# *MULTIMEDIA EXTENSIONS TO TINKER*

**Version 1.0**

**Matthew D. Jennings**

**TINKER Microprocessor Laboratory**
**North Carolina State University**
**http://www.ece.ncsu.edu/tinker**

***Direct correspondence to: conte@ncsu.edu***

# INTRODUCTION

The proposed multimedia extensions to the *Tinker* architecture encompass multimedia extensions found in both the Intel MMX technology and the PA-RISC 2.0 architecture. These extensions provide data types and operations typically found on other general purpose and dsp architectures for handling 2-D & 3-D graphics, full-motion video and audio. The multimedia extensions to *Tinker* will distinguish themselves from traditional multimedia extensions by providing for compiler based speculative and predicated execution. We also intend to use these new data types and operations to increase the performance of 32-bit integer code. For 32-bit integer code it will be possible to effectively double the number of ALUs by using the new data types and operations in a 64-bit Tinker Implementation where sufficient parallelism exists in the code. This is a draft for *Tinker* multimedia extensions that will undergo further review.

## New Data Types

The new data types for the multimedia extensions are packed integer where multiple integer data quantities (byte, half-word and word) are grouped into a single 64-bit quantity. General purpose registers will be used for the packed integer data types. The packed integer data types allow for traditional Single Instruction, Multiple Data (SIMD) operations that are typical of multimedia applications that operated on 8-bit graphics pixels and 16-bit audio quantities. Describing these new operations as SIMD is confusing though on a VLIW architecture that already groups multiple *operations or Ops* into a single *multiple operation or MultiOp*. For *Tinker*, the tradition SIMD philosophy will now apply for an individual *Op*, allowing for more than one SIMD instruction per *MultiOp*.

Packed Byte (eight 8-bit quantities)

| [63:56] quantity7 | [55:48] quantity6 | [47:40] quantity5 | [39:32] quantity4 | [31:24] quantity3 | [23:16] quantity2 | [15:8] quantity1 | [7:0] quantity0 |
|---|---|---|---|---|---|---|---|

Packed Half-Word (four 16-bit quantities)

| [63:48] quantity3 | [47:32] quantity2 | [31:16] quantity1 | [15:0] quantity0 |
|---|---|---|---|

Packed Word (two 32-bit quantities)

| [63:32] quantity1 | [31:0] quantity0 |
|---|---|

Long-Word (existing data type, a 64-bit quantity)

| [63:0] quantity0 |
|---|

The GPRs used to store these data types will include the speculative tag bit, used to support *sentinel scheduling* for operations on packed integer data types. See appendix A for information on supporting speculative execution using sentinel scheduling.

Predicated execution will also be supported. Predication will be possible for each individual quantity in a packed data type. See appendix B for information on supporting predicated execution for packed data types.

## New Packed Data Operations

The new operations on the packed integer data types include arithmetic, comparison, shift, conversion and sentinel operations.

**Arithmetic operations**

The arithmetic operations include packed versions of addition, logical addition, subtraction, logical subtraction, average, logical average, multiplication and multiply-add. The addition, logical addition, subtraction and logical subtraction operations include a saturation option that is new to *Tinker*. When the saturation version of an operation is selected, if the operation results in an overflow or underflow, the result is clamped to the largest or smallest representable value, respectively. Saturation is important for graphics applications that involve shading, for example to prevent a black pixel from becoming white if an overflow occurs.

The packed average and logical average operations take two packed data types as input, adds corresponding data quantities, divides each result by two placing the result in the corresponding data location. Unbiased rounding is performed to reduce the accumulation of rounding errors.

Packed Average on Half-Word Operation

| [63:48]<br>a3 | [47:32]<br>a2 | [31:16]<br>a1 | [15:0]<br>a0 |
|---|---|---|---|
| (Pave) | (Pave) | (Pave) | (Pave) |
| b3 | b2 | b1 | b0 |
| (a3+b3)/2 | (a2+b2)/2 | (a1+b1)/2 | (a0+b0)/2 |

The packed multiplication and packed multiply-add instructions use either byte, half-word or word source operands, from a packed data type, and provide half-word, word and long-word results, respectively. The packed multiplication operation gets its source quantities from different locations depending on whether the high or low version of the instruction is used. For the low version of multiplication, every other packed quantity is used as a source starting from the packed quantity in the low-order bits. For the high version of multiplication, every other packed quantity is used as a source starting from the packed quantity in the high-order bits.

For example, the multiplication low operation for half-words uses the quantities at bit locations [47:32] and [15:0] of the source GPRs as the source operands. The multiplication high instruction uses the quantities at bit locations [63:48] and [31:16] of the source GPRs as the source operands. The corresponding results are always destined to the [63:32] or [31:0] location that overlaps the source operands.

Packed Multiply Low on Half-Word Operation

| [63:48]<br>a3 | [47:32]<br>a2 | [31:16]<br>a1 | [15:0]<br>a0 |
|---|---|---|---|
| | * | | * |
| b3 | b2 | b1 | b0 |
| (a2*b2) | | (a0*b0) | |

Packed Multiply High on Half-Word Operation

| [63:48] a3 | [47:32] a2 | [31:16] a1 | [15:0] a0 |
|---|---|---|---|
| * | | * | |
| b3 | b2 | b1 | b0 |
| (a3*b3) | | (a1*b1) | |

The packed multiply-add operation for half-words uses all 16-bit half-word source quantities as illustrated below.

Packed Multiply-Add on Half-Word Operation

| [63:48] a3 | [47:32] a2 | [31:16] a1 | [15:0] a0 |
|---|---|---|---|
| (*+) | | (*+) | |
| b3 | b2 | b1 | b0 |
| (a3*b3)+(a2*b2) | | (a1*b1)+(a0*b0) | |

**Comparison operations**

The comparison operation for the packed integer data types still includes the condition field found in the integer comparison operation. The new comparison operation compares each of the multiple data quantities of the SIMD Op in parallel. The result of the comparison is 1 for true and 0 for false. The size of the result depends on whether it is the byte, half-word, or word or long-word version of the operation.

Parallel Compare on Packed Half-Word

| 12 | 32 | 8 | 33 |
|---|---|---|---|
| == ? | == ? | == ? | == ? |
| 11 | 32 | 47 | 33 |
| …0000 | …0001 | …0000 | …0001 |

The packed compare-to-predicate operation behaves as parallel compare-to-predicate operations. Predicate registers have been expanded from a 1-bit speculative tag bit plus 1 bit (one boolean value in a 2-bit register) to a 1-bit speculative tag bit plus 8 bits (eight boolean values in a 9-bit register). The expanded

predicate registers and packed compare-to-predicate operation are used to support predication of the individual parallel operations in a packed data type operation.

To support boolean reductions using the expanded predicate register the predicate reduction operation has also been added. The predicate reduction operation can be thought of as a packed compare-to-predicate operation with packed predicate sources in place of a comparison operation on two general purpose register sources. Only the logical AND or logical OR D-action specifiers (AN, AC, ON, OC) are used for the predicate reduction operation.

See appendix B for a detailed discussion of predication support for packed data types.

**Shift operations**

The shift operations on the packed integer data types include literal shift left (or shift left with immediate), literal shift right, literal arithmetic shift left and literal arithmetic shift right. The packed shift operations each perform multiple shifts, one for each data quantity in the byte, half-word, word or long-word packed data type.

**Conversion operations**

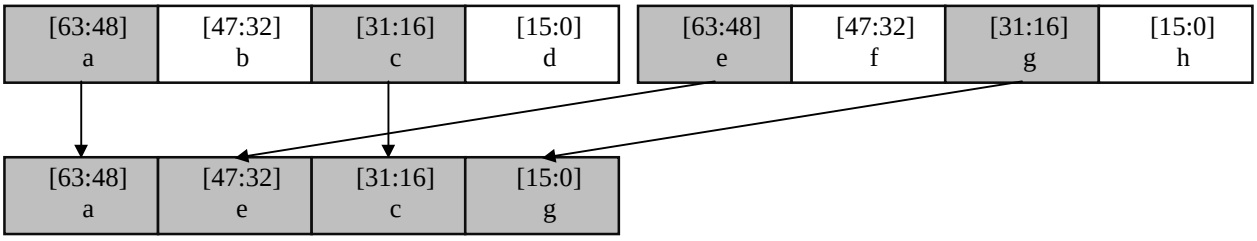
The conversion operations include pack (and unpack) instructions to convert long-word data quantities to packed word quantities (or vice-versa), and further pack (or unpack) other quantities already in packed integer data types. Saturation is employed on the pack instructions and there is a pack instruction for both logical and signed data. There is a high order and low order version of the unpack instructions for unpacking the packed data quantities in the bit ranges [63:32] and [31:0] respectively. A typical use of these instruction is when higher precision is required for intermediate operations, requiring conversion.

Pack predicate and unpack predicate operations are used for manipulating packed predicate registers. Their operation is analogous to the pack and unpack operations.

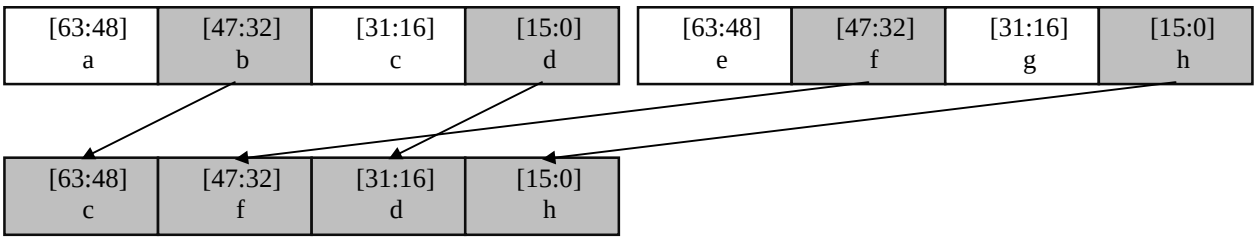
The other conversion operations are the pmix and permute operations. These instructions are used to re-arrange the order of quantities in packed data types. There is a version of pmix and permute for byte, half-word and word packed data types.

There is a high and low version of mix to distinguish which quantities are mixed together by the operation. Operation is best described through illustration. The high and low mix operation on half-word quantities is illustrated below. Operation of the byte and word version can be deduced from it.

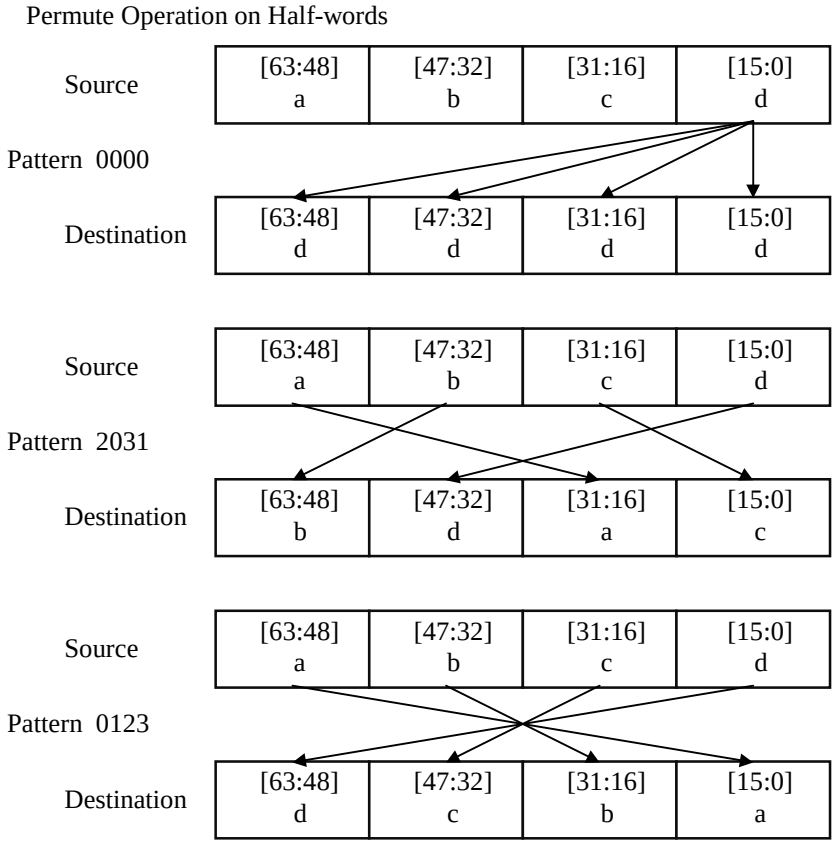
Pmix High Operation on Half-Words




Pmix Low Operation on Half-words



The permute operation takes a packed data type as a source and re-arranges the source quantities to provide for any combination of the source quantities as a result. The following figure illustrates some examples of how this works for packed half-word quantities.




**Sentinel operation**

The pcheck operation is used to provide the sentinel function when pack operations are scheduled speculatively. When pack instructions are scheduled speculatively, speculative tag bits and addresses of excepting instructions are *not* propagated through the pack instruction. This information is not propagated because the other source of the pack instruction may not be affected by an exception. Propagating the address of the exception would invalidate all of the data quantities from the pack operation, even though some of them may not be affected by an exception. Further unpacking of the register may result in the quantities affected by the exception not being used.

Pcheck is introduced to act as the sentinel for data speculatively packed, and possible unpacked, in the home block that the data is used in. Pcheck uses the source registers orphaned by the pack instruction(s) to check if a set speculative tag bit was left behind by the pack operation. Pcheck may be implemented with an

existing operation, such as moving a register to itself or moving a register to GPR0 (always zero). Therefore pcheck will not require a new opcode.

See appendix A for information on supporting speculative execution using sentinel scheduling.

The following chart summarizes the proposed multimedia extensions for *Tinker*:

| Category | Assembly Command | Description |
|---|---|---|
| Arithmetic | **PADD** | **Packed integer addition** |
| | **PADDL** | **Logical packed integer addition** |
| | **PSUB** | **Packed integer subtraction** |
| | **PSUBL** | **Logical packed integer subtraction** |
| | **PAVE** | **Packed integer average** |
| | **PAVEL** | **Logical packed integer average** |
| | **PMPY** | **Packed integer multiplication** |
| | **PMPYADD** | **Packed integer multiplication with addition of results** |
| Comparison | **PCMPR** | **Packed integer comparison using integer comparison conditions** |
| | **PCMPP** | **Packed compare-to-predicate using integer comparison conditions** |
| | **PREDUCE** | **Predicate reduction operation for boolean reduction of packed predicate inputs** |
| Shift | **PSHL** | **Logical packed shift left by amount in register or literal (immediate) value** |
| | **PSHR** | **Logical packed shift right by amount in register or literal (immediate) value** |
| | **PSHLA** | **Arithmetic packed shift left by amount in register or literal (immediate) value** |
| | **PSHRA** | **Arithmetic packed shift right by amount in register or literal (immediate) value** |
| Conversion | **PACK** | **Pack data quantities into next smallest packed data type** |
| | **PACKL** | **Pack logical data quantities into next smallest packed data type** |
| | **PUNPCK** | **Unpack data quantities to next largest packed (or non-packed) data type** |
| | **PUNPCKL** | **Unpack logical data quantities to next largest packed (or non-packed) data type** |
| | **PACKP** | **Pack boolean predicate values** |
| | **PUNPCKP** | **Unpack boolean predicate values** |
| | **PMIX** | **Mix packed data quantities** |
| | **PERM** | **Permute packed data quantities** |
| Sentinel | **PCHECK** | **Provide sentinel function for speculative execution** |

## General Comments

The assembly semantic for the operations on packed integer quantities always start with the prefix P. An option field is used for specifying the saturated or unsaturated versions of the addition and subtraction operations. The pack operations always employ saturation.

Bit-wise logical operations on the packed integer data types can be done using the existing *Tinker* operations (e.g. AND, NAND, OR, and XOR). Moving packed integer data types from GPR to GPR is also done using the integer move operation already found in *Tinker*.

# Appendix A: Sentinel Scheduling Support

The fundamental problem in supporting sentinel scheduling [1] for the speculative execution of packed data type quantities is that propagation of the address for the exception causing instruction will destroy all data quantities in the result register. This is true because the addresses themselves are 64-bit quantities, the same size as all of the packed data type registers. Fortunately all of the new operations on packed data type quantities are non-excepting. More importantly, none of the operations on packed data type quantities cause exceptions that affect only some of the quantities in a packed data type. For most operations on packed data type quantities it will be possible to propagate the address of a previously excepting speculative instruction, as all data quantities will be affected by that exception.

The unusual case is for the pack operation. The pack operation may have one source operand that is affected by an exception and one that is not. Since the pack operation is simply a data conversion operation a result in which some of the quantities are valid (not affected by an exception) and some of the quantities are invalid (affected by an exception) is required. For other operations in which one of the source operands is affected by an exception, all of the quantities of the result are also affected by the same exception, making the pack operation unique in this sense. Therefore, unless the instruction is a pack operation, it is possible to propagate the address of the exception causing instruction.

For a speculatively executed pack instruction in which one source operand is affected by an exception and the other is not it is possible that the pack operation itself should be executed but that the exception should not be handled. This is possible if, after unpacking, the speculative execution involving the quantities affected by the exception were predicted wrong. For this case we need execution to continue without exception handling for the quantities not affected by the exception of one of the source operands. For this reason it will *not be possible to propagate the address* of an exception causing instruction *for a speculatively executed pack instruction*.

## Speculative Execution of Packed Operations

The solution for the speculative execution of operations on packed data types will focus on the characteristics of the pack and unpack operations. The speculative scheduling of instruction sequences that include the pack operation may require the inclusion of the packed sentinel operation pcheck. The pcheck instruction will need to be included if the speculative scheduling of a pack instruction places it dependent on a speculatively scheduled potentially excepting instruction but before its sentinel. Assume each potentially excepting instruction and its sentinel delineate the endpoints of a restartable instruction interval as defined by Mahlke et al. [1]. Then, if the speculative scheduling of a pack instruction places it in a restartable instruction interval a pcheck instruction will be required as a sentinel for the pack instruction.

**Non-speculative packed operations**

The non-speculative execution of all of the new extensions for packed data type operations support speculative execution using sentinel scheduling as defined by Mahlke et al. [1]. When a source operand of a packed data type instruction has a set speculative tag bit exception handling is done. When no source operands have a set speculative tag bit the operation is executed. No exceptions result from the execution of any of the packed data type operations. The general description of the packed data type operations are found in the main text of this document.

**Speculative pack(l) within a restartable instruction interval**

The speculative execution of a pack(l) instruction scheduled within a restartable instruction interval has a number of distinctive features since the propagation of the address of an excepting instruction is not possible:

1. Source registers of the speculatively scheduled pack(l) instruction must be added to the live set of registers for the (each) restartable instruction interval that the pack(l) instruction is scheduled within.
2. The destination registers of a speculatively scheduled pack(l) instruction must be different from its sources registers.
3. The speculative tag bit of the destination is reset.

As it is impossible to propagate the address information for a potentially excepting instruction from the source register to the destination register of the speculative pack(l) instruction the source registers need to remain unmodified for later checking. These sources can be described as orphaned by the pack(l) instruction. The orphaned source registers still contain the information required for exception handing, the speculative tag bit and, possibly, the address of an excepting instruction. The pcheck operation will be performed non-speculatively using the orphaned sources to provide the sentinel function for the pack(l) instruction.

The orphaned sources are added to the (each) live set of registers for any restartable instruction interval that the pack(l) instruction is scheduled within. This will guarantee that the orphaned sources are left unmodified, permitting later checking just before data quantities are needed non-speculatively. Requiring that destination registers differ from their source registers is a side-effect of this requirement.

Resetting the speculative tag bit of the destination register allows more potentially excepting instructions to be speculatively scheduled after this speculatively scheduled pack operation. As the pack(l) operation does not cause an exception, the speculative tag bit of the destination will always be reset.

**Speculative pack not within a restartable instruction interval**

The speculative execution of a pack instruction *not* scheduled within a restartable instruction interval will not require a pcheck operation as it will not be possible for one of its source operands to have a set speculative tag bit. Therefore it will not be necessary to add its sources to any live sets of registers and its destination register can be the same as one of its source registers.

**Speculative packed operation other than pack(l) and pcheck**

If the speculative tag bit of one or more source registers is set, set the tag bit of the destination and propagate the address of the first source to have a set speculative tag bit. This follows the classical definition of sentinel scheduling. As none of the packed data type operations cause an exception the destination speculative tag bit will be set only if one or more of its sources has a set speculative tag bit.

**Pcheck operation**

The pcheck operation uses an orphaned source register of a pack(l) operation to perform the sentinel function for a potentially excepting instruction, scheduled prior to the pack operation, that the pack(l) operation is dependent on. The pcheck instruction is used to replace the sentinel function assigned to the potentially excepting instruction, as the sentinel information for that instruction will not be propagated. If the assigned sentinel is the sentinel portion of another operation, a pcheck will be inserted before that instruction. If the assigned sentinel is an explicit sentinel, pcheck will replace that instruction. A pcheck operation is required for each orphaned source of a speculative pack(l) instruction that is dependent on a potentially excepting instruction.

Pcheck's operation is to check an orphaned source register to a pack(l) instruction. If that source register has a set speculative tag bit, the register will also contain the address of an excepting instruction whose speculative execution was predicted correctly. Pcheck can be implemented with an existing instruction, such as move. Moving the contents of the orphaned source register to itself (or to GPR0) and using the existing sentinel function of the move operation will provide the right functionality for pcheck. Pcheck and the explicit sentinel are identical except that the explicit sentinel uses the propagated sentinel information as its

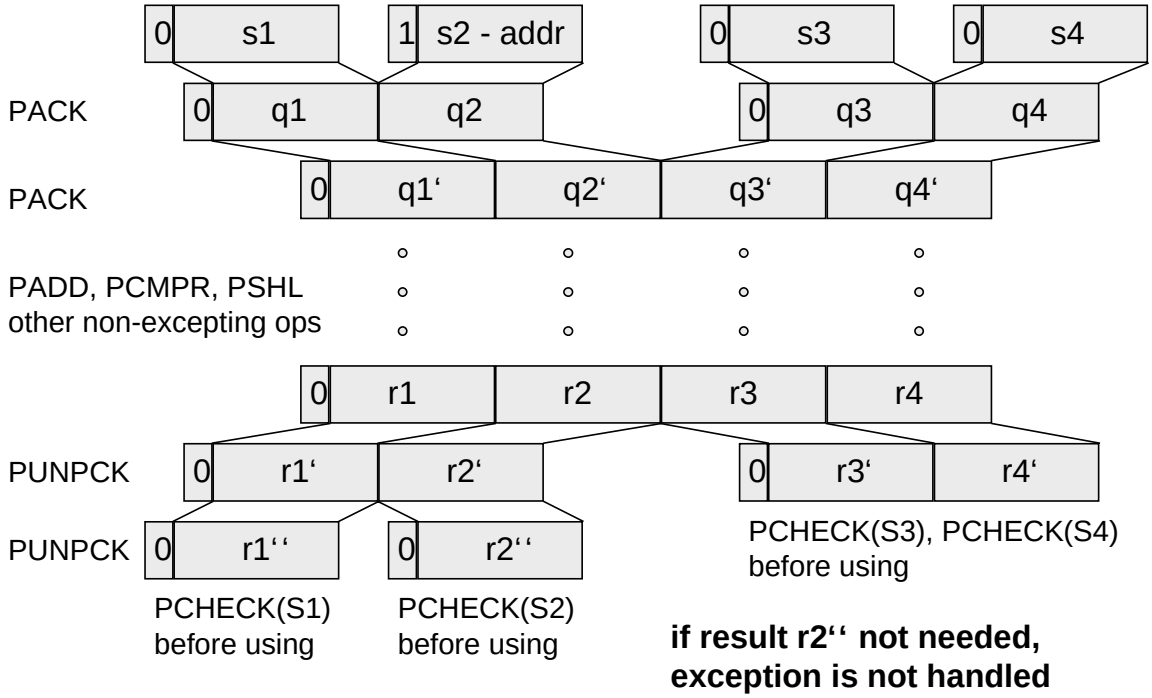
source register while pcheck uses the orphaned source register of a pack(l), containing the sentinel information, as its source.

## Examples of Speculative Packed Operations

The provided examples for speculatively executed operations on packed data types include packed operations between groups of pack and punpck instructions. Examples with pack(l) instructions provide for the interesting cases as sentinel scheduling can be supported without modification for packed operations that do not include the pack(l) instructions.

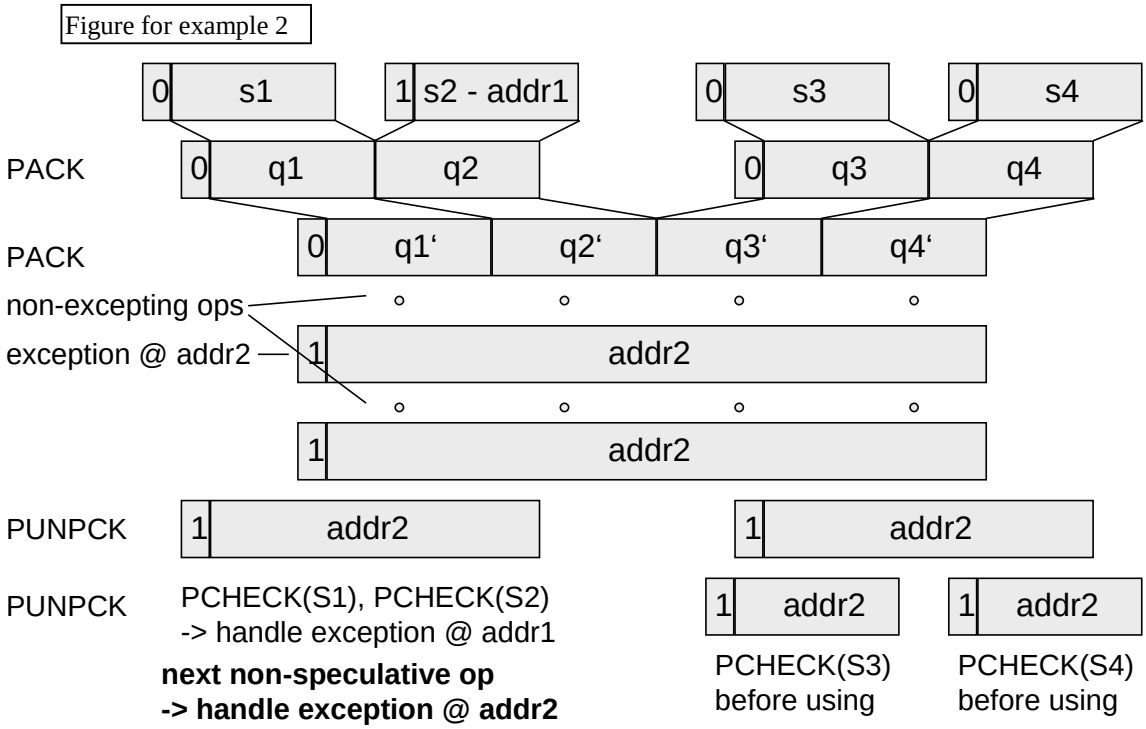
In the first example of speculatively scheduled operations on packed data types it is assumed that none of the operations between the pack and punpck instructions are potentially excepting instructions. It is also assumed that each source to the pack instructions are in a restartable instruction interval (delineated by a speculative potentially excepting instruction and it sentinel). Illustrated in the figure are source and destination registers, including the speculative tag bits, for the instructions of the example. In the first example, s2, the second source to a pack operation, has either caused an exception or is propagating the information for a previous exception. The exception occurred at PC address addr.

Figure for example 1




Since it is possible that each pcheck operation is in a different basic block it is possible for an exception to affect the result in r2‘‘ but that the exception will not be handled. This occurs when a branch is not predicted correctly and the speculative execution of the quantities associated with r2‘‘ are not needed.

In the second example potentially excepting instructions are assumed between the pack and punpck instructions. Again it is also assumed that each source to the pack instructions are in a restartable instruction interval. Again s2, the second source to a pack operation, has either caused an exception or is propagating the information for a previous exception. The exception occurred at PC address addr1. Also, now it is assumed that the potentially excepting instruction between the pack and punpck instructions causes an exception at PC address addr2.




Both exceptions will be handled in order as pcheck(s2) will be scheduled before the data quantity dependent on source s2 is used non-speculatively.

# Appendix B: Predicated Execution Support

Predicated execution will be supported for operations on packed data quantities by expanding the predicate registers to include eight predicate bits and introducing a packed compare-to-predicate operation for writing to the enlarged predicate register. The predicate registers will now be nine bits total, eight 1-bit predicates and an additional bit for the speculative tag bit. The new predicate register structure is illustrated below.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| spec. tag bit | quantity7 predicate | quantity6 predicate | quantity5 predicate | quantity4 predicate | quantity3 predicate | quantity2 predicate | quantity1 predicate | quantity0 predicate |

The same predicate registers will be used for all operations. How the predicate register is used will depend on the data type and options assumed by the operation (as explained below). Each data quantity in a packed data type will have one or more significant predicate bits.

For operations with packed byte results:
1. all fields in the predicate register will be significant to their corresponding parallel operation.

For operations with packed half-word results:
1. quantity0 and quantity1 predicates are significant to parallel operation 0.
2. quantity2 and quantity3 predicates are significant to parallel operation 1.
3. quantity4 and quantity5 predicates are significant to parallel operation 2.
4. quantity6 and quantity7 predicates are significant to parallel operation 3.

For operations with packed word results:
1. quantity0, quantity1, quantity2 and quantity3 predicates are significant to parallel operation 0.
2. quantity4, quantity5, quantity6 and quantity7 predicates are significant to parallel operation 1.

For operations with long-word results:
1. quantity0, quantity1, quantity2, quantity3, quantity4, quantity5, quantity6 and quantity7 predicates are significant to the operation.

## Predicated Packed Operations

The operation of packed instructions dependent on a predicate will be as though there were multiple operations in parallel, with each operation dependent on its own predicate value. Which data type is specified for the packed data type operation will determine how many parallel operations, and predicate values, there are. For example, operations on packed half-word quantities will behave as though there are four separate operations on half-word quantities, each with its own predicate value.

**Options for predicate values**

Predicate inputs to non-packed operations have one value, True or False. Predicate inputs to packed data type operations have the option of having a single value, the same value for each parallel operation, or multiple values, a separate value for each parallel operation. The S/M field for packed data type operations will specify whether single or multiple values will be used. If a single value is chosen, all eight predicate bits are significant to that value. If multiple values are chosen, the predicate bit(s) significant to each parallel operation are as discussed above.

Each predicate value can optionally be the logical AND or the logical OR of its significant predicate bit(s). If the result of the logical AND or logical OR, as specified by the AND/OR field, of the significant predicate bit(s) is 1 the predicate value is True. Conversely, if the logical AND or logical OR, as specified by the AND/OR field, of the significant predicate bit(s) is 0 the predicate value is False.

For each separate operation:
1. If the predicate value is True the corresponding quantity in the destination register will be modified to reflect the result of this operation.
2. If the predicate value is False the corresponding quantity in the destination register will not be modified

**Packed compare-to-predicate operation**

Packed compare-to-predicate operation will be similar to multiple instances of the non-packed compare-to-predicate operations, with the number of instances dependent on the packed data type specified in the B/H/W/X field. For example, for packed compare-to-predicate on packed byte quantities eight comparison results will be used, along with the D-action specifiers, for eight predicate results. The eight comparison results will be equivalent to the packed compare operation for the same integer comparison condition. One comparison result for each data quantity. The D-action specifiers for the packed compare-to-predicate will be the same for each of the eight comparison results. Also, the predicate input to the packed compare-to-predicate instruction will have the same options as predicate inputs to other instructions. Namely, the predicate input may have a single value for all packed quantities or multiply values, one for each packed quantity. The predicate values can optionally be the logical AND or the logical OR of the significant predicate bits, as described above.

For packed compare-to-predicate on packed byte quantities:
1. all fields in the predicate register will be set to the result of the D action on their corresponding compare result.

For packed compare-to-predicate on packed half-word quantities:
1. quantity0 and quantity1 predicates are set to the result of the D action on compare result 0.
2. quantity2 and quantity3 predicates are set to the result of the D action on compare result 1.
3. quantity4 and quantity5 predicates are set to the result of the D action on compare result 2.
4. quantity6 and quantity7 predicates are set to the result of the D action on compare result 3.

For packed compare-to-predicate on packed word quantities:
1. quantity0, quantity1, quantity2 and quantity3 predicates are set to the result of the D action on compare result 0.
2. quantity4, quantity5, quantity6 and quantity7 predicates are set to the result of the D action on compare result 1.

For packed compare-to-predicate on long-word quantities:
1. quantity0, quantity1, quantity2, quantity3, quantity4, quantity5, quantity6 and quantity7 predicates are set to the result of the D action on the compare result.

## Packed Predication's Affect on Speculative Execution

Instruction promotion (speculative execution of predicated instructions) as described in [2] can be allowed in special cases. An instruction can be promoted if its promotion does not make the instruction dependent on a potentially excepting instruction. Also, an instruction can be promoted even if its promotion makes the instruction dependent on a potentially excepting instruction, as long as each of the data quantities in the result have the same predicate value (originate from the same basic block). This restriction is required as propagating the PC address of an exception using sentinel scheduling, as described in appendix A, requires all 64-bits of the register. All data quantities of a result would therefore be invalidated by the propagation of an exception's PC address. This can only be allowed if all data quantities would have been affected by the same exception. If each of the data quantities in the result have the same predicate value (originate from the same basic block) they all should be affected by an exception.

The case of a packed operation with differing predicate values for some of its data quantities could be the result of active compiler grouping of operations on smaller data quantities into a single packed operation. If

the smaller data quantities originate from different basic blocks their corresponding predicate values could be different. For this special case, speculative execution can not be dependent on a potentially excepting instruction as it is now possible that some data quantities should be affected by the exception while others should not be affected by the exception.

# Packed Operation Example: String Copy

This string copy example for illustrating the usefulness of packed operations is derived from the paper on *Parallelization of Control Recurrences for ILP Processors* by M. Schlansker et. al. [3] . The example for illustrating control height reduction in the paper is string copy with count, the C code for which follows:

```
Count = -1 ;
do {
        *q++ = *p ;
        count++ ;
} while (*p++)
```

This string copy example for illustrating the usefulness of packed operations uses the intermediate code for this C code after a control height reduction transformation and optimizations have been applied. A comparison of this intermediate code (after transformation and optimization) will be made with intermediate code that assumes the availability of packed operations as described in the document *Proposed Multimedia Extensions to Tinker*. Also, implementation issues in Tinker assembly are discussed for this example.

**Intermediate code for string copy after transformation and optimization**

The intermediate code from [3] after a control height reduction and optimizations for the string copy with count is provide below: (Note: this intermediate code differs slightly from [3] due to corrections. The count register has been renamed to vr13 and the predicate is re-initialized at the beginning of the exit code.)

```
        vr1 = p
        vr9 = q
        vr13 = -1
        W = TRUE
loop:   vr2 = vr1 + 1    ;  vr3 = vr1 + 2    ;  vr4 = vr1 + 3
        vr5 = load vr1   ;  vr6 = load vr2   ;
        vr7 = load vr3   ;  vr8 = load vr4
        W = AND( W, (vr5 != 0), (vr6 != 0), (vr7 != 0), (vr8 != 0) )
        vr10 = vr9 + 1   ;  vr11 = vr9 + 2   ;  vr12 = vr9 + 3
        store vr9,   vr5              if W
        store vr10,  vr6              if W
        store vr11,  vr7              if W
        store vr12,  vr8              if W
        vr1  = vr1  + 4               if W
        vr9  = vr9  + 4               if W
        vr13 = vr13 + 4               if W
        if W go to loop
exit:   W = TRUE
        store vr9,   vr5              if W
        count = vr13 + 1              if W
        W = AND( W, (vr5 != 0) )
        store vr10,  vr6              if W
        count = vr13 + 2              if W
        W = AND( W, (vr6 != 0) )
        store vr11,  vr7              if W
        count = vr13 + 3              if W
        W = AND( W, (vr7 != 0) )
        store vr12,  vr8              if W
        count = vr13 + 4              if W
```

**Intermediate code assuming existence of packed operations**

The intermediate code for the string copy with count after a control height reduction and optimizations has been rewritten with the assumption that packed operations exist. Packed byte data types will be used in the loop code as each of the ascii characters are represented as eight bit quantities. In Tinker a packed byte data type represents eight byte quantities as Tinker is a 64-bit architecture. During the iterations of the loop **the intermediate code assuming packed byte quantities will perform twice as much work, with fewer instructions**, since the intermediate code from the paper [3] performs four character copies per loop iteration. The following intermediate code is for string copy with count assuming the existence of packed operations:

```
        vr1 = p
        vr9 = q
        vr13 = -1
        W = TRUE
  loop: vr5 = load vr1                                  /* load long-word of packed byte quantities */
        W = AND( W, (byte1 != 0), (byte2 != 0),  … , (byte8 != 0) )
        store vr9,    vr5             if W            /* long-word store of packed byte quantities */
        vr1   = vr1   + 8             if W
        vr9   = vr9   + 8             if W
        vr13 = vr13 + 8               if W
        if W go to loop
exit:   Wa = (byte1 != 0)
        Wb = AND( (byte1 != 0), (byte2 != 0) )
        Wc = AND( (byte1 != 0), (byte2 != 0), (byte3 != 0) )
        Wd = AND( (byte1 != 0), (byte2 != 0),  … , (byte4 != 0) )
        We = AND( (byte1 != 0), (byte2 != 0),  … , (byte5 != 0) )
        Wf = AND( (byte1 != 0), (byte2 != 0),  … , (byte6 != 0) )
        Wg = AND( (byte1 != 0), (byte2 != 0),  … , (byte7 != 0) )
        store vr9,    vr5
        vr13 = vr13 + 1
        store vr9 + 1,    vr5 + 1     if Wa           /* conditional byte store of char through end-of-str */
        vr13 = vr13 + 1               if Wa
        store vr9 + 2,    vr5 + 2     if Wb           /* conditional byte store of char through end-of-str */
        vr13 = vr13 + 1               if Wb
        store vr9 + 3,    vr5 + 3     if Wc           /* conditional byte store of char through end-of-str */
        vr13 = vr13 + 1               if Wc
        store vr9 + 4,    vr5 + 4     if Wd           /* conditional byte store of char through end-of-str */
        vr13 = vr13 + 1               if Wd
        store vr9 + 5,    vr5 + 5     if We           /* conditional byte store of char through end-of-str */
        vr13 = vr13 + 1               if We
        store vr9 + 6,    vr5 + 6     if Wf           /* conditional byte store of char through end-of-str */
        vr13 = vr13 + 1               if Wf
        store vr9 + 7,    vr5 + 7     if Wg           /* conditional byte store of char through end-of-str */
        vr13 = vr13 + 1               if Wg
        count = vr13
```

While the exit intermediate code has a comparable number of instructions as the exit code the paper [3] would have had if it also had been written for eight character copies per loop iteration there is a significant amount of parallelism in this exit code. An *extended* Tinker assembly implementation, as discussed below, will be able to exploit this parallelism.

**Tinker Assembly implementation issues**

Implementing the intermediate code that assumes packed operations into *extended* Tinker assembly is mostly straightforward. Byte or long-word versions of the load and store operations are specified by choosing the appropriate B/H/W/X field for the byte, half-word, word or long-word options for these operations. The loop code uses the long-word versions of load and store (L.X and S.X respectively) while the exit code uses the byte version of the store operation (S.B).

The loop code functional statement

W = AND( W, (byte1 != 0), (byte2 != 0), … , (byte8 != 0) )

can be implemented using the packed compare-to-predicate operation with the previous W value as the predicate input. The predicate input W will be True if all of the packed predicate bits are one and False otherwise (if one or more packed predicate bits equal 0). The unconditional D action specifier will be used so that the result, W, is zeros or False if the predicate input is False. Thus the intermediate code statement

W = AND( W, (byte1 != 0), (byte2 != 0), … , (byte8 != 0) )

can be replaced with the Tinker assembly statement

%PRn %PR0 = PCMPP.B.!=.UN.UC(vr5, vrPEOS, %PRn)

where the predicate input and 1st destination are the same, the 2nd destination is not used (%PR0 always False), B specifies packed byte version, UN and UC specify unconditional D action specifiers, and the vrPEOS source contains packed end-of-string characters.

In the exit code predicates Wa, Wb, … , Wg will be calculated using the unpack predicates (PUNPCKP) command and the predicate reduction (PREDUCE) command. (This use of the predicate reduction command will also give an indication of how packed predicates can be used as a powerful boolean reduction tool.)

Upon entering the exit code the packed predicate W, generated in the loop code, contains all of the information needed to calculate the predicates Wa, Wb, … , Wg, specifically, which character location contains the first end-of-string character. Unpacking of the predicates in W, followed by boolean reductions, will be used to calculate the correct dependencies of Wa, Wb, … , Wg on the first end-of-string character. This can best be illustrated using Tinker assembly instructions with discussion following. In the following Tinker assembly W$_*$'s are aliases for predicate registers, with the subscripts indicating which predicates remain after unpacking.

```
Wabcd, Wefgh = PUNPCKP (W)
Wab, Wcd = PUNPCKP (Wabcd)  ;  Wef, Wgh = PUNPCKP (Wefgh)
Wa, Wb = PUNPCKP (Wab)  ;  Wc, Wd = PUNPCKP (Wcd)
We, Wf = PUNPCKP (Wef)  ;  Wg, Wh = PUNPCKP (Wgh)

Wb = PREDUCE.AN (Wab.AND, %PR1)       /*  operation not required, Wab.AND can be used as is */
Wc = PREDUCE.AN (Wab.AND, Wc.AND)
Wd = PREDUCE.AN (Wabcd.AND, %PR1)     /*  operation not required, Wabcd.AND can be used as is */
We = PREDUCE.AN (Wabcd.AND, We.AND)
Wf = PREDUCE.AN (Wabcd.AND, Wef.AND)
Wg = PREDUCE.AN (Wf.AND, Wg.AND)
```

PR1 is always True and can be used as a source to the predicate reduction operation. The assembly instructions that do use %PR1 as an input though are not required in this example. This is because the W$_*$.AND option can be used in place of these results for the intermediate code of this example. Note that using the logical AND of the intermediate predicate unpacking results as sources to the predicate reduction operations is simpler than using the completely unpacked results. The other important assembly characteristic to note for this code is that there are few data dependencies. The predicate unpacking instructions are dependent on each other but in a tree fashion, revealing parallelism. Also, the predicate reduction of Wg is dependent on the predicate reduction of Wf.

All of the conditional statements in the intermediate code are non-packed operations; their packed predicate inputs will be True if all of their packed predicate bits are one and False otherwise (if one or more packed

predicate bits equal 0). This will be accomplished by choosing the AND option for the packed predicate inputs to these conditional operations.

# References


1. S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, *Sentinel Scheduling: A Model for Compiler-Controlled Speculative Execution*, ACM Transactions on Computer Systems, Vol. 11, No. 4, November 1993, pp. 377-408
2. S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, *Effective Compiler Support for Predicated Execution Using the Hyperblock,*
3. M. Schlansker, V. Kathail, S. Anik, *Parallelization of Control Recurrences for ILP Processors*

PADD OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 13 | 1 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RSRVD rv | S/ N | DESTINATION t | rv | PREDICATE p |

Format: t = PADD.B/H/W/X.S/N (s1, s2) if p;

Purpose: To do integer addition on packed integer data types.

Description: The two sources are added and the result is placed in the destination register t. Source1 and source2 can be a GPR or a CR which is specified by the 8-bit source register fields s1 and s2. The destination register can be a GPR or CR specified by the 8-bit destination register field t. This operation is valid for byte, half-word, word or long-word as specified by field B/H/W/X. The S/N field is used to specify saturating or not saturating operation.

Operation:
```
if PR(p)
        t =  s1 + s2;
```

PADDL OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 13 | 1 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RSRVD rv | S/ N | DESTINATION t | rv | PREDICATE p |

Format: t = PADDL.B/H/W/X.S/N (s1, s2) if p;

Purpose: To do logical integer addition on packed integer data types.

Description: The two sources are added and the result is placed in the destination register t. Source1 and source2 can be a GPR or a CR which is specified by the 8-bit source register fields s1 and s2. The destination register can be a GPR or CR specified by the 8-bit destination register field t. This operation is valid for byte, half-word, word or long-word as specified by field B/H/W/X. The S/N field is used to specify saturating or not saturating operation.

Operation:

```
if PR(p)
        t =  s1 + s2;
```

PSUB OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 13 | 1 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RSRVD rv | S/ N | DESTINATION t | rv | PREDICATE p |

Format: t = PSUB.B/H/W/X.S/N (s1, s2) if p;

Purpose: To do integer subtraction with carry on packed integer data types.

Description: The source s2 is subtracted from the source s1 and the result is placed in the destination register t. Source1 and source2 can be a GPR or a CR which is specified by the 8-bit source register fields s1 and s2. The destination register can be a GPR or CR specified by the 8-bit destination register field t. This operation is valid for byte, half-word, word or long-word as specified by field B/H/W/X. The S/N field is used to specify saturating or not saturating operation.

Operation:
```
if PR(p)
        t =  s1 - s2;
```

PSUBL OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 13 | 1 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE<br>0 | OPCODE | SOURCE<br>s1 | SOURCE<br>s2 | BH<br>WX | RSRVD<br>rv | S/<br>N | DESTINATION<br>t | rv | PREDICATE<br>p |

Format: t = PSUBL.B/H/W/X.S/N(s1, s2) if p;

Purpose: To do logical integer subtraction on packed integer data types.

Description: The source s2 is subtracted from the source s1 and the result is placed in the destination register t. Source1 and source2 can be a GPR or a CR which is specified by the 8-bit source register fields s1 and s2. The destination register can be a GPR or CR specified by the 8-bit destination register field t. This operation is valid for byte, half-word, word or long-word as specified by field B/H/W/X. The S/N field is used to specify saturating or not saturating operation.

Operation:

```
if PR(p)
        t =  s1 - s2;
```

PAVE OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 14 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE<br>0 | OPCODE | SOURCE<br>s1 | SOURCE<br>s2 | BH<br>WX | RSRVD<br>rv | DESTINATION<br>t | rv | PREDICATE<br>p |

Format: t = PAVE.B/H/W/X (s1, s2) if p;

Purpose: To do integer average with unbiased rounding on packed integer data types.

Description: The two sources are averaged (added together and then divided by 2) and the result is placed in the destination register t. Source1 and source2 can be a GPR or a CR which is specified by the 8-bit source register fields s1 and s2. The destination register can be a GPR or CR specified by the 8-bit destination register field t. This operation is valid for byte, half-word, word or long-word as specified by field B/H/W/X.

Operation:

```
if PR(p)
        t =  (s1 + s2) / 2;
```

PAVEL OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 14 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE<br>0 | OPCODE | SOURCE<br>s1 | SOURCE<br>s2 | BH<br>WX | RSRVD<br>rv | DESTINATION<br>t | rv | PREDICATE<br>p |

Format: t = PAVEL.B/H/W/X (s1, s2) if p;

Purpose: To do logical integer average with unbiased rounding on packed integer data types.

Description: The two sources are averaged (added together and then divided by 2) and the result is placed in the destination register t. Source1 and source2 can be a GPR or a CR which is specified by the 8-bit source register fields s1 and s2. The destination register can be a GPR or CR specified by the 8-bit destination register field t. This operation is valid for byte, half-word, word or long-word as specified by field B/H/W/X.

Operation:

```
if PR(p)
        t =  (s1 + s2) / 2;
```

PMPY OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 13 | 1 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RSRVD rv | HI L0 | DESTINATION t | rv | PREDICATE p |

Format: t = PMPY.B/H/W.HI/LO (s1, s2) if p;

Purpose: To do integer multiplication on packed integer quantities.

Description: The source s1 is multiplied by source s2 and the result is placed in the destination register t. Source1 and source2 can be a GPR or a CR which is specified by the 8-bit source register fields s1 and s2. The destination register can be a GPR or CR specified by the 8-bit destination register field t. The HI/LO field is used to specify which packed 16-bit quantities are used as sources. This operation is valid for byte, half-word or word sources as specified by field B/H/W/X. The larger results are half-word, word and long-word respectively.

Operation:
```
if PR(p)
        t =  s1 * s2;
```

PMPYADD OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 14 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RSRVD rv | DESTINATION t | rv | PREDICATE p |

Format: t = PMPYADD.B/H/W (s1, s2) if p;

Purpose: To do integer multiplication followed by addition on packed integer quantities

Description: The source s1 is multiplied by source s2 and the result is placed in the destination register t. Source1 and source2 can be a GPR or a CR which is specified by the 8-bit source register fields s1 and s2. The destination register can be a GPR or CR specified by the 8-bit destination register field t. This operation is valid for byte, half-word or word sources as specified by field B/H/W/X. The larger results are half-word, word and long-word respectively.

Operation:
```
if PR(p)
        t = s1 (*+) s2;
```

PCMPR OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 4 | 10 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | COND | RESERVED rv | DEST t | rv | PREDICATE p |

Format: t = PCMPR.B/H/W/X.COND (s1, s2) if p;

Purpose: To compare two integer values for packed integer data types.

Description: The two sources specified by fields, s1and s2, are arithmetically compared. The comparison condition is specified by the field, cond. The result of the comparison is stored in the GPR or CR specified by the field destination t. Source1 and source2 can be a GPR or a CR which is specified by the 8-bit source register fields s1 and s2. This operation is valid for byte, half-word, word or long-word operands as specified by field B/H/W/X.

Operation:

```
if PR(p)
        t = s1 cond s2;
```

PSHL OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 14 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RESERVED rv | DESTINATION t | L1 | PREDICATE p |

LITERAL PSHL OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 8 | 6 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | LITERAL s1 [0:7] | SOURCE s2 | BH WX | LIT [8:15] | RSV rv | DESTINATION t | L1 | PREDICATE p |

Format: t = PSHL.B/H/W/X (s1, s2) if p;

Purpose: To left shift s1 by s2 for packed integer data types.

Description: Each packed integer data quantity in source s1 is shifted left by s2 and the result is placed in the destination register t. Source2 can be a GPR or a CR which is specified by the 8-bit source register field s2. If field L1 is 1 then source1 is a 16-bit literal which is encoded using the second (literal) encoding. Otherwise source1 can be a GPR or a CR which is specified by the 8-bit source register field s1. The destination register can be a GPR or CR specified by the 8-bit destination register field t. This operation is valid for byte, half-word, word or long-word as specified by field B/H/W/X.

Operation:
```
if PR(p)
        t =  s1  << s2;
```

PSHR OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 14 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RESERVED rv | DESTINATION t | L1 | PREDICATE p |

LITERAL PSHR OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 8 | 6 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | LITERAL s1 [0:7] | SOURCE s2 | BH WX | LIT [8:15] | RSV rv | DESTINATION t | L1 | PREDICATE p |

Format: t = PSHR.B/H/W/X (s1, s2) if p;

Purpose: To right shift s1 by s2 for packed integer data types.

Description: Each packed integer data quantity in source s1 is shifted right by s2 and the result is placed in the destination register t. Source2 can be a GPR or a CR which is specified by the 8-bit source register field s2. If field L1 is 1 then source1 is a 16-bit literal which is encoded using the second (literal) encoding. Otherwise source1 can be a GPR or a CR which is specified by the 8-bit source register field s1. The destination register can be a GPR or CR specified by the 8-bit destination register field t. This operation is valid for byte, half-word, word or long-word as specified by field B/H/W/X.

Operation:
```
if PR(p)
        t =  s1  >> s2;
```

PSHLA OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 14 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RESERVED rv | DESTINATION t | L1 | PREDICATE p |

LITERAL PSHLA OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 8 | 6 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | LITERAL s1 [0:7] | SOURCE s2 | BH WX | LIT [8:15] | RSV rv | DESTINATION t | L1 | PREDICATE p |

Format: t = PSHLA.B/H/W/X (s1, s2) if p;

Purpose: To left shift s1 by arithmetic word s2 for packed integer data types.

Description: Each packed integer data quantity in source s1 is shifted left by arithmetic word s2 and the result is placed in the destination register t. Source2 can be a GPR or a CR which is specified by the 8-bit source register field s2. If field L1 is 1 then source1 is a 16-bit literal which is encoded using the second (literal) encoding. Otherwise source1 can be a GPR or a CR which is specified by the 8-bit source register field s1. The destination register can be a GPR or CR specified by the 8-bit destination register field t. This operation is valid for byte, half-word, word or long-word as specified by field B/H/W/X.

Operation:

```
if PR(p)
        t =  pshla(s1, s2);
```

PSHRA OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 14 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RESERVED rv | DESTINATION t | L1 | PREDICATE p |

LITERAL PSHRA OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 8 | 6 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | LITERAL s1 [0:7] | SOURCE s2 | BH WX | LIT [8:15] | RSV rv | DESTINATION t | L1 | PREDICATE p |

Format: t = PSHRA.B/H/W/X (s1, s2) if p;

Purpose: To right shift s1 by arithmetic word s2 for packed integer data types.

Description: Each packed integer data quantity in source s1 is shifted right by arithmetic word s2 and the result is placed in the destination register t. Source2 can be a GPR or a CR which is specified by the 8-bit source register field s2. If field L1 is 1 then source1 is a 16-bit literal which is encoded using the second (literal) encoding. Otherwise source1 can be a GPR or a CR which is specified by the 8-bit source register field s1. The destination register can be a GPR or CR specified by the 8-bit destination register field t. This operation is valid for byte, half-word, word or long-word as specified by field B/H/W/X.

Operation:
```
if PR(p)
        t =  pshra(s1, s2);
```

PACK OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 14 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RESERVED rv | DESTINATION t | rv | PREDICATE p |

Format: t = PACK.H/W/X (s1, s2) if p;

Purpose: To pack the contents of s1 and s2 into a packed word data type destination t. Or to further pack packed data quantities in s1 and s2 into the next smallest packed data type destination t.

Description: The contents of GPRs or CRs as specified by the source fields s1 and s2 are packed into the GPR or CR as specified by the field destination t. The field B/H/W/X specifies the size of the source operands: long-word for X, packed integer word for W, or packed integer half-word for H. The destination is a packed integer data type of next size smaller than the source operands.

Operation:
```
if PR(p)
        t = pack (s1, s2);
```

PACKL OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 14 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RESERVED rv | DESTINATION t | rv | PREDICATE p |

Format: t = PACKL.H/W/X (s1, s2) if p;

Purpose: To pack the logical contents of s1 and s2 into a packed word data type destination t. Or to further pack packed data quantities in s1 and s2 into the next smallest packed data type destination t.

Description: The contents of GPRs or CRs as specified by the source fields s1 and s2 are packed into the GPR or CR as specified by the field destination t The field B/H/W/X specifies the type of the source operands: long-word for X, packed integer word for W, or packed integer half-word for H. The destination is a packed integer data type of next size smaller than the source operands.

Operation:

```
if PR(p)
        t = pack_logical (s1, s2);
```

PUNPCK OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 13 | 1 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | RSRVED rv | BH WX | RSRVED rv | HI L0 | DESTINATION t | rv | PREDICATE p |

Format: t = PUNPCK.B/H/W.HI/LO (s1) if p;

Purpose: To unpack the high or low packed data quantities of source s1 and place in destination t.

Description: High or low data quantities of GPR or CR specified by the source field s1 are unpacked and placed into the GPR or CR as specified by the field destination t The field B/H/W/X specifies the type of the source operand: packed integer word for W, packed integer half-word for H, or packed integer byte for B. The destination is a packed integer data type of next size larger than the source operands. The HI/LO field specifies the high-order or low-order version of the unpack instruction for unpacking the packed data quantities in the bit ranges [63:32] and [31:0] respectively.

Operation:
```
if PR(p)
        t = punpck (s1);
```

PUNPCKL OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 13 | 1 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | RSRVED rv | BH WX | RSRVED rv | HI L0 | DESTINATION t | rv | PREDICATE p |

Format: t = PUNPCKL.B/H/W.HI/LO (s1) if p;

Purpose: To unpack the logical high or low packed data quantities of source s1 and place in destination t.

Description: High or low data quantities of GPR or CR specified by the source field s1 are unpacked and placed into the GPR or CR as specified by the field destination t The field B/H/W/X specifies the type of the source operand: packed integer word for W, packed integer half-word for H, or packed integer byte for B. The destination is a packed integer data type of next size larger than the source operands. The HI/LO field specifies the high-order or low-order version of the unpack instruction for unpacking the packed data quantities in the bit ranges [63:32] and [31:0] respectively.

Operation:
```
if PR(p)
        t = punpck_logical (s1);
```

PMIX OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 13 | 1 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | SOURCE s2 | BH WX | RSRVED rv | HI L0 | DESTINATION t | rv | PREDICATE p |

Format: t = PMIX.B/H/W.HI/LO (s1, s2) if p;

Purpose: To mix the data quantities of packed integer sources s1 and s2 and place in destination t.

Description: Every other quantity of the GPRs or CRs specified by the source fields s1 and s2 are mixed together and placed into the GPR or CR as specified by the field destination t. The field B/H/W/X specifies the type of the source and destination operands: packed integer word for W, packed integer half-word for H, or packed integer byte for B. The HI/LO field specifies the high-order or low-order version of the mix instruction. The high-order version of the instruction mixes every other quantity starting with the quantities in the high-order bits. The low-order version of the instruction mixes every other quantity starting with the quantities in the low-order bits.

Operation:
```
if PR(p)
        t = mix (s1, s2);
```

PERM OPERATION

| 1 | 1 | 1 | 5 | 2 | 6 | 8 | 8 | 2 | 14 | 8 | 1 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | R | SP | PAUSE | OPTYPE 0 | OPCODE | SOURCE s1 | RSRVED rv | BH WX | RESERVED not enough, need 24 have | DESTINATION t | rv | PREDICATE p |

Format: t = PERM.B/H/W (s1) if p;

Purpose: To select any combination of data quantities from source s1 and place into destination t.

Description: Some combination of data quantities from GPR or CR specified by the source field s1 placed into the GPR or CR as specified by the field destination t. The field B/H/W/X specifies the type of the source and destination operands: packed integer word for W, packed integer half-word for H, or packed integer byte for B. Fields p0, p1, … , p7 specify which combination is placed in the destination t. For packed byte p0-p7 are used, for packed half-word p0-p3 are used and for packed word p0 and p1 are used. Pn specifies which quantity from source s1 goes into the quantity location n of destination t. For packed byte, pn is a number from 0-7. For packed half-word pn is a number from 0-3. For packed word pn is a either 0 or 1. Pn specifies source quantity location and n specifies destination quantity location. Quantity 0 is defined to be in the low order bits of both the source and destination. ** Note are not enough reserve bits to specify p0-p7 for PERM.B **

Operation:

```
if PR(p)
        t = perm (s1);
```