset of next instruction`

Corresponding 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.

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`
Corresponding 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`
Corresponding 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.

INT_EQUAL

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

Corresponding 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`

Corresponding 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.`

Corresponding 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.`
Corresponding 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`
`input1`	`Second varnode to add`
`output`	`Varnode containing result of integer addition`
Corresponding 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`
Corresponding 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`
Corresponding 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`
`input1`	`Second varnode to add`
`output`	`Boolean result containing signed overflow condition`
Corresponding 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`
Corresponding 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`	`Varnode to negate.`
`output`	`Varnode result containing twos complement.`
Corresponding 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.`
`output`	`Varnode result containing bitwise negation.`
Corresponding 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.`
Corresponding 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.`

Corresponding 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.`

Corresponding 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`

Corresponding 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`

Corresponding 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`

Corresponding 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`

Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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`

Corresponding 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.

INT_SREM

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

Corresponding 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`

Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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`
`output`	`Varnode result containing absolute-value`
Corresponding semantic statement	
`output = abs(input0);`	

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

FLOAT_SQRT

Parameters	Description
`input0`	`Floating-point input`
`output`	`Varnode result containing square root`
Corresponding semantic statement	
`output = sqrt(input0);`	

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

FLOAT_CEIL

Parameters	Description
`input0`	`Floating-point input`
`output`	`Varnode result containing result of truncation`
Corresponding semantic statement	
`output = ceil(input0);`	

This is a floating-point truncation operator. The integer obtained by rounding input0 up is stored in output, as a floating-point value. Both input and output must be the same size. If input is **NaN**, output is set to **NaN**.

FLOAT_FLOOR

Parameters	Description
`input0`	`Floating-point input`
`output`	`Varnode result containing result of truncation`
Corresponding semantic statement	
`output = floor(input0);`	

This is a floating-point truncation operator. The integer obtained by rounding input0 down is stored in output, as a floating-point value. Both input and output must be the same size. If input is **NaN**, output is set to **NaN**.

FLOAT_ROUND

Parameters	Description
`input0`	`Floating-point input`
`output`	`Varnode result containing result of truncation`
Corresponding semantic statement	
`output = round(input0);`	

This is a floating-point rounding operator. The integer closest to the floating-point value in input0 is stored in output, as a floating-point value. Both input and output must be the same size. If input is **NaN**, output is set to **NaN**.

FLOAT_UNORDERED

Parameters	Description
`input0`	`First floating-point input`
`input1`	`Second floating-point input`
`output`	`Boolean result`
Corresponding semantic statement	
`output = unordered(input0);`	

This operator returns *true* in output if either input0 or input1 is interpreted as **NaN**. Both inputs must be the same size, and output must be size 1. This instruction is now deprecated.

FLOAT_NAN

Parameters	Description
`input0`	`Floating-point input`
`output`	`Boolean varnode containing result of NaN test`
Corresponding 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`
Corresponding 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`
Corresponding 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`
Corresponding 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. Input0 and output can be different sizes. The fractional part of input0 is dropped in the conversion.

Additional P-CODE Operations

The following opcodes are not generated as part of the raw translation of a machine instruction into p-code operations. 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`
Corresponding semantic statement	
	Cannot be explicitly coded

This operation represents a copy from one or more possible locations. In compiler theory parlance, this is a **phi-node**. It cannot be directly coded in a specification file, but the data-flow library uses it. So it will not turn up in the initial translation of machine instructions to p-code, but it may turn up in p-code that has been analyzed. 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
Corresponding semantic statement	
	Cannot be explicitly coded

This 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 cannot be generated by the initial translation of machine instructions but may turn up in p-code that has been analyzed.

PTRADD

Parameters	Description
input0	Varnode containing base integer offset
input1	Varnode containing integer index
input2(**constant**)	Constant varnode indicating element size
output	Varnode result containing input0 + input1 * input2
Corresponding semantic statement	
	Cannot be explicitly coded

Analysis algorithms sometimes produce this operator indirectly as a placeholder for a more complicated pointer calculation. Input0 is intended to be a pointer to the beginning of an 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.

Syntax Reference

Name	Syntax	Description
COPY	v0 = v1;	Copy v1 into v2.
LOAD	* v1 *[spc]v1 *:2 v1 *[spc]:2 v1	Dereference v1 as pointer into default space. Optionally specify space to load from and size of data in bytes.

STORE	*v0 = v1;
	*[spc]v0 = v1;
	*:4 v0 = v1;
	*[spc]:4 v0= v1;
		Store 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 v0 viewed as an offset in current space.

CALL	call v0;	Branch execution to address of v0. Hint that branch is subroutine call.
CALLIND	call [v0];	Branch execution to v0 viewed as an offset in current space. Hint that branch is subroutine call.
RETURN	return [v0];	Branch execution to v0 viewed as an offset in current space. Hint that branch is a subroutine return.
INT_EQUAL	v0 == v1	True if v0 equals v1.
INT_NOTEQUAL	v0 != v1	True if v0 does not equal v1.
INT_SLESS	v0 s< v1 v1 s> v0	True if v0 is less than v1 as a signed integer.
INT_SLESSEQUAL	v0 s<= v1 v1 s>= v0	True if v0 is less than or equal to v1 as a signed integer.
INT_LESS	v0 < v1 v1 > v0	True if v0 is less than v1 as an unsigned integer.
INT_LESSEQUAL	v0 <= v1 v1 >= v0	True if v0 is less than or equal to v1 as an unsigned 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 a 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 b1 bits.
INT_MULT	v0 * v1	Integer multiplication of v0 and v1.
INT_DIV	v0 / v1	Unsigned division of v0 by v1.
INT_SDIV	v0 s/ v1	Signed division of v0 by v1.
INT_REM	v0 % v1	Unsigned remainder of v0 modulo 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_NOTEQUAL	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.
FLOAT_LESSEQUAL	v0 f<= v1 v1 f>= v0	True if v0 is less than or equal to v1 as floating-point.
FLOAT_UNORDERED	unordered(v0,v1)	True if either v0 or v1 is not a valid number (NaN).
FLOAT_NAN	nan(v0)	True if v0 is not a valid floating-point number (NaN).
FLOAT_ADD	v0 f+ v1	Addition of v0 and v1 as floating-point numbers.
FLOAT_DIV	v0 f/ v1	Division of v0 by v1 as floating-point numbers.
FLOAT_MULT	v0 f* v1	Multiplication of v0 and v1 as floating-point numbers.
FLOAT_SUB	v0 f- v1	Subtraction of v1 from v0 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 floating point number.
FLOAT_SQRT	sqrt(v0)	Square root of v0 as floating-point number.
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.
FLOAT_CEIL	ceil(v0)	Nearest integer greater than v0.
FLOAT_FLOOR	floor(v0)	Nearest integer less than v0.
FLOAT_ROUND	round(v0)	Nearest integer to v0.
SUBPIECE	v0:2 v0(2)	The least significant n bytes of v0. All but least significant n bytes of v0.
PIECE	<na>	
MULTIEQUAL	<na>	
INDIRECT	<na>	
CAST	<na>	
PTRADD	<na>	
PTRSUB	<na>	