OVP VMI Morph Time Function Reference

Imperas Software Limited
Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com

Author:	Imperas Software Limited
Version:	6.13.0
Filename:	OVP_VMI_Morph_Time_Function_Reference.doc
Project:	OVP VMI Morph Time Function Reference
Last Saved:	Monday, 15 June 2015
Keywords:	

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

1 Introduction

This is reference documentation for **version 6.13.0** of the VMI *morph time* function interface, defined in `ImpPublic/include/host/vmi/vmiMt.h`.

It also gives details of the VMI *instruction fetch* interface, defined in `ImpPublic/include/host/vmi/vmiCxt.h`, and *instruction decoder* function interface which greatly simplifies the creation of robust and correct instruction decoders. This interface is defined in `ImpPublic/include/host/vmi/vmiDecode.h`.

The functions in the VMI morph time function interface are used to define instruction behavior of a simulated processor, and are callable only within or beneath the processor *morph callback* function (defined with the `VMI_MORPH_FN` macro, installed as the `morphCB` field of the processor `vmiIASAttr` structure).

The morph callback performs the following actions:

1. It fetches an instruction at a simulated address supplied as an argument.
2. It decodes the instruction (for example, by a cascaded `if` driven by bit fields extracted from the fetched instruction, or by using the decoder function interface);
3. It calls one or more of the routines specified here to describe the behavior of the instruction.

Functions in section 3 of this document show how the fetch and decode support routines are used to implement steps 1 and 2 above.

Remaining examples in this document describe step 3 only – the starting point for each is a small *emission* function that is assumed to be called with appropriate arguments extracted from a decoded instruction.

See the *Imperas Processor Modeling Guide* for a detailed explanation of the steps required to model a processor using the functions in this interface.

2 Interaction with Imperas Simulators

Processor models developed using this interface can be used with both Imperas OVP platforms and the Imperas Simulator (`imperas.exe`) simulation product.

It is important to understand at a high level how the simulators use the morph callback function, and what is happening when it is called. This is briefly described here.

1. When the simulator executes a branch to a simulated address that it has not previously encountered, it calls the morph callback to translate a sequence of simulated opcodes into native machine code. The code block is terminated when the simulator detects a subsequent branch or jump instruction[1]. It then executes that native code.
2. Previously-encountered translated sequences (*code blocks*) are cached in a *dictionary*. If the simulator executes the same code again, it will reuse the cached code block and not call the morph callback.
3. It is very important to understand that the morph callback does not execute simulated instructions: instead, **it describes the behavior of those instructions**, using a sequence of VMI morph time interface calls.
4. The VMI morph time interface routines generate an ordered list of *native machine interface* (NMI) nodes which, when processed in order, together describe the full behavior of an instruction.
5. When the simulator has assembled an NMI node list for a complete code block (which can contain many instructions), the list is passed to a compiler module which generates an equivalent native code block.

[1] Or by the `vmimtEndBlock` function, described later in this document.

3 Instruction Fetch and Decode Support Routines

The VMI morph-time routines described in this manual and processor model disassembler routines both require support routines for the fetch and decode of instructions.

File `ImpPublic/include/host/vmi/vmiCxt.h` provides an API for instruction fetch.

File `ImpPublic/include/host/vmi/vmiDecode.h` provides an API to simplify decode of fetched instructions.

3.1 vmicxtFetch[1248]Byte

Prototypes

```
Uns8  vmicxtFetch1Byte(vmiProcessorP processor, Addr simAddress);
Uns16 vmicxtFetch2Byte(vmiProcessorP processor, Addr simAddress);
Uns32 vmicxtFetch4Byte(vmiProcessorP processor, Addr simAddress);
Uns64 vmicxtFetch8Byte(vmiProcessorP processor, Addr simAddress);
```

Description

These four routines fetch (respectively) 1, 2, 4 and 8 byte instruction words from the passed address for the passed processor. The endianness of the fetch is specified by the current processor endianness.

Example

This example demonstrates usage of `vmicxtFetch4Byte` for the OR1K training examples.

```
//
// Decode the OR1K instruction at the passed address. If the decode succeeds,
// dispatch it to the corresponding function in the dispatch table and return
// True; otherwise, dispatch using the defaultCB and return False.
//
Bool or1kDecode(
    or1kP               or1k,
    Uns32               thisPC,
    or1kDispatchTableCP table,
    or1kDispatchFn      defaultCB,
    void               *userData,
    Bool                inDelaySlot
) {
    // get the instruction at the passed address - always 4 bytes on OR1K
    vmiProcessorP       processor   = (vmiProcessorP)or1k;
    Uns32               instruction = vmicxtFetch4Byte(processor, thisPC);
    or1kInstructionType type        = decode(instruction);

    // apply the callback, or the default if no match
    if(type!=OR1K_IT_LAST) {
        ((*table)[type])(or1k, thisPC, instruction, userData, inDelaySlot);
        return True;
    } else {
        defaultCB(or1k, thisPC, instruction, userData, inDelaySlot);
        return False;
    }
}
```

Notes and Restrictions

1. Multiple calls to `vmicxt` routines may be used to fetch parts of a single instruction. For example, a CISC processor mode (such as an x86) can use `vmicxtFetch1Byte` to get the first instruction byte and then, depending on the value fetched, use further `vmicxt` functions calls to get subsequent instruction bytes.

3.2 vmidNewDecodeTable

Prototype

```
vmidDecodeTableP vmidNewDecodeTable(Uns32 bits, Uns32 defaultValue);
```

Description

This function returns a new *decode table* object, that is used to construct robust and efficient instruction decoders. The decode table decodes instructions of width `bits`. `defaultValue` specifies a value that is returned by function `vmidDecode` if an unrecognized instruction is encountered.

Example

This example is part of the OR1K training examples.

```
//
// This macro adds a decode table entry for a specific instruction class
//
#define DECODE_ENTRY(_PRIORITY, _NAME) \
    vmidNewEntry(           \
        table,              \
        #_NAME,             \
        OR1K_IT_##_NAME,    \
        MASK_##_NAME,       \
        OP_##_NAME,         \
        _PRIORITY           \
    )

//
// Create the OR1K decode table
//
static vmidDecodeTableP createDecodeTable(void) {

    vmidDecodeTableP table = vmidNewDecodeTable(OR1K_BITS, OR1K_IT_LAST);

    // handle movhi instruction
    DECODE_ENTRY(0, MOVHI);

    // handle arithmetic instructions (second argument constant)
    DECODE_ENTRY(0, ADDI);
    DECODE_ENTRY(0, ADDIC);
    DECODE_ENTRY(0, ANDI);
    DECODE_ENTRY(0, ORI);
    DECODE_ENTRY(0, XORI);
    DECODE_ENTRY(0, MULI);
    … etc …
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64 currently.

3.3 vmidNewEntry

Prototype

```
Bool vmidNewEntry(
    vmidDecodeTableP table,
    const char      *name,
    Uns32            matchValue,
    Uns64            mask,
    Uns64            value,
    Int32            priority
);
```

Description

Given a previously-created decode table object, this function adds a new decode entry to that table. Each decode entry decodes a single instruction type. The name of the entry is given by the `name` argument (this is informative only and used in error messages).

An instruction matches the new entry if:

```
(instruction & mask) == value
```

If this decode entry matches an instruction, `vmidDecode` will return `matchValue` (which is typically a processor-model-specific enumeration member).

It is possible that multiple entries in a decode table match the same instruction pattern – for example, often a RISC move instruction is a special case of an arithmetic instruction (such as an add). If such conflicting entries are required, they must be given distinct `priority` values, and the entry with greatest priority is deemed to match. If two entries with the same priority both match a candidate instruction, a decode table entry conflict error will be generated when `vmidDecode` is first called for the table.

If the decode entry was successfully created, `vmidNewEntry` returns `True`. Otherwise (if the decode table is already in use or `value` specifies bits that are not selected by `mask`) it returns `False`.

Typically, calls to `vmidNewEntry` are used within a macro as in the example below.

See also function `vmidNewEntryFmtBin`, which enables decode table entries to be created from format strings.

Example

This example is part of the OR1K training examples. The decode for each opcode is specified by patterns in file `or1kInstructions.h`:

```
////////////////////////////////////////////////////////////////////////////
// OPCODE FORM 4
// OPCODE(6) D(5) UNUSED(4) OPCODE(1) I(16)
////////////////////////////////////////////////////////////////////////////
#define OP4(_OP1, _OP2) (Uns32)((WIDTH(6,_OP1)<<26) | (WIDTH(1,_OP2)<<16))

#define OP4_MASK        OP4(-1, -1)
#define OP4_D(_I)       WIDTH(5,(_I)>>21)
```

```
#define OP4_I(_I)       WIDTH(16,(_I)>>0)
#define MASK_MOVHI      OP4_MASK
#define OP_MOVHI        OP4(0x06, 0x0)
```

An enumeration specifying the different instruction types is in `or1kDecode.h`:

```
//
// Instruction type enumeration
//
typedef enum or1kInstructionTypeE {

    // movhi instruction
    OR1K_IT_MOVHI,

    // arithmetic instructions (second argument constant)
    OR1K_IT_ADDI,
    OR1K_IT_ADDIC,
    OR1K_IT_ANDI,
    OR1K_IT_ORI,
    OR1K_IT_XORI,
    OR1K_IT_MULI,
    … etc …
}
```

Then the decode table is filled by function `createDecodeTable` in file `or1kDecode.c`:

```
//
// This macro adds a decode table entry for a specific instruction class
//
#define DECODE_ENTRY(_PRIORITY, _NAME) \
    vmidNewEntry(           \
        table,              \
        #_NAME,             \
        OR1K_IT_##_NAME,    \
        MASK_##_NAME,       \
        OP_##_NAME,         \
        _PRIORITY           \
    )

//
// Create the OR1K decode table
//
static vmidDecodeTableP createDecodeTable(void) {

    vmidDecodeTableP table = vmidNewDecodeTable(OR1K_BITS, OR1K_IT_LAST);

    // handle movhi instruction
    DECODE_ENTRY(0, MOVHI);

    // handle arithmetic instructions (second argument constant)
    DECODE_ENTRY(0, ADDI);
    DECODE_ENTRY(0, ADDIC);
    DECODE_ENTRY(0, ANDI);
    DECODE_ENTRY(0, ORI);
    DECODE_ENTRY(0, XORI);
    DECODE_ENTRY(0, MULI);
    … etc …
}
```

Notes and Restrictions

1. Entries may not be added to a decode table after `vmidDecode` has been called on that table.
2. `mask` must select all non-zero bits in `value` (i.e. `(mask&~value)` must be zero).

3.4 vmidNewEntryFmtBin

Prototype

```
Bool vmidNewEntryFmtBin(
    vmidDecodeTableP table,
    const char      *name,
    Uns32            matchValue,
    const char      *format,
    Int32            priority
);
```

Description

Given a previously-created decode table object, this function adds a new decode entry to that table. Each decode entry decodes a single instruction type. The name of the entry is given by the `name` argument (this is informative only and used in error messages).

The instruction format string, `format`, contains three kinds of characters:

1. *constrained* characters (either '0' or '1') – the corresponding bit in the instruction must have the same value;
2. *spacer* characters (any of '|', '/', comma, space or tab) – these are ignored and can be freely used to improve readability of the format string;
3. *don't care* characters (any character not listed above) – these characters can be either 0 or 1 in the instruction and it will still match.

The format specifies bits in the instruction in most-significant bit to least-significant bit order. For example, the following pattern could be used to create a decode table entry that matches a 16-bit instruction with the five most-significant bits `01001` and the three least significant bits `110`:

```
"01001........110"
```

The case above uses the dot character as a *don't care* character. Here is another example that matches exactly the same instruction pattern, but using *x* as a don't care character and using vertical-bar spacer characters to improve readability:

```
"|01001|xxxxxxxx|110|"
```

It is possible that multiple entries in a decode table match the same instruction pattern – for example, often a RISC move instruction is a special case of an arithmetic instruction (such as an add). If such conflicting entries are required, they must be given distinct `priority` values, and the entry with greatest priority is deemed to match. If two entries with the same priority both match a candidate instruction, a decode table entry conflict error will be generated when `vmidDecode` is first called for the table.

If the decode entry was successfully created, `vmidNewEntry` returns `True`. Otherwise (if the decode table is already in use, or `value` specifies bits that are not selected by `mask`, or the pattern string has the wrong number of characters) it returns `False`.

Typically, calls to `vmidNewEntryFmtBin` are used within a macro as in the example below.

See also function `vmidNewEntry`, which enables decode table entries to be created from mask/value pairs.

Example

This example is part of the ARC processor model example. An enumeration specifying the different instruction types is in `arcDecodeTypes.h`:

```
typedef enum arcInstructionTypeE {

    ////////////////////////////////////////////////////////////////////////////
    // 32-BIT INSTRUCTIONS
    ////////////////////////////////////////////////////////////////////////////

    // nonary instructions
    ITYPE_SET_32_0 (SWI),
    ITYPE_SET_32_0 (SYNC),
    ITYPE_SET_32_0 (RTIE),
    ITYPE_SET_32_0 (BRK),
    ITYPE_SET_32_0 (NOP),

    // unary instructions (with major opcode 0x04)
    ITYPE_SET_32_1 (ASL),
    ITYPE_SET_32_1 (ASR),
    ITYPE_SET_32_1 (LSR),
    ITYPE_SET_32_1 (ROR),
    ITYPE_SET_32_1 (RRC),
    ITYPE_SET_32_1 (SEXB),
    ITYPE_SET_32_1 (SEXW),
    ITYPE_SET_32_1 (EXTB),
    ITYPE_SET_32_1 (EXTW),
    ITYPE_SET_32_1 (ABS),
    ITYPE_SET_32_1 (NOT),
    ITYPE_SET_32_1 (RLC),

    … etc …
}
```

Macros are used to specify the decode entries for each instruction type, for example:

```
#define ITYPE_SET_32_1(_NAME)       \
    ARC_IT_##_NAME##_B_C,           \
    ARC_IT_##_NAME##_B_U6,          \
    ARC_IT_##_NAME##_B_LIMM,        \
    ARC_IT_##_NAME##_0_C,           \
    ARC_IT_##_NAME##_0_U6,          \
    ARC_IT_##_NAME##_0_LIMM
```

Decode entries corresponding to each instruction type are specified in file `arcDecodeEntries.h` using `vmidNewEntryFmtBin`, as follows:

```
#define DECODE_ENTRY(_PRIORITY, _NAME, _PATTERN) \
    vmidNewEntryFmtBin(table, #_NAME, ARC_IT_##_NAME, _PATTERN, _PRIORITY)

#define DECODE_SET_32_1(_NAME, _F1, _F2) \
    DECODE_ENTRY(0, _NAME##_B_C,    "|"_F1"|...|00|101111|.|...|......|"_F2"|"); \
    DECODE_ENTRY(0, _NAME##_B_U6,   "|"_F1"|...|01|101111|.|...|......|"_F2"|"); \
    DECODE_ENTRY(1, _NAME##_B_LIMM, "|"_F1"|...|00|101111|.|...|111110|"_F2"|"); \
    DECODE_ENTRY(2, _NAME##_0_C,    "|"_F1"|110|00|101111|.|111|......|"_F2"|"); \
```

```
    DECODE_ENTRY(2, _NAME##_0_U6,   "|"_F1"|110|01|101111|.|111|......|"_F2"|"); \
    DECODE_ENTRY(3, _NAME##_0_LIMM, "|"_F1"|110|00|101111|.|111|111110|"_F2"|")
```

Decode tables are filled by functions `createDecodeTable16` and `createDecodeTable32` in file `arcDecode.c`:

```
static vmidDecodeTableP createDecodeTable32(void) {

    vmidDecodeTableP table = vmidNewDecodeTable(32, ARC_IT_LAST);

    // nonary instructions
    DECODE_SET_32_0 (SWI,  "010", "01", "101111", "000", "111111");
    DECODE_SET_32_0 (SYNC, "011", "01", "101111", "000", "111111");
    DECODE_SET_32_0 (RTIE, "100", "00", "101111", "000", "111111");
    DECODE_SET_32_0 (BRK,  "101", "01", "101111", "000", "111111");
    DECODE_SET_32_0 (NOP,  "110", "01", "001010", "111", "000000");

    // unary instructions (with major opcode 0x04)
    DECODE_SET_32_1 (ASL,      "00100", "000000");
    DECODE_SET_32_1 (ASR,      "00100", "000001");
    DECODE_SET_32_1 (LSR,      "00100", "000010");
    DECODE_SET_32_1 (ROR,      "00100", "000011");
    DECODE_SET_32_1 (RRC,      "00100", "000100");
    DECODE_SET_32_1 (SEXB,     "00100", "000101");
    DECODE_SET_32_1 (SEXW,     "00100", "000110");
    DECODE_SET_32_1 (EXTB,     "00100", "000111");
    DECODE_SET_32_1 (EXTW,     "00100", "001000");
    DECODE_SET_32_1 (ABS,      "00100", "001001");
    DECODE_SET_32_1 (NOT,      "00100", "001010");
    DECODE_SET_32_1 (RLC,      "00100", "001011");

    … etc …
}
```

Notes and Restrictions

1. Entries may not be added to a decode table after `vmidDecode` has been called on that table.
2. The number of constrained and don't care characters added together must equal the `bits` argument given when the decode table was created.

3.5 vmidDecode

Prototype

```
Uns32 vmidDecode(vmidDecodeTableP table, Uns64 instr);
```

Description

This function decodes the passed instruction value `instr` using the decode table. If the instruction matches some entry in the decode table, the `matchValue` associated with that entry is returned. Otherwise, the `defaultValue` specified when the table was created is returned.

Example

This example is part of the OR1K training examples.

```
//
// Decode the instruction and return an enum describing it
//
static or1kInstructionType decode(Uns32 instruction) {

    // get the OR1K decode table
    static vmidDecodeTableP decodeTable;
    if(!decodeTable) {
        decodeTable = createDecodeTable();
    }

    // decode the instruction to get the type
    or1kInstructionType type = vmidDecode(decodeTable, instruction);

    // some arguments to l.sf and l.sfi are invalid: filter them here
    if((type==OR1K_IT_SF) && !getCmpInfo(OP5_CMPOP(instruction))->name) {
        type = OR1K_IT_LAST;
    } else if((type==OR1K_IT_SFI) && !getCmpInfo(OP6_CMPOP(instruction))->name) {
        type = OR1K_IT_LAST;
    }

    return type;
}
```

Notes and Restrictions

None.

4 Basic Register Operations

This section describes emission functions for basic register operations: moves, unary operations, binary operations and comparisons.

4.1 Unary Operation Types

The available unary operations are described by the `vmiUnop` enumeration in `vmiTypes.h`:

```
typedef enum {
                    // MOVE OPERATIONS
    vmi_MOV,        // d <- a
    vmi_SWP,        // d <- byteswap(a)

                    // ARITHMETIC OPERATIONS
    vmi_NEG,        // d <- -a
    vmi_ABS,        // d <- (a<0) ? -a : a

                    // SATURATED ARITHMETIC OPERATIONS
    vmi_NEGSQ,      // d <- saturate_signed(-a)
    vmi_ABSSQ,      // d <- (a<0) ? saturate_signed(-a) : a

                    // BITWISE OPERATIONS
    vmi_NOT,        // d <- ~a

                    // MISCELLANEOUS OPERATIONS
    vmi_CLZ,        // d <- count_leading_zeros(a)
    vmi_CLO,        // d <- count_leading_ones(a)
    vmi_CTZ,        // d <- count_trailing_zeros(a)
    vmi_CTO,        // d <- count_trailing_ones(a)
    vmi_BSFZ,       // d <- least_significant_zero_index(a)
    vmi_BSFO,       // d <- least_significant_one_index(a)
    vmi_BSRZ,       // d <- most_significant_zero_index(a)
    vmi_BSRO,       // d <- most_significant_one_index(a)

    vmi_UNOP_LAST   // KEEP LAST

} vmiUnop;
```

Signed saturation instructions clamp overflowing values to the smallest negative or largest positive value. *Unsigned saturation* instructions clamp overflowing values to zero or the largest value.

Operations `vmi_CLZ`, `vmi_CLO`, `vmi_CTZ` and `vmi_CTO` count the number of leading (most significant) or trailing (least significant) one or zero bits in the argument value. If there are no bits of the required type, the result value is the size of the type in bits. For example, for an 8-bit type, the result will be 8 if there are no bits of the required type in the argument; otherwise it will be a value in the range 0-7.

Operations `vmi_BSFZ`, `vmi_BSFO`, `vmi_BSRZ` and `vmi_BSRO` return the bit index of the most significant (`BSR`) or least significant (`BSF`) one or zero bit in the argument value, where the least significant bit of a value is index 0. If there are no bits of the required type, the result value is the size of the type in bits.

4.2 Binary Operation Types

The available binary operations are described by the `vmiBinop` enumeration in `vmiTypes.h`:

```
typedef enum {
                    // ARITHMETIC OPERATIONS
    vmi_ADD,        // d <- a + b
    vmi_ADC,        // d <- a + b + C
    vmi_SUB,        // d <- a - b
    vmi_SBB,        // d <- a - b - C
    vmi_RSBB,       // d <- b - a - C
    vmi_RSUB,       // d <- b - a
    vmi_IMUL,       // d <- a * b (signed)
    vmi_MUL,        // d <- a * b (unsigned)
    vmi_IDIV,       // d <- a / b (signed)
    vmi_DIV,        // d <- a / b (unsigned)
    vmi_IREM,       // d <- a % b (signed)
    vmi_REM,        // d <- a % b (unsigned)
    vmi_CMP,        // a - b

                    // SATURATED ARITHMETIC OPERATIONS
    vmi_ADDSQ,      // d <- saturate_signed(a + b)
    vmi_SUBSQ,      // d <- saturate_signed(a - b)
    vmi_RSUBSQ,     // d <- saturate_signed(b - a)
    vmi_ADDUQ,      // d <- saturate_unsigned(a + b)
    vmi_SUBUQ,      // d <- saturate_unsigned(a - b)
    vmi_RSUBUQ,     // d <- saturate_unsigned(b - a)

                    // HALVING ARITHMETIC OPERATIONS
    vmi_ADDSH,      // d <- ((signed)(a + b)) / 2
    vmi_SUBSH,      // d <- ((signed)(a - b)) / 2
    vmi_RSUBSH,     // d <- ((signed)(b - a)) / 2
    vmi_ADDUH,      // d <- ((unsigned)(a + b)) / 2
    vmi_SUBUH,      // d <- ((unsigned)(a - b)) / 2
    vmi_RSUBUH,     // d <- ((unsigned)(b - a)) / 2
    vmi_ADDSHR,     // d <- round(((signed)(a + b)) / 2)
    vmi_SUBSHR,     // d <- round(((signed)(a - b)) / 2)
    vmi_RSUBSHR,    // d <- round(((signed)(b - a)) / 2)
    vmi_ADDUHR,     // d <- round(((unsigned)(a + b)) / 2)
    vmi_SUBUHR,     // d <- round(((unsigned)(a - b)) / 2)
    vmi_RSUBUHR,    // d <- round(((unsigned)(b - a)) / 2)

                    // BITWISE OPERATIONS
    vmi_OR,         // d <- a | b
    vmi_AND,        // d <- a & b
    vmi_XOR,        // d <- a ^ b
    vmi_ORN,        // d <- a | ~b
    vmi_ANDN,       // d <- a & ~b
    vmi_XORN,       // d <- a ^ ~b
    vmi_NOR,        // d <- ~(a | b)
    vmi_NAND,       // d <- ~(a & b)
    vmi_XNOR,       // d <- ~(a ^ b)

                    // SHIFT/ROTATE OPERATIONS
    vmi_ROL,        // d <- a << b | a >> <bits>-b
    vmi_ROR,        // d <- a >> b | a << <bits>-b
    vmi_RCL,        // (d,c) <- (a,c) << b | (a,c) >> <bits>-b
    vmi_RCR,        // (d,c) <- (a,c) >> b | (a,c) << <bits>-b
    vmi_SHL,        // d <- a << b
    vmi_SHR,        // d <- (unsigned)a >> b
    vmi_SAR,        // d <- (signed)a >> b

                    // SATURATED SHIFT OPERATIONS
    vmi_SHLSQ,      // d <- saturate_signed(a << b)
    vmi_SHLUQ,      // d <- saturate_unsigned(a << b)

                    // ROUNDING SHIFT OPERATIONS
    vmi_SHRR,       // d <- round((unsigned)a >> b)
```

```
    vmi_SARR,       // d <- round((signed)a >> b)

    vmi_BINOP_LAST  // KEEP LAST

} vmiBinop;
```

4.3 Handling Instruction Flags

Several functions in this section use a `vmiFlags` structure to indicate how processor flags are used and affected by translated native code. There is one input flag (carry) and five output flags (carry, parity, zero, sign, and overflow), the behaviour of which for any instruction is specified by entries in the structure.

4.3.1 Carry In Flag

The carry in flag is used by arithmetic operations `vmi_ADC` and `vmi_SBB` to provide the input carry or borrow value. For rotate-with-carry operations, the carry participates in an N+1 bit rotation with the N bit operand value.

4.3.2 Carry Out Flag

The carry out flag is set by arithmetic operations to indicate a carry out or a borrow. It is also set by shift and rotate operations to indicate the last bit shifted or rotated out of the operand. For rotate-with-carry operations, the carry participates in an N+1 bit rotation with the N bit operand value; for shifts or rotates of zero, the carry is unchanged. For an N-bit multiply operation, the carry flag is set if the result was truncated in order to fit into N bits.

4.3.3 Parity Flag

The parity flag indicates the parity of the least significant byte of the result of an operation. If the least significant byte contains an even number of one bits, the flag is set; otherwise, it is cleared.

4.3.4 Zero Flag

The zero flag is set if an operation result is zero and cleared otherwise.

4.3.5 Sign Flag

The sign flag is set if the most significant bit of an operation result is set and cleared otherwise.

4.3.6 Overflow Flag

The overflow flag is set if an arithmetic operation overflowed (the carry into the most significant bit of the result is different to the carry out). For an N-bit multiply operation, the overflow flag is set if the result was truncated in order to fit into N bits.

4.3.7 `vmiFlags` Structure Usage

The carry input flag is be represented in a processor structure by an `Uns8` entry. Each of the five output flags may also be represented by an `Uns8` entry, if required by that model. In order to tell the simulator how to use the flags, each emission function that uses them is passed a pointer to a `vmiFlags` structure:

```
typedef enum {
    vmi_CF=0,           // carry flag
    vmi_PF=1,           // parity flag
    vmi_ZF=2,           // zero flag
    vmi_SF=3,           // sign flag
    vmi_OF=4,           // overflow flag
    vmi_LF=5            // KEEP LAST
} vmiFlag;

//
// Bitmask indicating whether particular flags should be negated
//
typedef enum {
    vmi_FN_NONE  =0x00, // empty negate mask
    vmi_FN_CF_IN =0x01, // negate carry in flag
    vmi_FN_CF_OUT=0x02, // negate carry out flag
    vmi_FN_PF    =0x04, // negate parity flag
    vmi_FN_ZF    =0x08, // negate zero flag
    vmi_FN_SF    =0x10, // negate sign flag
    vmi_FN_OF    =0x20, // negate overflow flag
} vmiFlagNegate;

//
// Processor flag-related structures
//
typedef struct vmiFlagsS {
    vmiReg        cin;          // register specifying carry in
    vmiReg        f[vmi_LF];    // registers to hold operation results
    vmiFlagNegate negate;       // bitmask of negated flags
} vmiFlags;
```

If an emission function is passed a null pointer as its `flags` argument, then the function should neither use nor set any flags. Otherwise, the function should obtain the carry in (if required) from a register described by the `cin` field and write output flags to registers described in the `f` array. If any register is given as `VMI_NOFLAG`, it should be ignored or discarded by the emission function.

There is also a *negate mask* that allows the parity of flags in processor models to be inverted with respect to those in the native processor.

As an example, here is how flag settings could be used to use register `CPUX_CARRY` as an input flag (if required) and store the output carry to register `CPUX_CARRY` and the output overflow to `CPUX_OVERFLOW`:

```
// processor structure definition
```

```
typedef struct cpuxS {
    Uns8  carryFlag;        // carry flag
    Uns8  overflowFlag;     // overflow flag
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_CARRY      CPUX_OFFSET(carryFlag);
#define CPUX_OVERFLOW   CPUX_OFFSET(overflowFlag);

const vmiFlags flagsCO = {
    cin : CPUX_CARRY,               // offset to carry in flag
    f   : {
        [vmi_CF] = CPUX_CARRY,      // offset to carry out flag
        [vmi_PF] = VMI_NOFLAG,      // parity flag not used
        [vmi_ZF] = VMI_NOFLAG,      // zero flag not used
        [vmi_SF] = VMI_NOFLAG,      // sign flag not used
        [vmi_OF] = CPUX_OVERFLOW    // offset to overflow flag
    }
};
```

For some processors, model flags are required to have inverted polarity with respect to native flags. As an example, for ARM processors, `SBC` and `SUB` instructions use and emit a *borrow* instead of a carry. To simulate this, the carry in must be negated before use (by `SBC`) and the carry out must be negated before being written to the model structure. This can be specified by using the `negate` mask in the `vmiFlags` structure as follows:

```
const vmiFlags flagsCO = {
    cin : CPUX_CARRY,               // offset to carry in flag
    f   : {
        [vmi_CF] = CPUX_CARRY,      // offset to carry out flag
        [vmi_PF] = VMI_NOFLAG,      // parity flag not used
        [vmi_ZF] = VMI_NOFLAG,      // zero flag not used
        [vmi_SF] = VMI_NOFLAG,      // sign flag not used
        [vmi_OF] = CPUX_OVERFLOW    // offset to overflow flag
    },
    vmi_FN_CF_IN|vmi_FN_CF_OUT      // negate carry in and carry out
};
```

Not all operations use or set all the flags; the following table gives usage for unary operation types (defined in `vmiTypes.h`).

```
    --------- ----- --------------------
    OPERATION INPUT       OUTPUT
    --------- ----- --------------------
              CY    CY  PF  ZF  SF  OF
    --------- ----- --------------------
    vmi_MOV    -     -   A   A   A   -
    vmi_SWP    -     -   A   A   A   -
    vmi_NEG    -     A   A   A   A   A
    vmi_ABS    -     A   A   A   A   A
    vmi_NEGSQ  -     A   A   A   A   A
    vmi_ABSSQ  -     A   A   A   A   A
    vmi_NOT    -     -   A   A   A   -
    vmi_CLZ    -     0   A   A   0   0
    vmi_CLO    -     0   A   A   0   0
    vmi_CTZ    -     0   A   A   0   0
    vmi_CTO    -     0   A   A   0   0
    vmi_BSFZ   -     0   A   A   0   0
    vmi_BSFO   -     0   A   A   0   0
    vmi_BSRZ   -     0   A   A   0   0
    vmi_BSRO   -     0   A   A   0   0
    --------- ----- --------------------

    ------ ----------------------------
    SYMBOL MEANING
```

```
------ ----------------------------
-      Unused or unaffected
A      Flag affected (output)
0      Flag cleared (output)
------ ----------------------------
```

The following table gives usage for binary operation types (defined in `vmiTypes.h`).

```
---------- ----- --------------------
OPERATION  INPUT       OUTPUT
---------- ----- --------------------
            CY    CY  PF  ZF  SF  OF
---------- ----- --------------------
vmi_ADD     -     A   A   A   A   A
vmi_ADC     U     A   A   A   A   A
vmi_SUB     -     A   A   A   A   A
vmi_SBB     U     A   A   A   A   A
vmi_RSBB    U     A   A   A   A   A
vmi_RSUB    -     A   A   A   A   A
vmi_IMUL    -     A   A   A   A   A
vmi_MUL     -     A   A   A   A   A
vmi_IDIV    -     -   A   A   A   -
vmi_DIV     -     -   A   A   A   -
vmi_IREM    -     -   A   A   A   -
vmi_REM     -     -   A   A   A   -
vmi_CMP     -     A   A   A   A   A
vmi_ADDSQ   -     A   A   A   A   A
vmi_SUBSQ   -     A   A   A   A   A
vmi_RSUBSQ  -     A   A   A   A   A
vmi_ADDUQ   -     A   A   A   A   A
vmi_SUBUQ   -     A   A   A   A   A
vmi_RSUBUQ  -     A   A   A   A   A
vmi_ADDSH   -     A   A   A   A   -
vmi_SUBSH   -     A   A   A   A   -
vmi_RSUBSH  -     A   A   A   A   -
vmi_ADDUH   -     A   A   A   A   -
vmi_SUBUH   -     A   A   A   A   -
vmi_RSUBUH  -     A   A   A   A   -
vmi_ADDSHR  -     A   A   A   A   -
vmi_SUBSHR  -     A   A   A   A   -
vmi_RSUBSHR -     A   A   A   A   -
vmi_ADDUHR  -     A   A   A   A   -
vmi_SUBUHR  -     A   A   A   A   -
vmi_RSUBUHR -     A   A   A   A   -
vmi_OR      -     0   A   A   A   0
vmi_AND     -     0   A   A   A   0
vmi_XOR     -     0   A   A   A   0
vmi_ORN     -     0   A   A   A   0
vmi_ANDN    -     0   A   A   A   0
vmi_XORN    -     0   A   A   A   0
vmi_NOR     -     0   A   A   A   0
vmi_NAND    -     0   A   A   A   0
vmi_XNOR    -     0   A   A   A   0
vmi_ROL     U0    A   A   A   A   -
vmi_ROR     U0    A   A   A   A   -
vmi_RCL     U     A   A   A   A   -
vmi_RCR     U     A   A   A   A   -
vmi_SHL     U0    A   A   A   A   -
vmi_SHR     U0    A   A   A   A   -
vmi_SAR     U0    A   A   A   A   -
vmi_SHLSQ   -     A   A   A   A   A
vmi_SHLUQ   -     A   A   A   A   A
vmi_SHRR    -     A   A   A   A   -
vmi_SARR    -     A   A   A   A   -
--------- ----- --------------------

------ ------------------------------------------------------------------
SYMBOL MEANING
------ ------------------------------------------------------------------
```

```
    -      Unused or unaffected
    U      Flag used (input)
    U0     Flag used only if shift/rotate is zero (input)
    A      Flag affected (output)
    0      Flag cleared (output)
    ------ ----------------------------------------------------------------
```

For signed saturating operations (with `SQ` suffix) the flags represent the computed value *before* saturation. It is therefore possible to tell whether signed saturation has occurred using the *overflow* flag.

For unsigned saturating operations (with `UQ` suffix) the flags represent the computed value *before* saturation. It is therefore possible to tell whether unsigned saturation has occurred using the *carry* flag.

4.4 Handling Exceptions

Integer operations can cause two kinds of exception: *integer overflow* (for example, when the minimum negative integer is divided by -1) and *divide-by-zero*. In both cases, these exceptions cause the model *integer exception handler* to be called. The integer exception handler is of type `vmiArithExceptFn` and is specified as the `arithExceptCB` field of the processor `vmiIASAttrs` structure:

```
#define VMI_ARITH_EXCEPT_FN(_NAME) vmiIntegerExceptionResult _NAME( \
    vmiProcessorP           processor,          \
    vmiNumericExceptionType exceptionType,      \
    vmiExceptionContext     exceptionContext    \
)
typedef VMI_ARITH_EXCEPT_FN((*vmiArithExceptFn));
```

The exact reason it is being called is indicated by the `exceptionType` argument:

```
typedef enum vmiIntegerExceptionTypeE {
    VMI_INTEGER_DIVIDE_BY_ZERO,
    VMI_INTEGER_OVERFLOW
} vmiIntegerExceptionType;
```

The handler may modify processor state to reflect the result of the faulting operation, or possibly use `vmirtSetPCException` to jump to a simulated exception vector. The return value of the handler is of type `vmiIntegerExceptionResult`:

```
typedef enum vmiIntegerExceptionResultE {
    VMI_NUMERIC_UNHANDLED,      // not handled
    VMI_NUMERIC_ABORT,          // handled, abort current instruction
    VMI_NUMERIC_CONTINUE,       // handled, continue current instruction
} vmiIntegerExceptionResult;
```

A result of `VMI_INTEGER_UNHANDLED` indicates that the exception condition is unexpected and simulation should terminate.

A result of `VMI_INTEGER_ABORT` indicates that the current simulated instruction should be terminated and simulation should resume with the *next* simulated instruction.

A result of `VMI_INTEGER_CONTINUE` indicates that simulation of the current instruction should be resumed after the faulting operation. In this case, all results of the faulting operation will be discarded and simulation will resume with the next VMI operation (which could be in the next simulated instruction or a later operation in the current simulated instruction).

4.5 vmimtGetSMPParentRegister

Prototype

```
vmiReg vmimtGetSMPParentRegister(vmiReg r, Uns32 level);
```

Description

The VMI interfaces allow the specification of SMP processor clusters. These clusters are implemented as a number of levels of container processor objects, each of which can contain further levels of container processors or leaf processors (which are actually simulated). As an example, the MIPS 1004K OVP processor model has these hierarchy levels:

1. a root level CMP processor object, containing:
2. a number of CPU processor objects, each containing:
3. a number of VPE (*virtual processing element*) processor objects, each containing:
4. a number of TC (*thread context*) objects, which are actually simulated.

The thread context objects each represent a microthread running on a VPE. Instructions on a TC can refer to registers:

1. local to the TC itself (for example, the GPRs);
2. at the VPE level (for example, most system control registers are implemented at the VPE level);
3. at the CPU level (a few system control registers are shared by all the VPEs).

It is possible for a TC to be moved to a different VPE on the same CPU at run time. For example, TC0 might start life bound to VPE0 on CPU0, but later on be dynamically rebound to VPE1 of CPU0 instead[2]. When this rebinding occurs, *any reference to a register at the VPE level must be sure to use the new VPE*.

Function `vmimtGetSMPParentRegister` takes as an argument a `vmiReg` representing a register in the processor and a `level` argument, indicating a parent level (level 0 is the leaf level processor itself, level 1 is its parent, level 2 is its grandparent, and so on). It returns a new `vmiReg` representing that register at the indicated parent level. The register description remains valid *even if the leaf level processor is subsequently relocated in the SMP cluster so that the parent at that level changes*.

Example

This example is taken from the MIPS processor model. In file `mips32Morph.c`, there is a function `mips32VPEReg` which returns the `vmiReg` description for a VPE-level register for the current TC. If the processor does not implement the multithreaded ASE, the TC and VPE levels are equivalent; otherwise, function `vmimtGetSMPParentRegister` is used to construct the correct `vmiReg` description for the parent VPE:

[2] See function `vmirtSetSMPParent` in the *VMI Run Time Function Reference*.

```
vmiReg mips32VPEReg(mips32P tc, vmiReg r) {
    mips32P vpe = VPE_FOR_TC(tc);
    return (tc==vpe) ? r : vmimtGetSMPParentRegister(r, 1);
}
```

This function is used, for example, in `mips32MorphCop0.c` to return the `vmiReg` description for a control register:

```
static vmiReg getCOP0Reg(mips32P tc, Uns32 reg, Uns32 sel) {

    mips32Cop0RegId id     = cop0RegInfo[reg][sel].id;
    vmiReg          result = MIPS32_CPU_REG(cop0.regs[id]);

    switch(getCOP0Level(reg, sel)) {

        case COP0_RL_CPU:
            return VMI_REG_DELTA(result, tc->cpuDelta);

        case COP0_RL_VPE:
            return mips32VPEReg(tc, result);

        default:
            return result;
    }
}
```

It is only necessary to use `vmimtGetSMPParentRegister` when *the parent at that level can change dynamically at run time*. In the case of the MIPS processor, a TC can be bound to a different VPE at run time, but that VPE must lie on the same CPU. Therefore, references to CPU level registers for a TC do not need to use `vmimtGetSMPParentRegister` but can use the macro `VMI_REG_DELTA` instead. This macro constructs a `vmiReg` value referencing a register at a constant offset from the current processor.

Notes and Restrictions

None.

4.6 vmimtMoveRC

Prototype

```
void vmimtMoveRC(
    Uns32  bits,
    vmiReg rd,
    Uns64  c
);
```

Description

Emit code to move a constant value `c` into target register `rd` within the processor. The register has size `bits` within the processor structure.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitMoveConstToReg(Uns32 rd, Uns32 constVal) {
    vmimtMoveRC(CPUX_GBITS, CPUX_REG(rd), constVal);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. For target registers less than 64 bits wide, the unused most significant bits of `c` are silently discarded.

4.7 vmimtMoveRSimPC

Prototype

```
void vmimtMoveRSimPC(
    Uns32  bits,
    vmiReg rd
);
```

Description

Emit code to move the current simulated program counter into target register `rd` within the processor. The register has size `bits` within the processor structure.

If a processor model does not use physically-mapped code dictionaries, then this is equivalent to using `vmimtMoveRC`, specifying the current program counter as the constant argument. However, when processor models do use physically-mapped code dictionaries, `vmimtMoveRSimPC` **must** be used to obtain the current simulated address, because the same JIC compiled code block can be mapped at *different* simulated addresses.

See the description of `vmirtAliasMemoryVM` in the *VMI Run Time Function Reference* and also the *Imperas Processor Modeling Guide* for more information about physically-mapped code dictionaries.

Example

The template MIPS model uses this function when calculating link addresses:

```
// Emit code to set link address to thisPC+8
static void emitSetLinkAddress(vmiReg rl) {

    Uns32 bits = MIPS32_GPR_BITS;

    vmimtMoveRSimPC(bits, rl);
    vmimtBinopRC(bits, vmi_ADD, rl, 8, 0);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

4.8 vmimtMoveRR

Prototype

```
void vmimtMoveRR(
    Uns32  bits,
    vmiReg rd,
    vmiReg ra
);
```

Description

Emit code to move from source register `ra` to target register `rd` within the processor. Both registers are of size `bits` within the processor structure.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitMoveRegToReg(Uns32 rd, Uns32 ra) {
    vmimtMoveRR(CPUX_GBITS, CPUX_REG(rd), CPUX_REG(ra));
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32, 64 or 128.

4.9 vmimtMoveExtendRR

Prototype

```
void vmimtMoveExtendRR(
    Uns32  destBits,
    vmiReg rd,
    Uns32  srcBits,
    vmiReg ra,
    Bool   signExtend
);
```

Description

Emit code to move from source register `ra` to target register `rd` within the processor. The destination register is of size `destBits` and the source register of size `srcBits` (`destBits` must be equal to or larger than `srcBits`). If source and destination sizes are unequal, then the source will be zero-extended (if `signExtend` is `False`) or sign-extended (if `signExtend` is `True`) to the full destination size.

If source and destination sizes match, this function is exactly equivalent to `vmimtMoveRR`.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS      32  // number of GPRs
#define CPUX_GBITS      32  // size of GPR (bits)
#define CPUX_SHORT_BITS 16  // size of short

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitMoveSRegToReg(Uns32 rd, Uns32 ra, Bool extend) {
    vmimtMoveExtendRR(
        CPUX_GBITS,      CPUX_REG(rd),
        CPUX_SHORT_BITS, CPUX_REG(ra),
        extend
    );
}
```

Notes and Restrictions

1. `destBits` and `srcBits` must be 8, 16, 32 or 64.
2. `destBits` must be equal to or greater than `srcBits`.

4.10 vmimtCondMoveRRR

Prototype

```
void vmimtCondMoveRRR(
    Uns32  bits,
    vmiReg flag,
    Bool   select1,
    vmiReg rd,
    vmiReg ra,
    vmiReg rb
);
```

Description

Emit code to compare the (8-bit) register `flag` in the processor structure with `select1`. If `flag` equals select1, the emitted code will move from source register `ra` to target register `rd` within the processor. Otherwise, the emitted code will move source register `rb` to target register `rd` within the processor. All registers are of size `bits` within the processor structure.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitCondMoveRRR(Uns32 f, Uns32 rd, Uns32 ra, Uns32 rb) {
    vmimtCondMoveRRR(
        CPUX_GBITS, CPUX_REG(f), 1, CPUX_REG(rd), CPUX_REG(ra), CPUX_REG(rb)
    );
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32, 64 or 128.

4.11 vmimtCondMoveRRC

Prototype

```
void vmimtCondMoveRRC(
    Uns32  bits,
    vmiReg flag,
    Bool   select1,
    vmiReg rd,
    vmiReg ra,
    Uns64  c
);
```

Description

Emit code to compare the (8-bit) register `flag` in the processor structure with `select1`. If `flag` equals select1, the emitted code will move from source register `ra` to target register `rd` within the processor. Otherwise, the emitted code will move constant `c` to target register `rd` within the processor. All registers are of size `bits` within the processor structure.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitCondMoveRRC(Uns32 f, Uns32 rd, Uns32 ra, Uns32 c) {
    vmimtCondMoveRRC(CPUX_GBITS, CPUX_REG(f), 1, CPUX_REG(rd), CPUX_REG(ra), c);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32, 64 or 128.

4.12 vmimtCondMoveRCR

Prototype

```
void vmimtCondMoveRCR(
    Uns32  bits,
    vmiReg flag,
    Bool   select1,
    vmiReg rd,
    Uns64  c,
    vmiReg rb
);
```

Description

Emit code to compare the (8-bit) register `flag` in the processor structure with `select1`. If `flag` equals select1, the emitted code will move constant `c` to target register `rd` within the processor. Otherwise, the emitted code will move source register `rb` to target register `rd` within the processor. All registers are of size `bits` within the processor structure.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitCondMoveRCR(Uns32 f, Uns32 rd, Uns32 c, Uns32 rb) {
    vmimtCondMoveRCR(CPUX_GBITS, CPUX_REG(f), 1, CPUX_REG(rd), c, CPUX_REG(rb));
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32, 64 or 128.

4.13 vmimtCondMoveRCC

Prototype

```
void vmimtCondMoveRCC(
    Uns32  bits,
    vmiReg flag,
    Bool   select1,
    vmiReg rd,
    Uns64  c1,
    Uns64  c2
);
```

Description

Emit code to compare the (8-bit) register `flag` in the processor structure with `select1`. If `flag` equals select1, the emitted code will move constant `c1` to target register `rd` within the processor. Otherwise, the emitted code will move constant `c2` to target register `rd` within the processor. Register `rd` is of size `bits` within the processor structure.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitCondMoveRCC(Uns32 f, Uns32 rd, Uns32 c1, Uns32 c2) {
    vmimtCondMoveRCC(CPUX_GBITS, CPUX_REG(f), 1, CPUX_REG(rd), c1, c2);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32, 64 or 128.

4.14 vmimtUnopR

Prototype

```
void vmimtUnopR(
    Uns32      bits,
    vmiUnop    op,
    vmiReg     rd,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a unary operation on register `rd` in the processor structure, writing the result back to the same register. The argument `bits` gives the bit width for the operation.

Argument `op` is the unary operation to perform. Available unary operations are defined in `vmiTypes.h`: see *Unary Operation Types* for more information about this.

If the flags argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitUnopR(vmiUnop op, Uns32 rd) {
    vmimtUnopR(CPUX_GBITS, op, CPUX_REG(rd), NULL);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. The `vmi_SWP` unary operation entirely reverses the byte order of the argument. For example the 64-bit value `0x0102030405060708` becomes `0x0807060504030201`.

4.15 vmimtUnopRR

Prototype

```
void vmimtUnopRR(
    Uns32      bits,
    vmiUnop    op,
    vmiReg     rd,
    vmiReg     ra,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a unary operation on register `ra` in the processor structure, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the unary operation to perform. Available unary operations are defined in `vmiTypes.h`: see *Unary Operation Types* for more information about this.

If the flags argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

If `rd` and `ra` are the same, this is equivalent to `vmimtUnopR`.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitUnopRR(vmiUnop op, Uns32 rd, Uns32 ra) {
    vmimtUnopRR(CPUX_GBITS, op, CPUX_REG(rd), CPUX_REG(ra), NULL);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. The `vmi_SWP` unary operation entirely reverses the byte order of the argument. For example the 64-bit value `0x0102030405060708` becomes `0x0807060504030201`.
3. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

4.16 vmimtUnopRC

Prototype

```
void vmimtUnopRC(
    Uns32      bits,
    vmiUnop    op,
    vmiReg     rd,
    Uns64      c,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a unary operation on constant `c`, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the unary operation to perform. Available unary operations are defined in `vmiTypes.h`: see *Unary Operation Types* for more information about this.

If the flags argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitUnopRR(vmiUnop op, Uns32 rd, Uns32 c) {
    vmimtUnopRC(CPUX_GBITS, op, CPUX_REG(rd), c, NULL);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. The `vmi_SWP` unary operation entirely reverses the byte order of the argument. For example the 64-bit value `0x0102030405060708` becomes `0x0807060504030201`.
3. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

4.17 vmimtBinopRR

Prototype

```
void vmimtBinopRR(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rd,
    vmiReg     ra,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a binary operation on registers `rd` and `ra` in the processor structure, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  carryFlag;        // carry flag
    Uns8  overflowFlag;     // overflow flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_CARRY      CPUX_OFFSET(carryFlag);
#define CPUX_OVERFLOW   CPUX_OFFSET(overflowFlag);

// vmiFlags specifying:
// 1. Use ‘carryFlag’ as CF input (if appropriate);
// 2. Set ‘carryFlag’ from CF output;
// 3. Set ‘overflowFlag’ from OF output.
const vmiFlags flagsCO = {
    cin : CPUX_CARRY,           // offset to carry in flag
    f   : {
        [vmi_CF] = CPUX_CARRY,  // offset to carry out flag
        [vmi_PF] = VMI_NOFLAG,  // parity flag not used
        [vmi_ZF] = VMI_NOFLAG,  // zero flag not used
        [vmi_SF] = VMI_NOFLAG,  // sign flag not used
        [vmi_OF] = CPUX_OVERFLOW// offset to overflow flag
    }
};

static void emitBinopRR(vmiBinop op, Uns32 rd, Uns32 ra) {
    vmimtBinopRR(
        CPUX_GBITS, op, CPUX_REG(rd), CPUX_REG(ra), &flagsCO
```

```
    );
}
```

Notes and Restrictions

1. The `bits` argument must be 8, 16, 32 or 64.
2. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
3. For shift/rotate operations (`vmi_ROL`, `vmi_ROR`, `vmi_RCL`, `vmi_RCR`, `vmi_SHL`, `vmi_SHR` and `vmi_SAR`) the shift/rotate amount `b` is by default masked using `bits-1` before use. For example, if `bits` is 32 then the shift/rotate amount will be masked to the range 0..31 before use. This default behavior can be overridden by `vmimtSetShiftMask`.
4. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

4.18 vmimtBinopRRR

Prototype

```
void vmimtBinopRRR(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rd,
    vmiReg     ra,
    vmiReg     rb,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a binary operation on registers `ra` and `rb` in the processor structure, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

If `rd` and `ra` are equal, this is equivalent to `vmimtBinopRR`.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  carryFlag;        // carry flag
    Uns8  overflowFlag;     // overflow flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_CARRY      CPUX_OFFSET(carryFlag);
#define CPUX_OVERFLOW   CPUX_OFFSET(overflowFlag);

// vmiFlags specifying:
// 1. Use ‘carryFlag’ as CF input (if appropriate);
// 2. Set ‘carryFlag’ from CF output;
// 3. Set ‘overflowFlag’ from OF output.
const vmiFlags flagsCO = {
    cin : CPUX_CARRY,           // offset to carry in flag
    f   : {
        [vmi_CF] = CPUX_CARRY,  // offset to carry out flag
        [vmi_PF] = VMI_NOFLAG,  // parity flag not used
        [vmi_ZF] = VMI_NOFLAG,  // zero flag not used
        [vmi_SF] = VMI_NOFLAG,  // sign flag not used
        [vmi_OF] = CPUX_OVERFLOW// offset to overflow flag
    }
};
```

```
static void emitBinopRR(vmiBinop op, Uns32 rd, Uns32 ra, Uns32 rb) {
    vmimtBinopRRR(
        CPUX_GBITS, op, CPUX_REG(rd), CPUX_REG(ra), CPUX_REG(rb), &flagsCO
    );
}
```

Notes and Restrictions

1. Argument `bits` must be 8, 16, 32 or 64.
2. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
3. For shift/rotate operations (`vmi_ROL`, `vmi_ROR`, `vmi_RCL`, `vmi_RCR`, `vmi_SHL`, `vmi_SHR` and `vmi_SAR`) the shift/rotate amount `b` is by default masked using `bits-1` before use. For example, if `bits` is 32 then the shift/rotate amount will be masked to the range 0..31 before use. This default behavior can be overridden by `vmimtSetShiftMask`.
4. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

4.19 vmimtBinopRC

Prototype

```
void vmimtBinopRC(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rd,
    Uns64      c,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a binary operation on register `rd` in the processor structure and constant `c`, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  carryFlag;        // carry flag
    Uns8  overflowFlag;     // overflow flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_CARRY      CPUX_OFFSET(carryFlag);
#define CPUX_OVERFLOW   CPUX_OFFSET(overflowFlag);

// vmiFlags specifying:
// 1. Use 'carryFlag' as CF input (if appropriate);
// 2. Set 'carryFlag' from CF output;
// 3. Set 'overflowFlag' from OF output.
const vmiFlags flagsCO = {
    cin : CPUX_CARRY,           // offset to carry in flag
    f   : {
        [vmi_CF] = CPUX_CARRY,  // offset to carry out flag
        [vmi_PF] = VMI_NOFLAG,  // parity flag not used
        [vmi_ZF] = VMI_NOFLAG,  // zero flag not used
        [vmi_SF] = VMI_NOFLAG,  // sign flag not used
        [vmi_OF] = CPUX_OVERFLOW// offset to overflow flag
    }
};

static void emitBinopRC(vmiBinop op, Uns32 rd, Uns32 c) {
    vmimtBinopRC(CPUX_GBITS, op, CPUX_REG(rd), c, &flagsCO);
}
```

Notes and Restrictions

1. Argument `bits` must be 8, 16, 32 or 64.
2. For target registers less than 64 bits wide, the unused most significant bits of `c` are silently discarded.
3. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
4. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

4.20 vmimtBinopRCR

Prototype

```
void vmimtBinopRCR(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rd,
    Uns64      c,
    vmiReg     rb,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a binary operation on constant `c` and register `rb` in the processor structure, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  carryFlag;        // carry flag
    Uns8  overflowFlag;     // overflow flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_CARRY      CPUX_OFFSET(carryFlag);
#define CPUX_OVERFLOW   CPUX_OFFSET(overflowFlag);

// vmiFlags specifying:
// 1. Use 'carryFlag' as CF input (if appropriate);
// 2. Set 'carryFlag' from CF output;
// 3. Set 'overflowFlag' from OF output.
const vmiFlags flagsCO = {
    cin : CPUX_CARRY,           // offset to carry in flag
    f   : {
        [vmi_CF] = CPUX_CARRY,  // offset to carry out flag
        [vmi_PF] = VMI_NOFLAG,  // parity flag not used
        [vmi_ZF] = VMI_NOFLAG,  // zero flag not used
        [vmi_SF] = VMI_NOFLAG,  // sign flag not used
        [vmi_OF] = CPUX_OVERFLOW// offset to overflow flag
    }
};

static void emitBinopRR(vmiBinop op, Uns32 rd, Uns32 c, Uns32 rb) {
    vmimtBinopRCR(CPUX_GBITS, op, CPUX_REG(rd), c, CPUX_REG(rb), &flagsCO);
```

```
}
```

Notes and Restrictions

1. Argument `bits` must be 8, 16, 32 or 64.
2. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
3. For shift/rotate operations (`vmi_ROL`, `vmi_ROR`, `vmi_RCL`, `vmi_RCR`, `vmi_SHL`, `vmi_SHR` and `vmi_SAR`) the shift/rotate amount `b` is by default masked using `bits-1` before use. For example, if `bits` is 32 then the shift/rotate amount will be masked to the range 0..31 before use. This default behavior can be overridden by `vmimtSetShiftMask`.
4. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

4.21 vmimtBinopRCC

Prototype

```
void vmimtBinopRCC(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rd,
    Uns64      c1,
    Uns64      c2,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a binary operation on constants `c1` and `c2`, writing the result to register `rd`.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  carryFlag;        // carry flag
    Uns8  overflowFlag;     // overflow flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_CARRY      CPUX_OFFSET(carryFlag);
#define CPUX_OVERFLOW   CPUX_OFFSET(overflowFlag);

// vmiFlags specifying:
// 1. Use ‘carryFlag’ as CF input (if appropriate);
// 2. Set ‘carryFlag’ from CF output;
// 3. Set ‘overflowFlag’ from OF output.
const vmiFlags flagsCO = {
    cin : CPUX_CARRY,           // offset to carry in flag
    f   : {
        [vmi_CF] = CPUX_CARRY,  // offset to carry out flag
        [vmi_PF] = VMI_NOFLAG,  // parity flag not used
        [vmi_ZF] = VMI_NOFLAG,  // zero flag not used
        [vmi_SF] = VMI_NOFLAG,  // sign flag not used
        [vmi_OF] = CPUX_OVERFLOW// offset to overflow flag
    }
};

static void emitBinopRCC(vmiBinop op, Uns32 rd, Uns32 c1, Uns32 c2) {
    vmimtBinopRCC(CPUX_GBITS, op, CPUX_REG(rd), c1, c2, &flagsCO);
}
```

Notes and Restrictions

1. Argument `bits` must be 8, 16, 32 or 64.
2. For target registers less than 64 bits wide, the unused most significant bits of `c1` and `c2` are silently discarded prior to use.
3. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
4. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

4.22 vmimtBinopRRC

Prototype

```
void vmimtBinopRRC(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rd,
    vmiReg     ra,
    Uns64      c,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a binary operation on register `ra` in the processor structure and constant `c`, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  carryFlag;        // carry flag
    Uns8  overflowFlag;     // overflow flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_CARRY      CPUX_OFFSET(carryFlag);
#define CPUX_OVERFLOW   CPUX_OFFSET(overflowFlag);

// vmiFlags specifying:
// 1. Use 'carryFlag' as CF input (if appropriate);
// 2. Set 'carryFlag' from CF output;
// 3. Set 'overflowFlag' from OF output.
const vmiFlags flagsCO = {
    cin : CPUX_CARRY,           // offset to carry in flag
    f   : {
        [vmi_CF] = CPUX_CARRY,  // offset to carry out flag
        [vmi_PF] = VMI_NOFLAG,  // parity flag not used
        [vmi_ZF] = VMI_NOFLAG,  // zero flag not used
        [vmi_SF] = VMI_NOFLAG,  // sign flag not used
        [vmi_OF] = CPUX_OVERFLOW// offset to overflow flag
    }
};
```

```
static void emitBinopRRC(vmiBinop op, Uns32 rd, Uns32 ra, Uns32 c) {
    vmimtBinopRRC(CPUX_GBITS, op, CPUX_REG(rd), CPUX_REG(ra), c, &flagsCO);
}
```

Notes and Restrictions

1. Argument `bits` must be 8, 16, 32 or 64.
2. For target registers less than 64 bits wide, the unused most significant bits of `c` are silently discarded prior to use.
3. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
4. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

4.23 vmimtMulopRRR

Prototype

```
void vmimtMulopRRR(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rdh,
    vmiReg     rdl,
    vmiReg     ra,
    vmiReg     rb,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a multiply operation on registers `ra` and `rb` in the processor structure. Argument `bits` gives the bit width of these two registers. The result of the multiply has size `bits*2`; the most significant part of the result (size `bits`) is assigned to processor register `rdh`, and the least significant part of the result (also of size `bits`) is assigned to processor register `rdl`.

Either `rdh` or `rdl` may have the special value `VMI_NOREG`; this indicates that this part of the result is to be discarded (not saved in a processor register). If `rdh` is `VMI_NOREG`, this function is equivalent to `vmiBinopRRR`.

Available binary operations are defined in `vmiTypes.h`:

```
typedef enum {
    vmi_IMUL =  8,      // d <- a * b (signed)
    vmi_MUL  =  9,      // d <- a * b (unsigned)
} vmiBinop;
```

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  carryFlag;        // carry flag
    Uns8  overflowFlag;     // overflow flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_CARRY      CPUX_OFFSET(carryFlag);
#define CPUX_OVERFLOW   CPUX_OFFSET(overflowFlag);

// vmiFlags specifying:
// 1. Use 'carryFlag' as CF input (if appropriate);
// 2. Set 'carryFlag' from CF output;
// 3. Set 'overflowFlag' from OF output.
const vmiFlags flagsCO = {
    cin : CPUX_CARRY,           // offset to carry in flag
    f   : {
        [vmi_CF] = CPUX_CARRY,  // offset to carry out flag
        [vmi_PF] = VMI_NOFLAG,  // parity flag not used
        [vmi_ZF] = VMI_NOFLAG,  // zero flag not used
        [vmi_SF] = VMI_NOFLAG,  // sign flag not used
        [vmi_OF] = CPUX_OVERFLOW// offset to overflow flag
    }
};

static void emitMulopRRR(
    vmiBinop op, // vmi_IMUL or vmi_MUL only
    Uns32    rdh,
    Uns32    rdl,
    Uns32    ra,
    Uns32    rb
) {
    vmimtMulopRRR(
        CPUX_GBITS, op,
        CPUX_REG(rdh), CPUX_REG(rdl),
        CPUX_REG(ra), CPUX_REG(rb),
        &flagsCO
    );
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. Operations other than `vmi_IMUL` and `vmi_MUL` are not supported by this function.

4.24 vmimtDivopRRR

Prototype

```
void vmimtDivopRRR(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rdd,
    vmiReg     rdr,
    vmiReg     rah,
    vmiReg     ral,
    vmiReg     rb,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a divide operation, producing both result and remainder. The dividend is of size `bits*2` and is constructed from the register pair `rah:ral`. The divisor is in register `rb`. The result of the division is assigned to processor register `rdd`. The remainder is assigned to processor register `rdr`.

Either `rdd` or `rdr` may have the special value `VMI_NOREG`; this indicates that the result or remainder (as appropriate) is to be discarded (not saved in a processor register).

Available binary operations are defined in `vmiTypes.h`:

```
typedef enum {
    vmi_IDIV,           // d <- a / b (signed)
    vmi_DIV,            // d <- a / b (unsigned)
} vmiBinop;
```

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitDivopRRR(
    vmiBinop op, // vmi_IDIV or vmi_DIV only
    Uns32    rdd,
    Uns32    rdr,
    Uns32    rah,
    Uns32    ral,
    Uns32    rb
```

```
) {
    vmimtDivopRRR(
        CPUX_GBITS, op,
        CPUX_REG(rdd), CPUX_REG(rdr),
        CPUX_REG(rah), CPUX_REG(ral),
        CPUX_REG(rb),
        NULL
    );
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. Operations other than `vmi_IDIV` and `vmi_DIV` are not supported by this function.
3. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero or integer overflow). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).

4.25 vmimtCompareRR

Prototype

```
void vmimtCompareRR(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       ra,
    vmiReg       rb,
    vmiReg       flag
);
```

Description

Emit code to compare the two processor registers `ra` and `rb` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by subtracting the second argument from the first using twos complement arithmetic and discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h`:

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                       (OF==1)
    vmi_COND_NO  = 1,   // no overflow                    (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)               (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)           (CF==0)
    vmi_COND_Z   = 4,   // zero                           (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)          (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                       (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)  (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)      (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)  (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                       (SF==1)
    vmi_COND_NS  = 9,   // not negative                   (SF==0)
    vmi_COND_P   = 10,  // parity even                    (PF==1)
    vmi_COND_NP  = 11,  // not parity even                (PF==0)
    vmi_COND_L   = 12,  // less (signed)                  (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)              (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)         (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)     (ZF==0 && SF==OF)
} vmiCondition;
```

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_BF         CPUX_OFFSET(branchFlag)
```

```
static void emitCompareRR(vmiCondition cond, Uns32 ra, Uns32 rb) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

4.26 vmimtCompareCR

Prototype

```
void vmimtCompareCR(
    Uns32        bits,
    vmiCondition cond,
    Uns64        c,
    vmiReg       rb,
    vmiReg       flag
);
```

Description

Emit code to compare constant `c` and register `rb` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by subtracting the second argument from the first using twos complement arithmetic and discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h`:

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                         (OF==1)
    vmi_COND_NO  = 1,   // no overflow                      (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)                 (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)             (CF==0)
    vmi_COND_Z   = 4,   // zero                             (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)            (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                         (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)    (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)        (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)    (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                         (SF==1)
    vmi_COND_NS  = 9,   // not negative                     (SF==0)
    vmi_COND_P   = 10,  // parity even                      (PF==1)
    vmi_COND_NP  = 11,  // not parity even                  (PF==0)
    vmi_COND_L   = 12,  // less (signed)                    (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)                (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)           (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)       (ZF==0 && SF==OF)
} vmiCondition;
```

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_BF         CPUX_OFFSET(branchFlag)

static void emitCompareCR(vmiCondition cond, Uns32 c, Uns32 rb) {
    vmimtCompareCR(CPUX_GBITS, cond, c, CPUX_REG(rb), CPUX_BF);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

4.27 vmimtCompareRC

Prototype

```
void vmimtCompareRC(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       ra,
    Uns64        c,
    vmiReg       flag
);
```

Description

Emit code to compare the processor register `ra` and constant `c` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by subtracting the second argument from the first using twos complement arithmetic and discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h`:

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                       (OF==1)
    vmi_COND_NO  = 1,   // no overflow                    (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)               (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)           (CF==0)
    vmi_COND_Z   = 4,   // zero                           (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)          (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                       (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)  (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)      (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)  (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                       (SF==1)
    vmi_COND_NS  = 9,   // not negative                   (SF==0)
    vmi_COND_P   = 10,  // parity even                    (PF==1)
    vmi_COND_NP  = 11,  // not parity even                (PF==0)
    vmi_COND_L   = 12,  // less (signed)                  (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)              (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)         (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)     (ZF==0 && SF==OF)
} vmiCondition;
```

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_BF         CPUX_OFFSET(branchFlag)
```

```
static void emitCompareRC(vmiCondition cond, Uns32 ra, Uns32 c) {
    vmimtCompareRC(CPUX_GBITS, cond, CPUX_REG(ra), c, CPUX_BF);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. For `bits` less than 64, the unused most significant bits of `c` are silently discarded.

4.28 vmimtTestRR

Prototype

```
void vmimtTestRR(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       ra,
    vmiReg       rb,
    vmiReg       flag
);
```

Description

Emit code to compare the two processor registers `ra` and `rb` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by a bitwise-and of the two arguments, discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h` (but note that CF and OF are always set to zero by this comparison):

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                       (OF==1)
    vmi_COND_NO  = 1,   // no overflow                    (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)               (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)           (CF==0)
    vmi_COND_Z   = 4,   // zero                           (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)          (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                       (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)  (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)      (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)  (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                       (SF==1)
    vmi_COND_NS  = 9,   // not negative                   (SF==0)
    vmi_COND_P   = 10,  // parity even                    (PF==1)
    vmi_COND_NP  = 11,  // not parity even                (PF==0)
    vmi_COND_L   = 12,  // less (signed)                  (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)              (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)         (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)     (ZF==0 && SF==OF)
} vmiCondition;
```

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_BF         CPUX_OFFSET(branchFlag)
```

```
static void emitTestRR(vmiCondition cond, Uns32 ra, Uns32 rb) {
    vmimtTestRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

4.29 vmimtTestCR

Prototype

```
void vmimtTestCR(
    Uns32        bits,
    vmiCondition cond,
    Uns64        c,
    vmiReg       rb,
    vmiReg       flag
);
```

Description

Emit code to compare constant `c` and register `rb` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by a bitwise-and of the two arguments, discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h` (but note that CF and OF are always set to zero by this comparison):

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                       (OF==1)
    vmi_COND_NO  = 1,   // no overflow                    (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)               (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)           (CF==0)
    vmi_COND_Z   = 4,   // zero                           (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)          (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                       (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)  (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)      (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)  (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                       (SF==1)
    vmi_COND_NS  = 9,   // not negative                   (SF==0)
    vmi_COND_P   = 10,  // parity even                    (PF==1)
    vmi_COND_NP  = 11,  // not parity even                (PF==0)
    vmi_COND_L   = 12,  // less (signed)                  (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)              (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)         (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)     (ZF==0 && SF==OF)
} vmiCondition;
```

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_BF         CPUX_OFFSET(branchFlag)
```

```
static void emitTestCR(vmiCondition cond, Uns32 c, Uns32 rb) {
    vmimtTestCR(CPUX_GBITS, cond, c, CPUX_REG(rb), CPUX_BF);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

4.30 vmimtTestRC

Prototype

```
void vmimtTestRC(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       ra,
    Uns64        c,
    vmiReg       flag
);
```

Description

Emit code to compare the processor register `ra` and constant `c` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by a bitwise-and of the two arguments, discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h` (but note that CF and OF are always set to zero by this comparison):

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                       (OF==1)
    vmi_COND_NO  = 1,   // no overflow                    (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)               (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)           (CF==0)
    vmi_COND_Z   = 4,   // zero                           (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)          (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                       (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)  (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)      (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)  (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                       (SF==1)
    vmi_COND_NS  = 9,   // not negative                   (SF==0)
    vmi_COND_P   = 10,  // parity even                    (PF==1)
    vmi_COND_NP  = 11,  // not parity even                (PF==0)
    vmi_COND_L   = 12,  // less (signed)                  (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)              (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)         (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)     (ZF==0 && SF==OF)
} vmiCondition;
```

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_BF         CPUX_OFFSET(branchFlag)
```

```
static void emitTestRC(vmiCondition cond, Uns32 ra, Uns32 c) {
    vmimtTestRC(CPUX_GBITS, cond, CPUX_REG(ra), c, CPUX_BF);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. For `bits` less than 64, the unused most significant bits of `c` are silently discarded.

4.31 vmimtSetShiftMask

Prototype

```
void vmimtSetShiftMask(Uns8 mask);
```

Description

The shift/rotate amount for shift/rotate binops is normally masked to `bits-1`, where `bits` is the operand size. It is possible to override this default shift mask with any mask in the range 1..255 by immediately preceding the binop with a call to `vmimtSetShiftMask` giving the required shift mask.

Example 1

On the Intel x86 processor, all shifts, including byte and word size shifts, are masked to the range 0..31. This behavior can be specified as follows:

```
vmimtSetShiftMask(31);
vmimtBinopRR(CPUX_GBITS, vmi_ROR, CPUX_REG(rd), CPUX_REG(ra), 0);
```

Example 2

On ARM processors, the shift amount is byte sized. This behavior can be specified as follows:

```
vmimtSetShiftMask(255);
vmimtBinopRRR(CPUX_GBITS, vmi_ROR, CPUX_REG(rd), CPUX_REG(ra) , CPUX_REG(rb), 0);
```

Notes and Restrictions

1. The call to `vmimtSetShiftMask` must immediately precede the `vmimtBinop*` call to which the shift mask must be applied.
2. If `vmimtSetShiftMask` is called before a `vmimtBinop*` call that is not one of the shift/rotate opcodes (`vmi_ROR`, `vmi_ROL`, `vmi_RCL`, `vmi_RCR`, `vmi_SHL`, `vmi_SHR` or `vmi_SAR`) it is ignored.
3. If `vmimtSetShiftMask` is called before any VMI morph time interface function that is not a `vmimtBinop*` call, it is ignored.

5 Memory Operations

This section describes emission functions for memory operations (loads and stores).

In VMI versions prior to 4.3.0, only the *current processor data domain* could be targeted by load and store primitives.

From VMI version 4.3.0 onwards *any domain* can be targeted by a load or store primitive. This is useful in situations where loads or stores are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

From VMI version 6.2.0 onwards, load and store operations can handle any operand size from 1 to 128 bytes.

From VMI version 6.3.0 onwards, load and store functions can have *constraints* applied to their operation, as documented below.

5.1 Memory Constraints

The action of functions that emit code to load and store from memory can be refined using a *memory constraint*. Available constraints are defined by the `memConstraint` type in `vmiTypes.h`:

```
typedef enum memConstraintE {
    MEM_CONSTRAINT_NONE      = 0x0, // no constraint
    MEM_CONSTRAINT_ALIGNED   = 0x1, // access must not be misaligned
    MEM_CONSTRAINT_NO_DEVICE = 0x2, // access must not be to device memory
                                    // (with privilege MEM_PRIV_DEVICE)
} memConstraint;
```

The `memConstraint` type is a bitmask, so constraints may be combined.

Constraint `MEM_CONSTRAINT_ALIGNED` specifies that the load or store operation must use an address that is aligned to the size of the data element. If the address is misaligned, a simulated exception will be taken using the `rdAlignExceptCB` or `wrAlignExceptCB` callback functions specified in the attribute structure of the processor. See the *Imperas Processor Modeling Guide* for more information about the attributes structure.

Constraint `MEM_CONSTRAINT_NO_DEVICE` specifies that the load or store operation must not access a memory region of *device* type. Device type regions are specified by using a `memPriv` including `MEM_PRIV_DEVICE` when a memory region is defined using `vmirtAliasMemoryVM` or modified using `vmirtProtectMemory` (refer to the *VMI Run Time Function Reference* for more information about these functions). If the memory region is of device type, a simulated exception will be taken using the `rdDeviceExceptCB` or `wrDeviceExceptCB` callback functions specified in the attribute structure of the processor. See the *Imperas Processor Modeling Guide* for more information about the attributes structure.

5.2 vmimtStoreRRO

Prototype

```
void vmimtStoreRRO(
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    vmiReg        rb,
    memEndian     endian,
    memConstraint constraint
);
```

Description

Emit code to store the value of processor register `rb` (of size `bits`) to an address calculated from the value of `ra` plus the fixed displacement `offset`. The size of the calculated address (and the size in bits of the address register in the processor structure) is the specified with the processor is created (in an OVP platform, this is the `addressBits` parameter to `icmCreateProcessor`; in the Imperas Simulator (`imperas.exe`) product, this is the `addresswidth` attribute of the `DATA busmasterport` entity in the XML processor description).

If `ra` has the value `VMI_NOREG`, the address at which to store is `offset` alone. If the store address calculated by `ra+offset` is invalid (the platform defines no writable entity at that address), the simulator will call the store privilege exception handler (`wrPrivExceptCB`) defined for the processor model.

This function emits code that targets the current processor data domain. From VMI version 4.3.0, it is possible to specify *any* target domain – see related function `vmimtStoreRRODomain`.

Argument `bits` can be any multiple of eight between 8 (i.e. 1 byte) and 1024 (i.e. 128 bytes). Behavior is different when `bits` is 8, 16, 32 or 64 to all other cases; see the following subsections.

`bits` equal to 8, 16, 32 or 64

If `endian` is `MEM_ENDIAN_LITTLE`, then the store is performed little-endian. If it is `MEM_ENDIAN_BIG`, the store is performed big-endian.

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the store address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`. If the address is misaligned, the simulator will first call any store address snap handler (`wrSnapCB`) defined for the processor model. If there is no store address snap handler, or the store address snap handler returns zero (indicating no address snap is to be performed), the simulator will then call any store alignment exception handler (`wrAlignExceptCB`) defined for the processor model.

`bits` not equal to 8, 16, 32 or 64

The `endian` argument is ignored. Data is always stored little-endian.

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the store address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`; if the implied byte size is not a power of two, then the next smallest power of two is chosen. For example, if `bits` is 80 (implying a 10-byte store) then the simulator will verify that the address is aligned to an 8-byte boundary. Actions for misaligned address are the same as for `bits` of 8, 16, 32 or 64, as described above.

When the store is executed, it is broken down into individual transactions of 1, 2, 4 or 8 bytes in size, starting with the largest possible size. For example, a 10-byte store will be broken down into an 8-byte store followed by a 2-byte store.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitStoreRRO(Addr offset, Uns32 ra, Uns32 rb) {
    vmimtStoreRRO(
        CPUX_GBITS, offset, CPUX_REG(ra), CPUX_REG(rb),
        MEM_ENDIAN_BIG, MEM_CONSTRAINT_ALIGNED
    );
}
```

Notes and Restrictions

1. When the the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`).
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024..

5.3 vmimtStoreRCO

Prototype

```
void vmimtStoreRCO(
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    Uns64         c,
    memEndian     endian,
    memConstraint constraint
);
```

Description

Emit code to store the constant value `c` (of size `bits`) to an address calculated from the value of `ra` plus the fixed displacement `offset`. The size of the calculated address (and the size in bits of the address register in the processor structure) is the specified with the processor is created (in an OVP platform, this is the `addressBits` parameter to `icmCreateProcessor`; in the Imperas Simulator (`imperas.exe`) product, this is the `addresswidth` attribute of the `DATA busmasterport` entity in the XML processor description).

If `ra` has the value `VMI_NOREG`, the address at which to store is `offset` alone. If the store address calculated by `ra+offset` is invalid (the platform defines no writable entity at that address), the simulator will call the store privilege exception handler (`wrPrivExceptCB`) defined for the processor model.

This function emits code that targets the current processor data domain. From VMI version 4.3.0, it is possible to specify *any* target domain – see related function `vmimtStoreRCODomain`.

Argument `bits` can be any multiple of eight between 8 (i.e. 1 byte) and 1024 (i.e. 128 bytes). Behavior is different when `bits` is 8, 16, 32 or 64 to all other cases; see the following subsections.

`bits` equal to 8, 16, 32 or 64

If `endian` is `MEM_ENDIAN_LITTLE`, then the constant value is stored little-endian. If it is `MEM_ENDIAN_BIG`, the constant value is stored big-endian.

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the store address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`. If the address is misaligned, the simulator will first call any store address snap handler (`wrSnapCB`) defined for the processor model. If there is no store address snap handler, or the store address snap handler returns zero (indicating no address snap is to be performed), the simulator will then call any store alignment exception handler (`wrAlignExceptCB`) defined for the processor model.

bits not equal to 8, 16, 32 or 64

If endian is MEM_ENDIAN_LITTLE, then the constant value is stored little-endian. If it is MEM_ENDIAN_BIG, the constant value is stored big-endian.

If constraint does not include MEM_CONSTRAINT_ALIGNED, the simulator does not perform alignment checking for the store address. If constraint includes MEM_CONSTRAINT_ALIGNED, the simulator verifies that the address calculated by ra+offset is a multiple of the byte size implied by bits; if the implied byte size is not a power of two, then the next smallest power of two is chosen. For example, if bits is 80 (implying a 10-byte store) then the simulator will verify that the address is aligned to an 8-byte boundary. Actions for misaligned address are the same as for bits of 8, 16, 32 or 64, as described above.

When the store is executed, it is broken down into individual transactions of 1, 2, 4 or 8 bytes in size, starting with the largest possible size. For example, a 10-byte store will be broken down into an 8-byte store followed by a 2-byte store.

When bits is greater than 64, the constant value is written repeatedly into each 64-bit (8 byte) element of the written memory.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitStoreRCO(Addr offset, Uns32 ra, Uns32 c) {
    vmimtStoreRCO(
        CPUX_GBITS, offset, CPUX_REG(ra), c,
        MEM_ENDIAN_BIG, MEM_CONSTRAINT_ALIGNED
    );
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register ra in the processor structure is a 32-bit unsigned (Uns32).
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register ra in the processor structure is a 64-bit unsigned (Uns64).
3. Data address bus widths greater than 64 bits are not supported.
4. bits must be a multiple of 8 in the range 8 to 1024..
5. For bits less than 64, the unused most significant bits of c are silently discarded.

5.4 vmimtLoadRRO

Prototype

```
void vmimtLoadRRO(
    Uns32         destBits,
    Uns32         memBits,
    Addr          offset,
    vmiReg        rd,
    vmiReg        ra,
    memEndian     endian,
    Bool          signExtend,
    memConstraint constraint
);
```

Description

Emit code to load the value of processor register `rd` (of size `destBits`) from an address calculated from the value of `ra` plus the fixed displacement `offset`. The size of the calculated address (and the size in bits of the address register in the processor structure) is the specified with the processor is created (in an OVP platform, this is the `addressBits` parameter to `icmCreateProcessor`; in the Imperas Simulator (`imperas.exe`) product, this is the `addresswidth` attribute of the `DATA busmasterport` entity in the XML processor description).

If `ra` has the value `VMI_NOREG`, the address from which to load is `offset` alone. If the load address calculated by `ra+offset` is invalid (the platform defines no readable entity at that address), the simulator will call the load privilege exception handler (`rdPrivExceptCB`) defined for the processor model.

This function emits code that targets the current processor data domain. From VMI version 4.3.0, it is possible to specify *any* target domain – see related function `vmimtLoadRRODomain`.

Arguments `memBits` and `destBits` must be a multiple of eight between 8 (i.e. 1 byte) and 1024 (i.e. 128 bytes). Behavior is different when both values are 8, 16, 32 or 64 to all other cases; see the following subsections.

`destBits` and `memBits` both equal to 8, 16, 32 or 64

The size of the value to load from memory is given by `memBits`, which must be less than or equal to `destBits`. If `memBits` is less than `destBits`, the value will be sign-extended to `destBits` (if `signExtend` is `True`) or zero-extended (if `signExtend` is `False`).

If `endian` is `MEM_ENDIAN_LITTLE`, then the load is performed little-endian. If it is `MEM_ENDIAN_BIG`, the load is performed big-endian. Any required sign extension is done after endian swapping.

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the load address. If `constraint` includes

MEM_CONSTRAINT_ALIGNED, the simulator verifies that the address calculated by ra+offset is a multiple of the byte size implied by bits. If the address is misaligned, the simulator will first call any load address snap handler (rdSnapCB) defined for the processor model. If there is no load address snap handler, or the load address snap handler returns zero (indicating no address snap is to be performed), the simulator will then call any load alignment exception handler (rdAlignExceptCB) defined for the processor model.

destBits and memBits not equal to 8, 16, 32 or 64

The size of the value to load from memory is given by memBits, which must be equal to destBits. The signExtend argument is ignored

The endian argument is ignored. Data is always loaded little-endian.

If constraint does not include MEM_CONSTRAINT_ALIGNED, the simulator does not perform alignment checking for the load address. If constraint includes MEM_CONSTRAINT_ALIGNED, the simulator verifies that the address calculated by ra+offset is a multiple of the byte size implied by bits; if the implied byte size is not a power of two, then the next smallest power of two is chosen. For example, if bits is 80 (implying a 10-byte load) then the simulator will verify that the address is aligned to an 8-byte boundary. Actions for misaligned address are the same as for bits of 8, 16, 32 or 64, as described above.

When the load is executed, it is broken down into individual transactions of 1, 2, 4 or 8 bytes in size, starting with the largest possible size. For example, a 10-byte load will be broken down into an 8-byte load followed by a 2-byte load.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitLoadRRO(Addr offset, Uns32 memBits, Uns32 rd, Uns32 ra) {
    vmimtLoadRRO(
        CPUX_GBITS, memBits, offset, CPUX_REG(rd), CPUX_REG(ra),
        MEM_ENDIAN_BIG, True, MEM_CONSTRAINT_ALIGNED
    );
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register ra in the processor structure is a 32-bit unsigned (Uns32).

2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`).
3. Data address bus widths greater than 64 bits are not supported.
4. `destBits` and `memBits` `bits` must be a multiple of 8 in the range 8 to 1024.
5. `destBits` must be equal to or greater than `memBits` (if both are 8, 16, 32 or 64). Both values must be equal in all other cases.

5.5 vmimtTryStoreRC

Prototype

```
void vmimtTryStoreRC(
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    memConstraint constraint
);
```

Description

Emit code to trigger any exceptions that would be generated if a store of the passed bit size was made to the passed address. If no exceptions would be generated, the function has no effect.

If `ra` has the value `VMI_NOREG`, the address at which to store is `offset` alone. If the store address calculated by `ra+offset` is invalid (the platform defines no writable entity at that address), the simulator will call the store privilege exception handler (`wrPrivExceptCB`) defined for the processor model.

This function emits code that targets the current processor data domain. From VMI version 4.3.0, it is possible to specify *any* target domain – see related function `vmimtTryStoreRCDomain`.

Argument `bits` can be any multiple of eight between 8 (i.e. 1 byte) and 1024 (i.e. 128 bytes). Behavior is different when `bits` is 8, 16, 32 or 64 to all other cases; see the following subsections.

`bits` equal to 8, 16, 32 or 64

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the store address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`. If the address is misaligned, the simulator will first call any store address snap handler (`wrSnapCB`) defined for the processor model. If there is no store address snap handler, or the store address snap handler returns zero (indicating no address snap is to be performed), the simulator will then call any store alignment exception handler (`wrAlignExceptCB`) defined for the processor model.

`bits` not equal to 8, 16, 32 or 64

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the store address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`; if the implied byte size is not a power of two, then the next smallest power of two is chosen. For example, if `bits` is 80 (implying a 10-byte store) then the simulator will verify that the address is aligned to an

8-byte boundary. Actions for misaligned address are the same as for `bits` of 8, 16, 32 or 64, as described above.

Example

The example MIPS model uses this function to implement the `SC` instruction. This is defined as producing exceptions resulting from a bad address even if the subsequent store is not performed:

```
// Emit code for SC instruction
static void emitSC(mips32InstructionInfoP info, mips32P mips32) {

    Uns32     bits     = MIPS32_GPR_BITS;
    vmiLabelP ok       = vmimtNewLabel();
    vmiLabelP bad      = vmimtNewLabel();
    vmiLabelP done     = vmimtNewLabel();
    vmiReg    rt       = getR1(info);
    vmiReg    tempFlag = MIPS32_TEMPFLAG(0);

    // calculate LL load address (in MIPS32_TEMP1);
    doLLSCAddressCalculation(info);

    // do alignment/permissions check first
    vmimtTryStoreRC(bits, 0, MIPS32_TEMP1, MEM_CONSTRAINT_ALIGNED);

    // skip the store if LLbit is clear
    vmimtCondJumpLabel(MIPS32_LLBIT, False, bad);

    // compare the SC and LL virtual addresses
    vmimtCompareRR(
        MIPS32_GPR_BITS,
        vmi_COND_Z,
        MIPS32_COP0_LLAddr(mips32),
        MIPS32_TEMP1,
        tempFlag
    );

    // do the store if they match
    vmimtCondJumpLabel(tempFlag, True, ok);

    // clear result register and skip the store
    vmimtInsertLabel(bad);
    vmimtMoveRC(bits, rt, 0);
    vmimtUncondJumpLabel(done);

    // do the store and set result register
    vmimtInsertLabel(ok);
    emitStore(info, mips32);
    vmimtMoveRC(bits, rt, 1);

    // here on completion
    vmimtInsertLabel(done);
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`).
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.

5.6 vmimtTryLoadRC

Prototype

```
void vmimtTryLoadRC(
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    memConstraint constraint
);
```

Description

Emit code to trigger any exceptions that would be generated if a load of the passed bit size was made to the passed address. If no exceptions would be generated, the function has no effect.

If `ra` has the value `VMI_NOREG`, the address from which to load is `offset` alone. If the load address calculated by `ra+offset` is invalid (the platform defines no readable entity at that address), the simulator will call the load privilege exception handler (`rdPrivExceptCB`) defined for the processor model.

This function emits code that targets the current processor data domain. From VMI version 4.3.0, it is possible to specify *any* target domain – see related function `vmimtTryLoadRCDomain`.

Argument `bits` can be any multiple of eight between 8 (i.e. 1 byte) and 1024 (i.e. 128 bytes). Behavior is different when `bits` is 8, 16, 32 or 64 to all other cases; see the following subsections.

`bits` equal to 8, 16, 32 or 64

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the load address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`. If the address is misaligned, the simulator will first call any load address snap handler (`rdSnapCB`) defined for the processor model. If there is no load address snap handler, or the load address snap handler returns zero (indicating no address snap is to be performed), the simulator will then call any load alignment exception handler (`rdAlignExceptCB`) defined for the processor model.

`bits` not equal to 8, 16, 32 or 64

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the load address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`; if the implied byte size is not a power of two, then the next smallest power of two is chosen. For example, if `bits` is 80 (implying a 10-byte load) then the simulator will verify that the address is aligned to an

8-byte boundary. Actions for misaligned address are the same as for `bits` of 8, 16, 32 or 64, as described above.

Example

The example MIPS model uses this function to implement the `SYNCI` instruction. This is defined as producing exceptions resulting from a bad address, but otherwise having no effect:

```
// Emit code for SYNCI instruction - exceptions resulting from a bad address
// must be triggered, but otherwise no action
static void emitSYNCI(mips32InstructionInfoP info) {
    vmimtTryLoadRC(8, info->c, getR1(info), MEM_CONSTRAINT_ALIGNED);
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`).
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.

5.7 vmimtStoreRRODomain

Prototype

```
void vmimtStoreRRODomain(
    memDomainP    domain,
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    vmiReg        rb,
    memEndian     endian,
    memConstraint constraint
);
```

Description

This function is similar to `vmimtStoreRRO`, except for the `domain` argument, which allows the memory domain for the store to be specified.

If `domain` is `NULL`, the store is directed to the *current processor data domain* (in other words, behavior is identical to `vmimtStoreRRO`).

If domain is non-`NULL`, the store is directed to the *specified domain*.

This function is useful in situations where stores are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

See the description of `vmimtStoreRRO` for details of other arguments to this function.

Example

This example is from the OVP ARM model. This processor has *translating* load/store instructions that allow the user address space to be read or written from privileged mode. A utility functions selects either the user address space or the default address space, depending on attributes of the decoded instruction:

```
inline static memDomainP getDomain(armMorphStateP state) {
    return doTranslate(state) ? state->arm->dds.vmUser : 0;
}
```

This domain is then specified to `vmimtStoreRRODomain`:

```
void armEmitStoreRRO(
    armMorphStateP state,
    Uns32          bits,
    Uns32          offset,
    vmiReg         ra,
    vmiReg         rb
) {
    memDomainP    domain     = getDomain(state);
    memEndian     endian     = getEndian(state, bits);
    memConstraint constraint = getConstraint(state, bits);
    vmimtStoreRRODomain(domain, bits, offset, ra, rb, endian, constraint);
}
```

Notes and Restrictions

1. When the the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`).
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.

5.8 vmimtStoreRCODomain

Prototype

```
void vmimtStoreRCODomain(
    memDomainP    domain,
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    Uns64         c,
    memEndian     endian,
    memConstraint constraint
);
```

Description

This function is similar to `vmimtStoreRCO`, except for the `domain` argument, which allows the memory domain for the store to be specified.

If `domain` is `NULL`, the store is directed to the *current processor data domain* (in other words, behavior is identical to `vmimtStoreRCO`).

If domain is non-`NULL`, the store is directed to the *specified domain*.

This function is useful in situations where stores are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

See the description of `vmimtStoreRCO` for details of other arguments to this function.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitStoreRCODomain(memDomainP domain, Addr offset, Uns32 ra, Uns32 c) {
    vmimtStoreRCODomain(
        domain, CPUX_GBITS, offset, CPUX_REG(ra), c,
        MEM_ENDIAN_BIG, MEM_CONSTRAINT_ALIGNED
    );
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`).

2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`).
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.
5. For `bits` less than 64, the unused most significant bits of `c` are silently discarded.

5.9 vmimtLoadRRODomain

Prototype

```
void vmimtLoadRRODomain(
    memDomainP    domain,
    Uns32         destBits,
    Uns32         memBits,
    Addr          offset,
    vmiReg        rd,
    vmiReg        ra,
    memEndian     endian,
    Bool          signExtend,
    memConstraint constraint
);
```

Description

This function is similar to `vmimtLoadRRO`, except for the `domain` argument, which allows the memory domain for the store to be specified.

If `domain` is `NULL`, the load is directed to the *current processor data domain* (in other words, behavior is identical to `vmimtLoadRRO`).

If domain is non-`NULL`, the load is directed to the *specified domain*.

This function is useful in situations where loads are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

See the description of `vmimtLoadRRO` for details of other arguments to this function.

Example

This example is from the OVP ARM model. This processor has *translating* load/store instructions that allow the user address space to be read or written from privileged mode. A utility functions selects either the user address space or the default address space, depending on attributes of the decoded instruction:

```
inline static memDomainP getDomain(armMorphStateP state) {
    return doTranslate(state) ? state->arm->dds.vmUser : 0;
}
```

This domain is then specified to `vmimtLoadRRODomain`:

```
void armEmitLoadRRO(
    armMorphStateP state,
    Uns32          destBits,
    Uns32          memBits,
    Uns32          offset,
    vmiReg         rd,
    vmiReg         ra,
    Bool           signExtend,
    Bool           isReturn
) {
```

```
    memDomainP    domain     = getDomain(state);
    memEndian     endian     = getEndian(state, memBits);
    memConstraint constraint = getConstraint(state, bits);

    // emit single load
    vmimtLoadRRODomain(
        domain, destBits, memBits, offset, rd, ra, endian, signExtend, constraint
    );
    setVariable(state, rd, isReturn);
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`).
3. Data address bus widths greater than 64 bits are not supported.
4. `destBits` and `memBits` `bits` must be a multiple of 8 in the range 8 to 1024.
5. `destBits` must be equal to or greater than `memBits` (if both are 8, 16, 32 or 64). Both values must be equal in all other cases.

5.10 vmimtTryStoreRCDomain

Prototype

```
void vmimtTryStoreRCDomain(
    memDomainP    domain,
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    memConstraint constraint
);
```

Description

This function is similar to `vmimtTryStoreRC`, except for the `domain` argument, which allows the memory domain for the store to be specified.

If `domain` is `NULL`, the store is directed to the *current processor data domain* (in other words, behavior is identical to `vmimtTryStoreRC`).

If domain is non-`NULL`, the store is directed to the *specified domain*.

This function is useful in situations where stores are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

See the description of `vmimtTryStoreRC` for details of other arguments to this function.

Example

This example is from the OVP ARM model. This processor has *translating* load/store instructions that allow the user address space to be read or written from privileged mode. A utility functions selects either the user address space or the default address space, depending on attributes of the decoded instruction:

```
inline static memDomainP getDomain(armMorphStateP state) {
    return doTranslate(state) ? state->arm->dds.vmUser : 0;
}
```

This domain is then specified to `vmimtTryStoreRCDomain`:

```
void armEmitTryStoreRC(
    armMorphStateP state,
    Uns32          bits,
    Addr           offset,
    vmiReg         ra
) {
    memDomainP domain = getDomain(state);
    vmimtTryStoreRCDomain(domain, bits, offset, ra, getConstraint(state, bits));
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`).
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.

5.11 vmimtTryLoadRCDomain

Prototype

```
void vmimtTryLoadRCDomain(
    memDomainP    domain,
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    memConstraint constraint
);
```

Description

This function is similar to `vmimtTryLoadRC`, except for the `domain` argument, which allows the memory domain for the store to be specified.

If `domain` is `NULL`, the load is directed to the *current processor data domain* (in other words, behavior is identical to `vmimtTryLoadRC`).

If domain is non-`NULL`, the load is directed to the *specified domain*.

This function is useful in situations where loads are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

See the description of `vmimtTryLoadRC` for details of other arguments to this function.

Example

This example is from the OVP ARM model. This processor has *translating* load/store instructions that allow the user address space to be read or written from privileged mode. A utility functions selects either the user address space or the default address space, depending on attributes of the decoded instruction:

```
inline static memDomainP getDomain(armMorphStateP state) {
    return doTranslate(state) ? state->arm->dds.vmUser : 0;
}
```

This domain is then specified to `vmimtTryLoadRCDomain`:

```
void armEmitTryLoadRC(
    armMorphStateP state,
    Uns32          bits,
    Addr           offset,
    vmiReg         ra
) {
    memDomainP domain = getDomain(state);
    vmimtTryLoadRCDomain(domain, bits, offset, ra, getConstraint(state, bits));
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`).
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.

6 Control Flow Operations

This section describes emission functions for control flow operations: inter-instruction and intra-instruction unconditional and conditional jumps.

Inter-instruction jumps correspond to control transfers in the simulated processor instruction set.

Intra-instruction jumps are required when the translation of a simulated instruction requires loops or conditional branches (although if the control behavior is complicated, it is usually easier and more efficient to use a run-time call instead - see `vmimtCall` and related functions).

6.1 vmimtSetAddressMask

Prototype

```
void vmimtSetAddressMask(Uns64 mask);
```

Description

This call can be used *immediately prior* to any `vmimt*Jump*` call to specify masking of target and link addresses. For example specifying an address mask of `-2` will cause the least significant bit of target and return addresses to be ignored.

Example

On the MIPS processor, indirect jumps can switch to 16-bit mode, determined by whether the LSB of the target address is set. `vmimtSetAddressMask` is used to ensure that the LSB is not interpreted as part of the target address:

```
// Emit indirect jump, possibly with link
static void emitJumpR(mips32InstructionInfoP info, mips32P mips32, Bool link) {

    mips32P       vpe      = VPE_FOR_TC(mips32);
    vmiReg        rtgt     = link ? getR2(info) : getR1(info);
    Bool          isReturn = (VMI_REG_INDEX(rtgt)==MIPS32_REG_RA_INDEX);
    vmiJumpHint   hint     = link     ? vmi_JH_CALL :
                             isReturn ? vmi_JH_RETURN : vmi_JH_NONE;
    vmiPostSlotFn slotCB;

    // set up the link return address
    if(link) {
        emitSetLinkAddress(getR1(info));
    }

    // test for possible ISA mode switch if MIPS16e is available
    if(!COP0_FIELD(vpe, Config1, CA)) {

        // MIPS16e not available: target address is in rs, no slot callback
        // required
        slotCB = 0;

    } else {

        // MIPS16e available: LSB is used to control new mode
        slotCB = postDelaySlotJR;

        // set flag to indicate new mode - used in postDelaySlotJR
        vmimtBinopRRC(8, vmi_AND, MIPS32_MIPS16E_MODE, rtgt, 1, 0);

        // mask off LSB of target address
        vmimtSetAddressMask(~1);
    }

    // do the jump with one delay slot instruction
    vmimtUncondJumpRegDelaySlot(1, 0, rtgt, VMI_NOREG, hint, slotCB);
}
```

Notes and Restrictions

1. If `vmimtSetAddressMask` is not called immediately prior to a `vmimt*Jump*` function, it is ignored.

6.2 vmimtUncondJump

Prototype

```
void vmimtUncondJump(
    Addr        linkPC,
    Addr        toAddress,
    vmiReg      linkReg,
    vmiJumpHint hint
);
```

Description

Emit code to perform an unconditional inter-instruction branch to `toAddress`.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` as the branch is taken – this allows *branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum {
    vmi_JH_NONE      = 0,   // no jump hint
    vmi_JH_CALL      = 1,   // jump is a call
    vmi_JH_CALLINT   = 2,   // jump is an interrupt call
    vmi_JH_RETURN    = 3,   // jump is a return
    vmi_JH_RETURNINT = 4    // jump is an interrupt return
} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_LINKREG    CPUX_REG(9)

static void emitCall(Addr thisPC, Addr target) {
    vmimtUncondJump(thisPC+4, target, CPUX_LINKREG, vmi_JH_CALL);
}
```

Notes and Restrictions

1. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).

2. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).

6.3 vmimtUncondJumpDelaySlot

Prototype

```
void vmimtUncondJumpDelaySlot(
    Uns32         slotOps,
    Addr          linkPC,
    Addr          toAddress,
    vmiReg        linkReg,
    vmiJumpHint   hint,
    vmiPostSlotFn slotCB
);
```

Description

Emit code to perform an unconditional inter-instruction branch to `toAddress` with `slotOps` subsequent delay slot instructions, which will be executed prior to taking the branch.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` as the branch is taken – this allows *branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum {
    vmi_JH_NONE      = 0,   // no jump hint
    vmi_JH_CALL      = 1,   // jump is a call
    vmi_JH_CALLINT   = 2,   // jump is an interrupt call
    vmi_JH_RETURN    = 3,   // jump is a return
    vmi_JH_RETURNINT = 4    // jump is an interrupt return
} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken (if, for example, if there is a simulated exception in the delay slot instruction) the function is *not* called. The callback function is passed the processor as its only argument.

If `slotOps` and `slotCB` are 0, this function is equivalent to `vmiUncondJump`.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_LINKREG    CPUX_REG(9)

static void emitCallDS(Addr thisPC, Addr target) {
    vmimtUncondJumpDelaySlot(1, thisPC+4, target, CPUX_LINKREG, vmi_JH_CALL, 0);
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to 0 or 1.
2. When the the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
3. When the the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).
4. `vmimtUncondJumpDelaySlot` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

6.4 vmimtUncondJumpReg

Prototype

```
void vmimtUncondJumpReg(
    Addr         linkPC,
    vmiReg       toReg,
    vmiReg       linkReg,
    vmiJumpHint hint
);
```

Description

Emit code to perform an unconditional indirect jump to the address in processor register `toReg`. This function is typically used to generate code for calls through function pointers and return instructions.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` as the branch is taken – this allows *branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum {
    vmi_JH_NONE      = 0,   // no jump hint
    vmi_JH_CALL      = 1,   // jump is a call
    vmi_JH_CALLINT   = 2,   // jump is an interrupt call
    vmi_JH_RETURN    = 3,   // jump is a return
    vmi_JH_RETURNINT = 4    // jump is an interrupt return
} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_LINKREG    CPUX_REG(9)

static void emitCallIndirect(Addr thisPC, Uns32 toReg) {
    vmimtUncondJumpReg(thisPC+4, CPUX_REG(toReg), CPUX_LINKREG, vmi_JH_CALL);
}
static void emitReturn(void) {
    vmimtUncondJumpReg(0 /*unused*/, CPUX_LINKREG, VMI_NOREG, vmi_JH_RETURN);
}
```

Notes and Restrictions

1. When the instruction address bus width is 32 bits or less, the appropriate type for the address registers `toReg` and `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address registers `toReg` and `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).

6.5 vmimtUncondJumpRegDelaySlot

Prototype

```
void vmimtUncondJumpRegDelaySlot(
    Uns32         slotOps,
    Addr          linkPC,
    vmiReg        toReg,
    vmiReg        linkReg,
    vmiJumpHint   hint,
    vmiPostSlotFn slotCB
);
```

Description

Emit code to perform an unconditional indirect jump to the address in processor register `toReg` with `slotOps` subsequent delay slot instructions, which will be executed prior to taking the branch. This function is typically used to generate code for calls through function pointers and return instructions.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` as the branch is taken – this allows *branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum {
    vmi_JH_NONE      = 0,   // no jump hint
    vmi_JH_CALL      = 1,   // jump is a call
    vmi_JH_CALLINT   = 2,   // jump is an interrupt call
    vmi_JH_RETURN    = 3,   // jump is a return
    vmi_JH_RETURNINT = 4    // jump is an interrupt return
} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken (if, for example, if there is a simulated exception in the delay slot instruction) the function is *not* called. The callback function is passed the processor as its only argument.

If `slotOps` and `slotCB` are 0, this function is equivalent to `vmiUncondJumpReg`.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_LINKREG    CPUX_REG(9)

static void emitCallIndirectDS(Addr thisPC, Uns32 toReg) {
    vmimtUncondJumpRegDelaySlot(
        1, thisPC+4, CPUX_REG(toReg), CPUX_LINKREG, vmi_JH_CALL, 0
    );
}
static void emitReturnDS(void) {
    vmimtUncondJumpRegDelaySlot(
        1, 0 /*unused*/, CPUX_LINKREG, VMI_NOREG, vmi_JH_RETURN, 0
    );
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to 0 or 1.
2. When the the instruction address bus width is 32 bits or less, the appropriate type for the address registers `toReg` and `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
3. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address registers `toReg` and `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).
4. If register `toReg` is updated by delay slot instructions, the target address is not affected (it is the original value of `toReg`).
5. `vmimtUncondJumpRegDelaySlot` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

6.6 vmimtCondJump

Prototype

```
void vmimtCondJump(
    vmiReg      flag,
    Bool        jumpIfTrue,
    Addr        linkPC,
    Addr        toAddress,
    vmiReg      linkReg,
    vmiJumpHint hint
);
```

Description

Emit code to perform a conditional inter-instruction branch. The processor `flag` register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to target address `toAddress` if the value of the `flag` register is non-zero and execution will continue with the next `vmi` operation if the `flag` register is zero.

If `jumpIfTrue` is `False`, then a jump will be taken to target address `toAddress` if the value of the `flag` register is zero and execution will continue with the next `vmi` operation if the `flag` register is non-zero.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum {
    vmi_JH_NONE      = 0,   // no jump hint
    vmi_JH_CALL      = 1,   // jump is a call
    vmi_JH_CALLINT   = 2,   // jump is an interrupt call
    vmi_JH_RETURN    = 3,   // jump is a return
    vmi_JH_RETURNINT = 4    // jump is an interrupt return
} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_LINKREG    CPUX_REG(9)
#define CPUX_BF         CPUX_OFFSET(branchFlag)

// emit code for conditional jump to 'target'
static void emitCondJump(
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target
) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJump(CPUX_BF, True, 0 /*unused*/, target, VMI_NOREG, vmi_JH_NONE);
}

// emit code for conditional call to 'target' - if call is made, return
// address 'thisPC+4' is saved in register 'CPUX_LINKREG'
static void emitCondCall(
    Addr         thisPC,
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target
) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJump(CPUX_BF, True, thisPC+4, target, CPUX_LINKREG, vmi_JH_CALL);
}
```

Notes and Restrictions

1. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).

6.7 vmimtCondJumpDelaySlot

Prototype

```
void vmimtCondJumpDelaySlot(
    Uns32         slotOps,
    vmiReg        flag,
    Bool          jumpIfTrue,
    Addr          linkPC,
    Addr          toAddress,
    vmiReg        linkReg,
    vmiJumpHint   hint,
    vmiPostSlotFn slotCB

);
```

Description

Emit code to perform a conditional inter-instruction branch with `slotOps` subsequent delay slot instructions. The processor flag register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to target address `toAddress` only if the value of the `flag` register is non-zero.

If `jumpIfTrue` is `False`, then a jump will be taken to target address `toAddress` only if the value of the `flag` register is zero.

`slotOps` instructions after the current instruction will be executed prior to taking the branch. These instructions will be executed whether or not the branch is taken. If `slotOps` is 0, this function is equivalent to `vmiCondJump`.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum {
    vmi_JH_NONE      = 0,   // no jump hint
    vmi_JH_CALL      = 1,   // jump is a call
    vmi_JH_CALLINT   = 2,   // jump is an interrupt call
    vmi_JH_RETURN    = 3,   // jump is a return
    vmi_JH_RETURNINT = 4    // jump is an interrupt return
} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken the function is *not* called. The callback function is passed the processor as its only argument.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_BF         CPUX_OFFSET(branchFlag)

// emit code for conditional jump to 'target'
static void emitCondJump(
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target
) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJumpDelaySlot(
        1, CPUX_BF, True, 0 /*unused*/, target, VMI_NOREG, vmi_JH_NONE, 0
    );
}

// emit code for conditional call to 'target' - if call is made, return
// address 'thisPC+8' is saved in register 'CPUX_LINKREG'
static void emitCondCall(
    Addr         thisPC,
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target
) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJumpDelaySlot(
        1, CPUX_BF, True, thisPC+8, target, CPUX_LINKREG, vmi_JH_CALL, 0
    );
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to 0 or 1.
2. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
3. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).

4. `vmimtCondJumpDelaySlot` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

6.8 vmimtCondJumpDelaySlotAnnul

Prototype

```
void vmimtCondJumpDelaySlotAnnul(
    Uns32         slotOps,
    vmiReg        flag,
    Bool          jumpIfTrue,
    Addr          linkPC,
    Addr          toAddress,
    vmiReg        linkReg,
    vmiJumpHint   hint,
    vmiPostSlotFn slotCB

);
```

Description

Emit code to perform a conditional inter-instruction branch with `slotOps` subsequent delay slot instructions. The processor flag register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to target address `toAddress` only if the value of the `flag` register is non-zero.

If `jumpIfTrue` is `False`, then a jump will be taken to target address `toAddress` only if the value of the `flag` register is zero.

`slotOps` instructions after the current instruction will be executed prior to taking the branch, if the branch is taken. If the branch is *not* taken, some or all of the instruction actions may be annulled; the precise actions that are annulled are identified by a call to `vmimtSkipIfAnnul`, described elsewhere in this section.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum {
    vmi_JH_NONE      = 0,   // no jump hint
    vmi_JH_CALL      = 1,   // jump is a call
    vmi_JH_CALLINT   = 2,   // jump is an interrupt call
    vmi_JH_RETURN    = 3,   // jump is a return
    vmi_JH_RETURNINT = 4    // jump is an interrupt return
} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken the function is *not* called. The callback function is passed the processor as its only argument.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_BF         CPUX_OFFSET(branchFlag)

// emit code for conditional jump to 'target'
static void emitCondJumpAnnul(
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target
) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJumpDelaySlotAnnul(
        1, CPUX_BF, True, 0 /*unused*/, target, VMI_NOREG, vmi_JH_NONE, 0
    );
}

// emit code for conditional call to 'target' - if call is made, return
// address 'thisPC+8' is saved in register 'CPUX_LINKREG'
static void emitCondCallAnnul(
    Addr         thisPC,
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target
) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJumpDelaySlotAnnul(
        1, CPUX_BF, True, thisPC+8, target, CPUX_LINKREG, vmi_JH_CALL, 0
    );
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to 0 or 1.
2. When the the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).

3. When the the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).
4. `vmimtCondJumpDelaySlotAnnul` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

6.9 vmimtCondJumpReg

Prototype

```
void vmimtCondJumpReg(
    vmiReg      flag,
    Bool        jumpIfTrue,
    Addr        linkPC,
    vmiReg      toReg,
    vmiReg      linkReg,
    vmiJumpHint hint
);
```

Description

Emit code to perform a conditional indirect inter-instruction branch. The processor `flag` register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to the address in processor register `toReg` if the value of the `flag` register is non-zero and execution will continue with the next `vmi` operation if the `flag` register is zero.

If `jumpIfTrue` is `False`, then a jump will be taken to the address in processor register `toReg` if the value of the `flag` register is zero and execution will continue with the next `vmi` operation if the `flag` register is non-zero.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *indirect conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum {
    vmi_JH_NONE      = 0,   // no jump hint
    vmi_JH_CALL      = 1,   // jump is a call
    vmi_JH_CALLINT   = 2,   // jump is an interrupt call
    vmi_JH_RETURN    = 3,   // jump is a return
    vmi_JH_RETURNINT = 4    // jump is an interrupt return
} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_LINKREG    CPUX_REG(9)
#define CPUX_BF         CPUX_OFFSET(branchFlag)

// emit code for conditional jump to 'target'
static void emitCondJumpReg(
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target
) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJumpReg(CPUX_BF, True, 0 /*unused*/, target, VMI_NOREG, vmi_JH_NONE);
}

// emit code for conditional call to 'target' - if call is made, return
// address 'thisPC+4' is saved in register 'CPUX_LINKREG'
static void emitCondCallReg(
    Addr         thisPC,
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target
) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJumpReg(CPUX_BF, True, thisPC+4, target, CPUX_LINKREG, vmi_JH_CALL);
}
```

Notes and Restrictions

1. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).

6.10 vmimtCondJumpRegDelaySlot

Prototype

```
void vmimtCondJumpRegDelaySlot(
    Uns32         slotOps,
    vmiReg        flag,
    Bool          jumpIfTrue,
    Addr          linkPC,
    vmiReg        toReg,
    vmiReg        linkReg,
    vmiJumpHint   hint,
    vmiPostSlotFn slotCB

);
```

Description

Emit code to perform a conditional indirect inter-instruction branch with `slotOps` subsequent delay slot instructions. The processor `flag` register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to the address in processor register `toReg` only if the value of the `flag` register is non-zero.

If `jumpIfTrue` is `False`, then a jump will be taken to the address in processor register `toReg` only if the value of the `flag` register is zero.

`slotOps` instructions after the current instruction will be executed prior to taking the branch. These instructions will be executed whether or not the branch is taken. If `slotOps` is 0, this function is equivalent to `vmiCondJump`.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *indirect conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum {
    vmi_JH_NONE      = 0,   // no jump hint
    vmi_JH_CALL      = 1,   // jump is a call
    vmi_JH_CALLINT   = 2,   // jump is an interrupt call
    vmi_JH_RETURN    = 3,   // jump is a return
    vmi_JH_RETURNINT = 4    // jump is an interrupt return
} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken the function is *not* called. The callback function is passed the processor as its only argument.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_BF         CPUX_OFFSET(branchFlag)
// emit code for conditional jump to 'target'
static void emitCondJumpReg(
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target
) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJumpRegDelaySlot(
        1, CPUX_BF, True, 0 /*unused*/, target, VMI_NOREG, vmi_JH_NONE, 0
    );
}

// emit code for conditional call to 'target' - if call is made, return
// address 'thisPC+8' is saved in register 'CPUX_LINKREG'
static void emitCondCallReg(
    Addr         thisPC,
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target
) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJumpRegDelaySlot(
        1, CPUX_BF, True, thisPC+8, target, CPUX_LINKREG, vmi_JH_CALL, 0
    );
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to 0 or 1.
2. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
3. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).

4. `vmimtCondJumpRegDelaySlot` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

6.11 vmimtCondJumpRegDelaySlotAnnul

Prototype

```
void vmimtCondJumpRegDelaySlotAnnul(
    Uns32         slotOps,
    vmiReg        flag,
    Bool          jumpIfTrue,
    Addr          linkPC,
    vmiReg        toReg,
    vmiReg        linkReg,
    vmiJumpHint   hint,
    vmiPostSlotFn slotCB

);
```

Description

Emit code to perform a conditional indirect inter-instruction branch with `slotOps` subsequent delay slot instructions. The processor `flag` register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to the address in processor register `toReg` only if the value of the `flag` register is non-zero.

If `jumpIfTrue` is `False`, then a jump will be taken to the address in processor register `toReg` only if the value of the `flag` register is zero.

`slotOps` instructions after the current instruction will be executed prior to taking the branch, if the branch is taken. If the branch is *not* taken, some or all of the instruction actions may be annulled; the precise actions that are annulled are identified by a call to `vmimtSkipIfAnnul`, described elsewhere in this section.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *indirect conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum {
    vmi_JH_NONE      = 0,   // no jump hint
    vmi_JH_CALL      = 1,   // jump is a call
    vmi_JH_CALLINT   = 2,   // jump is an interrupt call
    vmi_JH_RETURN    = 3,   // jump is a return
    vmi_JH_RETURNINT = 4    // jump is an interrupt return
} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken the function is *not* called. The callback function is passed the processor as its only argument.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_BF         CPUX_OFFSET(branchFlag)

// emit code for conditional jump to 'target'
static void emitCondJumpRegAnnul(
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJumpRegDelaySlotAnnul(
        1, CPUX_BF, True, 0 /*unused*/, target, VMI_NOREG, vmi_JH_NONE, 0
    );
}

// emit code for conditional call to 'target' - if call is made, return
// address 'thisPC+8' is saved in register 'CPUX_LINKREG'
static void emitCondCallRegAnnul(
    Addr         thisPC,
    vmiCondition cond,
    Uns32        ra,
    Uns32        rb,
    Addr         target
) {
    vmimtCompareRR(CPUX_GBITS, cond, CPUX_REG(ra), CPUX_REG(rb), CPUX_BF);
    vmimtCondJumpRegDelaySlotAnnul(
        1, CPUX_BF, True, thisPC+8, target, CPUX_LINKREG, vmi_JH_CALL, 0
    );
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to 0 or 1.
2. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).

3. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).
4. `vmimtCondJumpRegDelaySlotAnnul` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

6.12 vmimtSkipIfAnnul

Prototype

```
void vmimtSkipIfAnnul(void);
```

Description

This routine has no action unless the current instruction is a delay slot instruction of a conditional jump specified by a call to `vmimtCondJumpDelaySlotAnnul` or `vmimtCondJumpRegDelaySlotAnnul`, described elsewhere in this section.

If the current instruction is indeed such a delay slot instruction, this routine causes all behavior following the call to `vmimtSkipIfAnnul` to be skipped if the branch is annulled (not taken).

`vmimtSkipIfAnnul` is typically called once in the main `morpher` callback function.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  branchFlag;       // branch flag
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_BF         CPUX_OFFSET(branchFlag)

//
// Create code for the CPUX instruction at the simulated address //referenced by
'thisPC'
//
VMI_MORPH_FN(cpuxMorphInstruction) {

    Uns32 instr = vmicxtFetch4Byte(processor, thisPC);

    // skip actions after this point if annulling the current instruction
    vmimtSkipIfAnnul();

    // instruction decode and morph
    if (instr == OP_BREAK(0x3ff)) {
                . . . etc . . .
      }
  }
```

Notes and Restrictions

None.

6.13 vmimtGetDelaySlotNextPC

Prototype

```
void vmimtGetDelaySlotNextPC(vmiReg targetReg, Bool getNextPC);
```

Description

This function is useful for determining an instruction address after a delay slot instruction or the branch target of a delay slot instruction.

If `getNextPC` is `True`, then register `targetReg` will be loaded with the address of the *next instruction to be executed after the delay slot instruction*. In the case of a conditional branch that is taken, this will be the branch address; in the case of a conditional branch that is *not* taken, this will be the address of the instruction following the delay slot instruction.

If `getNextPC` is `False`, then register `targetReg` will be loaded with the branch target address irrespective of whether the branch is taken.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

static void emitGetNextPC(Addr thisPC, vmiReg nextPCReg, Bool inDelaySlot) {
    if(inDelaySlot) {
        // if this is a delay slot instruction, get the next instruction address
        vmimtGetDelaySlotNextPC(nextPCReg, True);
    } else {
        // not in a delay slot: use the next program counter
        vmimtMoveRC(CPUX_ADDRESS_BITS, nextPCReg, thisPC+4);
    }
}
```

Notes and Restrictions

1. This function may only be used in the context of a delay slot instruction. Use outside this context will generate an assertion:
   `vmimtGetDelaySlotNextPC used outside delay slot context`
   The `inDelaySlot` parameter to the morph callback function specifies whether the current instruction is a delay slot instruction. The above example assumes that this has been passed as the third argument to `emitGetNextPC`.
2. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `targetReg` in the processor structure is a 32-bit unsigned (`Uns32`).

3. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `targetReg` in the processor structure is a 64-bit unsigned (`Uns64`).

6.14 vmimtEnterDelaySlotC

Prototype

```
void vmimtEnterDelaySlotC(
    Uns32         slotOps,
    Addr          simPC1,
    Addr          simPC2,
    vmiPostSlotFn slotCB
);
```

Description

This function is used to implement a special form of jump. Firstly, a jump is made to address `simPC1`, and address `simPC2` is scheduled as a delay slot instruction address. After `slotOps` instructions have been executed, control resumes at `simPC2`.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. The callback function is passed the processor as its only argument.

Notes and Restrictions

1. `slotOps` is currently restricted to 1.

6.15 vmimtEnterDelaySlotR

Prototype

```
void vmimtEnterDelaySlotR(
    Uns32         slotOps,
    vmiReg        toReg,
    Addr          simPC2,
    vmiPostSlotFn slotCB
);
```

Description

This function is used to implement a special form of jump. Firstly, a jump is made to the address held in register `toReg`, and address `simPC2` is scheduled as a delay slot instruction address. After `slotOps` instructions have been executed, control resumes at `simPC2`.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. The callback function is passed the processor as its only argument.

Example

This example is taken from the OVP ARC processor model. This processor has an *execute indexed* instruction (`EI_S`) that executes a single instruction at a computed address before resuming execution at the instruction after the `EI_S` instruction. The required target address is in register `rt`:

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

void arcEnterDelaySlotR(arcMorphStateP state, vmiReg rt) {

    Uns32 bits = ARC_GPR_BITS;

    // set BTA with target address
    vmimtMoveRC(bits, ARC_AUX_REG(bta), state->nextPC);

    // enter delay-slot block
    vmimtEnterDelaySlotR(1, rt, state->nextPC, 0);
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to 1.

6.16 vmimtNewLabel

Prototype

```
vmiLabelP vmimtNewLabel(void);
```

Description

This function is used to define a label for use with *intra-instruction jumps*.

The example below shows a case where the action that should be performed when a system call instruction is encountered depends on the run-time value of a processor pseudo-register `interceptTrap`: if this register is `True`, the system call should be handled by a run time call to a function (`vmic_InterceptTrap`, not shown here); if `interceptTrap` is `False`, the system call should be handled by jumping to a simulated exception handler at address `0x800`.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  interceptTrap;    // should traps be intercepted?
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
    Uns32 epcr;             // exception program counter register
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_EPCR       CPUX_OFFSET(epcr);
#define SYS_ADDRESS     0x800

static void emitSys(Addr thisPC) {

    Addr      nextAddress = thisPC + 4;
    vmiLabelP label1      = vmimtNewLabel();
    vmiLabelP label2      = vmimtNewLabel();
    vmimtCondJumpLabel(CPUX_OFFSET(interceptTrap), False, label1);

    // call trap routine
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)vmic_InterceptTrap);
    vmimtUncondJumpLabel(label2);

    vmimtInsertLabel(label1);
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)vmic_EnterKernelMode);
    vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);

    vmimtInsertLabel(label2);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use `vmimtCondJump` or `vmimtUncondJump` as appropriate.
2. Only use label-based jumps if *the correct branch to take is known only at run time*. If the correct branch is known at *morph time*, it is much more efficient to morph alternative code sequences instead. For example:

```
static void emitSys(Addr thisPC, Bool interceptTrap) {
    if(interceptTrap) {
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_InterceptTrap);
    } else {
        Addr nextAddress = thisPC + 4;
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_EnterKernelMode);
        vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);
    }
}
```

6.17 vmimtInsertLabel

Prototype

```
void vmimtInsertLabel(vmiLabelP label);
```

Description

This function is used to insert a label previously defined with `vmimtNewLabel` at the current position in the NMI node list. Labels are used to implement *intra-instruction jumps*.

The example below shows a case where the action that should be performed when a system call instruction is encountered depends on the run-time value of a processor pseudo-register `interceptTrap`: if this register is `True`, the system call should be handled by a run time call to a function (`vmic_InterceptTrap`, not shown here); if `interceptTrap` is `False`, the system call should be handled by jumping to a simulated exception handler at address `0x800`.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  interceptTrap;    // should traps be intercepted?
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
    Uns32 epcr;             // exception program counter register
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_EPCR       CPUX_OFFSET(epcr);
#define SYS_ADDRESS     0x800

static void emitSys(Addr thisPC) {

    Addr      nextAddress = thisPC + 4;
    vmiLabelP label1      = vmimtNewLabel();
    vmiLabelP label2      = vmimtNewLabel();
    vmimtCondJumpLabel(CPUX_OFFSET(interceptTrap), False, label1);

    // call trap routine
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)vmic_InterceptTrap);
    vmimtUncondJumpLabel(label2);

    vmimtInsertLabel(label1);
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)vmic_EnterKernelMode);
    vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);

    vmimtInsertLabel(label2);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use `vmimtCondJump` or `vmimtUncondJump` as appropriate.
2. Only use label-based jumps if *the correct branch to take is known only at run time*. If the correct branch is known at *morph time*, it is much more efficient to morph alternative code sequences instead. For example:

```
static void emitSys(Addr thisPC, Bool interceptTrap) {
    if(interceptTrap) {
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_InterceptTrap);
    } else {
        Addr nextAddress = thisPC + 4;
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_EnterKernelMode);
        vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);
    }
}
```

6.18 vmimtUncondJumpLabel

Prototype

```
void vmimtUncondJumpLabel(vmiLabelP toLabel);
```

Description

This function is used to perform an unconditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*.

The example below shows a case where the action that should be performed when a system call instruction is encountered depends on the run-time value of a processor pseudo-register `interceptTrap`: if this register is `True`, the system call should be handled by a run time call to a function (`vmic_InterceptTrap`, not shown here); if `interceptTrap` is `False`, the system call should be handled by jumping to a simulated exception handler at address `0x800`.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  interceptTrap;    // should traps be intercepted?
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
    Uns32 epcr;             // exception program counter register
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_EPCR       CPUX_OFFSET(epcr);
#define SYS_ADDRESS     0x800

static void emitSys(Addr thisPC) {
    Addr      nextAddress = thisPC + 4;
    vmiLabelP label1      = vmimtNewLabel();
    vmiLabelP label2      = vmimtNewLabel();
    vmimtCondJumpLabel(CPUX_OFFSET(interceptTrap), False, label1);

    // call trap routine
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)vmic_InterceptTrap);
    vmimtUncondJumpLabel(label2);

    vmimtInsertLabel(label1);
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)vmic_EnterKernelMode);
    vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);

    vmimtInsertLabel(label2);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use `vmimtCondJump` or `vmimtUncondJump` as appropriate.
2. The label argument to `vmimtUncondJumpLabel` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction.
   If the call to `vmimtInsertLabel` precedes the call to `vmimtUncondJumpLabel`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtUncondJumpLabel` (as in the above example), then this is a forward jump in the node list.
3. Only use label-based jumps if *the correct branch to take is known only at run time*. If the correct branch is known at *morph time*, it is much more efficient to morph alternative code sequences instead. For example:

```
static void emitSys(Addr thisPC, Bool interceptTrap) {
    if(interceptTrap) {
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_InterceptTrap);
    } else {
        Addr nextAddress = thisPC + 4;
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_EnterKernelMode);
        vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);
    }
}
```

6.19 vmimtCondJumpLabel

Prototype

```
void vmimtCondJumpLabel(
    vmiReg    flag,
    Bool      jumpIfTrue,
    vmiLabelP toLabel
);
```

Description

This function is used to perform a conditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*.

The processor flag register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to label `toLabel` if the value of the `flag` register is non-zero and execution will continue with the next `vmi` operation if the `flag` register is zero.

If `jumpIfTrue` is `False`, then a jump will be taken to target label `toLabel` if the value of the `flag` register is zero and execution will continue with the next vmi operation if the `flag` register is non-zero.

The example below shows a case where the action that should be performed when a system call instruction is encountered depends on the run-time value of a processor pseudo-register `interceptTrap`: if this register is `True`, the system call should be handled by a run time call to a function (`vmic_InterceptTrap`, not shown here); if `interceptTrap` is `False`, the system call should be handled by jumping to a simulated exception handler at address `0x800`.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  interceptTrap;    // should traps be intercepted?
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
    Uns32 epcr;             // exception program counter register
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_EPCR       CPUX_OFFSET(epcr);
#define SYS_ADDRESS     0x800
```

```
static void emitSys(Addr thisPC) {

    Addr      nextAddress = thisPC + 4;
    vmiLabelP label1      = vmimtNewLabel();
    vmiLabelP label2      = vmimtNewLabel();
    vmimtCondJumpLabel(CPUX_OFFSET(interceptTrap), False, label1);

    // call trap routine
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)vmic_InterceptTrap);
    vmimtUncondJumpLabel(label2);

    vmimtInsertLabel(label1);
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)vmic_EnterKernelMode);
    vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);

    vmimtInsertLabel(label2);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use `vmimtCondJump` or `vmimtUncondJump` as appropriate.
2. The label argument to `vmimtCondJumpLabel` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction. If the call to `vmimtInsertLabel` precedes the call to `vmimtCondJumpLabel`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtCondJumpLabel` (as in the above example), then this is a forward jump in the node list.
3. Only use label-based jumps if *the correct branch to take is known only at run time*. If the correct branch is known at *morph time*, it is much more efficient to morph alternative code sequences instead. For example:

```
static void emitSys(Addr thisPC, Bool interceptTrap) {
    if(interceptTrap) {
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_InterceptTrap);
    } else {
        Addr nextAddress = thisPC + 4;
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_EnterKernelMode);
        vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);
    }
}
```

6.20 vmimtCondJumpLabelFunctionResult

Prototype

```
void vmimtCondJumpLabelFunctionResult(Bool jumpIfTrue, vmiLabelP toLabel);
```

Description

This function is used to perform a conditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*.

Whether the branch should be taken is determined by the value returned by an embedded function call made immediately prior to the call to `vmimtCondJumpLabelFunctionResult` – see *Embedded Native Call Operations* for more information about embedded function calls.

If `jumpIfTrue` is `True`, then a jump will be taken to label `toLabel` if the embedded function call returned non-zero and execution will continue with the next instruction if the embedded function call returned zero.

If `jumpIfTrue` is `False`, then a jump will be taken to target label `toLabel` if the embedded function call returned zero and execution will continue with the next instruction if the embedded function call returned non-zero.

The example below shows a case where the action that should be performed when a system call instruction is encountered depends on the value returned by function `vmic_DoInterceptTrap`: if `True`, the system call should be handled by a run time call to a function (`vmic_InterceptTrap`, not shown here); if `False`, the system call should be handled by jumping to a simulated exception handler at address `0x800`.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8  interceptTrap;    // should traps be intercepted?
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
    Uns32 epcr;             // exception program counter register
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_EPCR       CPUX_OFFSET(epcr);
#define SYS_ADDRESS     0x800

static void emitSys(Addr thisPC) {

    Addr      nextAddress = thisPC + 4;
    vmiLabelP label1      = vmimtNewLabel();
    vmiLabelP label2      = vmimtNewLabel();
```

```
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)vmic_DoInterceptTrap);
    vmimtCondJumpLabelFunctionResult(False, label1);

    // call trap routine
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)vmic_InterceptTrap);
    vmimtUncondJumpLabel(label2);

    vmimtInsertLabel(label1);
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)vmic_EnterKernelMode);
    vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);

    vmimtInsertLabel(label2);
}

static Bool vmic_DoInterceptTrap(cpuxP cpux) {
    return cpux->interceptTrap;
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use `vmimtCondJump` or `vmimtUncondJump` as appropriate.
2. The label argument to `vmimtCondJumpLabelFunctionResult` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction.
   If the call to `vmimtInsertLabel` precedes the call to `vmimtCondJumpLabelFunctionResult`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtCondJumpLabelFunctionResult` (as in the above example), then this is a forward jump in the node list.
3. Only use label-based jumps if *the correct branch to take is known only at run time*. If the correct branch is known at *morph time*, it is much more efficient to morph alternative code sequences instead. For example:

```
static void emitSys(Addr thisPC, Bool interceptTrap) {
    if(interceptTrap) {
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_InterceptTrap);
    } else {
        Addr nextAddress = thisPC + 4;
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_EnterKernelMode);
        vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);
    }
}
```

6.21 vmimtTestRCJumpLabel

Prototype

```
void vmimtTestRCJumpLabel(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       r
    Uns64        c,
    vmiLabelP    toLabel
);
```

Description

This function is used to perform a conditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*.

The function emits code to compare register `r` of size `bits` with constant `c`. If the condition `cond` is satisfied, control branches to `toLabel`; otherwise, execution continues with the next vmi operation. The comparison is performed by performing a bitwise AND of the register and constant value.

Example

The MIPS processor model uses this to implement conditional move instructions, as follows:

```
static void emitMoveCond(mips32InstructionInfoP info, vmiCondition cond) {

    Uns32     bits = MIPS32_GPR_BITS;
    vmiReg    rd   = getR1(info);
    vmiReg    rs   = getR2(info);
    vmiReg    rt   = getR3(info);
    vmiLabelP skip = vmimtNewLabel();

    // don't write to register 0
    if(!VMI_ISNOREG(rd)) {
        vmimtTestRCJumpLabel(bits, cond, rt, -1, skip);
        vmimtMoveRR(bits, rd, rs);
        vmimtInsertLabel(skip);
    }
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use `vmimtCondJump` or `vmimtUncondJump` as appropriate.
2. The label argument to `vmimtCondJumpLabel` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction. If the call to `vmimtInsertLabel` precedes the call to `vmimtCondJumpLabel`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtCondJumpLabel` (as in the above example), then this is a forward jump in the node list.

6.22 vmimtCompareRCJumpLabel

Prototype

```
void vmimtCompareRCJumpLabel(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       r
    Uns64        c,
    vmiLabelP    toLabel
);
```

Description

This function is used to perform a conditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*.

The function emits code to compare register `r` of size `bits` with constant `c`. If the condition `cond` is satisfied, control branches to `toLabel`; otherwise, execution continues with the next vmi operation. The comparison is performed by subtracting the constant from the register value.

Example

The MIPS processor model uses this to implement `LWL`, `LWR`, `SWL` and `SWR` instructions, as follows:

```
// Terminate LWL, LWR, SWL or SWR if (address&3) is endValue
static void emitTerminateByteLoadStore(vmiLabelP endLabel, Int32 endValue) {
    Uns32 bits = MIPS32_GPR_BITS;
    vmimtCompareRCJumpLabel(bits, vmi_COND_Z, MIPS32_TEMP3, endValue, endLabel);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use `vmimtCondJump` or `vmimtUncondJump` as appropriate.
2. The label argument to `vmimtCondJumpLabel` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction. If the call to `vmimtInsertLabel` precedes the call to `vmimtCondJumpLabel`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtCondJumpLabel` (as in the above example), then this is a forward jump in the node list.

7 Embedded Native Call Operations

This section describes emission functions for embedding model function calls within translated native code. This technique is especially useful when there are no suitable `vmimt`-prefixed routines to implement required functionality: it provides a generic extension capability.

To embed a call to a model function:

1. Use the functions with the `vmimtArg` prefix to specify the arguments to the embedded call;
2. Use a function with the `vmimtCall` prefix to specify the function to call (and possibly what should happen to any function result).

7.1 vmimtArgProcessor

Prototype

```
void vmimtArgProcessor(void);
```

Description

`vmimtArgProcessor` specifies that the *current processor handle* should be passed as an argument to an embedded function call. Processor fields can then be accessed directly within the callback.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

// emit call to vmic_CountLeadingZeros
static void emitCountLeadingZeros(Uns32 rd, Uns32 rs) {
    vmimtArgProcessor();                        // argument 1: processor
    vmimtArgUns32(rd);                          // argument 2: index of rd
    vmimtArgReg(CPUX_GBITS, CPUX_REG(rs));      // argument 3: value of rs
    vmimtCall((vmiCallFn)vmic_CountLeadingZeros);
}

// function called at run time by embedded call
static void vmic_CountLeadingZeros(cpuxP cpux, Uns32 rd, Uns32 value) {
    Uns32 temp = 32;
    Int32 i;
    for(i = 31; i >= 0; i--) {
        if(value & (1 << i)) {
            temp = 31 - i;
            break;
        }
    }
    cpux->regs[rd] = temp;
}
```

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`vmic_CountLeadingZeros`, in the above example).

7.2 vmimtArgUns32

Prototype

```
void vmimtArgUns32(Uns32 arg);
```

Description

`vmimtArgUns32` specifies that a *32-bit unsigned value* (`Uns32`) should be passed as an argument to an embedded function call.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

// emit call to vmic_CountLeadingZeros
static void emitCountLeadingZeros(Uns32 rd, Uns32 rs) {
    vmimtArgProcessor();                    // argument 1: processor
    vmimtArgUns32(rd);                      // argument 2: index of rd
    vmimtArgReg(CPUX_GBITS, CPUX_REG(rs));  // argument 3: value of rs
    vmimtCall((vmiCallFn)vmic_CountLeadingZeros);
}

// function called at run time by embedded call
static void vmic_CountLeadingZeros(cpuxP cpux, Uns32 rd, Uns32 value) {
    Uns32 temp = 32;
    Int32 i;
    for(i = 31; i >= 0; i--) {
        if(value & (1 << i)) {
            temp = 31 - i;
            break;
        }
    }
    cpux->regs[rd] = temp;
}
```

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`vmic_CountLeadingZeros`, in the above example).

7.3 vmimtArgUns64

Prototype

```
void vmimtArgUns64(Uns64 arg);
```

Description

`vmimtArgUns64` specifies that a *64-bit unsigned value* (`Uns64`) should be passed as an argument to an embedded function call.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 cpuId;      // cpu Id number
    Uns32 epc;               // exception program counter (32-bit)
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_CPUID_REG  CPUX_OFFSET(cpuId)
#define CPUX_EPC_REG    CPUX_OFFSET(epc)

static void emitCall(Addr thisPC) {
    vmimtArgProcessor();
    vmimtArgUns32(32);
    vmimtArgUns64(64);
    vmimtArgReg(CPUX_GBITS, CPUX_CPUID_REG);
    vmimtArgRegSimAddress(CPUX_GBITS, CPUX_EPC_REG);
    vmimtArgSimAddress(thisPC);
    vmimtArgNatAddress(“hello world”);
    vmimtCall((vmiCallFn)vmic_TestCall);
}

// function called at run time by embedded call
static void vmic_TestCall(
    cpuxP cpux,         // processor handle
    Uns32 arg32,        // 32-bit argument
    Uns64 arg64,        // 64-bit argument
    Uns32 reg32,        // 32-bit argument from register
    Addr  regAddr,      // simulated address argument
    Addr  addr,         // simulated address argument
    char *natAddr       // native address argument
) {
    vmiPrintf(
        “Test: cpuId=%u arg32=%u arg64=%llu reg32=%u addr=%llu “
        “regAddr=%llu natAddr=%s\n”,
        cpux->cpuId, arg32, arg64, reg32, addr, regAddr, natAddr
    );
}
```

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`vmic_TestCall`, in the above example).

7.4 vmimtArgDouble

Prototype

```
void vmimtArgDouble(double arg);
```

Description

`vmimtArgDouble` specifies that a *double value* (64-bit floating point) should be passed as an argument to an embedded function call.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 cpuId;      // cpu Id number
    Uns32 epc;              // exception program counter (32-bit)
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_CPUID_REG  CPUX_OFFSET(cpuId)
#define CPUX_EPC_REG    CPUX_OFFSET(epc)

static void emitCall(Addr thisPC) {
    vmimtArgProcessor();
    vmimtArgUns32(32);
    vmimtArgDouble(1.2345);
    vmimtArgReg(CPUX_GBITS, CPUX_CPUID_REG);
    vmimtArgRegSimAddress(CPUX_GBITS, CPUX_EPC_REG);
    vmimtArgSimAddress(thisPC);
    vmimtArgNatAddress(“hello world”);
    vmimtCall((vmiCallFn)vmic_TestCall);
}

// function called at run time by embedded call
static void vmic_TestCall(
    cpuxP  cpux,        // processor handle
    Uns32  arg32,       // 32-bit argument
    double argDouble,   // double argument
    Uns32  reg32,       // 32-bit argument from register
    Addr   regAddr,     // simulated address argument
    Addr   addr,        // simulated address argument
    char  *natAddr      // native address argument
) {
    vmiPrintf(
        “Test: cpuId=%u arg32=%u argDouble=%g reg32=%u addr=%llu “
        “regAddr=%llu natAddr=%s\n”,
        cpux->cpuId, arg32, argDouble, reg32, addr, regAddr, natAddr
    );
}
```

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`vmic_TestCall`, in the above example).

7.5 vmimtArgReg

Prototype

```
void vmimtArgReg(vmiRegArgType argType, vmiReg r);
```

Description

`vmimtArgReg` specifies that the value of a processor register `r` should be passed as an argument to an embedded function call. The type of the register is given by the `argType` argument, defined in `vmiTypes.h` as follows:

```
typedef enum vmiRegArgTypeE {

    VPRRAT_8     =  8,              // 8-bit register argument
    VPRRAT_16    =  16,             // 16-bit register argument
    VPRRAT_32    =  32,             // 32-bit register argument
    VPRRAT_64    =  64,             // 64-bit register argument

    VPRRAT_FLT   = 0x80000000,      // floating-point argument identifier
    VPRRAT_FLT64 = 64|VPRRAT_FLT    // 64-bit floating point

} vmiRegArgType;
```

The argument type can be an 8-bit, 16-bit, 32-bit or 64-bit integer register or a floating-point register in double-precision format.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

// emit call to vmic_CountLeadingZeros
static void emitCountLeadingZeros(Uns32 rd, Uns32 rs) {
    vmimtArgProcessor();                    // argument 1: processor
    vmimtArgUns32(rd);                      // argument 2: index of rd
    vmimtArgReg(CPUX_GBITS, CPUX_REG(rs));    // argument 3: value of rs
    vmimtCall((vmiCallFn)vmic_CountLeadingZeros);
}

// function called at run time by embedded call
static void vmic_CountLeadingZeros(cpuxP cpux, Uns32 rd, Uns32 value) {
    Uns32 temp = 32;
    Int32 i;
    for(i = 31; i >= 0; i--) {
        if(value & (1 << i)) {
            temp = 31 - i;
            break;
        }
    }
    cpux->regs[rd] = temp;
}
```

Notes and Restrictions

1. In versions of the VMI API prior to 4.2.0, the first argument to this function was a number of bits (8, 16, 32 or 64). The function prototype has been enhanced to allow floating point operands to be explicitly identified, required for 64-bit host support.
2. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`vmic_CountLeadingZeros`, in the above example).

7.6 vmimtArgRegSimAddress

Prototype

```
void vmimtArgRegSimAddress(Uns32 bits, vmiReg r);
```

Description

`vmimtArgRegSimAddress` specifies that the value of a processor register `r` should be passed as an argument to an embedded function call. The register is of size `bits`, but it should be zero-extended to the size of the `Addr` type when passed as a function argument. This function is useful because many of the run time callbacks (defined in `vmiRt.h`) take generic `Addr` arguments to specify an address. The `Addr` type is 64-bit, but addresses in processor registers may well be less than this (32-bits or less).

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 cpuId; // cpu Id number
    Uns32 epc;          // exception program counter (32-bit)
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_CPUID_REG  CPUX_OFFSET(cpuId)
#define CPUX_EPC_REG    CPUX_OFFSET(epc)

static void emitCall(Addr thisPC) {
    vmimtArgProcessor();
    vmimtArgUns32(32);
    vmimtArgUns64(64);
    vmimtArgReg(CPUX_GBITS, CPUX_CPUID_REG);
    vmimtArgRegSimAddress(CPUX_GBITS, CPUX_EPC_REG);
    vmimtArgSimAddress(thisPC);
    vmimtArgNatAddress("hello world");
    vmimtCall((vmiCallFn)vmic_TestCall);
}

// function called at run time by embedded call
static void vmic_TestCall(
    cpuxP cpux,         // processor handle
    Uns32 arg32,        // 32-bit argument
    Uns64 arg64,        // 64-bit argument
    Uns32 reg32,        // 32-bit argument from register
    Addr  regAddr,      // simulated address argument
    Addr  addr,         // simulated address argument
    char *natAddr       // native address argument
) {
    vmiPrintf(
        "Test: cpuId=%u arg32=%u arg64=%llu reg32=%u addr=%llu "
        "regAddr=%llu natAddr=%s\n",
        cpux->cpuId, arg32, arg64, reg32, addr, regAddr, natAddr
    );
}
```

Notes and Restrictions

1. `bits` may be 8, 16, 32 or 64.
2. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`vmic_TestCall`, in the above example).

7.7 vmimtArgSimAddress

Prototype

```
void vmimtArgSimAddress(Addr arg);
```

Description

`vmimtArgSimAddress` specifies that the address `arg` should be passed as an argument to an embedded function call.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 cpuId; // cpu Id number
    Uns32 epc;          // exception program counter (32-bit)
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_CPUID_REG  CPUX_OFFSET(cpuId)
#define CPUX_EPC_REG    CPUX_OFFSET(epc)

static void emitCall(Addr thisPC) {
    vmimtArgProcessor();
    vmimtArgUns32(32);
    vmimtArgUns64(64);
    vmimtArgReg(CPUX_GBITS, CPUX_CPUID_REG);
    vmimtArgRegSimAddress(CPUX_GBITS, CPUX_EPC_REG);
    vmimtArgSimAddress(thisPC);
    vmimtArgNatAddress(“hello world”);
    vmimtCall((vmiCallFn)vmic_TestCall);
}

// function called at run time by embedded call
static void vmic_TestCall(
    cpuxP cpux,         // processor handle
    Uns32 arg32,        // 32-bit argument
    Uns64 arg64,        // 64-bit argument
    Uns32 reg32,        // 32-bit argument from register
    Addr  regAddr,      // simulated address argument
    Addr  addr,         // simulated address argument
    char *natAddr       // native address argument
) {
    vmiPrintf(
        “Test: cpuId=%u arg32=%u arg64=%llu reg32=%u addr=%llu “
        “regAddr=%llu natAddr=%s\n”,
        cpux->cpuId, arg32, arg64, reg32, addr, regAddr, natAddr
    );
}
```

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`vmic_TestCall`, in the above example).

7.8 vmimtArgSimPC

Prototype

```
void vmimtArgSimPC(Uns32 bits);
```

Description

`vmimtArgSimPC` specifies that the current simulated program counter should be passed as an argument to an embedded function call.

If a processor model does not use physically-mapped code dictionaries, then this is equivalent to using `vmimtArgUns32` or `vmimtArgUns64`, specifying the current program counter as the constant argument. However, when processor models do use physically-mapped code dictionaries, `vmimtArgSimPC` **must** be used to obtain the current simulated address, because the same JIC compiled code block can be mapped at *different* simulated addresses.

See the description of `vmirtAliasMemoryVM` in the *VMI Run Time Function Reference* and also the *Imperas Processor Modeling Guide* for more information about physically-mapped code dictionaries.

Example

The template ARM model uses this function when emitting an embedded call to an exception handler function (`armEmitArgSimPC` is a wrapper around `vmimtArgSimPC`):

```
// Emit call to exception function
static void emitExceptionCall(armMorphStateP state, exceptionFn cb) {

    Uns32 bits = ARM_GPR_BITS;

    // emit call to exception routine
    armEmitArgProcessor(state);
    armEmitArgSimPC(state, bits);
    armEmitArgUns32(state, state->info.bytes);
    armEmitCall(state, (vmiCallFn)cb);

    // terminate the current block
    armEmitEndBlock();
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

7.9 vmimtArgNatAddress

Prototype

```
void vmimtArgNatAddress(void *arg);
```

Description

`vmimtArgNatAddress` specifies that the void pointer `arg` should be passed as an argument to an embedded function call.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 cpuId; // cpu Id number
    Uns32 epc;          // exception program counter (32-bit)
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_CPUID_REG  CPUX_OFFSET(cpuId)
#define CPUX_EPC_REG    CPUX_OFFSET(epc)

static void emitCall(Addr thisPC) {
    vmimtArgProcessor();
    vmimtArgUns32(32);
    vmimtArgUns64(64);
    vmimtArgReg(CPUX_GBITS, CPUX_CPUID_REG);
    vmimtArgRegSimAddress(CPUX_GBITS, CPUX_EPC_REG);
    vmimtArgSimAddress(thisPC);
    vmimtArgNatAddress(“hello world”);
    vmimtCall((vmiCallFn)vmic_TestCall);
}

// function called at run time by embedded call
static void vmic_TestCall(
    cpuxP cpux,         // processor handle
    Uns32 arg32,        // 32-bit argument
    Uns64 arg64,        // 64-bit argument
    Uns32 reg32,        // 32-bit argument from register
    Addr  regAddr,      // simulated address argument
    Addr  addr,         // simulated address argument
    char *natAddr       // native address argument
) {
    vmiPrintf(
        “Test: cpuId=%u arg32=%u arg64=%llu reg32=%u addr=%llu “
        “regAddr=%llu natAddr=%s\n”,
        cpux->cpuId, arg32, arg64, reg32, addr, regAddr, natAddr
    );
}
```

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`vmic_TestCall`, in the above example).

7.10 vmimtCall

Prototype

```
void vmimtCall(vmiCallFn arg);
```

Description

`vmimtCall` emits a call to the function specified as an argument. The argument has type `vmiCallFn`:

```
typedef void (*vmiCallFn)(void);
```

In reality, the argument can be any function type that matches the arguments previously created by calls to functions with the `vmimtArg` prefix, and it will be necessary to cast the function to type `vmiCallFn`.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

// emit call to vmic_CountLeadingZeros
static void emitCountLeadingZeros(Uns32 rd, Uns32 rs) {
    vmimtArgProcessor();               // argument 1: processor
    vmimtArgUns32(rd);                      // argument 2: index of rd
    vmimtArgReg(CPUX_GBITS, CPUX_REG(rs));    // argument 3: value of rs
    vmimtCall((vmiCallFn)vmic_CountLeadingZeros);
}

// function called at run time by embedded call
static void vmic_CountLeadingZeros(cpuxP cpux, Uns32 rd, Uns32 value) {
    Uns32 temp = 32;
    Int32 i;
    for(i = 31; i >= 0; i--) {
        if(value & (1 << i)) {
            temp = 31 - i;
            break;
        }
    }
    cpux->regs[rd] = temp;
}
```

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`vmic_CountLeadingZeros`, in the above example).
2. If the function or any child uses floating point operations, include a call to `vmirtRestoreFPState` to ensure that native floating point status is restored to

that in force when the processor was created. See the *VMI Run Time Function Reference* for more information.

7.11 vmimtCallResult

Prototype

```
void vmimtCallResult(vmiCallFn arg, Uns32 bits, vmiReg rd);
```

Description

`vmimtCallResult` emits a call to the function specified as an argument. The argument `arg` has type `vmiCallFn`:

```
typedef void (*vmiCallFn)(void);
```

In reality, the argument can be any function type that matches the arguments previously created by calls to functions with the `vmimtArg` prefix, and it will be necessary to cast the function to type `vmiCallFn`.

The return value from the embedded function is stored in processor register `rd` which has size `bits`. If `rd` is `VMI_NOREG`, this function is equivalent to `vmimtCall`.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

// emit call to vmic_CountLeadingZeros
static void emitCountLeadingZeros(Uns32 rd, Uns32 rs) {
    vmimtArgReg(CPUX_GBITS, CPUX_REG(rs));     // argument 1: value of rs
    vmimtCallResult((vmiCallFn)vmic_CountLeadingZeros, CPUX_GBITS, CPUX_REG(rd));
}

// function called at run time by embedded call
static Uns32 vmic_CountLeadingZeros(Uns32 value) {
    Uns32 temp = 32;
    Int32 i;
    for(i = 31; i >= 0; i--) {
        if(value & (1 << i)) {
            temp = 31 - i;
            break;
        }
    }
    return temp;
}
```

Notes and Restrictions

1. `bits` may be 8, 16, 32 or 64.
2. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of

`vmimtArg`-prefixed functions exactly matches the function prototype (`vmic_CountLeadingZeros`, in the above example).

3. If the function or any child uses floating point operations, include a call to `vmirtRestoreFPState` to ensure that native floating point status is restored to that in force when the processor was created. See the *VMI Run Time Function Reference* for more information.

8 Connection Operations

Processor models may have *connections* associated with them. Connections are used to implement direct communication channels between processors. These communication channels allow the processors to communicate without sharing memory. Currently, the only form of connection object supported is a FIFO queue.

Section 2.3 in the *VMI Run Time Function Reference* describes functions that are used to create FIFO connections between processors and set the values of `cpux->inputConn` and `cpux->outputConn`. This section describes routines that are used to send and receive data using connection objects.

8.1 vmimtConnGetRB

Prototype

```
void vmimtConnGetRB(
    Uns32           bits,
    vmiReg          rd,
    vmiReg          connReg,
    Bool            peek,
    vmiConnUpdateFn updateCB
);
```

Description

This function emits code to perform a blocking read from a connection container object specified by the processor pseudo-register `connReg`, which must previously have been initialized. The data value to read has width `bits` and should be assigned to register `rd`. If `peek` is `False`, the value will be removed from the container; otherwise, it will be copied from the container.

In the case that the input connection container is empty prior to the attempted read, the processor will stop executing. It will remain stopped until some other processor writes to the container object using `vmimtConnPutRB` (or a related function). When this happens, the callback function `updateCB` is called, which determines how the waiting processor should respond: typically, the response should be to restart the waiting processor using either `vmirtRestartNow` or `vmirtRestartNext`. Upon restart, the current simulated instruction will be restarted, with the effect that the processor will retry the connection read.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32               // number of GPRs
#define CPUX_GBITS 32               // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32         regs[CPUX_GREGS]; // 32-bit GPRs
    vmiConnInputP inputConn;        // input connection
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

// See section 2.3 in the OVP Run Time Function Reference for an example of
// constructing connections and initializing cpux->inputConn

// Callback activated when the current connection is updated
static VMI_CONN_UPDATE_FN(restartAfterBlock) {
    vmirtRestartNow(processor);
}

// Emit code to get from connection and block if data not available
static void emitConnGetOrBlock(Uns32 rd, Bool peek) {

    vmimtConnGetRB(
        CPUX_BITS,
```

```
        CPUX_REG(rd),
        CPUX_OFFSET(inputConn),
        peek,
        restartAfterBlock
    );
}
```

Notes and Restrictions

1. See also section 10 in the *VMI Run Time Function Reference* which allows a run time read from a connection object.

8.2 vmimtConnGetRNB

Prototype

```
void vmimtConnGetRNB(
    Uns32  bits,
    vmiReg rd,
    vmiReg connReg,
    Bool   peek,
    vmiReg flag
);
```

Description

This function emits code to perform a nonblocking read from a connection container object specified by the processor pseudo-register `connReg`, which must previously have been initialized. The data value to read has width `bits` and should be assigned to register `rd`. If `peek` is `False`, the value will be removed from the container; otherwise, it will be copied from the container.

In the case that the input connection container is empty prior to the attempted read, the 8-bit processor register `flag` is assigned the value 0 and `rd` is unchanged; otherwise, `flag` is assigned the value 1.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32               // number of GPRs
#define CPUX_GBITS 32               // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8          readOK;           // FIFO readOK flag
    Uns32         regs[CPUX_GREGS]; // 32-bit GPRs
    vmiConnInputP inputConn;        // input connection
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_READOK     CPUX_OFFSET(readOK)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

// See section 2.3 in the OVP Run Time Function Reference for an example of
// constructing connections and initializing cpux->inputConn

// Emit code to get from connection without blocking if data not available
static void emitConnGet(Uns32 rd, Bool peek) {
    vmimtConnGetRNB(
        CPUX_BITS,
        CPUX_REG(rd),
        CPUX_OFFSET(inputConn),
        peek,
        CPUX_READOK
    );
}
```

Notes and Restrictions

1. See also Section 10 in *VMI Run Time Function Reference* which allows a run time read from a connection object.

8.3 vmimtConnPutRB

Prototype

```
void vmimtConnPutRB(
    Uns32           bits,
    vmiReg          connReg,
    vmiReg          ra,
    vmiConnUpdateFn updateCB
);
```

Description

This function emits code to perform a blocking write to a connection container object specified by the processor pseudo-register `connReg`, which must previously have been initialized. The data value to write has width `bits` and should obtained from register `ra`.

In the case that the input connection container is full prior to the attempted write, the processor will stop executing. It will remain stopped until some other processor reads from the container object using `vmimtConnGetRB` (or a related function) to make space in the container. When this happens, the callback function `updateCB` is called, which determines how the waiting processor should respond: typically, the response should be to restart the waiting processor using either `vmirtRestartNow` or `vmirtRestartNext`. Upon restart, the current simulated instruction will be restarted, with the effect that the processor will retry the connection write.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32               // number of GPRs
#define CPUX_GBITS 32               // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32          regs[CPUX_GREGS];// 32-bit GPRs
    vmiConnOutputP outputConn;      // output connection
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

// See section 2.3 in the OVP Run Time Function Reference for an example of
// constructing connections and initializing cpux->outputConn

// Callback activated when the current connection is updated
static VMI_CONN_UPDATE_FN(restartAfterBlock) {
    vmirtRestartNow(processor);
}

// Emit code to put to connection and block if data not available
static void emitConnPutOrBlock(Uns32 ra) {

    vmimtConnPutRB(
        CPUX_BITS,
        CPUX_OFFSET(outputConn),
        CPUX_REG(ra),
        restartAfterBlock
    );
```

```
}
```

Notes and Restrictions

1. See also section 10 in the *VMI Run Time Function Reference* which allows a run time write to a connection object.

8.4 vmimtConnPutRNB

Prototype

```
void vmimtConnPutRNB(
    Uns32  bits,
    vmiReg connReg,
    vmiReg ra,
    vmiReg flag
);
```

Description

This function emits code to perform a nonblocking write to a connection container object specified by the processor pseudo-register `connReg`, which must previously have been initialized. The data value to write has width `bits` and should obtained from register `ra`.

In the case that the output connection container is full prior to the attempted write, the 8-bit processor register `flag` is assigned the value 0 and the value is not written; otherwise, `flag` is assigned the value 1.

Example

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32               // number of GPRs
#define CPUX_GBITS 32               // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns8           writeOK;         // FIFO writeOK flag
    Uns32          regs[CPUX_GREGS];// 32-bit GPRs
    vmiConnOutputP outputConn;      // output connection
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_WRITEOK    CPUX_OFFSET(writeOK)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])

// See section 2.3 in the OVP Run Time Function Reference for an example of
// constructing connections and initializing cpux->outputConn

// Emit code to put to connection without blocking if container full
static void emitConnPut(Uns32 ra) {

    vmimtConnPutRNB(
        CPUX_BITS,
        CPUX_OFFSET(outputConn),
        CPUX_REG(ra),
        CPUX_WRITEOK
    );
}
```

Notes and Restrictions

1. See also section 10 in the *VMI Run Time Function Reference* which allows a run time write to a connection object.

9 Floating Point Operations

In general, modeling of floating point operations is hard. Although many processors claim to be IEEE Standard 754 compliant, there are usually implementation details that deviate from the Standard in some respects; for example, many processors implement variants of flush-to-zero mode (FZ) or denormals-are-zero mode (DAZ) which are not covered in the Standard and are inconsistently implemented in different hardware.

The VMI API has been designed so that a spectrum of implementation approaches is available for a particular instruction, depending on how closely the VMI primitives match the required behavior. For example:

1. It is possible to use VMI floating point primitives without modification. This provides fastest-possible simulation as floating point operations are efficiently mapped to native floating point instructions.
2. It is possible to use VMI floating point primitives directly with some result adjustment in cases where `NaN` or integer/unsigned indeterminate results are generated. This result adjustment is efficiently done using *handler* functions.
3. It is possible to use VMI floating point primitives directly with result adjustment applied to *every* result, whether a `NaN` or not.
4. It is possible to specify *user-defined* operation primitives, which are callback functions executed within the scope of a (possibly SIMD) floating point instruction.
5. If no other approach is possible, the instruction can be implemented using non-floating-point VMI primitives (usually an embedded call).

The *VMI Morph Time Function* API implements functions allowing many floating point operations to be implemented natively. Every API function is available in both a simple and a SIMD form. In the SIMD form, a number of operations are performed in parallel, with the results committed only if no operation raises an enabled exception. Unless otherwise stated, floating point operations comply with IEEE Standard 754 - 2008.

The *VMI Run Time Function Reference* manual describes how general characteristics of a floating point unit can be configured, and also describes functions to query and update the simulated floating point control word.

9.1 General Floating Point Operation Flow

The floating point operation primitives described later in this section all use a similar flow, outlined below in full SIMD form, as pseudo-code:

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whether stages in the flow are present or absent depends on FPU *configuration* settings, described below.

9.2 `vmiFPConfig` Structure

Every floating point operation is executed with a *configuration* that specifies implementation-specific details of its implementation. There is a *default configuration*, defined using the VMI Run Time function `vmirtConfigureFPU`. In addition, any floating point operation can specify an *operation-specific configuration*, which takes priority over the default configuration. Normally, a default configuration is set up when the model is initialized, and operation-specific configurations used only for those instructions which require different behavior.

Configuration information is given in a static constant structure of type `vmiFPConfig`, defined in `vmiTypes.h` as follows:

```
typedef struct vmiFPConfigS {
    Uns32               QNaN32;
    Uns64               QNaN64;
    Uns16               indeterminateUns16;
    Uns32               indeterminateUns32;
    Uns64               indeterminateUns64;
    vmiFPQNaN32ResultFn QNaN32ResultCB;
    vmiFPQNaN64ResultFn QNaN64ResultCB;
    vmiFPInd16ResultFn  indeterminate16ResultCB;
    vmiFPInd32ResultFn  indeterminate32ResultCB;
    vmiFPInd64ResultFn  indeterminate64ResultCB;
    vmiFPTinyResultFn   tinyResultCB;
    vmiFPTinyArgumentFn tinyArgumentCB;
    vmiFP32ResultFn     fp32ResultCB;
    vmiFP64ResultFn     fp64ResultCB;
    vmiFPArithExceptFn  fpArithExceptCB;
    vmiFPFlags          suppressFlags;
    Bool                stickyFlags;
    Bool                fzClearsPF;
    Bool                tininessAfterRounding;
```

```
} vmiFPConfig;
```

The structure fields are as follows:

1. `QNaN32` specifies the bit pattern produced when a floating point operation generates a 32-bit `QNaN` result. Normally this should be `0x7fc00000`, but older versions of IEEE Standard 754 permit the most significant bit of the significand to be reversed for `QNaN` and `SNaN`. On a processor such as the MIPS where `QNaN` and `SNaN` values are indeed reversed, a different value should be specified (for example, `0x7fbfffff` for MIPS).
2. `QNaN64` specifies the bit pattern produced when a floating point operation generates a 64-bit `QNaN` result. Normally this should be `0x7ff8000000000000ULL`, but a processor such as the MIPS where `QNaN` and `SNaN` values are reversed, a different value should be specified (for example, `0x7ff7fffffffffffULL` for MIPS).
3. `QNaN32ResultCB` is a callback function which, if given, is called whenever a 32-bit `QNaN` result is generated to give the processor model the opportunity to modify the resulting `QNaN` value. See the *VMI Morph Time Function Reference* for more information about `QNaN` result handlers.
4. `QNaN64ResultCB` is a callback function which, if given, is called whenever a 64-bit `QNaN` result is generated to give the processor model the opportunity to modify the resulting `QNaN` value. See the *VMI Morph Time Function Reference* for more information about `QNaN` result handlers.
5. `indeterminateUns16` specifies the bit pattern produced when a floating point operation generates a 16-bit indeterminate integer result. For processors compliant with IEEE Standard 754, this should be `0x8000`.
6. `indeterminateUns32` specifies the bit pattern produced when a floating point operation generates a 32-bit indeterminate integer result. For processors compliant with IEEE Standard 754, this should be `0x80000000`.
7. `indeterminateUns64` specifies the bit pattern produced when a floating point operation generates a 64-bit indeterminate integer result. For processors compliant with IEEE Standard 754, this should be `0x8000000000000000ULL`.
8. `indeterminate16ResultCB` is a callback function which, if given, is called whenever a 16-bit indeterminate result is generated to allow the processor model to provide the required indeterminate value. See the *VMI Morph Time Function Reference* for more information about indeterminate result handlers.
9. `indeterminate32ResultCB` is a callback function which, if given, is called whenever a 32-bit indeterminate result is generated to allow the processor model to provide the required indeterminate value. See the *VMI Morph Time Function Reference* for more information about indeterminate result handlers.
10. `indeterminate64ResultCB` is a callback function which, if given, is called whenever a 64-bit indeterminate result is generated to allow the processor model to provide the required indeterminate value. See the *VMI Morph Time Function Reference* for more information about indeterminate result handlers.
11. `tinyResultCB` is a callback function which, if given, is called whenever a tiny (denormalized) result is generated to give the processor model the opportunity to

modify the resulting tiny value (or take any other action). See the *VMI Morph Time Function Reference* for more information about tiny result handlers.

12. `tinyArgumentCB` is a callback function which, if given, is called whenever a denormalized argument is detected to give the processor model the opportunity to modify the argument value (or take any other action). See the *VMI Morph Time Function Reference* for more information about tiny argument handlers.
13. `fp32ResultCB` is a callback function which, if given, is called whenever a 32-bit floating point result is generated to allow the processor model to modify the result value or flags. Result callbacks are typically specified only for instruction-specific configurations, so this field should usually be `NULL` for a configuration used with `vmirtConfigureFPU`. See the *VMI Morph Time Function Reference* for more information about result handlers.
14. `fp64ResultCB` is a callback function which, if given, is called whenever a 64-bit floating point result is generated to allow the processor model to modify the result value or flags. Result callbacks are typically specified only for instruction-specific configurations, so this field should usually be `NULL` for a configuration used with `vmirtConfigureFPU`. See the *VMI Morph Time Function Reference* for more information about result handlers.
15. `fpArithExceptCB` is an exception handler callback function which is called whenever a floating point operation generates unmasked exceptions. The exception handler callback will typically update processor state and cause a jump to a vector address. See the *VMI Morph Time Function Reference* for more information about the exception handler callback.
16. `suppressFlags` is a field if type `vmiFPFlags` which enables flags generated by a floating point operation to be suppressed: any flag set to 1 in the bitmask will be masked out of the operation result flags.
17. `stickyFlags` is a boolean field which specifies whether the operation result flags should replace any current value of the output flags (if `False`) or whether operation flags should be combined with existing flags using bitwise-or (if `True`).
18. `fzClearsPF` is a boolean field that should be `True` if the processor implements *flush-to-zero* mode and when denormal results are flushed to zero *the precision flag in the floating point status word is not set*. If the processor does *not* implement flush-to-zero mode, or if the precision flag should be *set* when results are flushed to zero, then the argument should be `False`. Most floating point implementations set the precision flag when a denormal result is flushed to zero (e.g. x86, MIPS) but some do not (e.g. ARM).
19. `tininessAfterRounding` is a boolean field that indicates whether tininess should be detected before rounding a result or afterwards. This affects behavior for intermediate results that round to a minimum normal value of greater absolute magnitude. The boolean affects all floating point operations using IEEE types.

9.3 `vmiFPControlWord` Structure

The dynamic behavior of a processor simulated FPU is specified by a `vmiFPControlWord` structure, defined in `vmiTypes.h`:

```
typedef struct vmiFPControlWordS {
```

```
                        // INTERRUPT MASKS
    Uns32   IM  : 1;    // invalid operation mask
    Uns32   DM  : 1;    // denormal mask
    Uns32   ZM  : 1;    // divide-by-zero mask
    Uns32   OM  : 1;    // overflow mask
    Uns32   UM  : 1;    // underflow mask
    Uns32   PM  : 1;    // precision mask

                        // ROUNDING AND PRECISION
    Uns32   RC  : 2;    // rounding control
    Bool    FZ  : 1;    // flush to zero
    Uns32   DAZ : 1;    // denormals are zeros flag

} vmiFPControlWord;
```

The first six fields are *interrupt masks* that specify whether a floating-point arithmetic exception of the indicated type should by masked. If the exception is masked (the bit is 1), the exception will be ignored. If the exception is unmasked (the bit is 0), any exception of the indicated type will be signaled by calling the processor *floating point exception handler* (defined with the `VMI_FP_ARITH_EXCEPT_FN` macro in `vmiAttrs.h` and passed as the `fpArithExceptCB` field of the configuration). Masks other than `DM` are the standard IEEE Standard 754 exception masks; `DM` is a non-standard mask indicating denormal operands. Each mask corresponds to a flag described in the next section.

The `RC` field specifies the *rounding control* to use when arithmetic results cannot be exactly represented and precision exceptions are masked. The field value should be one of the first four members of the `vmiFPRC` enumeration:

```
typedef enum {

    // these values are valid in both conversion functions and in the rounding
    // control field of vmiFPControlWord, below
    vmi_FPR_NEAREST = 0,    // round towards nearest (even)
    vmi_FPR_NEG_INF = 1,    // round towards negative infinity
    vmi_FPR_POS_INF = 2,    // round towards positive infinity
    vmi_FPR_ZERO    = 3,    // round towards zero

    // these values are valid in conversion functions only
    vmi_FPR_CURRENT = 4,    // use currently-active rounding control
    vmi_FPR_AWAY    = 5,    // round towards nearest, tie away
    vmi_FPR_ODD     = 6     // round to odd (Von Neumann rounding)

} vmiFPRC;
```

The `FZ` field specifies that denormal results should be flushed to zero. The `DAZ` field specifies that denormal arguments should be treated as zero. Neither of these modes are IEEE 754 compliant, but many processors support variants of them.

The floating point control word in use for a processor can be set and fetched using two functions from the *VMI Run Time Function* API:

```
//
// Get the processor floating point control word
//
vmiFPControlWord vmirtGetFPControlWord(vmiProcessorP processor);

//
// Set the processor floating point control word
//
```

```
void vmirtSetFPControlWord(vmiProcessorP processor, vmiFPControlWord fpcw);
```

9.4 vmiFPFlags Structure

The exception flags generated by a floating point instruction are specified by a `vmiFPFlags` structure, defined in `vmiTypes.h`:

```
typedef struct vmiFPFlagsS {
    Uns32 I : 1;    // invalid operation flag
    Uns32 D : 1;    // denormal flag
    Uns32 Z : 1;    // divide-by-zero flag
    Uns32 O : 1;    // overflow flag
    Uns32 U : 1;    // underflow flag
    Uns32 P : 1;    // precision flag
} vmiFPFlags;
```

A brief description of each flag follows: refer to the IEEE 754 Standard for more information on all flags except the non-standard `D` flag.

`I`: *invalid operation flag*
This flag is set whenever an operation is considered invalid by the FPU. Examples are 0 divided by 0, subtracting infinity from infinity, `NaN` inputs to some instructions, or attempting to find the square root of a negative number. If the exception is masked by the `IM` bit in the control word, the result of the floating point operation is a `NaN`. The floating point configuration may specify a *NaN handler* to configure the exact `NaN` that is returned in these circumstances.

`Z`: *divide-by-zero flag*
This flag is set whenever division of a finite non-zero is attempted. If the exception is masked by the `ZM` bit in the control word, a properly-signed infinity is generated.

`O`: *overflow flag*
This flag is set whenever a value is too large to be represented. For example, multiplication of two very large numbers can generate an overflow. If the exception is masked by the `OM` bit in the control word, a properly-signed infinity is generated.

`U`: *underflow flag*
The behavior of this flag depends on the corresponding mask bit (`UM`) in the control word. If the underflow exception is *masked*, the flag is set only if a result is *both tiny and inexact*; if the underflow exception is unmasked, the flag is set for *any tiny result*. As an example, dividing a very small number by a large number can generate an underflow. If the exception is masked by the `UM` bit in the control word, a denormal or zero result is produced, as appropriate.

`P`: *precision flag*
This flag is set whenever some precision is lost by a floating point operation. For example, dividing 1.0 by 10.0 does not generate an exact result and causes the precision flag to be set. If the exception is masked by the `PM` bit in the control word, the result is rounded according to the rounding control specified by the `RC` field of the control word or the rounding control specified for the instruction (see above).

D: *denormal operand flag*
This flag is set whenever the input to a floating point operation is denormalized. If exceptions are enabled, this causes an exception *before* the floating point operation starts. If the exception is masked by the DM bit in the control word, the current floating point operation continues normally. If the DAZ bit is set in the control word, then denormal operands are rounded to zero. The floating point configuration may specify a *tiny operand handler* to configure the exact model behavior under which this happens. *This flag is not part of the IEEE 754 Standard, and is costly to simulate: set field `suppressFlags.f.D` in the floating point configuration if the flag is not required to improve floating point performance*.

Note that arithmetic instructions can signal more than one exception: for example, it is possible to get both an underflow and precision exception signalled by a single floating point instruction.

All floating point arithmetic instructions have a `vmiReg` target register that is assigned the instruction exception flags, provided that the floating point exception handler is not called. If the floating point exception handler *is* called (at least one of the exceptions raised by the instruction is not masked) the target flags register is not updated and the generated flags are instead passed as the `flags` argument of the `vmiFPArithExceptFn` handler function (see below).

9.5 `vmiFType` Enumeration

The type of the arguments for floating point instructions is specified by the `vmiFType` enumeration, defined in `vmiTypes.h`:

```
typedef enum {

    // these values specify that evaluation should be performed using IEEE 754
    // semantics (intermediates are the same type)
    vmi_FT_32_IEEE_754 = 32 | VMI_FT_IEEE_754,  // 32-bit floating point
    vmi_FT_64_IEEE_754 = 64 | VMI_FT_IEEE_754,  // 64-bit floating point

    // these values specify that evaluation should be performed using Intel x87
    // semantics (intermediates are promoted to 80-bit long double format)
    vmi_FT_32_X87      = 32 | VMI_FT_X87,       // 32-bit floating point
    vmi_FT_64_X87      = 64 | VMI_FT_X87,       // 64-bit floating point
    vmi_FT_80_X87      = 80 | VMI_FT_X87,       // 80-bit floating point

    // these values are valid in conversion operations only
    vmi_FT_16_INT      = 16 | VMI_FT_INT,       // 16-bit signed integer
    vmi_FT_32_INT      = 32 | VMI_FT_INT,       // 32-bit signed integer
    vmi_FT_64_INT      = 64 | VMI_FT_INT,       // 64-bit signed integer
    vmi_FT_16_UNS      = 16 | VMI_FT_UNS,       // 16-bit unsigned integer
    vmi_FT_32_UNS      = 32 | VMI_FT_UNS,       // 32-bit unsigned integer
    vmi_FT_64_UNS      = 64 | VMI_FT_UNS,       // 64-bit unsigned integer

} vmiFType;
```

Members `vmi_FT_32_IEEE_754` and `vmi_FT_64_IEEE_754` specify IEEE-compliant 32 and 64 bit floating point values and semantics, respectively (see below for semantic differences between IEEE and x87 modes).

Members vmi_FT_32_X87, vmi_FT_64_X87, and vmi_FT_80_X87 specify x87 32, 64 and 80 bit values and semantics, respectively (see below for semantic differences between IEEE and x87 modes).

Members vmi_FT_16_INT, vmi_FT_32_INT and vmi_FT_64_INT specify 16, 32 and 64 bit signed integer values and are valid as the source or target of floating point conversion functions only.

Members vmi_FT_16_UNS, vmi_FT_32_UNS and vmi_FT_64_UNS specify 16, 32 and 64 bit unsigned integer values and are valid as the source or target of floating point conversion functions only.

9.6 IEEE and x87 Semantic Differences

IEEE and x87 semantics differ in these ways.

Operand and Intermediate Size

When using IEEE semantics, calculations are performed using the operand size (32-bit float or 64-bit double). When using x87 semantics, operands are first converted to 80-bit long doubles and the result is rounded to float or double length on operation completion, if required. Note that x87 semantics can therefore cause *two* rounding events, firstly when an intermediate result is rounded to 80-bit precision, and a secondly when the final result is rounded to 32-bit or 64-bit precision from 80-bit precision.

For IEEE ternary floating point operations which are specified *not* to round intermediates (argument roundInt is False) the intermediate result of the multiply is represented using infinite precision. This means that such operations correspond to the IEEE definition of *fused-multiply-add* operations. When using x87 semantics with roundInt specified as False, the intermediate result is rounded to 80-bit precision. When roundInt is True, intermediate results of a ternary operation are rounded to the operand size in both cases.

NaN Operands

When operations have more than one NaN operand and a NaN result is generated, the results differ when using IEEE and x87 semantics, as follows:

Source Operands	Result
SNaN and QNaN, QNaN and SNaN	x87: QNaN source operand IEEE: First NaN operand, converted to QNaN
SNaN and SNaN	x87: SNaN operand with largest significand, converted to QNaN IEEE: First SNaN operand, converted to QNaN
QNaN and QNaN	x87: QNaN operand with largest significand IEEE: First QNaN operand

9.7 QNaN/SNaN Polarity Switch

This section describes how to control QNaN/SNaN polarity using a floating point configuration. This affects the highlighted stages in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Older versions of IEEE Standard 754 permit the most significant bit of the significand to be of either polarity to represent QNaN and SNaN. For most processors, a QNaN is indicated by the most significant bit being 1, and SNaN is indicated by the most significant bit being 0 (this allows any SNaN to be efficiently converted to a QNaN by setting the significand msb). Some processor architectures (e.g. legacy MIPS) define the values to be the other way round.

The QNaN/SNaN polarity is controlled by two fields in the configuration structure. QNaN32 specifies the bit pattern produced when a floating point operation generates a 32-bit QNaN result, and also implicitly the polarity of the signaling bit. QNaN64 defines an analogous value for a 64-bit QNaN.

If the default QNaN values imply a reversed signaling bit polarity, NaN values are automatically switched from reversed format as arguments are processed, and QNaN results are switched to reversed format on operation completion. Stages between the two highlighted lines in the above description always operate on NaN values with standard polarity.

9.8 Denormalized Argument Handler

This section describes how to control denormals-are-zeros mode (DAZ mode) using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whenever a denormalized argument is detected for a floating point operation and the denormals-are-zero (DAZ) bit is set in the current floating point control word, the tiny argument handler is called (specified using field tinyArgumentCB in the active configuration). The handler function is defined using the VMI_FP_TINY_ARGUMENT_FN macro from vmiTypes.h:

```
#define VMI_FP_TINY_ARGUMENT_FN(_NAME) vmiFP80Arg _NAME( \
    vmiProcessorP processor,        \
    vmiFP80Arg    value,            \
    vmiFPFlagsP   setFlags          \
)
```

The handler is passed the following arguments:

1. The current processor;
2. The tiny argument value, represented as a vmiFP80Arg;
3. An argument of setFlags of type vmiFPFlagsP, in which bits can be set to 1 to indicate a floating point exception caused by handling the tiny result.

It must return an appropriately-signed zero value as a vmiFP80Arg and may perform other updates to processor state.

The vmiFP80Arg type is defined as follows, and holds a floating point value in the Intel x87 80-bit format:

```
#define VMI_FP_80_BYTES 10
typedef union vmiFP80ArgU {
    Flt80      f80;
    Flt80Parts f80Parts;
    Uns8       bytes[VMI_FP_80_BYTES];
} vmiFP80Arg;
```

Type Flt80Parts is one of a set of types used to decode components of floating point numbers:

```
typedef struct Flt16PartsS {
    Uns32 fraction : 10;
    Uns32 exponent :  5;
    Bool  sign     :  1;
} Flt16Parts;

typedef struct Flt32PartsS {
    Uns32 fraction : 23;
    Uns32 exponent :  8;
    Bool  sign     :  1;
} Flt32Parts;

typedef struct Flt64PartsS {
    Uns64 fraction : 52;
    Uns32 exponent : 11;
    Bool  sign     :  1;
} Flt64Parts;

typedef struct Flt80PartsS {
    Uns64 fraction : 64;
    Uns32 exponent : 15;
    Bool  sign     :  1;
} Flt80Parts;
```

Example

This example is derived from the standard MIPS model.

```
static VMI_FP_TINY_ARGUMENT_FN(handleTinyArgument) {

    mipsP tc = (mipsP)processor;

    // when denormal arguments are flushed to zero, set Precision flag unless
    // attribute FA_DONT_SET_I_FLAG is also specified (floating point compare)
    if(!(tc->fopAttrs & FA_DONT_SET_I_FLAG)) {
        setFlags->f.P = 1;
    }

    // return appropriately-signed zero
    value.f80Parts.fraction = 0;
    value.f80Parts.exponent = 0;

    return value;
}
```

This returns an appropriately-signed zero value and updates processor state to force the *inexact* flag to be set on instruction completion.

If the DAZ bit is set in the current floating point control word and no tiny argument handler is specified in the configuration, denormal inputs are flushed to an appropriately-signed zero and the precision flag is set in all cases. Default behaviour cannot be used for the MIPS model because some operations do not set the precision flag, even though they flush their arguments to zero.

IMPORTANT NOTE: the returned result must not be tiny – if it is, the simulation may hang.

9.9 Tiny Result Handler

This section describes how to control flush-to-zero mode (FZ mode) using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whenever an underflow exception is generated by a floating point operation and the flush-to-zero (FZ) bit is set in the current floating point control word, the tiny result handler is called (specified using field tinyResultCB in the active configuration). The handler function is defined using the VMI_FP_TINY_RESULT_FN macro from vmiTypes.h:

```
#define VMI_FP_TINY_RESULT_FN(_NAME) vmiFP80Arg _NAME( \
    vmiProcessorP processor,        \
    vmiFP80Arg    value,            \
    Bool          isFlt32,          \
    Bool          isIntermediate,   \
    vmiFPFlagsP   setFlags          \
)
```

The handler function is passed the following arguments:

1. The processor;
2. The result value, represented as a vmiFP80Arg;
3. A Boolean indicating whether the result is 32-bit (if False, it is 64-bit);
4. A Boolean indicating whether the result is the final result from a VMI floating point operation (if False, it is an intermediate result of a ternary operation);
5. An argument of setFlags of type vmiFPFlagsP, in which bits can be set to 1 to indicate a floating point exception caused by handling the tiny result.

The handler function should return the desired result, encoded as a vmiFP80Arg.

Example

This example is derived from the standard MIPS model.

```
static VMI_FP_TINY_RESULT_FN(handleTinyResult) {

    mipsP     tc         = (mipsP)processor;
    Bool      isNegative = value.bytes[VMI_FP_80_BYTES-1] & 0x80;
    tinyValue tv;
```

```
    GET_VPE;

    // get FPU control bits
    Bool FS = COP1_FIELD(vpe, FENR, FS);
    Bool FN = COP1_FIELD(vpe, FCSR, FN);
    Bool FO = COP1_FIELD(vpe, FCSR, FO);

    // expect either FS or FN to be set if we get here
    VMI_ASSERT(
        FS || FN,
        "expected FS or FN to be set"
    );

    // should not be called for intermediates if FO is set
    VMI_ASSERT(
        !(isIntermediate && FO),
        "unexpected intermediate with FO bit set"
    );

    // when results are flushed to zero, set Underflow and Precision flags
    setFlags->f.U = 1;
    setFlags->f.P = 1;

    // indicate that tiny results are allowed for this function
    tc->fopAttrs |= FA_ALLOW_O_DENORMALS;

    // get current rounding mode
    vmiFPRC rc = getCurrentRoundingMode(tc);

    if(FN && !isIntermediate && (rc==vmi_FPR_NEAREST)) {

        // get minnorm/2 for the result
        vmiFP80Arg minNormDiv2 = (
            isFlt32 ?
            tinyValues32[TV_PLUS_MINNORM_DIV_2] :
            tinyValues64[TV_PLUS_MINNORM_DIV_2]
        );

        // is value >= minnorm/2?
        Bool geThanMinNormDiv2 = (
            ((value.bytes[9]&0x7f) >= minNormDiv2.bytes[9]) &&
            ( value.bytes[8]       >= minNormDiv2.bytes[8]) &&
            ( value.bytes[7]       >= minNormDiv2.bytes[7])
        );

        // here if FN should be applied
        if(isNegative) {
            tv = geThanMinNormDiv2 ? TV_MINUS_MINNORM : TV_MINUS_0;
        } else {
            tv = geThanMinNormDiv2 ? TV_PLUS_MINNORM : TV_PLUS_0;
        }

    } else {

        // here if FS should be applied
        if(isNegative) {
            tv = (rc==vmi_FPR_NEG_INF) ? TV_MINUS_MINNORM : TV_MINUS_0;
        } else {
            tv = (rc==vmi_FPR_POS_INF) ? TV_PLUS_MINNORM : TV_PLUS_0;
        }
    }

    return isFlt32 ? tinyValues32[tv] : tinyValues64[tv];
}
```

The MIPS processor does not generate denormalized results in the normal case – usually, operations producing such results generate Unimplemented Operation exceptions instead. However, it has three special mode bits (FS, FO and FN) that cause denormalized results

to be flushed either to zero or to the smallest normalized value, depending on the rounding mode (among other things).

The example tiny result handler examines the offending tiny result value and returns either zero or the smallest normalized value (appropriately signed) depending on the processor state. It also updates processor state to force the *inexact* and *underflow* flags to be set on instruction completion.

9.10 32-Bit and 64-Bit General Result Handlers

This section describes how to control handling of general operation results using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whenever an operation result is produced, a result handler can be supplied which can modify the resulting value if required (specified using `fp32ResultCB` and `fp64ResultCB` fields in the active configuration). Unlike `QNaN` result handlers (described next), general result handlers are called for *every* generated result, not just `QNaN` results. The 32-bit general handler function is defined using the `VMI_FP_32_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_32_RESULT_FN(_NAME) Uns32 _NAME( \
    vmiProcessorP processor,    \
    Uns32         result32,     \
    Uns32         argNum,       \
    vmiFPArgP     args,         \
    vmiFPFlagsP   setFlags      \
)
typedef VMI_FP_32_RESULT_FN((*vmiFP32ResultFn));
```

The handler function is passed the following:

1. The processor generating the result;
2. The result value, represented as an `Uns32`;
3. A count of the number arguments to the operation;
4. An array of `argNum` arguments to the operation;
5. An argument of `setFlags` of type `vmiFPFlagsP`, in which bits can be set to 1 to indicate a floating point exception caused by handling the result.

The handler function should return the desired 32-bit result, encoded as an `Uns32`. It is passed an ordered list of all arguments to the operation in the `args` array, which holds `argNum` values. Each value is a `vmiFPArg` structure, defined as follows:

```
#define VMI_FP_80_BYTES 10
typedef struct vmiFPArgS {
    vmiFType type;
```

```
    union {
        // use these for 16-bit types
        Uns16      u16;
        Int16      i16;
        Flt16Parts f16Parts;
        // use these for 32-bit types
        Uns32      u32;
        Int32      i32;
        Flt32      f32;
        Flt32Parts f32Parts;
        // use these for 64-bit types
        Uns64      u64;
        Int64      i64;
        Flt64      f64;
        Flt64Parts f64Parts;
        // use these for 80-bit types
        Flt80      f80;
        Flt80Parts f80Parts;
        Uns8       bytes[VMI_FP_80_BYTES];
    };
} vmiFPArg;
```

Field `type` specifies the argument type – note that floating point conversion operations may take argument values of different size and class to the required result, so the handler must cope with such cases.

There will be 1, 2 or 3 values in the `args` list, depending on whether the floating point operation is a conversion, unary, binary or ternary.

Example

This example is derived from the OVP MIPS model. In the MIPS processor, floating point `RECIP` and `RSQRT` instructions always set the precision (inexact) flag, unless the result is zero, a `QNaN` or infinity.

```
static VMI_FP_32_RESULT_FN(recipRsqrtResult32) {

    if(!(result32 & ~MIPS_SIGN_32)) {
        // no action if (signed) zero result
    } else if((result32 & MIPS_EXP_32) != MIPS_EXP_32) {
        // not a QNaN or infinite result - force the inexact flag
        setFlags->f.P = 1;
    }

    // return unmodified result
    return result32;
}
```

There is a similar handler for 64-bit general results, defined using the `VMI_FP_64_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_64_RESULT_FN(_NAME) Uns64 _NAME( \
    vmiProcessorP processor,    \
    Uns64         result64,     \
    Uns32         argNum,       \
    vmiFPArgP     args,         \
    vmiFPFlagsP   setFlags      \
)
typedef VMI_FP_64_RESULT_FN((*vmiFP64ResultFn));
```

The handler works in identical fashion to the 32-bit general result handler, the only difference being that it takes and returns values represented as an `Uns64`.

9.11 32-Bit and 64-Bit QNaN Handlers

This section describes how to control handling of QNaN operation results using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whenever a QNaN value is produced as a result, a QNaN handler can be supplied which can modify the resulting value if required (specified using QNaN32ResultCB and QNaN64ResultCB fields in the active configuration). The 32-bit QNaN handler function is defined using the VMI_FP_QNAN32_RESULT_FN macro from vmiTypes.h:

```
#define VMI_FP_QNAN32_RESULT_FN(_NAME) Uns32 _NAME( \
    vmiProcessorP processor,    \
    Uns32         QNaN32,       \
    Uns32         NaNArgNum,    \
    vmiFPArgP     NaNArgs,      \
    Uns32         allArgNum,    \
    vmiFPArgP     allArgs       \
)
```

The handler function is passed the following:

1. The processor generating the QNaN result;
2. The QNaN value, represented as an Uns32;
3. A count of the number of NaN arguments to the operation;
4. An array of NaNArgNum NaN arguments to the operation;
5. A count of *all* arguments to the operation;
6. An array of allArgNum arguments to the operation.

The handler function should return the desired 32-bit result, encoded as an Uns32.

If the operation had any NaN inputs (NaNArgNum is non-zero), then the handler can obtain an ordered list of those values from the NaNArgs array, which holds NaNArgNum values. Each value is a vmiFPArg structure, defined as follows:

```
#define VMI_FP_80_BYTES 10
typedef struct vmiFPArgS {
    vmiFType type;
```

```
    union {
        // use these for 16-bit types
        Uns16      u16;
        Int16      i16;
        Flt16Parts f16Parts;
        // use these for 32-bit types
        Uns32      u32;
        Int32      i32;
        Flt32      f32;
        Flt32Parts f32Parts;
        // use these for 64-bit types
        Uns64      u64;
        Int64      i64;
        Flt64      f64;
        Flt64Parts f64Parts;
        // use these for 80-bit types
        Flt80      f80;
        Flt80Parts f80Parts;
        Uns8       bytes[VMI_FP_80_BYTES];
    };
} vmiFPArg;
```

Field `type` specifies the argument type – note that floating point conversion operations may take argument values of different size and class to the required result, so the handler must cope with such cases.

The handler is also passed an ordered list of all arguments to the operation in the `allArgs` array, which holds `allArgNum` values. Each value is a `vmiFPArg` structure, as described above. There will be 1, 2 or 3 values in this list, depending on whether the floating point operation is a unary, binary or ternary.

The `QNaN` handler usually returns a result which is a `NaN`. However, it is also legal to return a *non-`NaN` result*. In this case, *any Invalid Operation exception signalled for the current floating point operation is cleared*. This is typically useful when modeling instructions with special behavior when multiplying infinity and zero: the default behavior is to produce a `QNaN` in these cases, but some processors instead produce a non-`QNaN` result.

Example

This example is derived from the OVP MIPS model.

```
inline static Bool is32BitSNaN(vmiFPArgP arg, Bool standardNaN) {
    return (
        (arg->type==vmi_FT_32_IEEE_754) &&
        !(arg->u32 & MIPS_SBIT_32) == standardNaN
    );
}

inline static Bool is64BitSNaN(vmiFPArgP arg, Bool standardNaN) {
    return (
        (arg->type==vmi_FT_64_IEEE_754) &&
        !(arg->u64 & MIPS_SBIT_64) == standardNaN
    );
}

inline static Bool isSNaN(vmiFPArgP arg, Bool standardNaN) {
    return is32BitSNaN(arg, standardNaN) || is64BitSNaN(arg, standardNaN);
}

static VMI_FP_QNAN32_RESULT_FN(handleQNaN32) {
```

```
    mipsP tc          = (mipsP)processor;
    Bool  standardNaN = cfgStandardNaN(tc);
    Uns32 i;

    // PASS 1: if any argument is a 32-bit SNaN, return that (as a QNaN) in
    // standard NaN mode, or if it is any SNaN, return the canonical QNaN (in
    // legacy mode)
    for(i=0; i<NaNArgNum; i++) {
        if(standardNaN && is32BitSNaN(&NaNArgs[i], standardNaN)) {
            return NaNArgs[i].u32 | MIPS_SBIT_32;
        } else if(!standardNaN && isSNaN(&NaNArgs[i], standardNaN)) {
            return MIPS_QNAN_32;
        }
    }

    // PASS 2: if any argument is a 32-bit QNaN, return that
    for(i=0; i<NaNArgNum; i++) {
        if(is32BitQNaN(&NaNArgs[i], standardNaN)) {
            return NaNArgs[i].u32;
        }
    }

    // otherwise, return positive canonical QNaN or the calculated result
    return standardNaN ? IEEE_QNAN_32 : QNaN32;
}
```

The MIPS floating point unit differs from IEEE 754 semantics because, if a `QNaN` result is generated, the pattern for this is based upon `QNaN` operands *only* and not `SNaN` operands. The `QNaN` handler above selects and returns the first `QNaN` from the argument list, or, if no `QNaN` is found, it returns the default `QNaN` pattern.

There is a similar handler for 64-bit `QNaN` results, defined using the `VMI_FP_QNAN64_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_QNAN64_RESULT_FN(_NAME) Uns64 _NAME( \
    vmiProcessorP processor,    \
    Uns64         QNaN64,       \
    Uns32         NaNArgNum,    \
    vmiFPArgP     NaNArgs,      \
    Uns32         allArgNum,    \
    vmiFPArgP     allArgs       \
)
```

The handler works in identical fashion to the 32-bit `QNaN` handler, the only difference being that it takes and returns `QNaN` values represented as an `Uns64`.

IMPORTANT NOTE: avoid using any floating point operations within this and all other floating point handler functions. Failure to do so may cause programs to hang on the Windows operating system.

9.12 16-Bit, 32-Bit and 64-Bit Indeterminate Handlers

This section describes how to control handling of integer/unsigned indeterminate operation results using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whenever an indeterminate value is produced as a result of a conversion, an indeterminate handler can be supplied which can provide the required result (specified using `indeterminate16ResultCB`, `indeterminate32ResultCB` or `indeterminate64ResultCB` fields in the active configuration). The 32-bit handler function is defined using the `VMI_FP_IND32_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_IND32_RESULT_FN(_NAME) Uns32 _NAME( \
    vmiProcessorP processor,    \
    vmiFPArg      value,        \
    Bool          isSigned      \
)
```

The handler function is passed the following:

1. The processor generating the indeterminate result;
2. The argument prior to conversion, represented as a `vmiFPArg`;
3. A Boolean indicating whether a signed conversion was performed.

The handler function should return the desired result as an `Uns32`.

Example

This example is extracted from the standard ARM model.

```
static Bool isNegative(vmiFPArg value) {

    if(value.type==vmi_FT_32_IEEE_754) {
        return value.f32Parts.sign;
    } else if(value.type==vmi_FT_64_IEEE_754) {
        return value.f64Parts.sign;
    } else if(value.type==vmi_FT_80_X87) {
        return value.f80Parts.sign;
    } else {
        return False;
    }
}
```

```
static VMI_FP_IND32_RESULT_FN(handleIndeterminate32) {

    Uns32 result;

    if(isNaN(value)) {
        result = 0;
    } else if(isNegative(value)) {
        result = isSigned ? ARM_MIN_INT32 : ARM_MIN_UNS32;
    } else {
        result = isSigned ? ARM_MAX_INT32 : ARM_MAX_UNS32;
    }

    return result;
}
```

In the ARM processor, out-of-range values are clamped to the minimum and maximum bounds, and `NaN` inputs are clamped to zero. The ARM processor supports both signed and unsigned conversion.

There are similar handlers for 16-bit and 64-bit indeterminate value handling, defined using the `VMI_FP_IND16_RESULT_FN` and `VMI_FP_IND64_RESULT_FN` macros from `vmiTypes.h`:

```
#define VMI_FP_IND16_RESULT_FN(_NAME) Uns16 _NAME( \
    vmiProcessorP processor,    \
    vmiFPArg      value         \
)
#define VMI_FP_IND64_RESULT_FN(_NAME) Uns64 _NAME( \
    vmiProcessorP processor,    \
    vmiFPArg      value         \
)
```

The handlers work in identical fashion to the 32-bit indeterminate handler, the only difference being that they return an `Uns16` and `Uns64` result, respectively.

9.13 Floating Point Exceptions

This section describes how to control floating point exceptions using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Any unmasked floating point exceptions cause the configured *floating point arithmetic exception handler* to be called. The floating point arithmetic exception handler is of type `vmiFPArithExceptFn` and is specified as the `fpArithExceptCB` field of the active configuration:

```
#define VMI_FP_ARITH_EXCEPT_FN(_NAME) vmiFloatExceptionResult _NAME( \
    vmiProcessorP processor,    \
    vmiFPFlagsP   flags         \
)
typedef VMI_FP_ARITH_EXCEPT_FN((*vmiFPArithExceptFn));
```

The `flags` argument to the arithmetic exception handler indicates the exception flags set by the faulting instruction. The handler may modify the processor state to reflect the exception conditions (for example, by changing simulated register state, or using `vmirtSetPCException` to jump to a simulated exception vector). It may also modify fields in the by-ref `flags` structure to simulate flag behavior that diverges from the IEEE standard. It should return `VMI_FLOAT_CONTINUE` to indicate that simulation should continue or `VMI_FLOAT_UNHANDLED` to indicate that an irrecoverable model error has occurred and simulation should terminate.

Example

This example is derived from the OVP MIPS model. Various implementation-specific adjustments are made to the flag settings, and then function `mipsTakeException` is called to enable execution at the exception vector address if required.

```
static VMI_FP_ARITH_EXCEPT_FN(handleFPException) {

    mipsP        tc       = (mipsP)processor;
    vmiFPFlags   enables  = {getEnabledExceptions(tc)};
    mipsFPOpAttr fopAttrs = tc->fopAttrs;

    // handle denormal arguments
    if(!flags->f.D) {
```

```
        // not a denormal argument
    } else if(fopAttrs & FA_ALLOW_I_DENORMALS) {
        // denormal arguments valid for this instruction
        flags->f.D = 0;
    } else {
        // take Unimplemented Operation exception
        FORCE_UNIMPLEMENTED_OPERATION(flags);
    }

    // handle tiny results
    if(!flags->f.U) {
        // not a tiny result
    } else if(fopAttrs & FA_ALLOW_O_DENORMALS) {
        // tiny results valid for this instruction
    } else {
        // take Unimplemented Operation exception
        FORCE_UNIMPLEMENTED_OPERATION(flags);
    }

    // clear underflow if this instruction requires it
    if(fopAttrs & FA_CLEAR_U_FLAG) {
        flags->f.U = 0;
    }

    // take any pending exception
    if((flags->bits & enables.bits)) {
        GET_FPU;
        fpu->cop1Cause = flags->bits;
        mipsTakeException(tc, excCode_FPE, 0, False);
    }

    return VMI_FLOAT_CONTINUE;
}
```

9.14 vmimtFConvertRR, vmimtFConvertSimdRR

Prototypes

```
void vmimtFConvertRR(
    vmiFType      destType,
    vmiReg        fd,
    vmiFType      srcType,
    vmiReg        fa,
    vmiFPRC       rc,
    vmiReg        flags,
    vmiFPConfigCP config
);

void vmimtFConvertSimdRR(
    Uns32         num,
    vmiFType      destType,
    vmiReg        fd,
    vmiFType      srcType,
    vmiReg        fa,
    vmiFPRC       rc,
    vmiReg        flags,
    vmiFPConfigCP config
);
```

Description

These functions emit code to convert a value in register `ra` that is in format `srcType`, placing the result in register `fd` in format `destType`. `srcType` and `destType` can be any members of the `vmiFType` enumeration, so this function allows conversion between any pair of floating point or integral types. For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fd` and `fa` indicate the first register in a contiguous vector.

If the result cannot be exactly represented using the target type, rounding to that type is controlled by the `rc` argument of type `vmiFPRC`, defined as follows:

```
typedef enum {

    // these values are valid in both conversion functions and in the rounding
    // control field of vmiFPControlWord, below
    vmi_FPR_NEAREST = 0,    // round towards nearest (even)
    vmi_FPR_NEG_INF = 1,    // round towards negative infinity
    vmi_FPR_POS_INF = 2,    // round towards positive infinity
    vmi_FPR_ZERO    = 3,    // round towards zero

    // these values are valid in conversion functions only
    vmi_FPR_CURRENT = 4,    // use currently-active rounding control
    vmi_FPR_AWAY    = 5,    // round towards nearest, tie away
    vmi_FPR_ODD     = 6     // round to odd (Von Neumann rounding)

} vmiFPRC;
```

Values `vmi_FPR_NEAREST`, `vmi_FPR_NEG_INF`, `vmi_FPR_POS_INF` or `vmi_FPR_ZERO` are standard rounding modes specified in IEEE 754-2008. `vmi_FPR_AWAY` specifies round-to-nearest, ties-away rounding, as defined in IEEE 754-2008, and is applicable only when converting from floating point to a signed or unsigned integer result. `vmi_FPR_ODD`

specifies round-to-odd (or Von Neumann) rounding, where inexact results always have the least significant fraction bit set, and is applicable only when converting to a floating point result.

If non-`NULL`, argument `config` specifies an operation-specific configuration that overrides the default FPU configuration for this operation.

It is possible for the conversion to generate exceptions: for example, converting from a non-integral floating point source to an integer result will always generate a precision exception. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `fd` and `flags` will be updated with the conversion result and flags. For the SIMD variant, the flags are a bitwise-or of flags resulting from each individual operation.

Example

This example is derived from the OVP MIPS model.

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

void mipsEmitFPUCVT(
    mipsInstructionInfoP info,
    mipsFPCtrl           ctrl,
    mipsP                tc,
    vmiFType             fdType,
    vmiReg               fd,
    vmiFType             fsType,
    vmiReg               fs,
    vmiReg               sticky,
    vmiReg               cause,
    vmiFPRC              rm,
    Uns32                num
) {
    vmiFPConfigCP config = getOpConfig(tc, ctrl);

    // set initial floating point operation attributes
    emitStartFPOperation(info, tc, cfgCVTAttrs(tc));

    // emit operation
    vmimtFConvertSimdRR(num, fdType, fd, fsType, fs, rm, cause, config),

    // set sticky flags on completion
    emitSetSticky(tc, sticky, cause);
}
```

Notes and Restrictions

None.

9.15 vmimtFUnopRR, vmimtFUnopSimdRR

Prototypes

```
void vmimtFUnopRR(
    vmiFType      type,
    vmiFUnop      op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        flags,
    vmiFPConfigCP config
);

void vmimtFUnopSimdRR(
    vmiFType      type,
    Uns32         num,
    vmiFUnop      op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        flags,
    vmiFPConfigCP config
);
```

Description

These functions emit code to perform a floating point unary operation on an argument in register `fa`, writing the result in register `fd`, both of type `type`. For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fd` and `fa` indicate the first register in a contiguous vector.

It is possible for the operation to generate exceptions. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `fd` and `flags` will be updated with the operation result and flags. For the SIMD variant, the flags are a bitwise-or of flags resulting from each individual operation.

If non-`NULL`, argument `config` specifies an operation-specific configuration that overrides the default FPU configuration for this operation.

Argument `op` is the unary operation to perform. The available unary floating point operations are specified using the `vmiFUnop` enumeration in `vmiTypes.h`:

```
typedef enum {
                    // BASIC ARITHMETIC OPERATIONS
    vmi_FMOV,       // d <- a
    vmi_FABS,       // d <- abs(a), signalling
    vmi_FQABS,      // d <- abs(a), IEEE 754-2008 (clear sign bit only)
    vmi_FNEG,       // d <- -a,     signalling
    vmi_FQNEG,      // d <- -a,     IEEE 754-2008 (toggle sign bit only)
    vmi_FRECIP,     // d <- 1/a
    vmi_FSQRT,      // d <- sqrt(a)
    vmi_FRSQRT,     // d <- 1/sqrt(a)

                    // ROUNDING OPERATIONS
    vmi_FCEIL,      // d <- roundTowardsPositiveInfinity(a)
    vmi_FFLOOR,     // d <- roundTowardsNegativeInfinity(a)
    vmi_FNEAREST,   // d <- roundToNearest(a)
    vmi_FTRUNC,     // d <- roundTowardsZero(a)
    vmi_FROUNDC,    // d <- roundUsingCurrentRoundingMode(a)
```

```
    vmi_FAWAY,      // d <- roundToNearestTiesAway(a)

                    // TRIGONOMETRIC OPERATIONS
    vmi_FSIN,       // d <- sin(a)
    vmi_FCOS,       // d <- cos(a)

                    // LOGARITHMIC OPERATIONS
    vmi_FLOG2,      // d <- log2(a)

                    // USER-DEFINED OPERATIONS
    vmi_FUNUD,      // (implemented with result handler functions)

    vmi_FUNOP_LAST  // KEEP LAST

} vmiFUnop;
```

A user-defined operation may be implemented by specifying an operation of `vmi_FUNUD` and a configuration with `fp32ResultCB` and/or `fp64ResultCB` callbacks.

Example

This example is derived from the OVP MIPS model.

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

void mipsEmitFPUUnop(
    mipsInstructionInfoP info,
    mipsFPCtrl           ctrl,
    mipsP                tc,
    vmiReg               fd,
    vmiReg               fs,
    vmiReg               sticky,
    vmiReg               cause,
    vmiFUnop             op,
    vmiFType             type,
    Uns32                num
) {
    vmiFPConfigCP config = getOpConfig(tc, ctrl);
    mipsFPOpAttr  attrs  = cfgUnopAttrs(tc, op);

    if((op==vmi_FABS) && cfgSimpleAbsNeg(tc)) {

        // IEEE 754-2008 behavior (clear sign bit only)
        vmimtFUnopSimdRR(type, num, vmi_FQABS, fd, fs, cause, 0);

    } else if((op==vmi_FNEG) && cfgSimpleAbsNeg(tc)) {

        // IEEE 754-2008 behavior (clear sign bit only)
        vmimtFUnopSimdRR(type, num, vmi_FQNEG, fd, fs, cause, 0);

    } else {

        // set initial floating point operation attributes
        emitStartFPOperation(info, tc, attrs);

        // emit operation
        vmimtFUnopSimdRR(type, num, op, fd, fs, cause, config),

        // set sticky flags on completion
        emitSetSticky(tc, sticky, cause);
    }
}
```

Notes and Restrictions

None.

9.16 vmimtFBinopRRR, vmimtFBinopSimdRRR

Prototypes

```
void vmimtFBinopRRR(
    vmiFType      type,
    vmiFBinop     op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    vmiFPConfigCP config
);

void vmimtFBinopSimdRRR(
    vmiFType      type,
    Uns32         num,
    vmiFBinop     op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    vmiFPConfigCP config
);
```

Description

These functions emit code to perform a floating point binary operation on arguments in registers `fa` and `fb`, writing the result to register `fd`, all of type `type` (except when `op` is `vmi_FSCALEI`, in which case the second argument is an integer). For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fd`, `fa` and `fb` indicate the first register in a contiguous vector.

It is possible for the operation to generate exceptions. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `fd` and `flags` will be updated with the operation result and flags. For the SIMD variant, the flags are a bitwise-or of flags resulting from each individual operation.

If non-`NULL`, argument `config` specifies an operation-specific configuration that overrides the default FPU configuration for this operation.

Argument `op` is the binary operation to perform. The available binary floating point operations are specified using the `vmiFBinop` enumeration in `vmiTypes.h`:

```
typedef enum {
                    // BASIC ARITHMETIC OPERATIONS
    vmi_FADD,       // d <- a + b
    vmi_FSUB,       // d <- a - b
    vmi_FMUL,       // d <- a * b
    vmi_FDIV,       // d <- a / b

                    // MIN/MAX OPERATIONS
    vmi_FMIN,       // d <- min(a, b)
    vmi_FMAX,       // d <- max(a, b)

                    // SCALE OPERATIONS
    vmi_FSCALEF,    // d <- a * 2^b (floating point b)
```

```
    vmi_FSCALEI,    // d <- a * 2^b (integer b)

                    // QUIET COMPARISON OPERATIONS
    vmi_FQCMPEQ,    // d <-  (a == b)     ? all_ones : all_zeros
    vmi_FQCMPNEQ,   // d <- !(a == b)     ? all_ones : all_zeros
    vmi_FQCMPLT,    // d <-  (a <  b)     ? all_ones : all_zeros
    vmi_FQCMPNLT,   // d <- !(a <  b)     ? all_ones : all_zeros
    vmi_FQCMPLE,    // d <-  (a <= b)     ? all_ones : all_zeros
    vmi_FQCMPNLE,   // d <- !(a <= b)     ? all_ones : all_zeros
    vmi_FQCMPGT,    // d <-  (a >  b)     ? all_ones : all_zeros
    vmi_FQCMPNGT,   // d <- !(a >  b)     ? all_ones : all_zeros
    vmi_FQCMPGE,    // d <-  (a >= b)     ? all_ones : all_zeros
    vmi_FQCMPNGE,   // d <- !(a >= b)     ? all_ones : all_zeros
    vmi_FQCMPOR,    // d <-  ordered(a,b) ? all_ones : all_zeros
    vmi_FQCMPNOR,   // d <- !ordered(a,b) ? all_ones : all_zeros

                    // SIGNALLING COMPARISON OPERATIONS
    vmi_FSCMPEQ,    // d <-  (a == b)     ? all_ones : all_zeros
    vmi_FSCMPNEQ,   // d <- !(a == b)     ? all_ones : all_zeros
    vmi_FSCMPLT,    // d <-  (a <  b)     ? all_ones : all_zeros
    vmi_FSCMPNLT,   // d <- !(a <  b)     ? all_ones : all_zeros
    vmi_FSCMPLE,    // d <-  (a <= b)     ? all_ones : all_zeros
    vmi_FSCMPNLE,   // d <- !(a <= b)     ? all_ones : all_zeros
    vmi_FSCMPGT,    // d <-  (a >  b)     ? all_ones : all_zeros
    vmi_FSCMPNGT,   // d <- !(a >  b)     ? all_ones : all_zeros
    vmi_FSCMPGE,    // d <-  (a >= b)     ? all_ones : all_zeros
    vmi_FSCMPNGE,   // d <- !(a >= b)     ? all_ones : all_zeros
    vmi_FSCMPOR,    // d <-  ordered(a,b) ? all_ones : all_zeros
    vmi_FSCMPNOR,   // d <- !ordered(a,b) ? all_ones : all_zeros

                    // USER-DEFINED OPERATIONS
    vmi_FBINUD,     // (implemented with result handler functions)

    vmi_FBINOP_LAST // KEEP LAST

} vmiFBinop;
```

Operations `vmi_FMIN` and `vmi_FMAX` differ from the IEEE definitions of `minNum` and `maxNum` in that if either argument is a `NaN` then the result is a `QNaN`. This difference is intentional: it allows an operation-specific `QNaN` handler to configure the precise result in this case (the IEEE 754 Specification allows alternative implementations in this case).

Unlike other binary operations, the second argument for operation `vmi_FSCALEI` is a signed integer, 4 bytes in size for 32-bit floating point types and 8 bytes in size for 64-bit and 80-bit floating point types. If this instruction is being used in a SIMD context, then the integer vector alignment must match the floating point vector; in other words, the integers need to have 4-byte alignment for 32-bit floating point operands, 8-byte alignment for 64-bit floating point operands, and 10-byte alignment for 80-bit floating point operands.

Operations `vmi_FQCMPEQ` . . . `vmi_FQCMPNOR` implement the IEEE 754 `compareQuiet` operations. Operations `vmi_FSCMPEQ` . . . `vmi_FSCMPNOR` implement the IEEE 754 `compareSignalling` operations.

A user-defined operation may be implemented by specifying an operation of `vmi_FBINUD` and a configuration with `fp32ResultCB` and/or `fp64ResultCB` callbacks.

Example

This example is derived from the OVP MIPS model.

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

void mipsEmitFPUBinop(
    mipsInstructionInfoP info,
    mipsFPCtrl           ctrl,
    mipsP                tc,
    vmiReg               fd,
    vmiReg               fs,
    vmiReg               ft,
    vmiReg               sticky,
    vmiReg               cause,
    vmiFBinop            op,
    vmiFType             type,
    Uns32                num
) {
    vmiFPConfigCP config = getOpConfig(tc, ctrl);

    // set initial floating point operation attributes
    emitStartFPOperation(info, tc, cfgBinopAttrs(tc, op));

    // emit operation
    vmimtFBinopSimdRRR(type, num, op, fd, fs, ft, cause, config),

    // set sticky flags on completion
    emitSetSticky(tc, sticky, cause);
}
```

Notes and Restrictions

None.

9.17 vmimtFTernopRRRR, vmimtFTernopSimdRRRR

Prototypes

```
void vmimtFTernopRRRR(
    vmiFType      type,
    vmiFTernop    op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        fc,
    vmiReg        flags,
    Bool          roundInt,
    vmiFPConfigCP config
);

void vmimtFTernopSimdRRRR(
    vmiFType      type,
    Uns32         num,
    vmiFTernop    op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        fc,
    vmiReg        flags,
    Bool          roundInt,
    vmiFPConfigCP config
);
```

Description

These functions emit code to perform a floating point ternary operation on arguments in registers `fa`, `fb` and `fc`, writing the result in register `fd`, all of type `type`. For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fd`, `fa`, `fb` and `fc` indicate the first register in a contiguous vector.

It is possible for the operation to generate exceptions. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `fd` and `flags` will be updated with the operation result and flags. The flags are a bitwise-or of flags resulting from each individual operation.

Argument `op` is the ternary operation to perform. The available ternary floating point operations are specified using the `vmiFTernop` enumeration in `vmiTypes.h`:

```
typedef enum {
                    // UNFUSED OPERATION            FUSED OPERATION
    vmi_FMADD,      // d <- (a * b) + c             d <- ( a * b) + c
    vmi_FMSUB,      // d <- (a * b) - c             d <- ( a * b) - c
    vmi_FNMADD,     // d <- -((a * b) + c)          d <- (-a * b) + c
    vmi_FNMSUB,     // d <- -((a * b) - c)          d <- (-a * b) - c
    vmi_FMSUBR,     // d <- c - (a * b)             d <- (-a * b) + c
    vmi_FMADDH,     // d <- ((a * b) + c) / 2.0     d <- (( a * b) + c) / 2.0
    vmi_FMSUBH,     // d <- ((a * b) - c) / 2.0     d <- (( a * b) - c) / 2.0
    vmi_FMSUBRH,    // d <- (c - (a * b)) / 2.0     d <- ((-a * b) + c) / 2.0

    vmi_FTERNOP_LAST// KEEP LAST

} vmiFTernop;
```

If `roundInt` is `True`, then each intermediate result will be rounded to the result type before being used as an operand to the next operation (in other words, the operation is *unfused*). The operations are carried out strictly in the precedence order implied by the `UNFUSED OPERATION` column in the table above: for example, `vmi_FMADDH` will first multiply `a` and `b`, then round the result, then add `c`, then round the result, then divide by 2. Unfused operations can therefore generate multiple rounding events.

If `roundInt` is `False`, then for x87 argument types intermediates are represented in the Intel x87 80-bit format, and for IEEE argument types intermediates are represented using infinite precision. This means that, for IEEE types, such operations correspond to the IEEE *fused-multiply-add* definition. In each case, the operations are carried out as indicated in the `FUSED OPERATION` column in the table above.

If non-`NULL`, argument `config` specifies an operation-specific configuration that overrides the default FPU configuration for this operation.

Example

This example is derived from the OVP MIPS model.

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

void mipsEmitFPUTernop(
    mipsInstructionInfoP info,
    mipsFPCtrl           ctrl,
    mipsP                tc,
    vmiReg               fd,
    vmiReg               fr,
    vmiReg               fs,
    vmiReg               ft,
    vmiReg               sticky,
    vmiReg               cause,
    vmiFTernop           op,
    vmiFType             type,
    Uns32                num,
    Bool                 roundInt
) {
    vmiFPConfigCP config = getOpConfig(tc, ctrl);

    // set initial floating point operation attributes
    emitStartFPOperation(info, tc, cfgTernopAttrs(tc, op));

    // emit operation
    vmimtFTernopSimdRRRR(type, num, op, fd, fs, ft, fr, cause, roundInt, config);

    // set sticky flags on completion
    emitSetSticky(tc, sticky, cause);
}
```

Notes and Restrictions

None.

9.18 vmimtFCompareRR, vmimtFCompareSimdRR

Prototypes

```
void vmimtFCompareRR(
    vmiFType      type,
    vmiReg        relation,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    Bool          allowQNaN,
    vmiFPConfigCP config
);

void vmimtFCompareSimdRR(
    vmiFType      type,
    Uns32         num,
    vmiReg        relation,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    Bool          allowQNaN,
    vmiFPConfigCP config
);
```

Description

These functions emit code to perform a floating point comparison operation on arguments in register `fa` and `fb`, writing the result in register `relation`. The argument registers are of type `type`; output register `relation` is an 8-bit value of type `vmiFPRelation`, which enumerates the four exclusive relations in IEEE Standard 754:

```
typedef enum {
    vmi_FPRL_UNORDERED = 0x1,   // unordered
    vmi_FPRL_EQUAL     = 0x2,   // equal
    vmi_FPRL_LESS      = 0x4,   // less than
    vmi_FPRL_GREATER   = 0x8    // greater than
} vmiFPRelation;
```

For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fa`, `fb` and `relation` indicate the first register in a contiguous vector (of byte size, in the case of `relation`).

It is possible for the comparison to generate exceptions. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `relation` and `flags` will be updated with the comparison result and flags. For the SIMD variant, the flags are a bitwise-or of flags resulting from each individual comparison.

Argument `allowQNaN` specifies the behaviour for IEEE `QNaN` inputs. If `allowQNaN` is `True`, `QNaN` inputs will not cause exceptions and the `vmi_FPRL_UNORDERED` relation will result from comparisons containing these operands. Otherwise, if `allowQNaN` is `False`, `QNaN` inputs will be treated as an error and the invalid operation exception will be raised.

Example

This example is derived from the OVP MIPS model.

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

void mipsEmitFPUCompare(
    mipsInstructionInfoP info,
    mipsP                tc,
    vmiReg               cc,
    vmiReg               fs,
    vmiReg               ft,
    vmiReg               sticky,
    vmiReg               cause,
    vmiFType             type,
    Uns32                num,
    Uns32                cond
) {
    Bool  allowQNaN = !(cond & 0x8);
    Uns8  mask      =  (cond & 0x7);
    Uns32 i;

    // set initial floating point operation attributes - note that when FS is
    // specified, compare flushes denormal operands to zero but *does not* set
    // the I flag
    emitStartFPOperation(info, tc, FA_ALLOW_I_DENORMALS|FA_DONT_SET_I_FLAG);

    // do the compare operation - following actions may not be done if there is
    // a floating-point exception
    vmimtFCompareSimdRR(type, num, cc, fs, ft, cause, allowQNaN, 0);

    // convert from vmiFPRelation bitmask to zero/non-zero flag value
    for(i=0; i<num; i++) {
        vmimtBinopRC(8, vmi_AND, VMI_REG_DELTA(cc, i), mask, 0);
    }

    // set sticky flags on completion
    emitSetSticky(tc, sticky, cause);
}
```

Notes and Restrictions

1. See also functions `vmimtFCompareRRC` and `vmimtFCompareSimdRRC` which allow result values to be specified in a more general form.

9.19 vmimtFCompareRRC, vmimtFCompareSimdRRC

Prototypes

```
void vmimtFCompareRRC(
    Uns8          rdBits,
    vmiFType      type,
    vmiReg        rd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    Bool          allowQNaN,
    Uns32         valueUN,
    Uns32         valueEQ,
    Uns32         valueLT,
    Uns32         valueGT,
    vmiFPConfigCP config
);

void vmimtFCompareSimdRRC(
    Uns8          rdBits,
    vmiFType      type,
    Uns32         num,
    vmiReg        rd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    Bool          allowQNaN,
    Uns32         valueUN,
    Uns32         valueEQ,
    Uns32         valueLT,
    Uns32         valueGT,
    vmiFPConfigCP config
);
```

Description

These functions emit code to perform a floating point comparison operation on arguments in registers `fa` and `fb`, writing the result in register `rd`. The argument registers are of type `type`; output register `rd` is of size `rdBits`. The value assigned to `rd` is selected as follows:

1. If the comparison between `fa` and `fb` produces an *unordered* result, `valueUN` is assigned to `rd`.
2. Otherwise, if `fa` is *equal to* `fb`, `valueEQ` is assigned to `rd`.
3. Otherwise, if `fa` is *less than* `fb`, `valueLT` is assigned to `rd`.
4. Otherwise (`fa` is *greater than* `fb`), `valueGT` is assigned to `rd`.

For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fa`, `fb` and `rd` indicate the first register in a contiguous vector (of size `rdBits`, in the case of `rd`).

It is possible for the comparison to generate exceptions. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `rd` and

`flags` will be updated with the comparison result and flags. For the SIMD variant, the flags are a bitwise-or of flags resulting from each individual comparison.

Argument `allowQNaN` specifies the behaviour for IEEE `QNaN` inputs. If `allowQNaN` is `True`, `QNaN` inputs will not cause exceptions and an *unordered* result will generated for comparisons containing these operands. Otherwise, if `allowQNaN` is `False`, `QNaN` inputs will be treated as an error and the invalid operation exception will be raised.

These functions generalize the behavior of `vmimtFCompareRR` and `vmimtFCompareSimdRR`. For example,

```
vmimtFCompareRR(
    type, relation, fa, fb, flags, allowQNaN, 0
);
```

is exactly equivalent to:

```
vmimtFCompareRRC(
    8, type, relation, fa, fb, flags, allowQNaN,
    vmi_FPRL_UNORDERED, vmi_FPRL_EQUAL, vmi_FPRL_LESS, vmi_FPRL_GREATER,
    0
);
```

Example

This example is derived from the OVP ARM model. The comparison is designed to directly assign to a value of type `armArithFlags` in the ARM processor structure.

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

// arithmetic flag indices
typedef enum armAFIE {
    AFI_Z,      // zero flag
    AFI_N,      // sign flag
    AFI_C,      // carry flag
    AFI_V,      // overflow flag
    AFI_LAST,   // KEEP LAST: for sizing
} armAFI;

// arithmetic flags
typedef struct armArithFlagsS {
    Uns8 f[AFI_LAST];
} armArithFlags, *armArithFlagsP;

//
// Return a mask bit that sets the given flag in an armArithFlags structure
//
#define FLAG_MASK(_ID)  (1<<(_ID*8))

//
// Compare fa to fb, setting armFlags and flags
//
void armEmitFCompareRRF(
    armMorphStateP state,
    vmiFType       type,
    vmiReg         armFlags,
    vmiReg         fa,
    vmiReg         fb,
    Bool           allowQNaN
) {
    // do prologue actions
```

```
    armStartFPOperation(state);

    // do the compare
    vmimtFCompareRRC(
        32,
        type,
        armFlags,
        fa,
        fb,
        ARM_FP_STICKY,
        allowQNaN,
        FLAG_MASK(AFI_C) | FLAG_MASK(AFI_V),    // unordered
        FLAG_MASK(AFI_Z) | FLAG_MASK(AFI_C),    // equal
        FLAG_MASK(AFI_N),                       // less
        FLAG_MASK(AFI_C),                       // greater
        armGetOpConfig(state)
    );
}

// processor structure
typedef struct armS {
    . . .
    armArithFlags  aflags;              // arithmetic flags
    armArithFlags  sdfpAFlagsAA32;      // FPU comparison flags (AArch32 state)
    . . .
} arm;

//
// morph-time macros to calculate variable offsets to flags in an arm structure
//
#define ARM_AFLAGS              ARM_CPU_REG(aflags)
#define ARM_AFLAGS_AA32         ARM_CPU_REG(sdfpAFlagsAA32)

//
// Common code to execute VFP VCMP instructions
//
static void emitVCmpVFP(armMorphStateP state, vmiReg rd, vmiReg rm, Uns32 ebytes) {

    Bool   allowQNaN = state->attrs->allowQNaN;
    vmiReg armFlags  = IS_AARCH64(state) ? ARM_AFLAGS : ARM_AFLAGS_AA32;

    armEmitFCompareRRF(state, bytesToFType(ebytes), armFlags, rd, rm, allowQNaN);
}
```

Notes and Restrictions

None.

10 Miscellaneous Operations

This section describes miscellaneous emission functions for simulator control and instruction counting.

10.1 vmimtYield

Prototype

```
void vmimtYield(void);
```

Description

This function emits code that explicitly suspends execution of the current processor on completion of the current simulated instruction to allow other processors in a multiprocessor simulation to run. The processor will run again in a delta cycle when all other runnable processors have executed.

Notes and Restrictions

None.

10.2 vmimtHalt

Prototype

```
void vmimtHalt(void);
```

Description

This function emits code that halts execution for the current processor. It is used to simulate hardware halt instructions.

A processor that has been halted may be restarted by a call to the run time functions `vmirtRestartNow` or `vmirtRestartNext` (defined in `vmiRt.h`). This call will typically be made within an event handler routine or as a call made by the implementation of a special instruction executed by another processor in the simulated platform.

Notes and Restrictions

None.

10.3 vmimtInterrupt

Prototype

```
void vmimtInterrupt(void);
```

Description

This function causes simulation to stop on completion of the current simulated instruction and return from the calling context (the `icmSimulate` or `icmSimulatePlatform` invocation) with a stop reason of `ICM_SR_INTERRUPT`.

Notes and Restrictions

None.

10.4 vmimtExit

Prototype

```
void vmimtExit(void);
```

Description

This simulation control function emits code that ends execution for the current processor. In a multiprocessor simulation, remaining processors will continue execution. If the current processor is the only running processor, then the simulation run will end.

This function is typically used to model special trap operations intended to cause simulation termination.

Example

```
#include "vmi/vmiMt.h"

#define SIM_EXIT_CODE 99

static void emitTrap(Uns32 trapCode) {
    if(trapCode==SIM_EXIT_CODE) {
        vmimtExit();
    }
}
```

Notes and Restrictions

None.

10.5 vmimtFinish

Prototype

```
void vmimtFinish(void);
```

Description

This simulation control function emits code that ends simulation. Note that this is different to `vmimtExit` because simulation will end even if other processors in a multiprocessor platform are still running.

This function is typically used to model special trap operations intended to cause simulation termination.

Example

```
#include "vmi/vmiMt.h"

#define SIM_FINISH_CODE 999

static void emitTrap(Uns32 trapCode) {
    if(trapCode==SIM_FINISH_CODE) {
        vmimtFinish();
    }
}
```

Notes and Restrictions

None.

10.6 vmimtEndBlock

Prototype

```
void vmimtEndBlock(void);
```

Description

This simulation control function forces the current native code block to be terminated after the current simulated instruction – the next simulated instruction is guaranteed to be in a different code block. The circumstances in which this might be useful are described below.

As explained in the section *Interaction with Imperas Simulators*, the simulator maintains native translations of simulated instructions in *code blocks* in a dictionary. The bounds of each code block are determined as follows:

1. Any instruction targeted by a jump instruction starts a code block.
2. Any jump instruction terminates a code block.

The simulator automatically determines the bounds of code blocks as it executes, and usually the two rules described above are adequate. There are, however, two circumstances that cause problems:

1. *Mode changes within a code block*
   The simulator can maintain many dictionaries for each simulated processor. This is useful because often instructions have different behavior depending on processor mode: for example, an instruction that writes to a status register may succeed in kernel mode but cause a privileged instruction trap in user mode. By maintaining separate dictionaries for user and kernel modes, the decision about the instruction behavior can be made at *morph time* instead of *run time*, which means the native code is much more efficient.
   If an instruction is executed that causes a mode change, then the next instruction must by definition be in a different code block (because it must be in a different dictionary). In this case, `vmimtEndBlock` should be called to force the current code block to be terminated.
2. *Bogus instruction patterns*
   A simulation may set unused memory to a specific pattern, for example `0xdeadbeef`. This pattern usually coincides with a possibly legal but highly unlikely instruction (although the simulator has no way of knowing this).
   If during simulation a branch is made to uninitialized memory (because of an application program error), the simulator will start translation of instructions with the unused memory pattern.
   If the pattern corresponds to an instruction that is not a branch, then the effect will be to create a very large code block consisting of very many repetitions of the translated unused pattern, which can cause an apparent simulator freeze while the vast code block is processed.

For performance reasons, it is therefore sensible to terminate the current code block whenever the unlikely-but-legal instruction is encountered.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiFetch.h”

#define DEAD 0xdeadbeef

VMI_MORPH_FN(cpuxMorphInstruction) {

    Uns32 instr = vmicxtFetch4Byte(processor, thisPC);

    // normal decode cascade – unlikely-but-legal instruction is translated here
    if(IS_OP_ADD(instr)) {
        emitBinopReg(instr, vmi_ADD, &flagsCO);
    } else if(IS_OP_ADDC(instr)) {
        emitBinopReg(instr, vmi_ADC, &flagsCO);
    } else {
  . . . etc . . .
    }

    // terminate the current code block if this was the unlikely-but-legal
    // instruction
    if(instr==DEAD) {
        vmimtEndBlock();
    }
}
```

Notes and Restrictions

None.

10.7 vmimtGetBlockMask

Prototype

```
void vmimtGetBlockMask(vmiReg blockMask);
```

Description

This simulation control function copies the current processor *block mask* to the 32-bit register `blockMask`. The block mask is used to validate that assumptions made when the current block was translated still apply when it is executed. If the assumptions no longer apply, the code block is automatically deleted and remorphed. The assumptions for the current block are specified using `vmimtValidateBlockMask`.

Example

This example shows how a block mask can be updated by a read-modify-write sequence specified at morph time:

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

// processor structure definition
typedef struct cpuxS {
    Uns32 tmp;
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_TMP        CPUX_OFFSET(tmp)

static void updateBlockMask(Uns32 toggleMask) {
    vmimtGetBlockMask(CPUX_TMP);
    vmimtBinopRC(32, vmi_XOR, CPUX_TMP, toggleMask, 0);
    vmimtSetBlockMaskR(CPUX_TMP);
}
```

Notes and Restrictions

See the description of `vmirtSetBlockMask` in the *VMI Run Time Function Reference* for extensive details on the usage of block masks.

10.8 vmimtSetBlockMaskC

Prototype

```
void vmimtSetBlockMaskC(Uns32 blockMask);
```

Description

This simulation control function sets the current processor *block mask* to the value `blockMask`. The block mask is used to validate that assumptions made when the current block was translated still apply when it is executed. If the assumptions no longer apply, the code block is automatically deleted and remorphed. The assumptions for the current block are specified using `vmimtValidateBlockMask`.

Notes and Restrictions

See the description of `vmirtSetBlockMask` in the *VMI Run Time Function Reference* for extensive details on the usage of block masks.

10.9 vmimtSetBlockMaskR

Prototype

```
void vmimtSetBlockMaskR(vmiReg blockMask);
```

Description

This simulation control function sets the current processor *block mask* to the value of the 32-bit register `blockMask`. The block mask is used to validate that assumptions made when the current block was translated still apply when it is executed. If the assumptions no longer apply, the code block is automatically deleted and remorphed. The assumptions for the current block are specified using `vmimtValidateBlockMask`.

Example

This example shows how a block mask can be updated by a read-modify-write sequence specified at morph time:

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

// processor structure definition
typedef struct cpuxS {
    Uns32 tmp;
} cpux, *cpuxP;

#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_TMP        CPUX_OFFSET(tmp)

static void updateBlockMask(Uns32 toggleMask) {
    vmimtGetBlockMask(CPUX_TMP);
    vmimtBinopRC(32, vmi_XOR, CPUX_TMP, toggleMask, 0);
    vmimtSetBlockMaskR(CPUX_TMP);
}
```

Notes and Restrictions

See the description of `vmirtSetBlockMask` in the *VMI Run Time Function Reference* for extensive details on the usage of block masks.

10.10 vmimtValidateBlockMask

Prototype

```
void vmimtValidateBlockMask(Uns32 modeMask);
```

Description

This simulation control function is used in combination with the *VMI Run Time API* function `vmirtSetBlockMask` to validate that assumptions made when the block was translated still apply when it is executed. If the assumptions no longer apply, the code block is automatically deleted and remorphed. Block mask behavior is implemented in two parts:

1. At *morph time*, the current value of bits in the built-in 32-bit block mask selected using a bitwise-and with `modeMask` are recorded with the code block.
2. At *run time*, bits in the built-in 32-bit block mask are again selected using a bitwise-and with `modeMask`. These selected bits are then compared with the morph-time value saved with the block. If these values differ, the block is deleted and retranslated; otherwise, the block is executed.

Block masks are typically used to efficiently implement instructions whose behavior differs based on enable bits that are likely to have a constant value when a block is executed. For example, floating point instructions are typically enabled by a processor configuration bit: if enabled, the instruction executes normally; if disabled, an exception is taken. By encoding the enable bit in a blockmask that is checked using `vmirtValidateBlockMask`, morph time code can create *either* the floating point instruction code *or* code to take an exception, not both. This results in much more compact JIT code with fewer branches.

Notes and Restrictions

1. See the description of `vmirtSetBlockMask` in the *VMI Run Time Function Reference* for extensive details and an example of the use of this function.
2. See related function `vmimtValidateBlockMaskR`, which allows block masks to be validated using any processor register instead of the built-in 32-bit block mask.

10.11 vmimtValidateBlockMaskR

Prototype

```
void vmimtValidateBlockMaskR(
    Uns32  bits,
    vmiReg r,
    Uns64  modeMask
);
```

Description

This simulation control function is used to validate that assumptions made when a block was translated still apply when it is executed. If the assumptions no longer apply, the code block is automatically deleted and remorphed. Block mask behavior is implemented in two parts:

1. At *morph time*, the current value of bits in register `r` selected using a bitwise-and with `modeMask` are recorded with the code block.
2. At *run time*, bits in register `r` are again selected using a bitwise-and with `modeMask`. These selected bits are then compared with the morph-time value saved with the block. If these values differ, the block is deleted and retranslated; otherwise, the block is executed.

Block masks are typically used to efficiently implement instructions whose behavior differs based on enable bits that are likely to have a constant value when a block is executed. For example, floating point instructions are typically enabled by a processor configuration bit: if enabled, the instruction executes normally; if disabled, an exception is taken. By encoding the enable bit in a blockmask that is checked using `vmirtValidateBlockMaskR`, morph time code can create *either* the floating point instruction code *or* code to take an exception, not both. This results in much more compact JIT code with fewer branches.

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. If a simulated instruction modifies register r, *ensure that the current code block is terminated using `vmimtEndBlock`*. If this is not done, then subsequent simulated instructions in the code block may operate incorrectly if their behavior depends on modified bits in the block mask register.
3. See the description of `vmirtSetBlockMask` in the *VMI Run Time Function Reference* for extensive details on the behavior of block masks.

10.12 vmimtTagBlock

Prototype

```
void vmimtTagBlock(vmiBlockTag tag);
```

Description

This simulation control function is used to tag a block at morph time to enable it to be conditionally preserved or deleted by a later call to VMI run time function `vmirtFlushTargetModeTagged`. The `tag` argument is an enumerated type defined in `vmiTypes.h`:

```
typedef enum vmiBlockTagE {
    VBT_NA = 0,         // no tag
    VBT_1  = (1<<0),    // tag 1
    VBT_2  = (1<<1),    // tag 2
    VBT_3  = (1<<2),    // tag 3
    VBT_4  = (1<<3),    // tag 4
    VBT_5  = (1<<4),    // tag 5
    VBT_6  = (1<<5),    // tag 6
    VBT_7  = (1<<6),    // tag 7
    VBT_8  = (1<<7),    // tag 8
} vmiBlockTag;
```

When a block is tagged, the tag value is combined with any existing tags for that block using bitwise-or. This means that multiple calls to `vmimtTagBlock` add tags cumulatively to the current code block, with up to eight distinct tags supported.

Example

This example is from the OVP ARC processor model. The ARC processor implements a *zero-overhead loop* construct in which a code block can only be reused if a limiting address held in a register holds a certain value. Setting the register to a new address must invalidate any blocks at that address unless they implement zero-overhead loop behavior, in which case they can be preserved.

Unnecessary block flushes are suppressed by using tagged blocks as follows:

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

void arcEmitZeroOverheadLoop(arcMorphStateP state) {

    if(state->atZOL) {

        // when executing this block, validate that lp_end has the same value
        // that it has when code for the block was generated
        vmimtValidateBlockMaskR(ARC_GPR_BITS, ARC_AUX_REG(lp_end), -1);

        // tag this block to avoid unnecessary deletion when lp_end changes
        vmimtTagBlock(VBT_1);

        . . .
    }
}
```

In a callback that is activated when the `lp_end` register changes, code blocks at the end address implied by the new value of the `lp_end` register are flushed using this idiom:

```
static void flushTargetZOL(arcP arc, Uns32 lpEnd) {

    for(mode=0; mode<ARC_MODE_LAST; mode++) {
        vmirtFlushTargetModeTagged(
            (vmiProcessorP)arc, lpEnd, mode, VBT_1, VBT_1, False
        );
    }
}
```

The call to `vmirtFlushTargetModeTagged` in this case will flush any code block at address lpEnd for which the expression

```
((block->tag & VBT_1) == VBT_1)
```

is `False`. This flushes any code block that was not tagged as a zero-overhead loop block when it was constructed.

Notes and Restrictions

None.

10.13 vmimtICount

Prototype

```
void vmimtICount(Uns32 bits, vmiReg rd);
```

Description

The simulator automatically maintains a count of executed instructions for every processor. This function emits code to load the instruction count for the current processor into simulator register `rd`.

Example

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

// processor structure definition
typedef struct cpuxS {
    Uns64 countReg;     // 64-bit count register
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_COUNT_REG  CPUX_OFFSET(countReg)

static void emitGetICount(void) {
    vmimtICount(64, CPUX_COUNT_REG);
}
```

Notes and Restrictions

1. The `bits` argument must be 8, 16, 32 or 64.

11 QuantumLeap Parallel Simulation Support

As of VMI version 6.0.0, Imperas Professional Simulation products implement a parallel simulation algorithm called *QuantumLeap*, which enables multicore platform simulation to be distributed over separate threads on multiple cores of the host machine for improved performance. Refer to the *OVP and CpuManager User Guide* for more information about QuantumLeap usage.

This section describes functions required to make processor models compatible with QuantumLeap.

11.1 vmimtAtomic

Prototype

```
void vmimtAtomic(void);
```

Description

This function indicates that the current instruction requires synchronization and that all other processor threads must be suspended while the instruction executes. `vmimtAtomic` should be used in three cases:

1. In a test-and-set instruction that reads, modifies and writes memory.
2. In the first instruction of a load/store exclusive or speculate/commit instruction pair.
3. When emitting code for any instruction which also emits an embedded call to a function which accesses shared data in an uncontrolled manner.

Refer to the *OVP Processor Modeling Guide* for a detailed explanation of when `vmimtAtomic` should be used.

Example

This example demonstrates the use of `vmimtAtomic` to indicate that a test-and-set instruction must execute atomically.

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32       // number of GPRs
#define CPUX_GBITS 32       // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
    Uns32 tmp;              // 32-bit temporary
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_REG(_R)    CPUX_OFFSET(regs[_R])
#define CPUX_TMP        CPUX_OFFSET(tmp)

static void emitSWP(Uns32 rd, Uns32 ra, Uns32 c) {
    // this instruction must execute atomically
    vmimtAtomic();
    // exchange rd and ra(c)
    vmimtLoadRRO(
        CPUX_GBITS, CPUX_GBITS, c, CPUX_TMP, CPUX_REG(ra),
        MEM_ENDIAN_BIG, True, True
    );
    vmimtStoreRRO(
        CPUX_GBITS, c, CPUX_REG(ra), CPUX_REG(rd),
        MEM_ENDIAN_BIG, True
    );
    vmimtMoveRR(CPUX_GBITS, CPUX_REG(rd), CPUX_TMP);
}
```

Notes and Restrictions

1. Refer to the *OVP Processor Modeling Guide*.